Introduction to RT–11

Order Number AA–5281E–TC

August 1991

This manual is an introductory manual for the RT–11 operating system. Its purpose is to acquaint new users with the RT–11 commands that perform common system operations. This manual presents the background material necessary to understand the system operations. It also contains a series of command examples and demonstration exercises that complement the text.

Revision/Update Information: This manual supersedes *Introduction to RT–11*, AA–5281D–TC.

Operating System: RT–11 Version 5.6

Digital Equipment Corporation
Maynard, Massachusetts

First Printing, March 1980
Revised, March 1983
Updated, July 1984
Updated, October 1985
Revised, February 1989
Revised, August 1991

S1524

This document was prepared using VAX DOCUMENT, Version 1.2.

Contents

Chapter 12 Using the RT–11 Communications Facilities

Chapter 13 Defining Your Own Commands

Chapter 21 Constructing Library Files

Chapter 22 Debugging a User Program

Glossary

Figures

Tables

Preface

This manual contains a totally revised introduction to the RT–11 operating system and is appropriate for both beginning and veteran users of RT–11. Unless indicated in the text, this manual describes the RT–11 operating system as it is distributed.

How you use this manual depends largely on your familiarity with RT–11. If you are not familiar with RT–11, you should start at the beginning and proceed in an orderly manner through the manual. If you are familiar with RT–11, this Preface indicates the appropriate course to follow.

This manual contains 22 chapters and a glossary. The chapters are logically divided into four parts. You are encouraged to begin with Part I and then proceed through successive parts. The first page of each part contains a short abstract of each chapter.

- Part I, Basics for the New User, describes the fundamentals of the operating system. If you are not familiar with RT–11, read this part very carefully and perform all indicated exercises. Chapter 4 describes and demonstrates the RT–11 text editor, KED. Other chapters in this manual assume you have worked through and understand the exercises in Chapter 4.

  If you are familiar with RT–11, you may still benefit from examining Part I, as the concepts presented there are referenced thoughout the manual. By reading Part I, you can be sure that your understanding of a concept agrees with the manual.

- Part II, Using the RT–11 Monitors, describes the monitors and environments that are available with RT–11.

  Read Chapter 5 for an explanation of the foreground and background job environments. Read Section 6.1.1 to determine if your processor can support the mapped monitors. If your processor can support mapped monitors, proceed through the rest of Part II.

- Part III, Using RT–11 Features, describes features that are available with RT–11.

  Initially, you should read the introductory sections in each chapter to determine if you are interested in a particular feature. Once you have discovered what is available, you can then go back through Part III and work through the explanations and examples in applicable chapters.

- Part IV, Programming with RT–11, describes the extensive RT–11 support for programming in the BASIC–PLUS, FORTRAN IV, FORTRAN–77, and MACRO languages.

  The chapters you read in Part IV is determined by which programming languages interest you. Chapter 16, Choosing a Programming Language, describes some of the major differences between programming languages and will help you determine which chapters in Part IV are appropriate.

Conventions

The following conventions are used in this manual.

Convention	Meaning

Black print	In examples, black print indicates output lines or prompting characters that the system displays. For example:
	`.BACKUP/INITIALIZE DL0:F*.FOR DU1:WRK`
	`Mount output volume in DU1:; continue? Y`

Red print	In examples, red print indicates user input.
Braces ({ })	In command syntax examples, braces enclose options that are mutually exclusive. You can choose only one option from the group of options that appears in braces.
Brackets ([ ])	Square brackets in a format line represent optional parameters, qualifiers, or values.
$\begin{bmatrix}\text{/option1}\\\text{/option2}\end{bmatrix}$	Square brackets in an option summary surrounding two or more options mean that you can specify any one of the enclosed options or any combination of them. The following example means you can specify either the /BEFORE or the /SINCE option or a combination of both to express one or two ranges of dates: $\begin{bmatrix}\text{/BEFORE[:date]}\\\text{/SINCE[:date]}\end{bmatrix}$
Bold options	**Bold** options in an option summary indicate default options; that is, the option that RT–11 uses if you do not specify any choice of action.
lowercase characters	In command syntax examples, lowercase characters represent elements of a command for which you supply a value. For example: `DELETE filespec`
UPPERCASE characters	In command syntax examples, uppercase characters represent elements of a command that should be entered exactly as given.
$\boxed{\text{RET}}$	$\boxed{\text{RET}}$ in examples represents the RETURN key. Unless the manual indicates otherwise, terminate all commands or command strings by pressing $\boxed{\text{RET}}$.
$\boxed{\text{RETURN}}$	$\boxed{\text{RETURN}}$ in the text represents the RETURN key.

Convention	Meaning
CTRL/*x*	CTRL/*x* indicates a control-key sequence. While pressing CTRL key, press another key. For example: CTRL/C

Associated Documents

Basic Books

- *Guide to RT–11 Documentation*
- *PDP–11 Keypad Editor User's Guide*
- *PDP–11 Keypad Editor Reference Card*
- *RT–11 Commands Manual*
- *RT–11 Quick Reference Manual*
- *RT–11 Master Index*
- *RT–11 System Message Manual*
- *RT–11 System Release Notes*

Installation Specific Books

- *RT–11 Automatic Installation Guide*
- *RT–11 Installation Guide*
- *RT–11 System Generation Guide*

Programmer Oriented Books

- *RT–11 IND Control Files Manual*
- *RT–11 System Utilities Manual*
- *RT–11 System Macro Library Manual*
- *RT–11 System Subroutine Library Manual*
- *RT–11 System Internals Manual*
- *RT–11 Device Handlers Manual*
- *RT–11 Volume and File Formats Manual*
- *DBG–11 Symbolic Debugger User's Guide*

Part I
Basics for the New User

Part I introduces the new user to the RT–11 operating system. This part is divided into four chapters.

- Chapter 1, RT–11 System Components, describes the major components that make up the operating system hardware and software.
- Chapter 2, Getting Started, describes basic concepts and procedures you should understand when using the operating system.
- Chapter 3, Managing Files, describes useful commands for managing files.
- Chapter 4, Using the Keypad Editor (KED), describes using the KED editor to create and edit text files.

Chapter 1

RT–11 System Components

A computer system is a collection of components that works together to process data. This chapter describes the following components of the RT–11 computer system:

- **Hardware**

  The hardware components are the mechanical devices in the system, the machinery and the electronics that perform physical functions.

- **Software**

  The software components are the programs written for the system. These programs perform logical and mathematical operations, provide the basic tools for manipulating information, and provide a means for you to control the system.

- **Documentation**

  Documentation includes the manuals and listings that tell you how to use the hardware and software.

Collectively, these components provide a complete computer system that lets both novice and expert use a computer.

1.1 System Hardware

The hardware components of the RT–11 computer system consist of the computer and its peripheral devices — machinery external to the computer that supplements the computer by providing resources for operations the computer cannot handle alone.

1.1.1 Computer

The computer has two basic components:

- **Processor**

  The *processor* or the *central processing unit* (CPU) is the control center of the computer. It performs all data processing, such as comparing information, calculating a value, and printing, displaying, or storing information. The RT–11 computer system is constructed around a Digital PDP–11 processor. Any PDP–11 model can be used in an RT–11 system.

- **Memory**

  Memory or *main memory* is a series of storage locations within the computer where the computer processes your data. The amount of memory contained in a PDP–11 computer varies depending on the computer: the smallest memory size is 32K bytes; the largest possible size is 4096K bytes. The minimum memory distributed with a MicroPDP–11 computer is 512K bytes.

1.1.2 Peripheral Devices

The RT–11 computer system requires a minimum of two peripheral devices: a terminal and a disk storage device with its medium.

1.1.2.1 Terminal

Your interaction with the computer system occurs through the terminal. You enter information — operating commands, for example — from the terminal's keyboard, which you use much like a typewriter keyboard. The computer, in turn, displays information and messages on the terminal's printer or screen.

Generally, an RT–11 computer system has one terminal through which all interaction between the system and the user takes place. This is called the *console terminal*. If the system has more than one terminal, one of them is still designated the console terminal and the system is still considered a single-user operating system. However, the other terminals can be controlled by one or more application programs.

1.1.2.2 Storage Devices

Although you should have more than one storage device, one disk may be adequate. Depending on your computer system, you can have any combination of disk drives, diskette drives, and magnetic tape drives.

With two storage devices you can use one to hold your operating system files, and the other to back up (make a copy of) the operating system files. This way, you have a means of replacing your RT–11 files if they are damaged or accidentally deleted.

Some disk drives contain *fixed* disks; that is, they come with a hard disk permanently installed in them. For example, the MicroPDP–11 computer is configured with a fixed hard disk.

Other storage devices are not distributed with a medium permanently installed in them. You insert the storage medium or volume (disk, diskette, magtape) in these drives when you want to store or retrieve information.

Each drive has a device name (2-letter mnemonic) and a unit number (0, 1, 2, . . . n). The mnemonic identifies the type of hardware interface to which the device is connected; and the unit number identifies the specific drive, since you can have more than one drive for each type of hardware interface.

Once the storage volume is in the drive, you access it by the device name and unit number. Then, the storage medium becomes an area apart from the computer's main memory to hold your information. This is further explained in Chapter 2.

Storage devices and volumes are classified in three ways according to their use:

- **System (SY)**

  The system device and volume contains the operating system files. The system boots (starts up) from this volume when you turn on the computer. The system volume can be identified by its physical device mnemonic and unit number or by the logical (symbolic) name SY.

- **Default storage (DK)**

  The default storage volume stores files you create when you do not specify a device in which to store them. This volume can be identified by its physical device memnonic and unit number or by the logical (symbolic) name DK.

  When you first install the operating system, the system device logical name (SY) and the default storage device logical name (DK) represent the same device. However, after you install the system files, you can reassign the logical name DK to a different device. See Chapter 2 for an explanation of why you would want to do this and the commands that enable you to do this.

- **Backup**

  You use the backup device to create backup copies of your own or of the system files. You do this by copying the files from the volume on the system device to the volume on the backup device. If your operating system does not come with a backup copy of the system files, then you should make one when you install the system.

  The backup device should have a removable volume (rather than a fixed one). This device has no logical name automatically assigned to it by the operating system.

1.1.2.3 Optional Devices

You can add optional peripheral devices to your computer system according to your needs. For example, if your computer system is used primarily for program development, it may have extra storage devices and a high-speed printing device. Computer systems used in a laboratory environment may have graphics display hardware and an analog-to-digital converter. Computer systems that provide (or use) information in conjunction with another kind of computer system often have a magtape device and an auxiliary communication port.

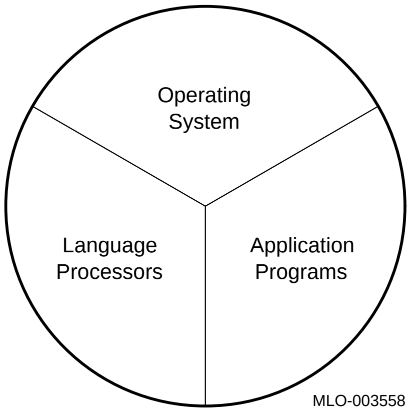
The hardware configuration of your RT–11 computer system includes the computer, the terminal, storage devices, and any other devices you choose to add.

1.2 System Software

System software is a collection of programs that effectively transforms the system hardware components into usable tools. These programs include operations, functions, and routines that make it easier for you to use the hardware to solve problems and produce results. For example, some system programs store and retrieve data among the peripheral devices. Others perform difficult or lengthy mathematical calculations. Some programs let you create, edit, and process application programs of your own. Still others handle entire applications for you.

As shown in Figure 1–1, system software always includes an operating system, which is the *intelligence* of the computer system. In addition, the system software usually includes one or several language processors and it sometimes includes specific applications. These components make the whole called system software. However, they vary in size depending on what each component includes in a specific software system.

Figure 1–1: System Software

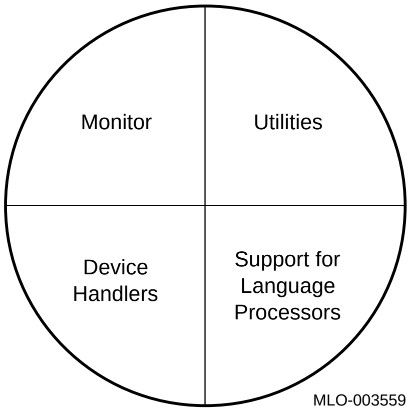


1.2.1 The RT–11 Operating System

The operating system is a collection of programs that organizes all the hardware and software resources of the computer system into a working unit and gives you control. It lets you create, store, and manage files and create and run programs of your own.

The RT–11 operating system consists of the four types of programs shown in Figure 1–2. The size of each of these components varies depending on what each includes in a specific operating system.

Figure 1–2: RT–11 Operating System



- **Monitor**

  The monitor (executive) program is the link between the system hardware, the system software, and you. You operate the computer through the monitor. Its major functions are to:

  Accept, acknowledge, and process commands that you or a program give the system

  Control all input to and output from the system

  Time or schedule when jobs (programs) are run

  Maintain the file system

  Check for abnormal conditions in the system and issue clear, informative messages

  Control the way the system uses memory

  Monitor Components

  The monitor consists of three major components:

  — **Resident monitor (RMON)**

  The resident monitor provides the console terminal service and central program code necessary for both system and user programs. It always remains in computer's main memory, regardless of system operations, and is the basic source of control in the computer.

  — **Keyboard monitor (KMON)**

  The keyboard monitor, the most visible part of the monitor, interprets the command language you use to communicate with the computer. When you see the system prompt (a period), KMON is waiting for you to give it a command.

  Because you need KMON only when you issue commands, KMON sometimes does not stay in memory but is replaced, or *overlaid*, by other programs if they need the space. However, KMON returns as soon as the other program terminates.

  — **User service routines (USR)**

  The user service routines access information stored on disks and tapes; for example, routines in the USR open and close files. Depending on how your system is set up, the USR can be swapped in and out of main memory in the same way as KMON.

  Monitor Types

  RT–11 has three types of monitors, each with its own RMON, KMON, and USR components. Which monitor type you choose is dictated by the hardware and the application.

When you install your operating system, you choose one of the following monitors to run the system:

— **Unmapped (SB and FB)**

  RT–11 distributes two unmapped monitors; the single job (SB) and foreground /background (FB). The SB monitor is the simplest and smallest monitor and can support only one utility or user program at a time. This monitor could be ideal for small systems being used for one specific application with little system overhead.

  The FB monitor includes all the features of the SB monitor as well as its foreground/background feature. This feature lets one job run at high priority in the *foreground*, while KMON or a user-written program runs at low priority in the *background*. For example, with the FB monitor, you can be editing a file in the background while a time-critical program is collecting, storing, and analyzing data in the foreground.

— **Single-mapped (XM)**

  The XM monitor gives you all the features of the unmapped monitors and extends the amount of memory normally available on a PDP–11 computer.

  The XM monitor also gives you the system job feature, which means you can run programs as system jobs to keep the background free for entering commands and running other jobs and the foreground free for a time-critical job. Chapter 7 further explains the system job feature.

  The XM monitor lets you run up to eight programs at the same time. You are limited only by the size of the programs and the amount of memory in your computer.

  (A single-job single-mapped monitor can be built through the system generation process. Look for the XB entry in the *RT–11 Master Index* for references throughout the documentation.)

— **Fully-mapped (ZM)**

  The ZM monitor gives you all the features of the XM monitor and also supports separated Instruction and Data address space and Supervisor Mode. The ZM monitor lets you take advantage of all the features in the current PDP–11 processors.

  (A single-job fully-mapped monitor can be built through the system generation process. Look for the ZB entry in the *RT–11 Master Index* for references throughout the documentation.)

You can tell which monitor your system has from the first line the system displays on your terminal screen when you boot (start) your system. That line identifies the operating system, the monitor, and the version number of the operating system. For example, the following line says that you are running Version 5.6 of the RT–11 operating system with the XM monitor controlling system operations:

```
RT-11XM V05.6
```

Once you have booted your system, you can identify your monitor by issuing the following command at the period (.) prompt:

```
.SHOW CONFIGURATION RET
```

The first line of the configuration display identifies the operating system, the monitor, and the version number of the operating system. See Chapters 5 and 6 for more complete descriptions of the RT–11 monitors.

- **Device handlers**

  Device handlers provide the interface to the hardware devices that are part of the computer system. Device handlers transmit control signals and data to and from your system's peripheral hardware devices (printer, terminal, and so on). A device handler exists for every type of peripheral device that the system supports.

- **Utilities**

  Utilities are programs that cover a wide range of resources; such programs let you create and edit text files, manage programs and files, and locate user-programming errors. Some of the utility programs in the RT–11 operating system are as follows:

  — *Editors*, which let you create and modify programs, memos, or any text file you wish to create.

  — *File management programs*, which let you manipulate and maintain your programs and data — to transfer them between devices, to update them, and to delete them when you are done with them.

  — *Communication programs*, which let you communicate with other computers and operating systems and to transfer files between RT–11 and another operating system.

  — *Debugging programs*, which help you find and correct errors in your programs.

  — *HELP* and *INDEX*, which are designed to help you find information about RT–11: HELP gives on-line KMON command descriptions; INDEX, gives on-line references to topics described in the RT–11 manuals.

  — A *librarian*, which lets you to store and retrieve often-used programming routines.

  — A *linking program*, which converts programs into a format suitable for loading and execution.

  — A *macro assemble*, which converts assembly language source code into object modules.

- **Support for language processors**

  The RT–11 operating system supports programming languages. This support is in the form of the libraries SYSMAC and SYSLIB provided with the operating system. The macro library SYSMAC contains operating system macros and calls used in MACRO–11 programs, and the System Subroutine Library (SYSLIB) allows access to those macros by high-level languages such as FORTRAN.

1.2.2 Language Processors

You use a language processor to translate the programs you create with a text editor into a series of binary codes the computer can understand. A language processor exists for every programming language supported by the system, whether it is a low-level language or a high-level language.

- **Low-Level (Assembler)**

  RT–11 distributes one low-level assembler language, MACRO–11. The assembler translates assembly code (low-level language) into binary code (machine-level language) in the form of an object file. Assemblers are usually one-for-one translators; that is, each instruction in the language usually becomes one instruction to the computer.

  An advantage of programming in assembly language is that it gives you control over such factors as program size and execution speed. A disadvantage is that it is a more intricate language than a higher level one, and low-level languages require much more knowledge of the computer and its devices than a higher level language requires.

- **High-Level (Compilers and Interpreters)**

  High-level languages can be divided into compiled and interpreted languages. Compilers and interpreters are the respective language processors for these languages. These processors usually translate each high-level language instruction into a series of assembly-language or machine-level instructions. This means high-level languages require you to write less code for a computer operation than assembly languages require, which makes programming in a high-level language easier.

  Usually a compiler makes more than one pass through an entire program before translating it. In this way, the compiler checks for errors at many levels and optimizes (eliminates) code when possible. The compiler-translated language is called the object code, while the high-level language text file is called the source code.

  PDP–11 C, FORTRAN IV, FORTRAN–77, and DIBOL are languages that use a compiler. RT–11 support for FORTRAN IV and FORTRAN–77 is described in Chapter 17.

  Like compilers, intrepreters also translate instructions written in a high-level language. However, while compilers translate an entire program before letting you run the program, interpreters translate and run a program at the same time, on an instruction-by-instruction basis. The interpreter does not produce an object file. This is a slower way to run a program than first compiling it and then running it. But one advantage of an interpreter is that you do not have to compile and link files. Another advantage is that you can interactively debug errors in a program as they appear.

BASIC–PLUS is a high-level language that uses an interpreter. Also, optionally, BASIC–PLUS lets you preserve the intermediate translated program and thereby lessen the waiting period before a BASIC program is ready to run. RT–11 support for BASIC–PLUS is described in Chapter 18.

1.2.3 Application Programs

Digital and several other companies sell application programs that run on the RT–11 operating system. Application programs cover many categories, such as marketing, purchasing, inventory management, manufacturing, and engineering. Some general application programs include:

- Word processing
- Spreadsheet analysis
- Communications
- Accounting
- Database management
- Graphics

Other specialized application programs are geared to particular businesses such as law, medicine, dentistry, real estate, and municipal administration.

To help keep you up to date on what application programs from Digital and other companies are available for the PDP–11, Digital publishes the *PDP–11 Software Source Book*. The source book describes more than 2000 application programs, many of which run on the RT–11 operating system. If you want to order the source book, contact your local Digital sales office.

1.3 System Documentation

System documentation can include manuals telling you how to use the hardware and software and also source listings of programs that make up the operating system. Source listings, however, have to be purchased separately and are not included in the software documentation set.

- **Hardware manuals**

  Hardware manuals describe the devices in the computer system. RT–11 hardware documentation includes a processor handbook or technical manual that describes the PDP–11 computer you are using, and an owner's manual or user's guide for each peripheral device in your computer system. These manuals tell you how to operate the devices and give you special programming information that you may need if you intend to write device drivers or special system software involving the devices.

- **Software manuals**

  Software manuals describe the operating system and the language processors.

All RT–11-related software manuals are listed and described in the *Guide to RT–11 Documentation*. Many of these manuals are provided with your system; others can be ordered through your local Digital sales office.

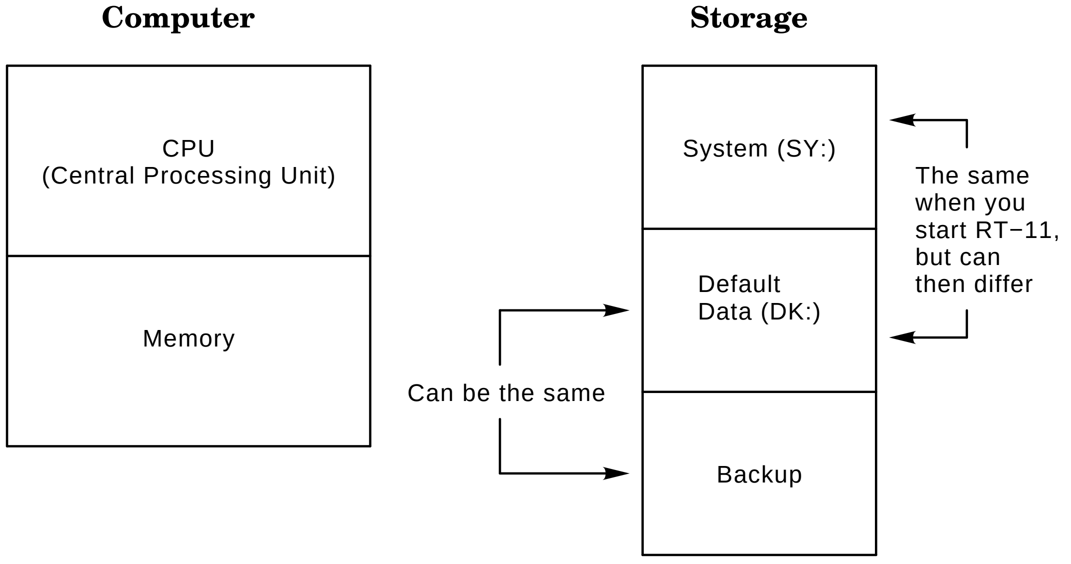
- **Source listings**

  Source listings are listings of the commented assembly language code that makes up the RT–11 operating system. These detailed listings are generally needed only if you intend to modify the system software. They can be ordered in microfiche form through your local Digital sales office, which also sells the source kits for the operating system.

1.4 Chapter Summary

The RT–11 computer system consists of the following components.

Hardware	Software	Documentation
Basic components	RT–11 Operating system	Hardware manuals
Computer	Monitor	
Terminal	Device handlers	Software manuals
1 disk storage devices	Utilities	Source listings
Optional components	Support for language processors	
Extra terminals		
Extra storage devices	Language processors	
Printer(s)		
Other devices	Application programs	



Chapter 2

Getting Started

This chapter assumes that the RT–11 operating system is installed on your computer. If RT–11 is not installed, see the *RT–11 Automatic Installation Guide* or the *RT–11 Installation Guide* and the hardware manuals provided with your computer for instructions on installing RT–11.

This chapter describes the following basic concepts and procedures you need to understand when using the RT–11 operating system:

- Using mass storage volumes
- Knowing your terminal keyboard
- Starting RT–11
- Communicating with the computer
- Setting the date and time
- Using the SETUP command
- Listing volume directories
- Using storage devices
- Using files
- Knowing the general command format
- Responding to command prompts
- Using HELP and INDEX
- Running utilities

2.1 Using Mass Storage Volumes

Mass storage volumes (for example, disks, diskettes, and magtapes) provide a storage area apart from the computer's main memory. That information storage can be user-application programs, data needed by a program, the results of a program run, textual information, and so on. For example, the RT–11 operating system is stored on a mass storage volume called the system volume. When information is needed, as it is when you start the computer, information from the storage volume is transferred into main memory.

2.1.1 Storing Files

You store information on a mass storage volume in the form of files. Each file is a collection of binary codes that represents data or computer instructions. Files may be parts of programs or entire programs, program input data, or text, such as a letter or report. Whatever its content, each file is treated as a unit and occupies a fixed area of the volume.

Every file on a mass storage volume has a unique name composed of a file name and a file type. The file name and file type identify the file and distinguish it from other files on the volume.

You can instruct RT–11 to display on your terminal the names of all files on a volume. The resulting list is called the volume directory listing. By referring to the volume directory, you can find the name, size, and creation date of each file on that volume and delete old files that you no longer need. Whenever you perform an operation that affects the number or size of files on a volume, the volume's directory reflects the change.

2.1.2 Protecting Files

You can protect your files in several ways:

- By backing them up
- By using the protection features of the media or hardware
- By using the protection features of RT–11

2.1.2.1 Backing Up Files

A storage volume may become damaged, lost, stolen, or worn through use. For these reasons, protect your files against accidental erasure or loss. One way to protect a file is to make a copy of it on a second storage volume. That copy, called a **backup file**, ensures you against the loss of your file should the original copy be lost or become unreadable.

Before you continue, make sure you have a backup copy of the files on your *working system* volume, the volume on device SY. If you cannot locate a backup for this volume, create one before you continue. For manual backup instructions, see the *RT–11 Installation Guide*. An experienced user should perform the backup operation.

2.1.2.2 Volume and Hardware File-Protection Features

Some storage volumes provide a mechanism that protects files against accidental erasure. This mechanism is generally a switch on the volume itself or on the device unit in which the volume is placed. You can set it to a write-protect or write-enable condition.

When a volume is write protected, information can be copied from that volume but not written to or deleted from the volume. When a volume is write enabled, information can be both copied from or to the volume and deleted from the volume.

To protect important files against accidental modification or deletion, you can write-protect the volume on which they reside.

2.1.2.3 Operating System File-Protection Features

The RT–11 operating system provides several protection features:

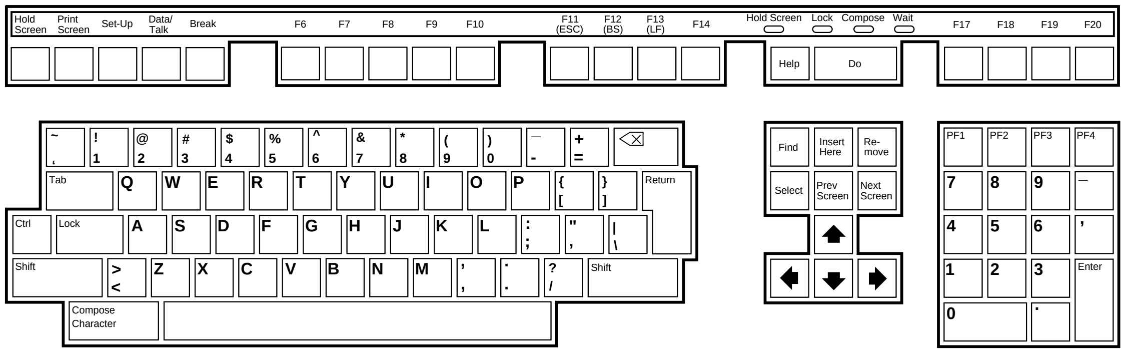
- Distributed RT–11 files cannot be deleted unless you use the UNPROTECT command to remove the protection against deletion from the files.
- You must confirm certain RT–11 commands that might otherwise erase important information. You can override this feature if you do not need it.
- RT–11 prompts you for file information when it is needed by a command.
- You can use the PROTECT command to protect files against deletion or modification by an editor.

2.2 Knowing Your Terminal Keyboard

All terminals have a keyboard and a video screen or paper output device. You enter commands or information at the keyboard, and the screen or paper output device echos the characters typed at the keyboard and displays RT–11 messages and responses.

The names or symbols and the position of some keyboard keys vary among different terminals. Using Figure 2–1 as a guide, notice the layout of your own keyboard.

Figure 2–1: LK201 Keyboard Layout



MLO-003561

The keys for the alphanumeric characters and the SPACE BAR, SHIFT, TAB, and RETURN keys work like keys on a standard typewriter. The SHIFT LOCK key, however, locks only capital letters.

The DELETE, CTRL, and HOLD SCREEN or NO SCROLL keys are specific to computer terminals and have functions you will learn in this chapter and the next.

Table 2–1 summarizes the use of the major keyboard keys you will be using. This manual uses the key symbols given in the table to identify the major keyboard keys.

Other key and keypad (the keys to the right of the keyboard) functions are described in this manual as you use them.

Table 2–1: Keyboard Keys

Key	Function
CAPS LOCK or LOCK	Locks the SHIFT key so you can type letters in uppercase. Press this key again to unlock the SHIFT key.
CTRL	When used with another character key, performs various RT–11 functions. While pressing the CTRL key, you also press another key to execute the function. This is indicated in examples by the symbol $\boxed{\text{CTRL/X}}$ where X represents the other key pressed, like the C key in the symbol $\boxed{\text{CTRL/C}}$.
DELETE or ⌫	The DELETE key has the word DELETE or the ⌫ symbol on it, depending on your terminal type. Pressing this key once erases the last character typed (the character to the left of the cursor). Pressing the key twice erases the last two characters, and so on.
HOLD SCREEN or NO SCROLL	Stops a screen display from scrolling (moving) off the screen. Press the key again to resume scrolling.
RETURN or RET	Advances the cursor to the beginning of the next line. You press this key to execute most RT–11 commands.
SHIFT	Allows you to select between numeric and special characters, between uppercase and lowercase characters, and, in general, between the two characters displayed on those keys having two characters.
SPACE BAR or SP	Identifies the space bar. Press this key once for each space you need. SP is used in examples only if there could be doubt as to where you should add a space.
TAB	Moves the cursor to the beginning of the next tab stop. Tab stops on a computer terminal are usually positioned every eight spaces across the line, beginning at column 1 of the display.
Any other key	Transmits the alphanumeric or special character to the computer.

This manual uses the following conventions for representing keys. When keys are listed in a table column by themselves or when you are asked to press a specific key, that key is enclosed in a box. But, when the key is only referred to in text, it is represented in uppercase characters without the box.

See your terminal hardware manual for additional information on using your terminal and its keyboard.

2.3 Starting RT–11

Starting RT–11 involves turning on the computer and the hardware devices and loading the appropriate software components from the system volume into computer memory.

The start-up procedure varies depending on the computer, how it is set up, and the devices connected to the computer. Follow the directions given in your hardware manuals.

In computer terminology, *bootstrapping* or *booting* an operating system means *starting* the operating system. So, the word *boot* is often used in place of *start* when referring to starting an operating system.

2.3.1 Start-up Procedure

The following start-up procedure varies depending on your PDP–11 computer:

1. Press the power-on switch on your terminal.
2. Press the power-on switch on your computer.
    a. The computer takes a few seconds to run some diagnostic self-tests and displays some diagnostic memory messages on your terminal screen.
    b. The computer either boots the operating system or displays a boot prompt.
        - An example of a boot prompt on a MicroPDP–11 or a PDP–11/84 is the following:

          ```
          Commands are Help, Boot, List, Setup, Map and Test.
          Type a command and press the RETURN key:
          ```

          If you get a boot prompt, follow the directions in your hardware manual for booting the operating system. In response to the preceding example prompt, you would type the following command if you have your operating system on device DU0:

          ```
          BOOT DU0: RET
          ```
        - When the period (.) prompt, called the monitor prompt, appears at the left margin of the terminal screen, RT–11 is ready to accept a command.
3. Turn on any peripheral devices (for example, a printer) that you may need.

2.3.2 What the Computer Does When You Turn It On

What happens when you turn on a computer differs depending on the computer and how you have set it up. The following is a description of a startup procedure in a MicroPDP–11 or a PDP–11/84:

1. If the computer is in dialog mode, it prompts you for the device from which to boot the operating system.

    If the computer is in automatic mode, it searches the devices connected to it for the bootstrap (start-up routine). For example, the computer might look for the bootstrap first on a diskette device, then on a fixed hard-disk device, and finally on a magtape device — if these devices are a part of your computer system. If the computer fails to find a bootable device, it prompts you for one.

The search pattern is preprogrammed into the computer by Digital. However, you can usually reprogram this pattern to your own choosing or have the computer immediately prompt you for the bootable device by SETUP commands explained in the hardware manual.

NOTE

It is wise to remove from your storage devices any bootable (one having a bootstrap routine in the first few blocks) volumes you do not want to boot, unless you know the computer will not look there before it finds the one you want.

2. When the computer finds the bootstrap, the computer copies it from the storage volume into memory.
3. The bootstrap next takes over, loads, and starts the RT–11 operating system monitor.
4. The monitor then displays on your terminal screen a message identifying itself with its version and update number. For example, the following message identifies the XM monitor of Version 5.6 of the RT–11 operating system:

   ```
   RT-11XM V05.06
   ```

5. Next, the monitor looks on the system volume device (SY) for a start-up command file. This file configures RT–11 the way you want it. Start-up command files for the RT–11 monitors are distributed with RT–11. You can edit these files to suit your needs. See Chapter 8 on how to edit a start-up command file.
6. If there is no start-up command file on your system device, a message is displayed on your terminal screen saying that the file cannot be found. However, RT–11 should start normally.
7. When the period (.) prompt, called the monitor prompt, appears at the left margin of the terminal screen, RT–11 is ready to accept a command from the terminal.

2.3.3 Restarting RT–11

You may have to restart the operating system if you want to boot from a different disk or monitor. When the hardware is turned on and RT–11 is running, you can restart RT–11 with the BOOT command. This command runs the bootstrap sequence described in Section 2.3.2, beginning at step 3. This procedure is called a *soft* (software) *boot*, whereas starting the computer when the hardware has been turned off is called a *hard* (hardware) *boot*.

Restart RT–11 by typing the command BOOT SY: and pressing $\boxed{\text{RETURN}}$; for example:

```
.BOOT SY: [RET]
```

If you have a foreground or system job running on RT–11 when you issue the BOOT command, you will get the prompt:

```
Foreground loaded; Are you sure?
```

All jobs (foreground and system) are aborted by rebooting RT–11. Chapters 5 and 7 explain what a job is and how to run foreground/background and system jobs.

The message `Foreground loaded; Are you sure?` means you have a foreground job running and it will be aborted if you continue restarting your computer. If the job is printing a file, stop the print job first; if the job is an editor with an open file, close that file or you will lose the information in the file.

If you want to abort restarting RT–11, type N and press RETURN in response to the preceding prompt. Conclude any jobs you want to conclude and then again restart RT–11 with the BOOT command.

If you want to continue restarting RT–11 after getting the preceding warning prompt, type Y and press RETURN in response to the prompt.

If you do not get a prompt with a message ending `Are you sure?`, RT–11 will restart automatically.

Once you have created your own start-up command file, you can restart RT–11 with the BOOT command to reinitialize the operating system with the features of your start-up command file. (Chapter 8 describes startup command files and how to create them.) When you are more familiar with RT–11, you will learn other reasons for using the BOOT command.

2.4 Communicating with the Computer

You communicate with the computer through commands. These commands can be in any of the following languages:

- Keyboard monitor (KMON)/Digital Command Language (DCL)
- Command String Interpreter language (CSI)
- Concise Command Language (CCL)
- User-defined languages (defined through the UCL and the UCF features)

These languages are different from the traditional computer programming languages, such as FORTRAN and COBOL, in that their purpose is to communicate between the user and the operating system, not to create programs. Accordingly, they are called *command* (not *programming*) languages.

The Digital Command Language (DCL) is a standard interface between a user and a Digital operating system. DCL as implemented on RT–11 is also called the keyboard monitor command language or the KMON command language, since KMON (short for keyboard monitor) is the program that interprets the language. The implementations of DCL on other Digital operating systems, such as RSX and VMS, are somewhat different.

CSI and CCL are are not as easy to use as DCL. In all three languages you can do many of the same things, but if you are a systems programmer, you may want to use CSI or CCL for some additional functionality.

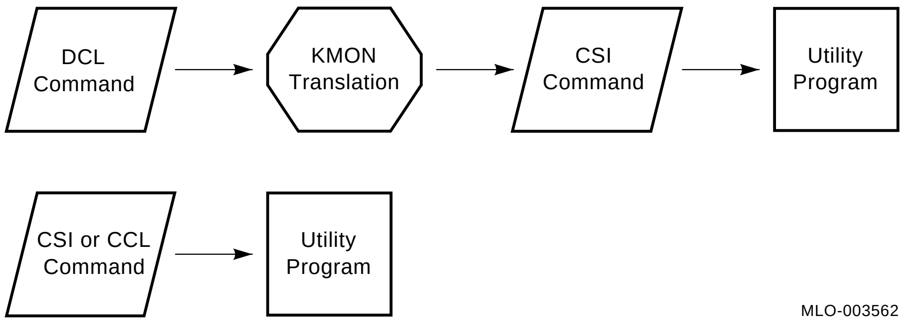
The User Command Linkage (UCL) and the User Command First (UCF) features let you define your own commands. These features are different from the three preceding languages and are explained in Chapter 13.

Whenever you issue a command, KMON inteprets that command, no matter what language it is in.

- If the command is DCL syntax, KMON first checks to see if the command is simple or complex. If the command is a simple one, KMON itself executes the command; if the command is complex (that is, requires action from a utility), KMON determines the correct utility program to execute that command, and then passes the command to the utility to execute.
- If the command is a CSI or CCL syntax, KMON passes that command to the utility you have named in the command; then the utility executes the operation.

Figure 2–2 shows the difference in the sequence of events when you issue a DCL complex command or a CSI or CCL command.

Figure 2–2: Sequence of Events with DCL and CSI or CCL Commands



This chapter explains how to use DCL, the language you should learn first. See the *RT–11 System Utilities Manual* and the *RT–11 Commands Manual* for an explanation of CSI and CCL commands.

2.4.1 DCL Commands

DCL commands are letters and English or English-like words that instruct RT–11 to perform specific tasks. These commands usually approximate the name of the task you want to accomplish. TYPE, PRINT, and RUN are examples of DCL commands.

A prompt character — a period at the left margin of the terminal printer or screen — appears whenever the RT–11 monitor is waiting for you to type a command.

On a terminal screen, a blinking symbol, called the **cursor**, indicates where the character you type will appear. The cursor can be a blinking rectangle or a blinking underline, depending on how you set your terminal. The cursor moves one character to the right each time you type a character.

You can abbreviate a command to its fewest unambiguous letters. For example, you can type PRI for the PRINT command or PRO for the PROTECT command. But you would get an error message if you just typed PR since that could be interpreted as either PRINT or PROTECT.

Three letters of a command are usually all you need, but sometimes less than three letters are enough. Consistently abbreviating commands to three letters helps avoid confusion. A few commands, like FORMAT and FORTRAN, require four letters.

2.4.2 Issuing a Command

After typing a command, you *issue* or execute it by pressing RETURN. This is indicated by RET in the examples in this manual. Pressing RETURN tells the monitor to initiate the command and to perform the operation.

If you type a command correctly and completely, the period (.) prompt is again displayed on the screen when the command has completed its function.

If you make a mistake or omit a required part of a command when you issue the command, RT–11 displays an RT–11 error message or prompts you for more information.

2.4.3 Correcting Typing Mistakes

If you make a typing mistake, you can correct it in several ways:

- By pressing DELETE or CTRL/U.
  - To delete the single character to the left of the cursor, press DELETE once.
  - To delete *all* the characters to the left of the cursor, press CTRL/U; that is, while holding down the CTRL key, press the U key.

  For example:

  1. At the monitor prompt, type in any ten characters.
  2. Delete the last five by pressing DELETE five times.
  3. Press CTRL/U to delete the other five.

- By using the Single-Line (SL) Command Editor

  The RT–11 SL Editor lets you do many editing functions (from moving the cursor to any point in a command line and inserting text within a command already typed, to recalling and repeating commands).

  For now, all you need know is how to delete typing mistakes on the command line. However, after you finish this chapter, you should read and do the exercises in Chapter 10, which describes how to load and use the SL editor.

2.4.4 Aborting or Canceling a Command

If you change your mind about issuing a command, you can either abort or cancel the command.

- To abort a command that is executing, press CTRL/C CTRL/C. That is, while pressing the CTRL key, you also press the C key twice. For example:

  1. Issue the DIRECTORY command.

     ```
     .DIR  RET
     ```

     A listing of the files on your default storage volume will start to be displayed on the terminal screen.

  2. Before the display of file names is finished, press CTRL/C CTRL/C. Each CTRL/C is echoed on your terminal as

     ```
     ^C
     ```

     This stops the display and returns you to the monitor prompt even though the display may not be finished.

- To cancel a command, press CTRL/C.

  You can cancel a command if the command has not yet been executed (you typed in a command, but then changed your mind before you pressed RETURN, or if a command (or program) is waiting for you to enter information at a prompt. For example:

  1. Type the DIRECTORY command and then press CTRL/C once. The CTRL/C is echoed on the terminal screen as ^C beside the canceled command. For example:

     ```
     .DIR^C

     .
     ```

     This leaves the command line on your terminal screen as you had typed it, but it does not execute the command. Instead, the cursor is placed on a new line beside the monitor prompt beneath the canceled command line.

  2. Next, issue the COPY command, but press CTRL/C when you receive a prompt. This action cancels the command at the command prompt.

     ```
     .COPY  RET
     From? ^C

     .
     ```

2.4.5 Responding to Error Messages

If you make a mistake when you issue a command, you will receive an error message, beginning with a question mark, displayed on your terminal's screen. For example:

```
?KMON-F-Invalid option
```

or

```
?PIP-E-Protected file already exists <dev:filnam.typ>
```

These messages are in the format:

?ProgramName–ErrorLevel–MessageText

where:

ProgramName	is the name of the program that found the error. This may be the command interpreter (KMON or CSI) or any one of the utilities to which KMON forwards the command. For example, PIP (in the preceding example) is one of the utilities that the COPY command uses.
ErrorLevel	is a single letter indicating the severity of the error.
MessageText	is the error description.

Table 2–2 lists the error levels, their meanings, and their effects.

Table 2–2: Meaning and Effect of Error Levels

Error Level	Meaning	Effect
I	Information	Command execution continues. The program detected a condition that you should be informed of. The message appears at the terminal or in the listing file. The condition may affect execution at a later time and may require future action.
W	Warning	Command execution continues. The program detected a condition that may cause errors in execution. Corrective action may be necessary. The message appears at the terminal or in the listing file.
E	Error	Command execution may terminate. The program detected an error that can cause other errors during execution. Corrective action is necessary. The message appears at the terminal or in the listing file.
F	Fatal	Execution terminates. The program detected a serious error. You must enter another command to continue processing. The message appears at the terminal or in the listing file.
U	Unconditional Abort	Execution terminates. An extremely serious error occurred that prevents further processing.

All RT–11 error messages are listed in the *RT–11 System Message Manual*. They are listed alphabetically first by the program that displayed them, then by the error code, and finally by the message text. Under each message is an explanation of it and a possible way or ways to correct the situation.

When you get an error message and are not sure what it means, look it up in the *RT–11 System Message Manual* and read the explanation and suggested correction given for that message. For practice, look up the explanations for the two sample error messages given in the previous example.

2.4.6 Command Options

Most commands can take one or more options that qualify how the command acts. You place the options after the command and separate them from the command and from each other by a slash character (/). For example, the DIRECTORY command gives you a directory listing of the files on your default storage volume, in the order they are stored on that volume. However, you also have the option of alphabetizing that listing; and the DIRECTORY/ALPHABETIZE command gives you the same directory listing, but in alphabetical order.

Options can also be abbreviated to the fewest unambiguous letters, which is usually three; for example, the DIRECTORY/ALPHABETIZE command can be abbreviated to DIR/ALP.

Issue the DIRECTORY/ALPHABETIZE command as in the following example, but after a few lines press CTRL/C twice to abort the listing, since it could be long:

```
.DIR/ALP  RET
     .
     .
     .
 ^C ^C
```

As a second example, notice the DIRECTORY/ALPHABETIZE/VOLUMEID command. This command displays the volume ID and owner name of your storage volume in addition to an alphabetical directory listing.

Issue the DIRECTORY/ALPHABETIZE/VOLUMEID command:

```
.DIR/ALP/VOL  RET
```

2.5 Setting the Date and Time

To set the date and time you enter them with the DATE and TIME commands. Once you accurately enter them, the computer keeps them both current as long as the computer is kept on. This helps in record keeping for RT–11 operations and helps you identify when RT–11 operations were performed.

For example, by entering the current date you instruct RT–11 to assign this date to all files you create on that day. The date will also appear in volume directories and listings produced by the various language processors and utility programs.

If your processor has a clock, by specifying the current time of day, you instruct RT–11 to keep track of time based on the time you set. The current time is printed on listings when they are produced, and may also be used to control certain program operations.

For PDP–11 computers, you must *reset* the date and time each time you turn on your computer.

2.5.1 Setting the Date

If you used the automatic installation procedure to install RT–11, you set the date when you installed your files. However, if you have turned off your computer since then, you will need to reset the date.

To set the date, use the DATE command in the following command format:

DATE dd-mmm-yy

where:

`dd-mmm-yy`	represents the day, month, and year
`dd`	is a number from 1 to 31
`mmm`	is the first three characters of the name of the month
`yy`	is the last two digits of the year

Do the following exercise:

1. Set the date to January 10, 1990, by issuing the monitor DATE command with the day, month, and year as follows:

   ```
   .DATE 10-JAN-90 RET
   ```

2. Since the preceding date is not current, enter the correct date by using the same command format. Entering a new date resets the date.

Remember: the date that is set is temporary. You must reenter it whenever you turn on your computer.

2.5.2 Displaying the Date

Once you set the date, display it with the DATE command. For example:

```
.DATE RET
10-Sep-90
```

If RT–11 responds to the DATE command with the message `?KMON-W-No date`, the date has not been set.

2.5.3 Setting the Time

To set the time, use the TIME command in the following command format:

TIME hh:mm:ss

where:

`hh:mm:ss`	represents the time in 24-hour notation
`hh`	is the hour
`mm`	is the minutes
`ss`	is the seconds

RT–11 keeps track of time in hours, minutes, and seconds, based on the initial time that you enter in the command.

Set the time by issuing the monitor TIME command with the hour, minutes, and seconds in 24-hour notation as follows:

```
.TIME 15:01:00 RET
```

If your processor does not have a clock, the monitor prints a message on the terminal; this message informs you that the command is not valid for your computer configuration:

```
?KMON-W-No clock
```

Otherwise, the time is set to 3:01 p.m. If your processor has a clock, enter the correct time by using the preceding command format. Entering the new time resets the clock.

The clock stops when you turn off your computer. If you want the time to be kept current, you must reenter it whenever you turn on your computer.

If your processor has a clock and you do not set the time, the TIME command will return the time elapsed since the last hardware boot.

2.5.4 Displaying the Time

To check the time, issue the TIME command. For example:

```
.TIME RET
15:06:19
```

RT–11 responds by printing the time, based on the information you previously entered.

2.6 Using the SETUP Command

The SETUP command you set various features of your terminal and printer. For brief on-line descriptions of these, check the help files for those two devices by issuing the following commands:

```
.SETUP TERMINAL HELP RET
.SETUP PRINTER HELP RET
```

Not all the features displayed in the help text are available for all terminals or printers. See your hardware manuals for setup features specific to your devices.

The SETUP CLEAR command, which clears your terminal screen, is a command you may find useful. To see how this command works, do the following:

1. Type ten letter Xs in a row and then press CTRL/C.

   ```
   .XXXXXXXXXX  ^C
   ```

2. Next, to clear your screen, issue the SETUP CLEAR command:

   ```
   .SETUP CLEAR RET
   ```

Chapter 8 explains how to incorporate some of the setup features you might want in an RT–11 start-up command file so that you will always have RT–11 set up your way each time you start it.

2.7 Listing Volume Directories

Every storage volume with files on it has a directory, which is a list of all the files stored on the volume. You can display a volume directory on your terminal, using the DIRECTORY command as described in the examples illustrating the DCL command format.

To list the directory of your system volume (the volume from which RT–11 was booted), type:

```
.DIRECTORY SY: RET
```

Since the directory of the system volume may be quite long, after approximately 10 lines have printed on the terminal, press CTRL/O.

This control command is echoed as:

```
^O
```

This command sequence inhibits the remainder of the listing output from printing on the terminal, although the information on the total number of files and blocks is still given. When control returns to monitor command mode, look at the directory listing. For example:

```
 22-Feb-91
SWAP  .SYS    28P  03-Jan-91      RT11XM.SYS   122P  03-Jan-90
VMX   .SYS     3P  16-Jan-91      PIX   .SYS    68P  03-Jan-90
DWX   .SYS     6P  03-Jan-91      DZX   .SYS     4P  03-Jan-90
LSX   .SYS     5P  03-Jan-91      SPX   .SYS    11P  03-Jan-90
XCX   .SYS     4P  03-Jan-91      LDX   .SYS    10P  03-Jan-90
SLX   .SYS    20P  03-Jan-91      DUP   .SAV    51P  03-Jan-90
DIR   .SAV    19P  03-Jan-91      PIP   .SAV    30P  03-Jan-90
KEX   .SAV    74P  03-Jan-91      VTCOM .SAV    24P  03-Jan-90
SPOOL .SAV    21P  03-Jan-91      DUMP  .SAV     9P  03-Jan-90
SETUP .SAV    42P  03-Jan-91      RESORC.SAV    30P  03-Jan-90
UCL   .SAV    16P  03-Jan-91      ^O

 36 Files, 740 Blocks
 650 Free blocks
```

The listing contains the current date, a list of file names and types, file sizes, file protection status, file creation dates, and a directory summary.

- **Current Date**

  At the top of the listing is the current date, as you entered it earlier in the DATE command.

- **List of File Names and Types**

  Following the date, in 2-column format, is a list of the files on the volume. They are listed in the order they are stored on the volume. Read across the columns, moving from left to right, one row at a time.

  First the file name appears, followed by a dot and a file type that is used to identify the file's format and contents. Table 2–3 lists some common RT–11 file types with their meanings.

Table 2–3: Common File Types

File Type	Meaning
.OBJ	Object file
.BAK	Editor backup file
.BAS	BASIC source file
.COM	Command file
.DAT	Data file
.DSK	Logical disk file
.FOR	FORTRAN source file
.JOU	Journal file
.JBK	Journal backup file
.SAV	Executable program file
.SYS	Operating system files and handlers
.TXT	ASCII test file

Note that while monitor and device handler files have a .SYS file type, other RT–11 operating system files do not have that file type.

- **File Sizes**

  After the file type is a number indicating the size of the file. The size is given in blocks, the smallest unit of space into which a mass storage volume can be divided. A block can contain 512 characters.

- **File Protection Status**

  The protected status of a file is indicated by the letter P following the file size in a directory listing. The P means the indicated file cannot be deleted or modified by an editor. Files furnished on the distribution medium have a protected status. However, you can change the protected status of a file with the UNPROTECT command, which is explained in Chapter 3.

- **File Creation Date**

  The date on which a file was created is displayed at the right of each file listing. However, this space is empty for files created on the day you started RT–11 without entering a date.

  If you start RT–11, do not enter a date, and let RT–11 run overnight, the date program tries to increment a zero date. This is one way RT–11 can store a BAD (meaning *invalid*) date in your directory. If, in this situation, you create a file and get a directory listing for that file, the message –BAD– appears in the date column next to your file.

- **Directory Summary**

  At the bottom of the directory listing, you are told how many files are on the volume, their total length, and the number of free blocks available for your use.

You can also obtain an abbreviated directory, which lists only file names and file types in 5-column format. To do this, use the DIRECTORY command with its /BRIEF option. After several lines have listed, interrupt the directory listing by pressing CTRL/C twice to abort the listing and return to monitor command mode. For example:

```
.DIRECTORY/BRIEF RET
 10-Feb-91
SWAP  .SYS    RT11XM.SYS    VMX   .SYS    PIX   .SYS    DWX   .SYS
DZX   .SYS    LSX   .SYS    SPX   .SYS    XCX   .SYS    LDX   .SYS
SLX   .SYS    DUP   .SAV    DIR   .SAV    PIP   .SAV    KEX   .SAV
VTCOM .SAV    SPOOL .SAV    DUMP  .SAV    SETUP .SAV    RESORC.SAV
UCL   .SAV    UCL   .DAT    VBGEXE.SAV    SRCCOM.SAV    TECO  .SAV
TABFIX.TEC    GNC   .TEC    DISK  .TEC    TAGEXP.TEC    TEXT  .TEC
DOC   .TEC    RNO   .TEC    MEM   .TEC    MACRO .TEC    LOCAL .TEC
STRTXM.COM
^C ^C
```

Volume directories can be printed on a printer if one is available in your computer system. Use the /PRINTER option to print a directory listing. If you have a printer, make sure it is turned on, and then issue the DIRECTORY command as shown:

```
.DIRECTORY/PRINTER RET
```

The listing may be quite long. When the printer has finished printing, retrieve the listing.

2.8 Using Storage Devices

RT–11 identifies storage volumes (and consequently the files on those volumes) by the names of the devices in which the volumes reside. These device names (or device specifications) are divided into two categories: *physical names* (names that represent specific devices) and *logical names* (symbolic names that can represent any storage device connected to your computer).

This section describes:

- How devices are referenced by RT–11
- The names of the handlers enabling you to use devices
- The names of the files containing those handlers
- How you can create your own logical names to reference devices
- How you can create your own logical disks as parts of a physical disk

2.8.1 Physical Device Names

A physical device name consists of a 2- or 1-letter mnemonic, a unit number, and a colon; for example, DU1:.

The *physical* name is not the hardware name for the device. Rather, the physical name is a *device specification*; that is, a name RT–11 uses to specify (represent) a specific device.

The format for writing the name is:

DDn: (for devices with 1 to 8 units)

or

Dnn: (for extended-unit devices)

where:

DD or **D**	is a 2- or 1-letter mnemonic identifying the type of device and naming a device handler (a program that lets you interact with that type of device). For example, the device could be a printer, a terminal, or a storage device. A storage device in turn could be a a magtape, diskette, a disk device, or a partition of a large disk. In the example device name, DU1:, DU names the DU handler, which controls MSCP disk devices. This handler lets you interact with a disk drive or disk partition connected to an MSCP controller. The single letter (extended-unit) form is available only for certain device handlers after you have done an optional system generation. When you have more than one device of the same type connected to your computer, the individual devices of the same type are identified as units of the device. If there are 8 or less units of this device attached to your computer, you use the 2-letter mnemonic. If there are more than 8 units of this device attached to your computer and you generated support for the extended-unit handler of that device, you use the 1-letter mnemonic. See the *RT–11 System Release Notes* for information on extended device-unit handlers.

n or **nn**	is an octal number identifying the unit of the specified type of device. This enables RT–11 to distinguish between different devices of the same type. RT–11 assumes unit 0 of a device if no unit number is given. For example, DU: is equivalent to DU0:. The first 8 units of a device are numbered 0 to 7. If you have generated support for the extended device-unit handler (which can handle up to 64 units of a device), then units 8 through 63 (decimal) are numbered 10 through 77 (octal). For example, D10: is the device name for the 9th device unit.
: (colon)	tells RT–11 that the alphanumeric characters preceding the colon represent a device. You must use the colon in a device specification when RT–11 could confuse a device name for a file name. Otherwise, you do not need the colon (in a device specification), though you can use it. This means:

- When a device name is part of a file specification, you must end that name with a colon so as to separate it from the file name that follows. For example, a colon separates the device name DL1 from the file name MYFILE in the file specification DL1:MYFILE.TXT. Section 2.9.1 describes how to specify a file in RT–11.
- When a command can take either a file specification or a device specification as its object, you must use the colon if you intend to specify a device. For example, the command DIRECTORY SY: gives you a directory listing of all the files on the system device SY:; but the command DIRECTORY SY gives you a directory listing of only those files whose name is SY on the default storage device. If you have no files whose name is SY, the command DIRECTORY SY displays the following message:

  ```
  0 Files, 0 Blocks
  ```
- When a command can take only a device specification as its object, then you do not need a colon. For example, the ASSIGN, DEASSIGN, and MOUNT commands work only with devices and so do not require a colon when you specify a device name.

- RT–11 on-line listings of logical device assignments omit the colon from the device name.
- In RT–11 documentation, the device name is often spelled without the colon. However the colon is included in table listings of device names and in command examples using device names.

Some standard RT–11 physical device names and their corresponding devices are listed in Table 2–4. These names represent not only storage devices but also other kinds of devices that might be connected to your operating system. The letters *n* and *nn* (in the case of extended units) indicate the unit number.

Table 2–4: Physical Device Names

Name	Device
DLn:	RL01/02 disk
DMn:	RK06/07 disk
DUn: or Dnn:	MSCP disks and diskettes
DXn:	RX01 diskette
DYn:	RX02 diskette
LD: or Lnn:	Logical disk
LP:	Parallel-interface printer
LS:	Serial-interface printer
MMn:	Magtape
MSn:	Magtape
MTn:	Magtape
MUn:	TMSCP Magtape
RKn:	RK05/06 disk
SPn:	Spooler
TT:	Console terminal
VM:	Virtual memory device
XL:	Serial communication

2.8.2 Device Handler Names

Device handlers let you use devices, whether they are storage devices, terminals, printers, or whatever other devices you have connected to your computer. RT–11 refers to a handler by the same 2-letter mnemonic as the device names but without the unit numbers or colons. In fact, physical device names are the handler names minus the unit numbers and colon, with one exception. A handler with extended device-unit support keeps its 2-letter name, while the physical device names for its units over 8 have 1 letter.

Whenever you specify a physical device name, you are specifying a device handler.

To see what device handlers your operating system currently knows (that is, are installed), issue the SHOW command. For example:

```
.SHOW RET
```

A listing like the following should appear:

```
.SHOW RET

TT (Resident)
DL (Resident)
DU0 = DK , SY
MQ (Resident)
VM
LD
SP
XL (loaded=VTCOM)
LS (Loaded)
LS0 = (SPOOL)
NL
19 free slots

.
```

TT, DL, MQ, and so on are names of device handlers in the preceding display. The line `DU0 = DK , SY` illustrates a logical device assignment, which is be explained in Section 2.8.4.

Some RT–11 commands (for example, INSTALL and LOAD) affect device handlers. These commands and the installing and loading of handlers are explained in the *RT–11 Commands Manual*.

2.8.3 Device Handler Files

The device handlers are contained in files whose names (or file-name specifications) begin with the same 2-letter mnemonic that identifies the device handler and end with the .SYS file type.

Issue the DIRECTORY command as follows:

```
.DIRECTORY DU.SYS RET
```

If you have the DU handler file for the unmapped monitors, RT–11 lists on your terminal something like this in response to the previous command:

```
 10-Feb-91
DU    .SYS    10P 25-Jan-91
 1 Files, 10 Blocks
 13313 Free blocks
```

If you do not have this file, RT–11 gives you a message like this:

```
10-Feb-91

0 Files, 0 Blocks
13313 Free blocks
```

Device handler file names for the mapped monitors have an X added to the end of the handler mnemonic. This X, however, does not appear in the handler name — only in the handler file name. So, if you are using a mapped monitor, issue the DIRECTORY command as follows:

```
.DIRECTORY DUX.SYS RET
```

If you have the DU handler file for a mapped monitor, RT–11 lists on your terminal something like this in response to the previous command:

```
 10-Feb-91
DUX   .SYS    10P 25-Jan-91
 1 Files, 10 Blocks
 13313 Free blocks
```

So, a physical device name, the name of the handler that services that device, and the name of the file containing that handler are all similar but slightly different. These differences are summarized in Table 2–5.

Table 2–5: The Differences Between Device, Handler, and File Names

Name	Representation
DU0:	An RT–11 unit of a storage device
DU	A handler for that device
DU.SYS	A file containing the DU handler for the unmapped monitors
DUX.SYS	A file containing the DU handler for the mapped monitors

2.8.4 Logical Device Names

In addition to physical device names, RT–11 uses two special logical (symbolic) device names, as listed in Table 2–6.

Table 2–6: Logical Device Names

Name	Representation
SY:	The operating system device; that is, the device from which the monitor was bootstrapped
DK:	The default storage device (initially the same as SY:)

Note that a logical device name ends with a colon just as a physical device name ends with a colon. However, you must specify the colon only when RT–11 needs it to know you are specifying a device rather than a file (see Section 2.8.1).

This section describes a simple way to reference your system device and your default storage device through the preceding two logical names.

You can use the logical device name SY for your system device instead of its physical device name, and you can use the logical device name DK for your default storage device instead of its physical device name. These two names, however, represent the same device *whenever* you start your operating system. That is, the system device SY and the default storage device DK are the same when you start your operating system, even if in a past operating system session you changed them to represent different devices. As an illustration of this, issue the SHOW command again:

```
.SHOW RET
```

In the listing shown on your terminal, you should see a line similar to the following:

```
DU0 = DK , SY
```

This means that the physical device name DU0 is represented by both the logical device names SY and DK. Notice also that this listing showing the logical name assignments is an example of when RT–11 displays device names without the colon ending.

Unlike DU or SY, the name DK is not tied to any one device. Rather, you can assign this name to represent any storage device connected to your computer. Section 2.8.5 explains how to do this.

Whatever device DK represents is your default storage device. This means if you do not specify a device name after a command, the command will work on whatever device is named DK. For example, issue the following DIRECTORY command by itself without a device name to get a listing of the files on DK. However, press CTRL/C twice to abort the command after the first few lines of the listing, since the listing could be long.

```
.DIR RET
   .
   .
   .
^C ^C
```

Although the name DK is not tied to one device, the name SY is tied to one device:

- The device from which RT–11 is booted. You get an error message if you attempt to change the assignment of that name.
- The default device used by KMON commands to run their programs. For example, the DIRECTORY command uses the DIR utility program. If the file DIR.SAV, the file containing the DIR utility, is not on SY, the DIRECTORY command cannot function.

  To see the file name of the DIRECTORY utility program, issue the DIRECTORY command as follows:

  ```
  .DIR SY:DIR RET
  ```

  You should get a file listing similar to the following:

  ```
   10-Feb-91
  DIR   .SAV    19P 25-Jan-91
   1 Files, 19 Blocks
   13313 Free blocks
  ```

2.8.5 Assigning Logical Names

You can assign a specific storage device on RT–11 to be your default storage device (DK). To do so, use the following command format:

ASSIGN physical-device-name logical-device-name

For example:

```
.ASSIGN DU1: DK: RET
```

Note in the command format that the `physical-device-name` precedes the `logical-device-name`. When you assign a logical device name to a physical device name, you first specify the physical device in the command line.

If you are not sure what devices are connected to your operating system, issue the SHOW command to find a device name for a storage device different from your system device:

```
.SHOW RET
```

Using the preceding ASSIGN command example as a guide, assign DK to an available storage device different from your system device. Then issue the SHOW command again to check that the assignment has been made. You should get a display similar to the following:

```
.SHOW RET
```

```
TT (Resident)
DU (Resident)
   DU0 = SY
   DU1 = DK
MQ (Resident)
VM
LD
SP
XL (loaded=VTCOM)
LS (Loaded)
   LS0 = (SPOOL)
NL
19 free slots

.
```

In addition to using RT–11's two default logical names, you may also want to assign your own logical device names to physical devices. For example, when programmers cannot predict which physical device will be available for use, they use logical device names in their programs. Then, before running the programs, they assign the logical device names they created to the physical device names on their operating system. A logical device name can be from 1 to 3 alphanumeric characters followed by a colon.

Use the ASSIGN command again to assign a 3-character name of your own choosing to the storage device you also assigned to DK. For example:

```
.ASSIGN DU1: VOL: RET
```

You can assign more than one logical name to a device. Issue the SHOW command to check your assignment:

```
.SHOW RET
```

Logical device names are temporary; that is, you must reassign them each time you boot RT–11. However, you do not have to reboot RT–11 to remove a logical name. To delete a logical name assignment, use the DEASSIGN command in the following format:

DEASSIGN logical-device-name

Now deassign the logical name assignment you just made as in the following example:

```
.DEASSIGN VOL: RET
```

To check that the logical name assignment is deassigned, issue the SHOW command:

```
.SHOW RET
```

To deassign all your own logical name assignments in one step and to reassign DK to the same device to which it was originally assigned by RT–11, issue the DEASSIGN command by itself. For example:

```
.DEASSIGN RET
```

Finally, issue the SHOW command to check that the logical name assignments are back in their original state:

```
.SHOW RET
```

2.8.6 Using Logical Disks

You can subset your physical disks into smaller logical disks. (Do not confuse logical device names with logical disks; although similar in name, they are completely different.) Logical disks, though actually just sections of a physical disk, are treated by RT–11 as if they were totally separate physical disks. This section introduces you to logical disks. See Chapter 9 for a complete explanation of why and how to use them.

The logical disk handler is named LD. To understand what that handler is and how to use it, do the following:

1. Issue the SHOW command to see if the LD handler is installed:

   ```
   .SHOW  RET
   ```

   If the LD handler is installed, it is listed in the display of known handlers. Go to step 2.

   If the handler is not listed in the display, then you must install it with the following command:

   ```
   .INSTALL LD  RET
   ```

   If you get no error messages, go to step 2. If you get either of the following error messages, the LD handler file is not on your system volume:

   ```
   ?KMON-F-File not found SY:LD.SYS
   ```

   or

   ```
   ?KMON-F-File not found SY:LDX.SYS
   ```

   If you get one of the preceding error messages, copy the file indicated in the error message (LD.SYS or LDX.SYS) from your distribution volume to your system volume. Use the following command format:

   COPY device-name:LD.SYS SY:

   or

   COPY device-name:LDX.SYS SY:

   Then, install the LD handler with the INSTALL command:

   ```
   .INSTALL LD  RET
   ```

2. Set aside a section of your default storage disk as a logical disk file, named INTRO.DSK, by issuing the following CREATE command:

   ```
   .CREATE INTRO.DSK/ALLO:400  RET
   ```

   The /ALLO:400 is an abbreviation for /ALLOCATE:400, which means you want the file INTRO.DSK to have 400 blocks allocated to it.

   Once you have created the file, INTRO.DSK can be treated by the logical disk handler as if it were a separate disk volume.

3. Verify the creation of INTRO.DSK on your default data device by issuing the following command:

   ```
   .DIR INTRO.DSK RET
   ```

4. Associate your logical disk file INTRO.DSK with a logical disk unit, by issuing the following MOUNT command:

   ```
   .MOUNT LD0: DK:INTRO.DSK RET
   ```

   This is similar to placing a physical volume into a physical disk drive. And the logical disk unit LD0 is treated by the operating system as a physical device name. In the preceding command, you can picture yourself as having *placed* your logical disk *into* LD0 (logical disk, unit 0). You can use 8 logical disks at a time since there are 0 through 7 logical disk units for the LD handler.

   If you need to use more than 8 logical disks simultaneously, you can do a system generation and select extended device-unit support for the LD handler. This extends the available logical disk units to 32. See Chapter 9 for a description of the LD handler.

5. Create a file directory structure on your logical disk by issuing the following INITIALIZE command:

   ```
   .INIT LD0: RET
   ```

   In response to the preceding command, you should get a prompt like the following:

   ```
   LD0:/Initialize; Are you sure?
   ```

   Type Y in response to this prompt.

   A new (unused) volume, even a logical disk volume, must be initialized before it is used. You can, however, also initialize a used volume. In that case, the INITIALIZE command clears the directory and effectively deletes the files on the specified volume. To caution you against initializing a volume containing files you may want, RT–11 gives you the prompt, `Are you sure?`, when you issue the INITIALIZE command.

6. Examine the directory of your new logical disk by issuing the following DIRECTORY command:

   ```
   .DIR LD0: RET
   ```

   You should get a directory listing something like the following:

   ```
   25-Jan-91

   0 Files, 0 Blocks
   392 Free blocks
   ```

   Note the 392 free blocks. With the CREATE/ALLO:400 command, you set aside 400 blocks for this logical disk file. However, when you used the INITIALIZE command, 8 blocks of the 400 were set aside for a directory and volume header information.

Once you have completed the preceding, you have a logical disk that is ready for use. For the purposes of this introduction, we will use that logical disk as your default storage device. To do so:

1. First, make sure you understand what physical devices DK and SY represent. Review Section 2.8.4 if necessary.
2. Next, with the following command, assign to DK the logical disk unit (LD0) on which you have mounted your logical disk file:

   ```
   .ASSIGN LD0: DK: RET
   ```
3. To check this logical disk assignment, issue the SHOW command:

   ```
   .SHOW RET
   ```

   One line in the display should read:

   ```
   LD0 = DK
   ```
4. Finally, issue the following four commands. However, after the first few lines of the system volume listing, press CTRL/C twice to abort the display:

   ```
   .DIR SY: RET
      .
      .
      .
   ^C ^C
   .DIR DK: RET
      .
      .
      .
   .DIR LD0: RET
      .
      .
      .
   .DIR RET
      .
      .
      .
   ```

Note that your system volume and your default storage volume (the volume RT–11 assumes when you do not specify a volume) are now different. The system volume remains as it was, but the default storage volume is now your logical disk volume. Since you have not yet created any files for your logical disk volume, that volume's directory listing should indicate that it has 0 files.

For the rest of this chapter and for the other chapters in Part I of this book, keep the volume INTRO.DSK (LD0) as your default storage volume DK. Also, reassign LD0 as DK each time you boot RT–11, since RT–11 automatically assigns SY as DK each time it is booted. This way, the files you create while doing the exercises in Part I will remain together in one place and you will have practice distinguishing between SY and DK.

2.9 Using Files

Most DCL commands work on files. You identify a file by its file specification, abbreviated as *filespec*.

2.9.1 Specifying Files

A complete filespec consists of a device name, the file name, and the file type in the following format:

devicename:filename.filetype

where:

: and **.**	The colon and period are called *delimiters*. The colon, indicating the *limit* or end of the device name, separates the device name from the file name. The period, indicating the end of the file name, separates the file name from the file type. For example, the following filespec refers to the file called MYFILE.TXT stored on the volume in device DU1:. `DU1:MYFILE.TXT`
devicename	is the name of the device containing the volume holding the file. This part of the filespec specifies on which volume information can be found or to which volume information should be written, and can be the physical or logical device name. Each device supported by RT–11 is identified by a device name, as explained in Section 2.8.1. A logical device name consists of any three alphanumeric characters followed by a colon. If you do not include a device name in your file specification, RT–11 assumes DK, the default storage device. It is a convention in RT–11 documentation to indicate a generic device name with the symbol *ddn*.
filename	is one to six alphanumeric characters assigned by the person or program that created the file.
filetype	is zero to three alphanumeric characters that indicate the format or contents of the information in the file and assigned by the person (or system program) that created the file. Note, however, that to create a file with a blank file type, you must specify a period (.) after the file name when you create the file.

If you create a file without specifying a file type or the period ending the file name, the name will include whatever file type is the default for the file creation operation you perform. For information on creating files with the KED editor, see Chapter 4.

Categorize your files according to type, using the file names to distinguish between individual files. For example, you might use the file type .TXT to identify the following text files:

```
MYFILE.TXT
FORM.TXT
MEMO1.TXT
```

Table 2–3 lists some common file types.

2.9.2 Abbreviating File Names and Types

When you specify files, you can abbreviate their names and types in two ways: through wildcards and through factoring.

2.9.2.1 Using Wildcards

When you want a command to act on a number of files with similar file specifications, use wildcards in the filespec to abbreviate the file names. The two symbols used as wildcards in RT–11 are the asterisk (*) and the percent sign (%).

The asterisk (*) can replace any character or string of characters in a file name or type (up to 6 for a file name, up to 3 for a file type). For example, if you have a series of files with the same file type, like MEMO.TXT, EXAMP.TXT, and INTRO.TXT, you could use the notation *.TXT to represent all these files. The following exercises illustrate some of the ways you can use the asterisk wildcard:

1. Find out how many files on your system volume have the file type .SAV, by issuing the command:

   ```
   .DIR SY:*.SAV  RET
   ```

2. Find out how many files on your system volume have a file name beginning with RT11, by issuing the command:

   ```
   .DIR SY:RT11*.*  RET
   ```

3. Find out which files on your system volume have a name beginning with the letter S and ending with the letter P, by issuing the command:

   ```
   .DIR SY:S*P.*  RET
   ```

If you omit the file type or the file name and file type in a file specification, RT–11 assumes a wildcard for that part or parts of the specification. For example, the following two commands are equivalent:

```
.DIR SY:
.DIR SY:*.*
```

So too, the next two commands are equivalent:

```
.DIR DU1:PROG1
.DIR DU1:PROG1.*
```

However, these two commands are not equivalent:

```
.DIR DU1:PROG1.
.DIR DU1:PROG1.*
```

PROG1. is a file having no characters in its file type, and the command DIR DU1:PROG1. means:

```
Display a directory listing of any file on device DU1: having the name PROG1.
```

While the command DIR DU1:PROG1.* means:

```
Display a directory listing of all files on device DU1: having the file name
PROG1
```

To avoid making mistakes as a new user of RT–11, it is better to use wildcards for each part of a file specification than to omit the last part or parts of the specification.

The percent sign (%) can replace any single character in a file name or type. For example, if you had two files named MEMO1.TXT and MEMO2.TXT, you could use the notation MEMO%.TXT to represent both files. Find out how many files on your system volume begin with the letter D and have only two letters, by issuing the command:

```
.DIR SY:D%.*  RET
```

You can use as many % wildcards in a filespec as required, each wildcard specifying one letter. For example, one or the other of the following two commands displays your file handler files. If you have a mapped monitor running your operating system, the first command displays your handler files; otherwise, the second command displays those files:

```
.DIR SY:%%X.SYS
.DIR SY:%%.SYS
```

You cannot use wildcards to specify devices.

2.9.2.2 Factoring File Specifications

Another way to specify several files at once on the same device is to use factoring. Factoring is valid for any command in the KMON command language.

In factoring, you enclose in parentheses the part of a multiple file specification that differs. This part can be multiple file names, multiple sections of a file name, multiple file types, or multiple sections of a file type. For example, the following two lines are equivalent:

```
MEMO(1,2,30).TXT
```

```
MEMO1.TXT,MEMO2.TXT,MEMO30.TXT
```

Find out what files (if any) on your system volume have names beginning with the two letters KE and ending with a D or an X, by issuing the command:

```
.DIR SY:KE(D,X).* RET
```

Factoring is useful when you are performing an operation on a group of files stored on a particular storage device. Take the following command as an example:

```
.COPY DU0:(FILE1.RNO,FILE2.RNO,FILE3.RNO) DU1:*.*
```

In this case, using factoring saves you from having to type in the device name for each file specification, if all the files are on DU0.

However, if DU0 is not your default storage device, and you issue the following command, you will get an error message:

```
.COPY DU0:FILE1.RNO,FILE2.RNO,FILE3.RNO DU1:*.*
```

Because RT–11 assumes FILE2.RNO and FILE3.RNO are on your default storage device, it will not be able to find the files. Notice the difference between these last two examples. This is a case where factoring the file specifications helps you correctly abbreviate them.

Note that the example command could also be expressed this way:

```
.COPY DU0:FILE(1,2,3).RNO DU1:
```

You can even factor a unit number in a device specification. For example, the following two commands are equivalent. They both tell RT–11 to copy the file MYPROG.FOR from device DU0 to device DU1:

```
.COPY DU(0 1):MYPROG.FOR
```

```
.COPY DU0:MYPROG.FOR DU1:MYPROG.FOR
```

This example illustrates how to factor multiple device specifications. The other examples illustrate how to factor multiple file specifications.

2.10 Knowing the General Command Format

The correct way to type a DCL command with its options and parameters (what the command acts on — usually filespecs) is called its format or syntax. This differs slightly for each command, depending on the nature of the command and its options, if it has any.

A command will not function unless it is issued with its correct syntax, including correct spelling, spacing, and punctuation. In the following general format, square brackets indicate optional qualifiers and characters, and the ellipsis symbol (. . .) indicates repetition; that is, you can repeat the item preceding the ellipsis:

COMMAND[/option . . .] INPUT[/option . . .] OUTPUT[/option . . .]

where:

COMMAND[/option]

First you indicate, by command, which system operation you want performed. Command options change the way the command normally operates.

INPUT[/option]

You next indicate input information that is to be used during the operation. This is usually one or more file specifications, though some commands affect only the in-memory image of the monitor rather than files (for example, the ASSIGN and DEASSIGN commands).

When you do not specify input, the command may take a *default* input; that is, the command assumes input. For example, the default input of the DIRECTORY command is the directory of the current storage volume (DK).

The input is separated from the command by one or more spaces so as to differentiate it from the command; for example, MEMO.TXT, located on DK, is the input filespec in the following command:

```
.PRINT MEMO.TXT
```

Some commands can take more than one input filespec; for example, the PRINT command can take more than one. To specify more than one filespec as input, separate them from one another by commas (,). For example:

```
.PRINT DU0:FILE1.TXT,DU1:FILE2.TXT,DU1:FILE3.TXT
```

If you do not specify a device name with a filespec, that file must be on DK. For example, in the following example all files must reside on DK:

```
.PRINT FILE1.TXT,FILE2.TXT,FILE3.TXT
```

Three limitations to issuing multiple filespecs on a command line are:

- You can include no more than six file specifications at a time (you can use wildcards in a filespec).
- You cannot add spaces after the commas (between the filespecs).
- The command cannot exceed 80 characters after factor expansion.

An input filespec can also be qualified by options. As with command options, you place the option after the information it qualifies, and you separate the option from the filespec and from other options by a slash character. For example, the command

```
.DIRECTORY INDEX.SAV/BEGIN
```

gives a directory listing of the default storage device beginning with the file INDEX.SAV. Issue the command:

```
.DIR INDEX.SAV/BEGIN  RET
```

OUTPUT[/option]

Finally, some commands require output information to be created as a result of the operation. For example, the COPY command requires input (the files you want to copy from) and output (the files you want to copy to). You separate input from output by leaving a space between them on the command line. You usually have only one output filespec. In the following example, the input and the output are on DK.

```
COPY MEMO.TXT MEMO.OLD
```

Output file options are available to let you alter default output operations.

Some commands do not use OUTPUT.

2.11 Responding to Command Prompts

If you issue a command without specifying required parameters (input and output necessary for the command to function) and the command does not take default input, the monitor prompts you for the needed information. For example, if you issue the following command:

```
.COPY  RET
```

The monitor prompts with:

```
From?
```

At the `From?` prompt, type in the specification of the files you want copied and press RETURN. The monitor next prompts you with:

```
To  ?
```

Respond to this prompt with the specification of the files to which you want the copied file transferred.

2.12 Monitor Command Descriptions

If you want information about a monitor command, you can either look it up in one of the following manuals or use the on-line HELP utility:

RT–11 Quick Reference Manual	Lists monitor commands alphabetically and gives a summary explanation of each command.
RT–11 Commands Manual	Lists monitor commands alphabetically and gives a complete description of each command with examples.

2.13 Using the On-Line HELP Utility

The HELP utility, distributed with the RT–11 operating system, lists the keyboard monitor commands and describes how to use them. This utility is activated by the HELP command. However, the file (HELP.SAV) must be on SY for the HELP command to work. You can determine whether this file is present on your system disk by issuing the command DIRECTORY SY:HELP. For example:

```
.DIR SY:HELP  RET
```

The exercises in this section assume the file HELP.SAV is on SY.

To familiarize yourself with using the HELP command, do the following:

- Display at your terminal a list of all the keyboard monitor commands with a brief description of their functions, by issuing the command HELP *. For example:

  ```
  .HELP *  RET
  ```

- Get a detailed description of the use of the HELP command itself, by issuing only the HELP command. For example:

  ```
  .HELP  RET
  ```

  When you do this, notice that the description of the HELP command begins with the name of the command and its function. The rest of the description is divided into four parts (called *subtopics*). Every complete HELP description of a command is divided into these same four parts, which are listed in Table 2–7.

Table 2–7: The Four Parts of a Help Descripton

Part	Description
Syntax	Shows how to type a command
Semantics	Briefly explains the meaning of the command elements used in the preceding *syntax* section
Options	Lists and briefly describes all of a command's options
Examples	Lists a few examples illustrating how to use the command

The HELP display should look something like this:

```
HELP           Lists helpful information

  SYNTAX
        HELP[/options] ❶ [ topic ❷ [ subtopic ❸ [:items... ❹ ]...]]
     or HELP *

  SEMANTICS
        HELP * lists the items for which help is available.
        HELP lists the HELP text (of which this is a part).
        HELP topic  lists information on the specific topic only.
        HELP topic subtopic  lists information on the specific subtopic
          only (for example, HELP HELP SEMANTICS lists the paragraph of
          which this text is a part).
        HELP topic subtopic:item  lists only the text associated with
          the specific item.
        HELP topic/item ❺ lists the text associated with the specific item
          under the subtopic OPTIONS.
        Valid topics are the keyboard monitor commands.
        Subtopics are "SYNTAX", "SEMANTICS", "OPTIONS", and "EXAMPLES".
        Items are specific command options.

  OPTIONS
   PRINTER
        Prints the HELP text on the line printer
   TERMINAL     (default)
        Types the HELP text on the terminal

   EXAMPLES
        HELP COPY              !Lists information about COPY command
        HELP/PRINTER EXECUTE   !Prints information about EXECUTE
                               !command
        HELP PRINT OPTION:COPIES!Describes the COPIES option for PRINT
        HELP COPY/BOOT/DEVICE  !Describes the listed options for COPY
```

The following list of descriptions and operations explains the items in the command syntax section of the preceding example:

❶ **/Options** specify HELP command options; there are only two: /PRINTER and /TERMINAL. The /PRINTER option sends the help information to a printer if one is available. The /TERMINAL option (the default mode) sends the information to the terminal.

As when issuing a monitor command, when you use the HELP utility, you can also abbreviate a command or option to the least number of ambiguous letters, usually three letters.

Print the HELP listing given in the previous example by issuing the command:

```
.HELP/PRI RET
```

❷ **Topic** specifies a monitor command about which you want information, for example: DIRECTORY. Get a complete help description of the DIRECTORY command by issuing the command:

```
.HELP DIR RET
```

❸ **Subtopic** specifies any one of the four sections of a complete HELP listing. Display the EXAMPLES section of the HELP DIRECTORY listing by issuing the command:

```
.HELP DIR EXA RET
```

❹ **Item** specifies one of the members in the subtopic group. For example, you might want a brief explanation of one option to a command. Display an explanation of the /ALPHABETIZE option for the DIRECTORY command by issuing the command:

```
.HELP DIR OPT:ALP RET
```

Note that you can specify more than one item in the command line if you separate the items with a colon (:). Display explanations for both the ALPHABETIZE and NEWFILES options of the DIRECTORY command by issuing the command:

```
.HELP DIR OPT:ALP:NEW RET
```

❺ For information about an option or options, you do not have to use the OPTIONS subheading. Instead, all you need to type is the command with its options. Issue the command:

```
.HELP DIR/ALP/NEW RET
```

2.14 Running Utility Programs

RT–11 has many programs, called *utilities*. Two utilities already mentioned in this chapter are the DIR utility and the HELP utility. Other utilities, such as PIP (Peripheral Interchange Program), SRCCOM (Source Comparison program), and KEX (Keypad Editor program for mapped monitors) are mentioned in Chapters 3 and 4. Most utility programs are described in the *RT–11 System Utilities Manual*.

You can run RT–11 utilities in the following ways:

- By issuing a KMON command, if the utility you want to run (or a function of a utility you want to run) has a KMON command that interacts with it. Many KMON commands run utilities, as described in Section 2.4. Two examples in this chapter of utilities that KMON interacts with are the DIR utility (see Section 2.8.4) and the HELP utility (see Section 2.13).
- By issuing the KMON R command with the name of the utility (or the name of the file containing the utility). The utility must be on SY, unless you include a file-structured device in the file specification. For example:

  ```
  .R DIR RET
  *
  ```

  or

  ```
  .R DK:MYPROG.SAV RET
  *
  ```

- By issuing the KMON RUN command with the name of the utility (or the name of the file containing the utility). The utility must be on DK, unless you include a file-structured device in the file specification. For example:

  ```
  .RUN DIR RET
  *
  ```

  or

  ```
  .RUN MYPROG.SAV RET
  *
  ```

  The asterisk (*) is the utility prompt. Once you receive this prompt, you issue CSI (Command String Interpreter) commands to control the utility.

- By issuing the KMON V command with the name of the utility (or the name of the file containing the utility). The V command calls the virtual program loader, VBGEXE.SAV, which must reside on SY. VBGEXE loads and runs the utility, which must be on SY, unless you include a file-structured device in the file specification. For example:

  ```
  .V INDEXX RET
  *
  ```

  or

  ```
  .V DK:MYPROG.SAV RET
  *
  ```

- By issuing the KMON VRUN command with the name of the utility (or the name of the file containing the utility). The VRUN command calls the virtual program loader, VBGEXE.SAV, which must reside on SY. VBGEXE then loads and runs the utility, which must be on DK, unless you include a file-structured device in the file specification. For example:

  ```
  .VRUN INDEXX RET
  *
  ```

  or

  ```
  .VRUN MYPROG.SAV RET
  *
  ```

  The asterisk (*) is the utility prompt. Once you receive this prompt, you issue CSI (Command String Interpreter) commands to control the utility.

- By issuing the KMON FRUN or SRUN command with the name of some utility files: For example:

  ```
  .FRUN SY:KEX.SAV RET
  *
  ```

  or

  ```
  .SRUN SY:KEX.SAV RET
  *
  ```

  See Chapter 5 and 7 for an explanation of how to use the FRUN and SRUN commands.

- By issuing the name of the utility as a CCL (Concise Command Language) command with input and output information (in CCL format) on one command line. For example:

```
.KEX filename RET

.SRCCOM file1,file2 RET

.PIP input-file output-file RET

.PIP output-file=input-file RET
```

Table 2–8 lists the ways of running a utility with the utility location RT–11 assumes for each way. The R, RUN, FRUN, and SRUN commands also let you specify a file-structured device with the utility name. If the utility is not in the assumed or specified location, RT–11 cannot find the utility and the command fails.

Table 2–8: Relationship Between Running a Utility and Its Location

Way of Running Utility	Assumed Location of Utility
KMON (DCL) command	SY:
R utility_name	SY:
RUN utility_name	DK:
V utility_name	SY:
VRUN utility_name	DK:
FRUN utility_name	SY:
SRUN utility_name	DK:
utility_name	SY:

For further information on running utilities, see the *RT–11 System Utilities Manual* and the on-line index entry for the specific utility.

2.15 Using the On-Line Index

You can use the on-line index, INDEX, to find the most current information about topics located in the RT–11 documentation set. INDEX immediately displays almost all the information contained in the *RT–11 Master Index* on your terminal screen.

The on-line package consists of the utility files INDEX.SAV or INDEXX.SAV, and the index data files INDEXA.IMG, INDEXB.IMG, and INDEX.IDX. Use the virtual version of the index utility, INDEXX.SAV, if you are using a mapped monitor. Otherwise, use INDEX.SAV. The data files (type .IMG and .IDX) are used with all monitors.

You access the on-line index as you would any other utility, as previously described in Section 2.14. The index data files can reside on the system (SY) or default data (DK) volume.

The rest of this section assumes you have INDEX.SAV or INDEXX.SAV on SY.

INDEX is not a DCL command; you must run the INDEX utility. For example, under a mapped monitor:

```
.INDEXX RET
```

INDEX then displays an introductory frame on your screen. That frame lists the mnemonics for the manuals in the RT–11 documentation set that are contained in the index. The spines (legends) on the manual binders contain those mnemonics for easy reference. These manuals are:

INS	*RT–11 Installation Guide*
INT	*Introduction to RT–11*
MRM	*RT–11 Quick Reference Manual*
PRM	*RT–11 Programmer's Reference Manual*
RLN	*RT–11 System Release Notes*
SSM	*RT–11 Software Support Manual*
SUG	*RT–11 Commands Manual*
SUM	*RT–11 System Utilities Manual*
SYG	*RT–11 System Generation Guide*
V5N	On-Line Release Notes (V5NOTE.TXT)

Manuals not listed in the introductory frame are not included in the index.

Press RETURN to get into the index. Then, in response to the search prompt, type a search string and press RETURN. For example, look for references to the word MONITOR by typing the letters MONI:

```
Search string: MONI RET
```

INDEX displays any entries matching the string. Continue to press RETURN to scroll through the index or type another search string and press RETURN.

Press CTRL/C to exit from INDEX.

As you proceed through this manual, use INDEX to find more information on a given topic.

2.16 Chapter Summary

This section summarizes how to specify a file and format a DCL command. The section also lists some useful keys, key sequences, and helpful DCL commands.

Complete information on DCL commands is in the *RT–11 Commands Manual* and a summary of DCL commands is in the *RT–11 Quick Reference Manual*.

2.16.1 File Specification Format

You store information in files on mass storage volumes. You then identify a stored file by a file specification, which names the device containing the volume holding the file. A complete file specification includes the device name, the file name, and the file type in the format:

devicename:filename.filetype

For example:

```
DU1:MEMO1.TXT
```

2.16.2 DCL Command Format

The primary way of interacting with the computer is through the DCL command language. The following is the general format of this language:

COMMAND[/OPTION...] INPUT[/OPTION...] OUTPUT[/OPTION...]

where:

COMMAND is a word indicating the task you want to accomplish. Word commands can be abbreviated to the least ambiguous letters, usually three or four. For example:

```
DIRECTORY
```

or

```
DIR
```

/OPTION alters the normal (default) operation of a command. For example:

```
DIRECTORY/ALPHABETIZE/VOLUMEID
```

or

```
DIR/ALP/VOL
```

INPUT is information to be used during the operation. The MEMO files are the input in the following two examples:

```
PRINT MEMO1.TXT
PRINT MEMO1.TXT,MEMO2.TXT,MEMO3.TXT
```

OUTPUT is information to be created as a result of the operation. The file MEMO1.OLD is the output in the following example:

```
COPY DU0:MEMO1.TXT DU1:MEMO1.OLD
```

2.16.3 Useful Keys and Key Sequences

The following list gives useful keys and key sequences discussed in this chapter:

CTRL/C

Exits a program (for example, the INDEX or HELP program) when the program is prompting you for input and returns control to the monitor. Also cancels a command that has not yet been executed.

CTRL/C CTRL/C

Aborts a command or program that is in the process of executing and returns control to the monitor.

CTRL/O

Inhibits the remainder of the output from being displayed on the terminal screen.

CTRL/U

Deletes the line of characters to the left of the cursor.

RETURN or RET

Executes a command line.

DELETE or <X

Removes one character at a time to the left of the cursor.

2.16.4 DCL Commands

The following alphabetical list gives the major DCL commands discussed in this chapter:

ASSIGN *physical-device-name logical-device-name*

Defines a logical name equivalent for a physical device.

CREATE *filespec/ALLOCATE:size*

Creates a file entry in a volume's directory and allocates the specified amount of space for the file on that volume. This command does not store any data in a file.

DATE *[dd-mmm-yy]*

Sets or displays the current system date, if it has been set.

DEASSIGN *[logical-device-name]*

Removes logical name assignments made with ASSIGN.

DIRECTORY *[ddn:]*

Lists a directory of the volume in the specified device. If you do not specify a device, this command lists a directory of the default storage volume (DK). The listing is displayed on the terminal screen in the order in which the files are on the volume.

DIRECTORY/ALPHABETIZE *[ddn:]*

Lists a directory of the specified volume. The listing is displayed on your terminal screen in alphabetical order.

DIRECTORY/BRIEF *[ddn:]*

Lists a brief directory of the specified volume. The listing is displayed on the terminal screen, showing only file names.

DIRECTORY/NEWFILES *[ddn:]*

Lists on your terminal screen only those files on the specified volume having the current system date.

DIRECTORY/PRINTER *[ddn:]*

Lists on the system printer the directory of the specified volume.

DIRECTORY/VOLUMEID *ddn:*

Displays on your terminal screen the volume ID and owner name along with the directory listing of the the specified volume.

HELP

Displays brief descriptions of monitor commands and command options.

INITIALIZE *ddn:*

Creates and clears a file directory structure on the specified volume.

MOUNT *logical-disk-unit logical-disk-file*

Associates a logical disk unit with a logical disk file.

R *filespec*

Runs a program located on the specified file-structured device, or, if not specified, on the system (SY) volume.

RUN *filespec*

Runs a program located on the specified file-structured device, or, if not specified, on the default storage (DK) device.

SETUP *CLEAR*

Clears the terminal screen and places the cursor at the top left of the screen.

SETUP *PRINTER HELP*

Lists SETUP commands letting you set various printer characteristics.

SETUP *TERMINAL HELP*

Lists SETUP commands letting you set various terminal characteristics.

SHOW *[optional-information]*

Lists on the terminal screen the installed devices and logical device assignments. Options are also available to make this command show other configuration information.

TIME *[hh:mm:ss]*

Sets or displays the current system time.

Chapter 3
Managing Files

This chapter describes how to manage files, including how to:

- Display files
- Print files
- Copy files
- Combine files
- Compare similar files
- Rename files
- Delete files
- Protect and unprotect files from deletion
- Backup files on a volume
- Restore backed-up files to their original state
- Consolidate fragmented free space on your storage volume

The exercises in this chapter assume you have created the logical disk file INTRO.DSK on your default storage volume, mounted the file as logical device LD0, initialized it, and assigned it as your default storage volume. If you have not, issue the following command sequence (for information on the commands, see Chapter 2, Section 2.8.6):

```
.CREATE INTRO.DSK/ALLO:400  RET
.MOUNT LD0: DK:INTRO.DSK  RET
.INIT/NOQ LD0:  RET
.ASSIGN LD0: DK:  RET
```

Whenever you start your computer, RT–11 assigns your system device as your default storage device. So, if you have followed the instructions in Chapter 2, but have restarted your computer since reading that chapter, reassign your logical disk device as your default storage device:

```
.ASSIGN LD0: DK:  RET
```

3.1 Displaying Files

The TYPE command displays the contents of a file on your terminal screen. This command runs the PIP (peripheral interchange program) utility contained in the file PIP.SAV on your system volume. Issue the command in the following format:

TYPE filespec

If you are displaying a file that contains more than 24 lines of information, the file scrolls off the screen. The contents of the file appear at the bottom of the screen and eventually disappear off the top as if the file were on a scroll being unrolled at the bottom and rolled at the top.

Display the contents of the file DEMOF3.FOR:

```
.TYPE SY:DEMOF3.FOR  RET
```

Since this file contains more than 24 lines, when the contents of the file stop scrolling, you will not be able to see the beginning of the file.

Issue the preceding command again, but this time press NO SCROLL or HOLD SCREEN before the beginning of the file scrolls from view. Pressing NO SCROLL or HOLD SCREEN once stops the screen display from scrolling.

Resume the scrolling by pressing NO SCROLL or HOLD SCREEN a second time.

3.2 Printing Files

RT–11 supports many kinds of printers from line printers to laser printers.

3.2.1 Checking the Printer Interface

Check your hardware manual to see which interface your printer uses. PDP–11 computers can use either serial- or parallel-interface printers.

RT–11 uses the LS handler for serial-interface printers and the LP handler for parallel-interface printers. However, by default RT–11 utilities send print jobs to a device named LP. So, if your printer is a serial-interface printer, you must logically associate the name LP with the LS handler by issuing the following ASSIGN command before you use the PRINT command for the first time:

```
.ASSIGN LS LP  RET
```

Since you must use the ASSIGN command each time you start RT–11, you may want to include this command in your startup command file (see Chapter 8).

3.2.2 Managing the Printing Process

The PRINT command prints one or more files on your printer. This command runs the PIP utility and can cause the SPOOL utility (if it is running) to manage the printing process, freeing your terminal for your use while your file is printing. See Chapter 7 for an explanation of how to use SPOOL.

If you do not use SPOOL, you may be delayed in getting the monitor prompt back until your file has finished printing.

3.2.3 Issuing the PRINT Command

Issue the PRINT command in the following format:

PRINT filespec

Print a listing of the file DEMOF3.FOR:

```
.PRINT SY:DEMOF3.FOR RET
```

If you do not use wildcards, you can specify up to six individual files with the PRINT command. Separate each filespec with a comma:

```
PRINT MEMO1.TXT,MEMO2.TXT,MEMO3.TXT,MEMO4.TXT,MEMO5.TXT,MEMO6.TXT
```

By using wildcards in a filespec, you can print more than six files at a time.

```
PRINT MEMO%.*
```

Issue the following command to print two short FORTRAN demonstration files located on your system volume:

```
.PRINT SY:DEMOF1.FOR,SY:DEMOF2.FOR RET
```

One of the several options of this command is the /COPIES:n option. For example, if you wanted to print three copies of the file DEMOF2.FOR, you would issue the command:

```
.PRINT SY:DEMOF2.FOR/COPIES:3 RET
```

3.2.4 Stopping a File from Printing

You can stop RT–11 from sending file text to the printer while it is printing in several ways. (Note that if your printer has a large text storage buffer, the file will not stop printing until the buffer is empty.)

- If you are not running SPOOL, then press CTRL/C twice to stop the print job.
- If you are running the SPOOL utility, issue the following KMON command to stop the print job:

  ```
  .SET SP KILL RET
  ```

  SP is the handler for the SPOOL utility. See the *RT–11 System Utilities Manual* for a description of all SPOOL SET commands.

3.3 Copying Files

The COPY command makes a duplicate of a file or files and places it on the volume and with the file name you specify. The original version of the file is unaffected; that is, a copy of the original version is made and moved to the new location. You should copy important files to a backup volume for safekeeping.

Issue the command in the following format:

COPY in-filespec out-filespec

This command has a wide selection of options letting you perform many types of copy operations such as copying the bootstrap code to the bootstrap blocks on the same volume or copying all the files on a large volume to several small volumes.

Do the following exercise:

1. Copy the file DEMOF1.FOR from the system volume to your storage volume and name it EXAMP.ONE:

   ```
   .COPY SY:DEMOF1.FOR EXAMP.ONE RET
   ```

2. Next, copy the file DEMOF2.FOR from the system volume to your storage volume and name it EXAMP.TWO:

   ```
   .COPY SY:DEMOF2.FOR EXAMP.TWO RET
   ```

3. Then, issue the DIRECTORY command to verify that your storage volume contains the two files EXAMP.ONE and EXAMP.TWO. You should get a listing similar to the following:

   ```
   .DIR RET
    11-SEP-88
   EXAMP .ONE   1P 26-AUG-88   EXAMP .TWO   1P 26-AUG-88
    2 Files, 2 Blocks
    390 Free blocks
   ```

   Notice that the files have a P next to their block size. When you copied those two files from the system volume, you not only copied the files' contents but also automatically copied the protected status and creation dates of the original files. The P means these two files cannot be deleted by the DELETE command unless you first unprotect them with the UNPROTECT command (see Section 3.9).

4. Finally, verify that the original files are still on the system volume, by issuing the following command:

   ```
   .DIR SY:DEMOF%.FOR RET
   ```

3.4 Combining Files

Use the COPY/CONCATENATE form of COPY to combine several input files into one output file.

You can use wildcards in the input filespec. For example, the following command copies all files on volume DU1 with a .TXT file type into the one file MEMOS.TXT on that volume.

```
.COPY/CON DU1:*.TXT DU1:MEMOS.TXT RET
```

If you do not use wildcards, you must use commas. For example:

```
.COPY/CON DU1:MEMO1.TXT,MEMO2.TXT,MEMO3.TXT DU1:MEMOS.TXT RET
```

In these two examples, the input files remain the same and the output file MEMOS.TXT is either created (if it does not exist) or is replaced to include only the joined files (if it already exists and is *unprotected* — See Sections 3.8 and 3.9).

Combine the two files EXAMP.ONE and EXAMP.TWO into one file EXAMPS.TXT:

```
.COPY/CON EXAMP.* EXAMPS.TXT RET
```

Check your directory to verify that you now have the new file EXAMPS.TXT; and note that the new file is two blocks in length, since it is a combination of two 1-block files.

```
.DIR RET
 11-SEP-88
EXAMP .ONE   1P 26-AUG-88   EXAMP .TWO   1P 26-AUG-88
EXAMPS.TXT   2  10-Sep-88
 3 Files, 4 Blocks
 388 Free blocks
```

The file EXAMP.ONE is almost identical to the file EXAMP.TWO. So the file EXAMPS.TXT has repetitious information. For a further check of EXAMPS.TXT, display the file with the TYPE command:

```
.TYPE  EXAMPS.TXT RET
```

3.5 Comparing Text Files

The DIFFERENCES command compares two ASCII text files and produces a listing of any differences between them. This command runs the SRCCOM (source comparison program) utility and requires that the file SRCCOM.SAV be on the system (SY) volume.

Issue the command in the following format:

DIFFERENCES filespec-1 filespec-2

During a text comparison, RT–11 compares the two specified files, character for character, and lists any lines that contain differences. Differences between uppercase and lowercase characters are listed. By default, the listing is displayed on the terminal screen. However, you can redirect the listing from the terminal to a file by using the /OUTPUT:*filespec* option, or to the printer, by using the /PRINTER option.

Usually, you perform a text comparison between two files that you expect to be the same or at least similar. For example, if an individual has copied one of your files to make changes to it, you can quickly scan the changes by performing a text comparison between the new version and your original.

Another use of a text comparison is to check edits you have made to a file yourself. By comparing the backup file against the edited version, you can proofread the changes since only the portions of text that are different are printed.

If you compare two files that are identical, RT–11 does not create a listing, but displays the following message on the terminal screen:

```
?SRCCOM-I-No differences found
```

If you compare two files that are different, RT–11 produces a listing of the differences and displays the following message on the terminal screen:

```
?SRCCOM-W-Files are different
```

3.5.1 Two Example Files

To understand how to interpret the output listing, first look at the following two sample FORTRAN text files. These are the two files, EXAMP.ONE and EXAMP.TWO, that you copied onto your default storage device according to the instructions of the preceding section. Note the two differences between the files:

- In line 7, the first file has *go to 10*, while the second file has *go to 100*.
- In line 14, the first file has the variable *radamg*, while the second file has the variable *radang*.

Example 1: FORTRAN File with Errors (EXAMP.ONE)

```
        real function ASIND( x)
        real x
c
c       This FORTRAN callable function returns the ARCSINE
c       of  a specified value as an angle in degrees.
c
        if (ABS( x) .lt. 1.0) go to 10
        ASIND = x * 90.0
        return
c
c       Use trigonometric identity to calculate ARCSINE of X.
c       Then convert radians to degrees.
c
  100   radamg = ATAN( x / SQRT( 1.0-x**2))
        ASIND = radang * 57.29577951
        return
c
        end
```

Example 2: FORTRAN File Without Errors (EXAMP.TWO)

```
        real function ASIND( x)
        real x
c
c       This FORTRAN callable function returns the ARCSINE
c       of  a specified value as an angle in degrees.
c
        if (ABS( x) .lt. 1.0) go to 100
        ASIND = x * 90.0
        return
c
c       Use trigonometric identity to calculate ARCSINE of X.
c       Then convert radians to degrees.
c
```

```
  100   radang = ATAN( x / SQRT( 1.0-x**2))
        ASIND = radang * 57.29577951
        return
c
        end
```

Display the files so you can see their contents on your terminal screen and can compare them with the display in this manual:

```
.TYPE EXAMP.ONE  RET
```

```
.TYPE EXAMP.TWO  RET
```

3.5.2 Comparing the Example Files

Issue the following command to compare the two files EXAMP.ONE and EXAMP.TWO and to create the file EXAMP.DIF containing a listing of any differences:

```
.DIFFERENCES/OUTPUT:EXAMP EXAMP.ONE EXAMP.TWO  RET
```

Since the files are different, RT–11 will display the following message on your screen:

```
?SRCCOM-W-Files are different
```

Next, issue the DIRECTORY command to verify that you have a new file EXAMP.DIF and then display that file. You will get a directory listing similar to the following plus the following file display:

```
.DIR  RET
 10-Sep-88
EXAMP .ONE   1P 26-Aug-88   EXAMP .TWO   1P 26-Aug-88
EXAMPS.TXT   2  10-Sep-88   EXAMP .DIF   1  10-Sep-88
 4 Files, 5 Blocks
 387 Free blocks
```

```
.TYPE EXAMP.DIF  RET
1) DK:EXAMP.ONE
2) DK:EXAMP.TWO
**********
1)1             if (ABS( x) .lt. 1.0) go to 10
1)              ASIND = x * 90.0
****
2)1             if (ABS( x) .lt. 1.0) go to 100
2)              ASIND = x * 90.0
**********
1)1       100   radamg = ATAN( x / SQRT( 1.0-x**2))
1)              ASIND = radang * 57.29577951
****
2)1       100   radang = ATAN( x / SQRT( 1.0-x**2))
2)              ASIND = radang * 57.29577951
**********
```

The first two lines identify the two files being compared. Each file name and the device on which the file resides are printed. For example:

```
1) DK:EXAMP.ONE
2) DK:EXAMP.TWO
```

The numbers at the left margin have the form *n)m*, where *n* specifies the files (either 1 or 2) and *m* specifies the page of that file on which the specific line is located. In this case, both input files contain only one page.

RT–11 displays 10 asterisks both before and after a section showing one or more differences between two files. In addition, within each section, a line of 4 asterisks separates the two files being compared, thus dividing each difference section into two subsections. For example:

```
**********
1)1             if (ABS( x) .lt. 1.0) go to 10
1)              ASIND = x * 90.0
****
2)1             if (ABS( x) .lt. 1.0) go to 100
2)              ASIND = x * 90.0
**********
```

Each difference section ends with a matching line, used as a reference to identify the location of the differing lines. For example:

```
ASIND = x * 90.0
```

and

```
ASIND = radang * 57.29577951
```

3.6 Renaming Files

The RENAME command assigns a new name (or a new file type or both) to a file in a volume directory without altering or moving the file itself. Since this command does not move a file, the volume indicated in the input and output portions of the command must be the same; otherwise, you get an error message. The command uses the PIP utility. Issue the command in the following format:

RENAME in-filespec out-filespec

Rename the file EXAMP.ONE to EXAMP1.FOR and the file EXAMP.TWO to EXAMP2.FOR:

```
.RENAME EXAMP.ONE EXAMP1.FOR RET
.RENAME EXAMP.TWO EXAMP2.FOR RET
```

Issue the DIRECTORY command to verify that your files are renamed. You should get a listing similar to the following:

```
.DIR RET
 10-Sep-88
EXAMP1.FOR   1P 26-Aug-88   EXAMP2.FOR   1P 26-Aug-88
EXAMPS.TXT   2  10-Sep-88   EXAMP .DIF   1  10-Sep-88
 4 Files, 5 Blocks
 387 Free blocks
```

3.7 Deleting Files

Use the DELETE command when you no longer need a file. DELETE removes information about a file from a volume's directory; and the space that the file occupies on the volume becomes available for reuse. You can specify up to six individual files at a time for deletion, though if you use wildcards, you can delete any number of files at once. Issue the command in the following format:

DELETE filespec,filespec,...

If you use wildcards with the DELETE command or if you use the /QUERY option, RT–11 requests confirmation from you for each file to be deleted. The /QUERY option is a helpful way of making sure you do not delete the wrong file by accidentally typing in a wrong letter or letters in the file specification. Typing Y in response to the confirmation prompt instructs RT–11 to delete the indicated file; typing N in response instructs RT–11 not to delete that file and go on to the next.

The DELETE command, however, will not delete files that have a protected status, indicated by the P next to their file size in a directory listing. You must first UNPROTECT such files before you can delete them (see Section 3.9).

Issue the DELETE command either with the asterisk wildcard and factoring or with the /QUERY option, as in the following examples. Then, delete the file EXAMPS.TXT:

```
.DELETE *.(FOR,TXT) RET
PIP-W-Protected file LD0:EXAMP1.FOR
PIP-W-Protected file LD0:EXAMP2.FOR
 Files deleted:
LD0:EXAMPS.TXT ? Y RET
```

or

```
.DELETE/QUERY EXAMP1.FOR,EXAMP2.FOR,EXAMPS.TXT,EXAMP.DIF RET
PIP-W-Protected file LD0:EXAMP1.FOR
PIP-W-Protected file LD0:EXAMP2.FOR
 Files deleted:
LD0:EXAMPS.TXT ? Y RET
LD0:EXAMP.DIF ? N RET
```

Issue the DIRECTORY command to see that you still have the files EXAMP1.FOR, EXAMP2.FOR, and EXAMP.DIF, but not the file EXAMPS.TEX. You should get a listing similar to the following:

```
.DIR RET
 10-Sep-88
EXAMP1.FOR   1P 26-Aug-88   EXAMP2.FOR   1P 26-Aug-88
EXAMP .DIF   1  10-Sep-88
 3 Files, 3 Blocks
 389 Free blocks
```

If you use wildcards with the DELETE command, RT–11 automatically excludes from deletion any files with a .SYS file type. To include .SYS files in a deletion command that uses wildcards, you must include the /SYSTEM option and you must first have unprotected the files.

3.8 Protecting Files

A file that has a *protected* status cannot be deleted or modified by an editor until that status is removed.

DIGITAL gave the files DEMOF2.FOR and DEMOF3.FOR a protected status so that you would not accidentally delete them. These files kept their protected status both when you copied the files with the COPY command and when you renamed them with the RENAME command. RT–11 does this so you do not accidentally delete a protected file after you copy or rename it.

The PROTECT command, which runs the PIP utility program, gives a file protected status. You can include up to six file specifications (separated by commas) with this command, and wildcards are supported. However, if you specify the /QUERY option with the command, RT–11 requests confirmation for each file specified before it will be protected.

Issue the command in the following format:

PROTECT filespec

Give the file EXAMP.DIF protected status and then display a listing of that file to verify that a P is placed to the right of the file's block number (size):

```
.PROTECT EXAMP.DIF  RET

.DIR  RET
 10-Sep-88
EXAMP1.FOR   1P 26-Aug-88   EXAMP2.FOR   1P 26-Aug-88
EXAMP .DIF   1P 10-Sep-88
 3 Files, 3 Blocks
 389 Free blocks
```

3.9 Unprotecting Files

The UNPROTECT command removes a file's protected status so you can delete the file or edit it with an editor. This command also runs the PIP utility program. Issue the command in the following format:

UNPROTECT filespec

You can include up to six file specifications (separated by commas) with this command, and wildcards are supported. However, if you specify the /QUERY option with the command, RT–11 requests confirmation for each file specified before it will be unprotected. You can specify up to six separate files (separated by commas) with this command, or you can specify any number of files if you include wildcards in your specification. However, if you specify the /QUERY option with the command, RT–11 requests confirmation for each file specified before it will unprotect it.

Remove the protected status from the file EXAMP1.FOR. Then list the directory of your default storage volume to verify that the P is removed from the block number beside the listing of the file:

```
.UNPROTECT EXAMP1.FOR RET

.DIR RET
 10-Sep-88
EXAMP1.FOR   1  26-Aug-88   EXAMP2.FOR   1P 26-Aug-88
EXAMP .DIF  1P 10-Sep-88
 3 Files, 3 Blocks
 389 Free blocks
```

Finally, to verify that the file EXAMP1.FOR is unprotected, delete that file from your directory, and check the directory listing to see that the file is no longer there:

```
.DELETE EXAMP1.FOR RET

.DIR RET
 10-Sep-88
EXAMP2.FOR   1P 26-Aug-88   EXAMP .DIF   1P 10-Sep-88
 2 Files, 2 Blocks
 390 Free blocks
```

3.10 Backing Up and Restoring Files

Issue the BACKUP command to make a backup copy of a whole disk, many files, or a large file. The BACKUP command runs BUP, the backup utility program, contained in the file BUP.SAV on your system volume. BUP is especially designed to efficiently and speedily back up large amounts of information. Unlike the COPY command, the BACKUP command can copy files across volumes; that is, the backup procedure can copy one portion of a backed-up file to one volume and the rest of the file (if it does not fit on the first volume) to a second volume.

Issue the BACKUP command in the following format:

BACKUP in-filespec out-filespec

The BACKUP command can do three kinds of back up operations. It can backup:

- One or more files into a container saveset file
- An entire disk into a container saveset file
- One or more files into a new logical disk file that BUP creates as part of the operation

While you can use the first two operations with a disk, diskette, or magtape as the backup volume, you can use the last operation only with a disk or large enough diskette as the backup volume.

NOTE

Unless you are doing the third backup operation, the directory structure of a backup volume made by BUP is not the same as the directory structure of a volume made by other RT–11 utilities, such as PIP. This means you have to *restore* a backup volume before you can use the files on that volume (except for logical disk backup volumes).

This section describes how to use the BACKUP command to do the following:

- Back up files on a volume
- List a directory on a backup volume
- Restore backed-up files

For a complete description of the BACKUP command and utility, see Chapter 14.

3.10.1 Backing Up Files

One advantage of backing up only files, rather than an entire disk, is that you do not copy unused areas on a disk.

Use the /VERIFY option with the BACKUP command to verify that the copied (backed-up) files are an exact match of the original ones. The verification procedure for the BACKUP command is relatively fast since it is especially designed to compare large areas of a volume at a time.

It is not necessary to initialize a new volume separately when using the BACKUP command, since the backup procedure has its own initialization procedure as a part of the backup. Once, however, BACKUP initializes a volume, the backup procedure does not automatically reinitialize it if the procedure uses the same backup volume again. This is useful since it lets you store more than one set of backed-up files on a volume having extra storage space.

BACKUP compartmentalizes all the files it backs up at one time; that is, it saves them in one file called a *saveset*. The default file type for this file is .BUP, and the default file name with file type is BACKUP.BUP. The following example command, however, gives the name MYBACK.BUP to the backup file:

```
.BACKUP/VERIFY *.* DU1:MYBACK RET
```

During the BACKUP procedure, RT–11 prompts you with messages asking for operator confirmation. For example, BACKUP prompts you to confirm the initialization of a new backup volume. In response to the initialization prompt, type Y if there are no files on the new backup volume that you want to save. Otherwise, type N and start over with a volume you can use as a backup.

When you issue the BACKUP command, your interaction with RT–11 should be similar to what follows, depending on the type of volume you use as a backup and your response to the prompts. The ellipsis indicates the names of the files that are copied:

```
.BACKUP/VERIFY DU0:*.* DU1:MYBACK RET
Mount output volume in DU1: Continue? Y RET
?BUP-W-Not a BACKUP volume
DU1:/BUP Initialize; Are you sure? Y RET
?BUP-I-Bad block scan started
?BUP-I-No bad blocks detected
?BUP-I-Creating output volume 1
 Files backed up:
    .
    .
    .
?BUP-I-Verify pass started
?BUP-I-Backup/Verify operation is complete
```

3.10.2 Listing a BACKUP Volume Directory

Backup volumes are uniquely formatted and have directories structured differently from other RT–11 volumes. So, you need the /DIRECTORY option of the BACKUP command (rather than the DIRECTORY command) to get a directory listing of what is on a backup volume.

Use the BACKUP/DIRECTORY command to display on your terminal a directory listing of what is on your your backup volume. For example:

```
.BACKUP/DIRECTORY DU1: RET

 RT-11 BACKUP
 10-Sep-88 10:59

 Saveset      Section      Blocks      Date

 MYBACK.BUP   1/1          11/11       10-Sep-87

 1 Saveset Section, 11 Blocks
 781 Free blocks
```

The following explains the *section* and *blocks* columns of the preceding display:

- **Section**

  A section of a saveset is the amount of the saveset that fits on one backup volume. So, the number of sections in a saveset is the same as the number of volumes used to back up a disk.

  The number after the slash in the *section* column indicates how many sections a saveset file is divided into. And the number before the slash in that column indicates the number of the saveset section on that backup volume. For example, the 1/1 for the saveset MYBACK.BUP means the saveset is undivided and the entire saveset file is on that volume. However, a section column of 1/2 means the saveset file is divided into 2 sections (since it did not fit on the volume), and the first section is contained on that backup volume.

- **Blocks**

  The number after the slash in the *blocks* column indicates how many blocks a saveset has. The number before the slash in that column indicates how many of those blocks are on that volume.

For example, the saveset MYBACK.BUP has 11 blocks and all of them are on the one backup volume. However, if MYBACK.BUP had 1200 blocks, the entire saveset would not fit on one diskette. In that case, the *blocks* column would indicate 792/1200; that is, the saveset contains 1200 blocks, but only 792 of them are on that backup volume.

Though the two files contained in the saveset file MYBACK.BUP contain only 2 blocks, a directory of 9 blocks is also part of the saveset file, making the total block number 11.

One or more additional backup savesets (depending on their block size) could also be copied to the example backup volume since that volume still has 781 free blocks.

3.10.3 Restoring BACKUP Files

BACKUP volumes cannot be used as you would use other volumes of RT–11 files. The purpose of a backup volume is only to store files. To use backed-up files, you must *restore* them onto another volume.

Issue the BACKUP/RESTORE command (as in the following example) to restore a volume(s) of files from one saveset file back into one volume of separate, usable files.

The /SAVESET option tells RT–11 which saveset on the specified device you want restored. If you do not use this option, RT–11 retrieves the files from the default saveset BACKUP.BUP on disk volumes or from the first available saveset on tape volumes.

After the /SAVESET option, type a comma and indicate which files in the saveset you want to restore. The wildcards in the following example tell RT–11 to restore all the files in the saveset. However, you do not have to restore all the files in a saveset, and you can put individual file names here rather than wildcards.

Specify the SAVESET option after the saveset filespec as in the following example. In this example, only one backup volume was used to back up the files. The ellipsis indicates the names of the files contained in the saveset:

```
.BACKUP/RESTORE DU1:MYBACK/SAVESET,*.* DU0:*.* RET
Mount input volume 1 in DU0; Continue? Y RET
?BUP-I-Restore operation started from volume 1
 Files restored:
   .
   .
   .
?BUP-I-Restore operation is complete
```

3.10.4 An Exercise in Backing Up and Restoring Files

To see what happens when you use the BACKUP utility both to *back up* and to *restore* files, do the following exercise:

1. Insert a diskette (either a new one or one having files you do not want to save) into one of your diskette drives so it can be used as a backup diskette.

2. Issue the following command to back up your logical disk files onto the diskette and type Y in response to the prompts. The example presumes a new, unused diskette:

```
.BACKUP/VERIFY *.* DU1:MYBACK RET
Mount output volume in DU1; Continue? Y RET
?BUP-W-Volume not RT-11 initialized
DU1:/BUP Initialize; Are you sure? Y RET
?BUP-I-Bad block scan started
?BUP-I-No bad blocks detected
?BUP-I-Creating output volume 1
 Files backed up:
LD0:EXAMP2.FOR
LD0:EXAMP.DIF
?BUP-I-Verify pass started
?BUP-I-Backup/Verify operation is complete
```

3. Unprotect and delete the files on LD0, the logical disk device you should be using as your default storage volume (DK):

```
.UNPROTECT *.* RET
 Files unprotected:
DK:EXAMP2.FOR
DK:EXAMP.DIF

.DELETE *.* RET
 Files deleted:
DK:EXAMP2.FOR  ? Y RET
DK:EXAMP.DIF   ? Y RET
```

4. Check the directory of your default storage device to see that it is now empty:

```
.DIR RET
 10-Sep-88

 0 Files, 0 Blocks
 392 Free blocks
```

5. With the BACKUP/DIRECTORY command, check the directory of your backup volume to see what is recorded:

```
.BACKUP/DIRECTORY DU1: RET

 RT-11 BACKUP
 10-Sep-88 10:59

 Saveset      Section      Blocks      Date

 MYBACK.BUP   1/1          11/11       10-Sep-87

 1 Saveset Section, 11 Blocks
 781 Free Blocks
```

6. With the BACKUP/RESTORE command and the /SAVESET input option, restore your backed-up files on LD0. Type Y in response to the mount prompt:

```
.BACKUP/RESTORE DU1:MYBACK/SAVESET,*.* LD0:*.* RET
Mount input volume 1 in LD0; Continue? Y RET
?BUP-I-Restore operation started from volume 1
 Files restored:
DK:EXAMP2.FOR
DK:EXAMP.DIF
?BUP-I-Restore operation is complete
```

When RT–11 restores files in this way, it does not initialize the volume it is restoring, but rather adds the restored files to the volume. This is useful when you do not want to delete files already on a volume, but only want to add some backed-up files to that volume.

However, if a file of the same name as a backed-up file already exists on a volume, BACKUP does either of two things. If the file already on the restoration volume is unprotected, the backed-up file of the same name takes its place. But if the file already on that volume is protected, the backed-up file is not restored.

7. Check the directory of your restored volume:

```
.DIR RET
 10-Sep-88
EXAMP2.FOR   1P 26-Aug-88   EXAMP .DIF   1P 10-Sep-88
 2 Files, 2 Blocks
 390 Free blocks
```

3.11 Consolidating Fragmented Free Space

At some point, you may run out of room to expand or add more files to a volume. If you run out of room on a volume, RT–11 displays a message telling you the device is full or that RT–11 is unable to open the output file.

Sometimes, however, there is still plenty of free space on a volume, but as you create and delete files, the free space becomes fragmented and therefore not always usable. For example, a large file might not fit on a volume because no single area of free space is large enough to accommodate the file, though if the free spaces were gathered together in one place, the large file could fit on the volume.

When the free space on a volume becomes too fragmented, use the SQUEEZE command to consolidate all the free space into a single area on the volume.

The SQUEEZE command runs DUP, the device utility program contained in the file DUP.SAV on your system volume. SQUEEZE consolidates all free space or unused areas on a random-access volume into a single area on that volume. This command is useful, since the RT–11 file system requires that each file occupy one contiguous space on a volume.

Issue the command in the following format:

SQUEEZE ddn:

To prevent you from using bad blocks on a volume, the SQUEEZE command does not move files with a .BAD file type. Because monitor or handler files might be moved if you perform a squeeze operation on the system device, RT–11 automatically reboots the running monitor when the compressing operation finishes.

CAUTION

- You should not squeeze the system volume (SY) while a foreground job is running since the job may be using the volume and squeezing the volume at this time could destroy data or cause a system crash. (See Chapter 5 and Chapter 7 for information on foreground jobs.)
- You should be careful when squeezing any disk volume while a foreground job is loaded. If foreground jobs have open files on the volume being squeezed, you could lose data in those files and/or corrupt that volume.
- If a computer error or a power failure occurs during a squeeze operation, the contents of the volume will be lost. Make a backup copy of an important or large volume before using the SQUEEZE command.

3.11.1 A Procedure for Using the SQUEEZE Command

A general procedure for using the SQUEEZE command follows:

1. With the DIRECTORY/FULL command, check your volume directory for unwanted files and for how much free space you have between files.
2. Delete all unwanted files. For example, files with a .BAK file type (backup files made by the editor program) might be files to delete.
3. Having deleted unwanted files, get the directory size (the number of occupied blocks left on your storage volume) by issuing the DIRECTORY/SUMMARY command.
4. With the BACKUP/VERIFY command, make a backup copy of your volume:

   ```
   .BACKUP/VERIFY *.* DU1:MYBACK RET
   ```

5. Issue the SQUEEZE command:

   ```
   .SQUEEZE DK: RET
   ```

6. With the /FREE option of the DIRECTORY command, check that all free blocks are now contiguous:

   ```
   .DIR/FREE DK: RET
   ```

7. Once you have verified that the SQUEEZE operation performed as you expected, you can if necessary use the backup volumes for other purposes.

3.11.2 An Exercise in Using the SQUEEZE Command

To familiarize yourself with the SQUEEZE command, do the following exercise:

1. Issue the DIRECTORY command with the /FULL option to see how much free space is on your default storage volume (INTRO.DSK) and to see where the free space is located. You should get a display similar to the following:

   ```
   .DIR/FULL RET
    10-Sep-88
   < UNUSED >   1             EXAMP2.FOR   1P 26-Aug-88
   < UNUSED >   2             EXAMP .DIF   1P 10-Sep-88
   < UNUSED >   387
    2 Files, 2 Blocks
    390 Free blocks
   ```

   The preceding display means that there is one block of unused space before the file EXAMP2.FOR, two blocks of unused space between the two files listed, and the remaining unused blocks after the listed files.

2. To make the free blocks contiguous, issue the SQUEEZE command and type Y in response to the confirmation prompt:

   ```
   .SQUEEZE DK: RET
   LD0:/Squeeze; Are you sure? Y RET
   ```

3. Issue the DIRECTORY/FREE command to verify that your files and the free block have been moved. You should get a display similar to the following:

   ```
   .DIR/FREE RET
    10-Sep-88
   < UNUSED >    390
    0 Files, 0 Blocks
    390 Free blocks
   ```

   The one < UNUSED > entry means that all the unused blocks are together.

3.12 Chapter Summary

This chapter describes the major KMON commands that help you manage files. You should use these commands to manage and maintain your files. The rest of this book presumes that you know how to do this. See the *RT–11 Commands Manual* for further information on these commands and for other less commonly used commands. The next two sections list the commands that run each utility named in this chapter and summarize those commands.

3.12.1 Commands and Utilities

Table 3–1 contains the commands and utilites discussed in this chapter. Each command is matched with the utility it runs. That utility must reside on your system (SY) device.

Table 3–1: Commands and Utilities

Command	Required Utility
BACKUP/DIRECTORY	BUP
BACKUP/RESTORE	BUP
BACKUP/VERIFY	BUP
COPY	PIP
COPY/CONCATENATE	PIP
DELETE	PIP
DELETE/QUERY	PIP
DIFFERENCES	SRCCOM
DIFFERENCES/OUTPUT:filespec	SRCCOM
DIFFERENCES/PRINTER	SRCCOM
DIRECTORY/FREE	DIR
DIRECTORY/FULL	DIR
PRINT	PIP (and SPOOL, if running)
PROTECT	PIP
RENAME	PIP
SQUEEZE	DUP
TYPE	PIP
UNPROTECT	PIP

3.12.2 File Management Commands

The following alphabetical list gives the monitor commands discussed in this chapter:

BACKUP/DIRECTORY *ddn:*

Lists on the terminal screen the saveset files on a volume.

BACKUP/RESTORE *in-filespec/SAVESET,file-1,file-2, . . . out-filespec*

Restores one or more files from the specified saveset to a one or more files on an output volume.

BACKUP/VERIFY *in-filespec,filespec, . . . out-filespec*

Backs up (copies) one or more files from an input volume to one or more output backup volumes and verifies that the input matches the output.

COPY *in-filespec,filespec, . . . out-filespec*

Copies a file or files from one location to another.

COPY/CONCATENATE *in-filespec,filespec, . . . out-filespec*

Combines two or more input files into one output file.

DELETE *filespec,filespec, . . .*

Deletes a file or files from a volume's directory.

DELETE/QUERY *filespec,filespec, . . .*

Deletes a file or files from a volume's directory only if you confirm each deletion at each confirmation prompt.

DIFFERENCES *filespec-1 filespec-2*

Compares two files and lists on the terminal screen the differences between them.

DIFFERENCES/OUTPUT:filespec *filespec-1 filespec-2*

Compares two files and lists in the file specified by the /OUTPUT option the differences between them.

DIFFERENCES/PRINTER *filespec-1 filespec-2*

Compares two files and prints on the system printer a listing of the differences between the files.

DIRECTORY/FREE *ddn:*

Lists the size of all free areas on a volume directory.

DIRECTORY/FULL *ddn:*

Lists an entire volume directory including unused areas and their sizes in blocks.

PRINT *filespec,filespec, . . .*

Prints the contents of one or more files on the system printer.

PROTECT *filespec,filespec, . . .*

Protects files so they cannot be deleted, or changed by an editor program.

RENAME *in-filespec,filespec, . . . out-filespec,filespec, . . .*

Assigns a new name (or names) to an existing file (or files).

SET *SP KILL*

Stops a file from printing, if SPOOL is managing the print job.

SQUEEZE *ddn:*

Consolidates all unused blocks and directory entries on a volume.

TYPE *filespec,filespec, . . .*

Displays the contents of one or more files on the terminal screen.

UNPROTECT *filespec,filespec, . . .*

Removes the protected status from files so they can be deleted or changed by an editor program.

Chapter 4

Using the Keypad Editor (KED)

This chapter provides a brief introduction to the KED editor and shows you how to:

- Create text files and inspect and edit existing text files.
- Use the on-line help facility.
- Store and retrieve text files.
- Use a few basic KED features, such as setting line wrap and screen width and how to turn on journal file protection.

The information in this chapter also applies to KEX, which is the default editor for the mapped monitors. KEX is a virtual KED program, so its functions and commands are the same and provide the same operations as the KED functions and commands.

Most of the information in this chapter and additional information in the *PDP–11 Keypad Editor User's Guide* are summarized in the *PDP–11 Keypad Editor Reference Card*.

4.1 Software Requirements

To run the KED editor, you need the file KED.SAV or KEX.SAV on your system (SY) volume.

- If you are using a mapped monitor, use KEX.SAV.
- If you are using any other monitor, use KED.SAV.

Verify that KED.SAV or KEX.SAV is on your system volume:

```
.DIRECTORY SY:KE%.SAV RET
```

If KED.SAV or KEX.SAV is not on your system volume, copy it to there.

4.2 Introduction to KED

KED is an interactive editor. You type text into a file as though your terminal is an electric typewriter. You can push combinations of keypad and keyboard keys to move the cursor through the file and manipulate the text. The results of text entry and manipulation operations are displayed on your terminal screen as they happen.

Basic KED Operations

Basic KED operations can be divided into the following three catagories:

- Creating text files

  This operation involves creating the original version of a text file.

- Inspecting (displaying) existing text files

  This operation lets you examine the contents of an existing text file but prohibits you from making any additions or changes to the file, thus ensuring the integrity of the file.

- Editing existing text files

  This operation lets you make additions and changes to an existing text file. KED copies the existing text file to a temporary file location and all editing is performed on the temporary file. The temporary file you are editing is made permanent only after you explicitly close it. At that point, the previous version of the file is preserved as a backup copy. Therefore, you need not worry about somehow ruining an existing text file while editing it, because you are in fact editing a copy of the existing file. You explicitly close the temporary file and make it permanent by exiting from the file with the EXIT command.

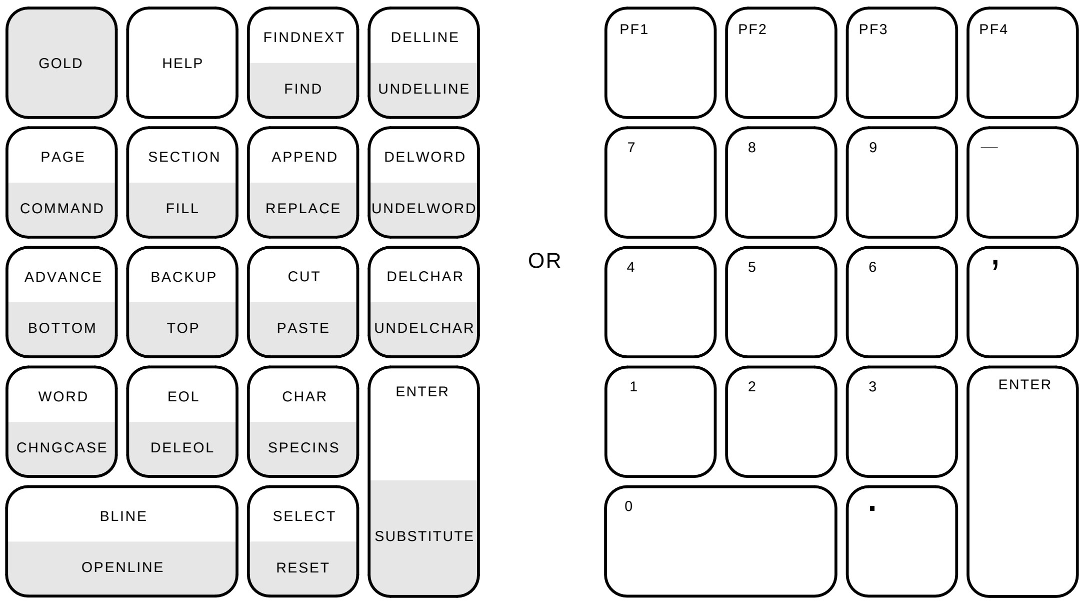
  As a further safeguard, you can stop the editing process and throw away the temporary copy of the existing file by quiting the editing process and not explicitly closing the temporary file. You throw away the temporary file (and therefore lose all edits) by exiting from the file with the QUIT command.

Keypad Function Keys

You use the keypad keys to move the cursor through text files and manipulate text. The keypad function keys are located on the right side of your keyboard.

The following diagram illustrates the relationship between the legend on the actual keypad keys and KED function for each key. The diagram also appears on the front of the *PDP–11 Keypad Editor Reference Card*. Many users place the reference card diagram near the terminal screen until the key legend/function relationship is learned.

Examine the diagram but do not worry now about remembering the key legend /function relationship. You will use most of the KED keypad functions in this chapter and the relationship between each keypad key legend and its KED function is presented throughout the chapter. The keypad key legend is included parenthetically next to the function.



MLO-003502

4.3 Creating a Text File

You create a text file by issuing the EDIT command with the /CREATE option and specifying the file you want to create. Issue the following command to create the text file WEEK.TXT. Because the command line does not include a device specification, WEEK.TXT is created on your default data (DK) volume.

```
.EDIT/CREATE WEEK.TXT  RET
```

Your terminal screen clears and you see a blinking cursor at the beginning of the text file.

4.4 Using On-Line Help

The KED editor includes an on-line help function. The keypad key marked PF2 is the HELP key.

Press HELP.

The display on your terminal screen is replaced with the first HELP screen, containing a display of the keypad function keys. The HELP key provides a handy reference to the keypad function keys. Each box in the display corresponds to a keypad key.

The HELP screen shows that many of the keypad keys can perform two functions:

- Standard function

  The standard function, indicated in the top half of the box, is performed by pressing the key alone.

- Alternate function

  A second, or alternate function, is indicated in the lower half of the box.

  In the display, note the GOLD key in the upper left corner. The GOLD key (marked PF1 on your keypad) is the KED alternate function key. The GOLD key is displayed in a different rendition (darker, lighter, or a different color) than the other keys. Each key's alternate function is indicated in the same rendition as the GOLD key, and the alternate function is performed by sequentially pressing the GOLD key and then the function key.

You can return to displaying your text file by pressing ENTER, but instead again press HELP. KED displays another information screen, containing commands that are issued from within KED (as opposed to RT–11 DCL commands). Again, you can return at any time to displaying your text file by pressing ENTER. Instead, repeatedly press HELP, displaying more screens. The final information screen illustrates KED functions that are a combination of the GOLD key and various keyboard keys.

If you again press the HELP key, KED displays the original keypad screen. Instead, press ENTER to return to your text file.

4.5 Entering Text

You have now created and entered the file WEEK.TXT. The cursor is resting at the beginning of the file. Although not apparent, the cursor is also resting at the end of the text file. To make that fact more apparent, press RETURN a few times and then press ↑ a couple of times. Note a mark is left at the furthest point in the file to which the cursor advanced; that mark is the end-of-file marker. The size of the file expands as necessary to contain entered text or space.

Press the keypad sequence GOLD TOP (PF1 5), which returns the cursor to the beginning of the file. Type the following text, using <X to correct any mistakes and RETURN to advance to the next line.

```
Monday was cold and rainy.
On Tuesday, it snowed.
Wednesday was sunny and warm.
```

Now, press RETURN sufficient times for the text you typed to scroll off the top of the screen, thereby simulating a more typical size file. Press CTRL/L, thereby entering a form feed (page separation character).

Type the following text:

```
FF
Here's a new page.
```

Press RETURN often enough for the form-feed symbol (FF) to scroll off the top of the screen. Then, type the following text and press RETURN:

```
On Thursday, the lilacs bloomed.  RET
```

The cursor rests at the beginning of the line below the text.

Finally, consolidate the space at the end of the file by pressing the keypad key DELLINE (PF4) until the terminal beeps. Press HELP (PF2). KED displays a message indicating you have deleted all the empty space beyond the last text line and then attempted to delete a nonexistent line. (Doing this consolidation is required here only to ensure your example text file ends at a known place, and is not normally performed.)

You can exit from a file with the cursor at any position in the file. Now, exit from KED and preserve the file WEEK.TXT on volume DK.

Press the keypad sequence GOLD COMMAND (PF1 7) and issue the EXIT command in response to the command prompt:

```
GOLD  COMMAND
Command:EXIT  ENTER
.
```

Verify that the file WEEK.TXT exists on your default data (DK) volume:

```
.DIRECTORY WEEK.TXT  RET
```

4.6 Inspecting an Existing Text File

You open a text file for inspection by issuing the EDIT command with the /INSPECT option for the specified file. Open the file WEEK.TXT for inspection:

```
.EDIT/INSPECT WEEK.TXT  RET
```

KED displays:

```
Monday was cold and rainy.
On Tuesday, it snowed.
Wednesday was sunny and warm.
```

Because WEEK.TXT is opened for inspection only, any attempt to alter the file causes the terminal to beep. If KED is unable to perform any operation you request, your terminal signals with a beep. If you are unsure what caused the terminal to beep, you can press HELP (PF2), and KED displays a short message indicating the reason.

Although WEEK.TXT is open for inspection and cannot be edited, you can use the keypad keys to move the cursor and thereby inspect the file. The keypad contains two keys that govern the direction of cursor movement, ADVANCE (4) and BACKUP (5). Initially, the cursor movement is ADVANCE, which continues to apply until explicitly changed.

You can move the cursor by character, word, line, section, page, or length of file. Perform the following keystrokes to move the cursor through the file. Although single keystrokes are used, repeating the keystroke repeats the cursor movement. The cursor is currently resting at the beginning of the file:

```
Monday was cold and rainy.
On Tuesday, it snowed.
Wednesday was sunny and warm.
```

Move the cursor one character position by pressing CHAR (3):

```
Monday was cold and rainy.
On Tuesday, it snowed.
Wednesday was sunny and warm.
```

Move the cursor one word position by pressing WORD (1):

```
Monday was cold and rainy.
On Tuesday, it snowed.
Wednesday was sunny and warm.
```

Move the cursor one line position by either pressing BLINE (0) to advance to the beginning of the next line

```
Monday was cold and rainy.
On Tuesday, it snowed.
Wednesday was sunny and warm.
```

or by pressing EOL (2) to move the cursor to the end of the line:

```
Monday was cold and rainy.
On Tuesday, it snowed.
Wednesday was sunny and warm.
```

Move the cursor to the next form feed by pressing PAGE (7):

```
FF
Here's a new page.
```

Move the cursor to the end-of-file marker by pressing GOLD ADVANCE (PF1 4):

```
On Thursday, the lilacs bloomed.
_
```

You have been advancing from the beginning to the end of WEEK.TXT because KED initially sets cursor movement direction to ADVANCE. Now press keypad key BACKUP (5) to redirect cursor movement toward the beginning of the file. You only need to press BACKUP once; cursor movement direction remains in effect until changed.

Press SECTION (8) to demonstrate moving the cursor one section (16 lines).

Finally, perform a search operation by pressing GOLD FIND (PF1 PF3). Respond to the prompt as follows, where BACKUP is 5:

```
Model: monday BACKUP
```

KED searches back through WEEK.TXT for the first occurrence of the target string *monday* and places the cursor at the beginning of the target string:

```
Monday was cold and rainy.
```

If KED is unable to find the target string, the terminal beeps.

When you have finished inspecting WEEK.TXT, you can exit from KED. There is no output file to close because this is a file inspection. Issue the commands:

```
 GOLD  COMMAND
Command: EXIT ENTER
.
```

4.7 Editing an Existing Text File

You open an existing text file for editing by issuing the EDIT command and specifying the file. The file must not be protected. Open the file WEEK.TXT for editing:

```
.EDIT WEEK.TXT  RET
```

Your terminal screen clears and you see a blinking cursor at the beginning of the text file:

```
Monday was cold and rainy.
On Tuesday, it snowed.
Wednesday was sunny and warm.
```

You can press HELP to display the on-line help screens and press ENTER to return to the text file.

In Section 4.6, you learned to move the cursor through the file. In this section, you will learn to manipulate text by using the keypad function keys. Text manipulation operations involve moving the cursor to a portion of text and then manipulating the existing text and/or adding new text. Like cursor movement operations, text manipulation also involves units of characters, words, lines, sections, and pages. The search function is helpful for placing the cursor at known points (targets) where you want to edit a file or for determining if a target string exists in a file.

As in previous sections, the legend on the keypad keys is included parenthetically with the function name in the following exercises.

The first part of the exercise involves searching for the following line, editing the line, and then adding the line to the first section of text. This is the line:

```
On Thursday, the lilacs bloomed.
```

Do this exercise:

1. Perform a search operation for the shortest unambiguous portion of the target string. You do not have to search for the first part of the string, but rather get to the general area. Therefore, a good target to search for would be the string *thur*, as follows:

   ```
   GOLD  FIND   (or)  PF1  PF3
   Model: thur  ADVANCE   (or) 4
   ```

   Those commands produce the following display:

   ```
   On Thursday, the lilacs bloomed.
   ```

2. Assume you want to change the word *lilacs* to *roses*. Do the following:

   a. Press WORD (1) two times, producing:

      ```
      On Thursday, the lilacs bloomed.
      ```

   b. Press DELWORD (-). The word *lilacs* is deleted.

      To demonstrate the restoration function, restore the deleted word. Press GOLD UNDELWORD (PF1 -) and see that *lilacs* returns.

Restoring deleted text elements is a general procedure. Text elements such as characters, words, and lines, are deleted by pressing a function key. The text element can be restored by subsequently pressing GOLD and the function key that deleted the element. This is because a deleted text element is stored in a buffer, and the most recently stored text element can be restored from the buffer.

Again press DELWORD and delete *lilacs*. Now, type *roses* and press the space bar, producing the following line:

```
On Thursday, the roses bloomed.
```

3. Press BACKUP BLINE (5 0) to move the cursor to the beginning of the text line.
4. The following functions *cut* the line out of the file and place the line in the *paste buffer*:

   SELECT EOL CUT (or) . 2 6
5. The following functions move the cursor to the beginning of the file:

   GOLD BACKUP (or) PF1 5

   ```
   Monday was cold and rainy.
   ```
6. Assuming that *Thursday*'s line should follow *Wednesday*, press FORWARD (4) to redirect cursor movement, then press BLINE (0) three times. The cursor now rests on the first open line.
7. The procedure now *pastes* the line we cut out and placed in the paste buffer on the open line:

   GOLD PASTE (or) PF1 6

The first four lines of text in the file now read:

```
Monday was cold and rainy.
On Tuesday, it snowed.
Wednesday was sunny and warm.
On Thursday, the roses bloomed._
```

The next part of this exercise replaces the line following the form feed, as follows:

1. Press ADVANCE PAGE (4 7), which displays the following text:

   ```
   FF
   Here's a new page.
   ```
2. Delete the line by pressing DELLINE (PF4).
3. Type in the following line:

   ```
   Welcome to new england.
   ```
4. Recognizing that *new england* should start with upper case letters, correct that error, using the following keypad functions and typing the letters N and E:

```
BACKUP WORD WORD  (or) 5 1 1

CTRL/U ADVANCE WORD CTRL/U  (or)
CTRL/U 4 1 CTRL/U
```

The line now reads:

```
Welcome to New England.
```

Exit from KED and preserve the output file:

```
GOLD COMMAND  (or) PF1 7
Command: EXIT ENTER
.
```

Two versions of the text file now exist on your default data (DK) volume, WEEK.TXT and WEEK.BAK. Verify that by issuing the following command:

```
.DIRECTORY WEEK.* RET
```

The file WEEK.TXT is the result of your most recent editing. The file WEEK.BAK is the previous version.

If your system has a printer, print WEEK.TXT:

```
.PRINT WEEK.TXT RET
```

The first page of the file is the poetry. The second page of the file, caused by the form feed you entered when you created WEEK.TXT, is a statement you could have deduced from the poetry.

4.8 Other Useful Features

KED provides many useful features, as described in the *PDP–11 Keypad Editor User's Guide*. The following brief list describes those features that may be helpful when performing exercises in other chapters in this manual. A more complete description of these features is found in *PDP–11 Keypad Editor User's Guide*.

Line Wrap

KED provides a SET WRAP command that forces a RETURN when text will exceed a specified right margin value. Using SET WRAP lets you type text without being concerned that the text will extend beyond the visible right margin—extend off your terminal screen.

You enable the SET WRAP feature as follows:

1. While running KED, press GOLD COMMAND (PF1 7).
2. Determine which right margin value you want to specify. You cannot specify a value higher than 78.
3. In response to the command prompt, issue the SET WRAP command for the value you want, such as 70:

   ```
   Command: SET WRAP 70 ENTER
   ```

If you want to subsequently disable the line wrap feature, press GOLD COMMAND (PF1 7) and issue:

```
Command: SET NOWRAP ENTER
```

Screen Width

You can determine how many columns are displayed on your terminal screen. By default, KED displays 80 columns. You can issue a KED command that changes the display to 132 columns. This feature is especially useful when inspecting program listings, which typically extend beyond column 80.

To change the screen width from 80 to 132 columns:

1. While running KED, press GOLD COMMAND (PF1 7).
2. In response to the command prompt, issue the following command:

   ```
   Command: SET SCREEN 132 ENTER
   ```

3. When you want to return to the 80-column display (the 132-column display is difficult to read), press GOLD COMMAND (PF1 7) and issue the following:

   ```
   Command: SET SCREEN 80 ENTER
   ```

Keeping a Journal File

You can protect the file you are creating or editing by keeping a journal file. KED maintains a continuously updated copy of changes and additions you make to a file, so if you experience an unplanned power-down or other system malfunction, you can recover your file.

You keep a journal file by coupling the /JOURNAL option with the EDIT command (and other options). For example, assuming the example file WEEK.TXT, you create WEEK.TXT and keep a journal file with the following command:

```
.EDIT/CREATE/JOURNAL WEEK.TXT RET
```

You edit WEEK.TXT and keep a journal file with:

```
.EDIT/JOURNAL WEEK.TXT RET
```

If an unplanned power-down or other system malfunction occurs, you recover the created or edited WEEK.TXT by using the /RECOVER option:

```
.EDIT/RECOVER WEEK.TXT RET
```

Other Features Described in This Manual

KED is a powerful editor. The *PDP–11 Keypad Editor User's Guide* describes many KED features that you will find useful. Also, in Part II of this manual, you will find information on:

- How to run the mapped monitor version of KED (KEX) as a system job.
- How to run multiple copies of KEX at the same time and instantly go from one file to another.
- How to easily move text from one file to another.

4.9 Chapter Summary

The following basic KED editing functions and commands were described in this chapter:

- Obtaining the on-line help screens:

  Press PF2

- Creating text files:

  EDIT/CREATE filespec

- Inspecting text files:

  EDIT/INSPECT filespec

- Edit existing text files:

  EDIT filespec

- Obtaining journal file protection:

  EDIT/CREATE/JOURNAL filespec

  or:

  EDIT/JOURNAL filespec

- Performing the journal file recovery operation:

  EDIT/RECOVER filespec

Part II
Using the RT–11 Monitors

Part II describes how to use the RT–11 monitors. It is divided into four chapters.

- Chapter 5, Using the Foreground/Background Monitor, describes the unmapped FB monitor. Chapter 5 also describes in some detail the foreground and background environments that are used with FB and the mapped monitors.
- Chapter 6, Introduction to the Mapped Monitors, describes the hardware requirements for running mapped monitors and how to determine if your processor meets those requirements. It also describes the advantages of mapped monitors and provides a short discussion of the three program environments available to you when you use mapped monitors.
- Chapter 7, Using the System Job Feature, describes and demonstrates the powerful system job feature. Of the distributed RT–11 monitors, only XM and ZM let you use the system job feature. Chapter 7 presents complete information and exercises for running, stopping, and unloading system jobs. It describes moving between the background environment and system job environment and from system job to system job.
- Chapter 8, Using the Start-Up Command File to Configure Your System, describes how you can use a start-up command file to configure your system. Information is presented to help you set up devices in your system to perform in the manner you want. The chapter illustrates the start-up command files for mapped monitors, but the discussion is useful for setting-up start-up command files for unmapped monitors as well.

Chapter 5

Using the Foreground/Background Monitor

If your processor can support a mapped monitor, you should investigate using it rather than the unmapped FB monitor, as mapped monitors are capable of utilizing the full power of your processor. See Section 6.1 to determine if your processor can support mapped monitors.

If the test you performed in Section 6.1 indicates you cannot use a mapped monitor, you can use the FB monitor and run two programs concurrently.

Use the on-line index utility, INDEX, and see the on-line help utility, HELP, for more information on topics described in this chapter.

5.1 The Foreground/Background Environment

The foreground/background environment is designed so that two programs can run concurrently. One of these programs you designate as the foreground program. The system gives priority to the foreground program (or job, as it is usually called) and lets it run until some condition, perhaps waiting for an I/O completion, causes it to relinquish control to the other program (the background job). The system then lets the background job run until the foreground job again requires control, and so on. In this way, the two programs share system resources. Whenever the foreground program is idle, the background program can run. Yet whenever the foreground program requires service, its requests are immediately satisfied. To the user at the terminal, the two programs appear to run simultaneously.

Foreground priority programs are generally time critical. For example, you may want to designate as the foreground job a program that collects and analyzes data. Background programs are usually non-time critical. Because the keyboard monitor and many RT–11 utilities are designed to run as the background job, you can continue to do program development by using monitor commands to run the editor, the FORTRAN compiler, the linker, and so forth.

To perform the following exercises your system must have a clock. Verify whether your system has a clock by entering the TIME command twice. If the time displayed changes, your system has a clock. If your system does not have a clock you should go to Part III.

5.2 Running the Foreground/Background Programs

Two programs are provided for you to run a foreground/background demonstration. The background job is called DEMOBG, and the foreground job DEMOFG. The function of the foreground job is to send messages every two seconds to the background job, telling it to cause the terminal to beep. The background job recognizes these messages and beeps once for each message sent by the foreground job.

Although the foreground job is always active, sending messages to the background job every two seconds, other programs besides DEMOBG can be executed in the background. Only when DEMOBG is active, however, is the circuit complete so that messages can be successfully received and honored. During the periods when DEMOBG is not running, the foreground program enters the messages in the monitor message queue. Once you restart DEMOBG in the background, the system immediately dequeues all the messages since the last exit of DEMOBG, resulting in many successive beeps. When the queue is empty, the normal send/receive cycle resumes, and the terminal beeps every two seconds as each current message is sent and honored.

5.2.1 Creating the Background Job

The background program DEMOBG.MAC is an assembly language source file and must be assembled and linked before you can use it. When you execute DEMOBG, it displays a message on the terminal. It is assumed that you have set the date.

5.2.2 Editing the Background Job

Use KED to modify the background job, DEMOBG.MAC. One of the lines of the message that is output by the program has a semicolon character preceding it, which makes the line a comment field. This will prevent the line from being printed as part of the message. Thus, the semicolon must be deleted from that line.

Change the line

; .ASCII /WELL DONE./

to

.ASCII /WELL DONE./

5.2.3 Running the Background Job

The demonstration in Chapter 19 illustrates assembling and linking MACRO programs. If you are not familiar with assembly/link operations, type the command lines as shown.

Assemble the background job.

```
.MACRO DEMOBG/LIST RET
```

Link the .OBJ file produced by the assembler to create a runnable job.

```
.LINK DEMOBG RET
```

Now run the background job and check the results.

```
.RUN DEMOBG RET
RT-11 DEMONSTRATION PROGRAM
IF INCORRECTLY EDITED, THIS IS THE LAST LINE.
WELL DONE.
```

If you did not delete the semicolon character, the last line will not be output. Return to the monitor by typing two successive CTRL/Cs.

$\boxed{\text{CTRL/C}}$
$\boxed{\text{CTRL/C}}$

```
^C
^C
.
```

5.3 Using the FB Monitor

The FB monitor provides you with commands that let you control the two-job environment. They let you interact with the two jobs and let the two jobs interact with one another.

5.3.1 Communication in a Two-Job Environment

When two jobs run simultaneously, you must have some means of indicating the job to which you are directing commands. Likewise, the two jobs must have the means to identify themselves when they have messages to print. The following conventions apply to system communication in a two-job environment.

1. The foreground job has priority. If both the foreground and the background job are ready to print output at the same time, the foreground job prints first. The foreground job prints a complete line, then the background job prints a complete line, and so on.
2. Either job can interrupt your input at the terminal if it has a message to print.
3. Messages printed by the background job are preceded by the characters B>.
4. Messages printed by the foreground job are preceded by the characters F>.
5. Typed commands are initially directed to the background job. You can redirect control alternately to the foreground and background jobs by using the CTRL/F and CTRL/B commands.

   To direct typed input to the foreground job, press $\boxed{\text{CTRL/F}}$. This command instructs the monitor that all subsequent terminal input—commands and text—is directed to the foreground job. Typing this command causes the system to print an F> on the terminal, unless output is already coming from the foreground job. Command input remains directed to the foreground job until the foreground job terminates, or until it is redirected to the background job through CTRL/B.

   To direct typed input to the background job, press $\boxed{\text{CTRL/B}}$. This command instructs the monitor that all subsequent terminal input—commands and text—is directed to the background job. Typing this command causes the system to print a B> on the terminal, unless output is already coming from the background job. Command input remains directed to the background job until redirected to the foreground job through CTRL/F.

5.3.2 Creating the Foreground Job

The foreground program DEMOFG is an assembly language source file; it must be assembled and linked before you can use it.

```
.MACRO DEMOFG/LIST  RET
```

The output resulting from this MACRO command includes an object file called DEMOFG.OBJ and a listing file called DEMOFG.LST. The command creates both files on your default data (DK) volume. You must link the .OBJ file to produce a runnable foreground program. Chapter 20 describes linking operations. If you are not familiar with linking operations, issue the following command as shown. Note that you use the LINK command here with the /FOREGROUND option.[1] This option produces a load module with a .REL file type which signifies to the system that the file can be run as the foreground program; the priority job.

```
.LINK/FOREGROUND DEMOFG  RET
```

5.3.3 Executing the Foreground and Background Jobs

Now, you are ready to operate the two-job environment. Many times, you have to consider the devices that are used for output in a foreground/background environment. For example, if your program assumes that the output device is a printer and you want to output to another device, use the ASSIGN command, as described in Section 2.8.5.

You do not have to consider that information for the demonstration programs that are provided, since the foreground job communicates with the background job, and both jobs send their output to the terminal.

When you use the FB monitor, you must always load into memory the peripheral device handlers needed by the foreground job. You can determine if a device handler is already loaded by issuing the SHOW command. You use the LOAD command to load a device handler in memory. For example, if your foreground job requires the use of the printer, you must load the printer (LP or LS) device handler. You should also specify the job name with the command. For a foreground job, the job name is F; for a background job, the job name is B. You can display numerous examples of device handler loading and job name assignments by typing or printing the FB monitor start-up command file, STRTFB.COM.

Use the FRUN command to run a foreground job, which is similar to RUN except that the system automatically loads and starts the execution of the foreground .REL program. (To execute a FORTRAN foreground job, you should use the /BUFFER:n option with the FRUN command. The argument *n* represents, in octal, the number of words of memory to allocate.)

[1] This command option also applies to compiled FORTRAN programs that are to be linked as a foreground job.

Use this command to start the execution of DEMOFG.REL.

```
.FRUN DEMOFG RET

F>
FOREGROUND DEMONSTRATION PROGRAM
SENDS A MESSAGE TO THE BACKGROUND PROGRAM "DEMOBG"
EVERY 2 SECONDS, TELLING IT TO BEEP.

B>
```

The foreground program DEMOFG is now running and queuing the message for the background program every two seconds. You now execute the background program DEMOBG to let it receive the messages that were queued and to beep the terminal.

```
.RUN DEMOBG  RET
RT-11 DEMONSTRATION PROGRAM
IF INCORRECTLY EDITED, THIS IS THE LAST LINE.
WELL DONE.
```

The terminal beeps several times in rapid succession as the monitor dequeues the messages, and then every two seconds as the foreground job sends its message to the background job.

You can run other jobs in the background. First, terminate the background job DEMOBG by pressing CTRL/C twice:

```
CTRL/C
CTRL/C

.
```

Execute a DIRECTORY command in the background to get a listing of all the .OBJ files on the default data (DK) volume by typing:

```
.DIRECTORY *.OBJ RET
```

The foreground job is still running and queuing its messages to the monitor. Rerun the background program to collect all the foreground messages while the background job was stopped and the directory was printing.

```
.RUN DEMOBG  RET
RT-11 DEMONSTRATION PROGRAM
IF INCORRECTLY EDITED, THIS IS THE LAST LINE.
WELL DONE.
```

The terminal again beeps several times in succession and then beeps once every two seconds. Stop the background job by pressing CTRL/C twice:

```
CTRL/C
CTRL/C

.
```

Now, stop the foreground job and remove it from memory. To do this, you must first press CTRL/F to direct terminal input to the foreground:

```
. CTRL/F
F>
```

The system prints the characters F> to remind you that you are now directing command input to the foreground job. Press CTRL/C twice to interrupt and terminate the execution of the foreground job and return control to the background job:

```
CTRL/C
CTRL/C
B>
```

You should unload the foreground job to reclaim memory space for background use. Use the UNLOAD command as follows:

```
UNLOAD F  RET

.
```

F represents the foreground job; you should use this code whenever you want to unload the foreground job. To unload any loaded device handlers, you must use their 2-letter device mnemonics.

The foreground program has access to all the system features available to a background program—opening and closing files, reading and writing data, and so on. However, before you begin to write and use programs in the foreground, read the *RT–11 Software Support Manual* for coding restrictions.

5.4 Useful Information in Part II

System jobs, as described in Chapter 7, are a variant of foreground jobs. System jobs can also be run as the foreground job, as shown in the FB monitor start-up command file, STRTFB.COM. The main difference is you can run multiple system jobs at the same time while you can run only one foreground job. Therefore, even if your processor cannot support a mapped monitor, you can learn a lot about using the foreground environment by:

1. Printing STRTFB.COM.
2. Examining Sections 7.1 through 7.2.2, Section 7.4, and Section 7.5.

   Remember that those sections describe running multiple system jobs and you are interested in the information in terms of running a single foreground job. Also, remember that you can run the SL command line editor and still run a foreground and background job; SL behaves like a handler and not like a job.
3. Examine the information in Chapter 8 while looking at your listing of STRTFB.COM. Again, much of the information in Chapter 8 is useful. The requirements for running a system job are much like those for running the same job as the foreground job.

5.5 Chapter Summary

The following is a summary of control commands used in a foreground/background environment.

CTRL/B

Directs all keyboard input to the background job (until CTRL/F).

CTRL/F

Directs all keyboard input to the foreground job (until CTRL/B).

FRUN

Loads and starts execution of the foreground job.

LOAD *dd*

Brings the indicated device handler into memory; the handler becomes resident in memory.

UNLOAD *dd*

Takes the indicated device handler out of memory, reclaiming its memory space; the handler becomes nonresident in memory.

UNLOAD *F*

Reclaims the memory space used by the foreground job.

Chapter 6

Introduction to the Mapped Monitors

This chapter contains the following information:

- Requirements for using the single-mapped and fully-mapped monitors
- Advantages of using a mapped monitor
- Determining if you are currently using a mapped monitor and, if not, how to make it your monitor
- Program environments available under the mapped monitors

Use the on-line index utility, INDEX, and see the on-line help utility, HELP, for more information on the topics discussed in this chapter.

6.1 Can You Use a Mapped Monitor?

RT–11 distributes two mapped monitors: XM and ZM. The XM monitor is single-mapped and the ZM monitor is fully-mapped. Whether you can use either monitor is initially determined by your hardware and subsequently by your application.

The hardware requirements are described in the following sections. Assuming your hardware supports a mapped monitor, which mapped monitor you use is determined by your application. If your application supports separated Instruction and Data address spaces and/or Supervisor Mode, you should look at ZM; otherwise, consider XM.

6.1.1 Can You Use XM?

Your computer must have more than 56K bytes of memory and a memory management unit (MMU) for you to use the XM monitor. RT–11 provides a command you can use to determine your computer characteristics:

```
SHOW CONFIGURATION
```

That command displays a lot of information; the information in which you are now interested is in the third section of the display. That section contains the name and ID of your processor (computer), the amount of memory contained in your processor, whether your processor contains a memory management unit, and other information.

Issue the following command and let the display scroll on your screen until it stops:

```
.SHOW CONFIGURATION RET
```

To use the XM monitor, the line displayed below your processor type must show that you have more than 56K bytes (for example, 512KB) of memory, and a line further down must show that you have a memory management unit.

If the processor hardware does not support the XM monitor, read Chapter 5 for information on using the FB monitor.

6.1.2 Can You Use the ZM Monitor?

The ZM monitor has the same memory requirements as XM. The ZM monitor also requires hardware support for separated I & D address space and Supervisor Mode. Assuming your hardware meets the memory requirements in Section 6.1.1, perform the following procedure to determine if it supports ZM.

1. First, if your processor uses the J–11 CPU chipset, then it supports the ZM monitor and you need not go further.
2. If your processor does not use the J–11 CPU chipset (or you are not sure), assuming the distributed program, CONFIG.SAV, resides on device *ddn*, issue the following command:

   ```
   .RUN ddn:CONFIG /D/V:040000/M:040000/R:466
   ```

   If that command returns no error message, your processor supports ZM. If that command returns the error message, *?CONFIG-E-Values do not match*, your processor does not support ZM.

6.2 Should You Use a Mapped Monitor?

You should use a mapped monitor for the following reasons:

- The mapped monitors let you take advantage of all the memory in your computer.

  The memory in your computer is divided into *low memory* and *extended memory*. If you recall the amount of memory in your computer (determined in the previous section) and subtract 56K bytes from that amount, you can see how many thousands of bytes of extended memory are available with the mapped monitors.
- Many of the programs included with your RT–11 distribution are designed to use the mapped monitors; they take advantage of your computer's extended memory feature.

  Many RT–11 programs are designed to reside mainly in your larger extended memory, with as little of the program as possible residing in your limited low memory. Without the mapped monitors, all the memory-resident portion of each program must reside in low memory.
- The mapped monitors let you run more than two programs at the same time with the system job feature.

  As RT–11 is distributed, only the mapped monitors give you the system job feature. The system job feature lets you run up to six system jobs at the same time, along with a foreground and background job.
- The fully-mapped monitors take advantage of all PDP–11 hardware features, including separated Instruction and Data address space and Supervisor Mode.

6.3 Are You Now Running a Mapped Monitor?

If you are not sure which monitor you are using now, issue the following command. Be ready to press the NO SCROLL or HOLD SCREEN key because the information may scroll quickly off your screen. The first word of the first line in the display will show you which monitor you are now using. If you are now running the XM monitor, that word is RT–11XM. If you are running the ZM monitor, that word is RT–11ZM.

```
.SHOW CONFIGURATION RET
```

When you know which monitor you are using, press NO SCROLL or HOLD SCREEN again to let the rest of the display scroll by. If that display shows RT–11XM or RT–11ZM as the first word, you can proceed to the next section. If that display does not show a mapped monitor, perform the following procedure:

1. If you are currently editing any files (if you have any files open on an editor), close them. If you are running any application in the foreground or system environment, exit from the application.
2. If you want to run the XM monitor, issue the following command to copy the XM monitor bootstrap to your system device, making XM the default monitor:

   ```
   .COPY/BOOT SY:RT11XM.SYS SY: RET
   ```

   If you want to run the ZM monitor, issue the following command to copy the ZM monitor bootstrap to your system device, making ZM the default monitor:

   ```
   .COPY/BOOT SY:RT11ZM.SYS SY: RET
   ```

3. Then, boot the XM or ZM monitor on your system device:

   ```
   .BOOT SY: RET
   ```

4. You have now soft-booted your computer and are running the XM or ZM monitor from your system (SY) device.

If you now repeat the command, SHOW CONFIGURATION, you should see the monitor you chose as the first word in the display.

6.4 Running Programs Under the Mapped Monitors

Mapped monitors let you run programs in the background environment, foreground environment, or system job environment. Proper use of those three environments lets you run up to eight jobs at the same time. You are limited only by the size of the programs and the amount of memory in your computer.

The following sections describe the three environments.

6.4.1 Background Environment

You direct operations to the background environment in response to the monitor prompt (.). In the previous section, you issued the command, SHOW CONFIGURATION. You issued that command to the background.

It is helpful to keep access open to the background environment while you are using your computer. While a program is running in the background, you are not able to issue a command to perform another operation or run another program.

Chapter 5 further explains and demonstrates using the background environment under the FB monitor. The background environment under the XM monitor is similar to the background environment under the FB monitor.

6.4.2 Foreground Environment

The foreground environment is generally used to run a time-critical program. The program running in the foreground has priority over all other jobs in the system. Chapter 5 further explains and demonstrates using the foreground environment under the FB monitor. The foreground environment under the mapped monitors is similar to the foreground environment under the FB monitor.

6.4.3 System Job Environment

Of the distributed RT–11 monitors, only XM and ZM let you use the system job environment to run programs. You generally run programs as system jobs to keep the background and foreground free for entering commands and running other jobs. You can run up to six system jobs at the same time and still have access to the background and foreground environments.

RT–11 is distributed with the following programs that can be run as system jobs:

- ERROR LOGGER

  You must perform the system generation (SYSGEN) procedure before you can run the error logger. Therefore, the error logger is not discussed. You can get complete information on the error logger in the *RT–11 System Utilities Manual*.

- INDEXX

  INDEXX is the mapped monitor version of the on-line INDEX utility. Running INDEXX as a system job gives you constant access to the INDEX utility.

- KEX

  KEX is the mapped monitor version of the KED editor and is identical in functionality to KED. You can run up to six KEX editors as system jobs at the same time. Running KEX as a system job can provide constant access to the monitor prompt.

  The following two chapters describe how to run KEX as a system job.

- SPOOL

  SPOOL is the transparent spooling package that manages printer operations for you. When using SPOOL, you direct output to your printer and are immediately returned to the monitor prompt.

  The following two chapters describe how to run SPOOL as a system job.

- VTCOM

  VTCOM is the virtual terminal communications package that lets you maintain communications and transfer files between a host and your own computer. While running VTCOM, you can perform operations on the host computer and instantly return to the RT–11 background, foreground, or system job environment.

  The following two chapters describe how to run VTCOM as a system job.

When you run KEX, SPOOL (or QUEUE), and VTCOM as system jobs, you can edit files, print files, and maintain communications with another computer — all at the same time. You also continue to have access to the background environment to issue commands and run other programs in the background, foreground, or system job environment.

Some application programs (programs you write or buy) can also be run in the system job environment.

Chapter 7, Using the System Job Feature, describes the system job environment in detail.

Chapter 7
Using the System Job Feature

The following sections describe how to use the system job feature. It is assumed you have already read Chapter 6 and are now running under the XM or ZM monitor.

This chapter contains the following information:

- Points you should consider before starting any system jobs
- Three ways you can start a system job
- Calling a system job once you have started it
- Stopping (aborting) a system job once you no longer need it
- Unloading a system job to regain the memory it used
- Creating indirect command files to abort and unload system jobs

Each section contains exercises to familiarize you with the system job environment.

You will find it helpful to have already read and worked through the exercises in Chapter 4.

Use the on-line index utility, INDEX, and see the on-line help utility, HELP, for more information on the topics described in this chapter.

7.1 Before Starting System Jobs

You should consider the following points before you start a system job:

- All device handlers used by a system job must reside on your system device and be loaded. The system device handler is always resident and loaded, so it is of no concern here.

  Think for a moment about the devices used by a system job, such as SPOOL. SPOOL manages files that you direct to your printer. Therefore, the device handlers you are concerned with are those for the device containing the file and the device handler for your printer. If you are unsure about the status of the device handlers, examine the directory of your system device by using the following command:

  ```
  .DIRECTORY SY:%%X.SYS  RET
  ```

  The percent character (%) in that command is a single-character wildcard. All handlers you use under mapped monitors must have the X suffix. If a handler that is used by a system job does not reside on your system device, you must copy it to there.

Use the SHOW command to determine the status of the device handlers. Issue the command:

```
.SHOW  RET
```

The SHOW command displays all the device handlers installed on your computer system and their status. Handlers are displayed in their general form; the X suffix is not displayed. (Any handler displayed as being Resident is required by RT–11, is loaded, and cannot be unloaded.) If the SHOW command does not display a handler you require, install that handler. Use the following command where *dd* represents the 2-character general form of the handler you want to install:

```
.INSTALL dd  RET
```

If the SHOW command displays the handler you require but does not indicate it is loaded, you must load that handler. Use the following command where *dd* represents the 2-character general form of the handler you want to load:

```
.LOAD dd  RET
```

You can load a handler from an indirect command file, your start-up command file, or directly at the monitor prompt as shown here.

- System jobs, foreground jobs, and jobs that run as pseudohandlers like SL, are loaded into memory in the order that you start them.

  This is an important consideration because you will occasionally stop (abort) and unload jobs when you need the memory allocated to a job.

  After you boot your computer, each job you run as a foreground or system job is loaded at the highest place possible in low memory. To keep from fragmenting the memory in your computer, you unload jobs from the lowest point in memory and proceed upward. Therefore, you should plan to run jobs first that you will rarely want to unload. You should be more willing to unload each successive job you load.

  For example, if you are going to use the single-line (SL) command editor, you will normally always use SL and therefore not want to unload it. Therefore, SL is a good candidate for loading first.

7.2 Starting System Jobs

The method you use to start a system job should be determined by how you are going to use that system job and the complexity of the startup procedure. You can start a system job three ways:

- With the start-up command file for the monitor you are using

  If you are going to use a system job whenever you first boot your computer, you should probably start that job from your start-up command file. If you are going to normally keep that job loaded in memory, you should run that job early in the start-up command file.

- By running an indirect command file that starts the job

  If you are going to use a system job periodically and the startup procedure for that system job requires more than a single command, you should probably start that job by running an indirect command file.

- Directly at the keyboard using the SRUN command

  System jobs that require only a single command line to start should probably be started directly from the keyboard.

The following three sections describe starting system jobs.

7.2.1 Starting System Jobs from Start-Up Command Files

RT–11 is distributed with default start-up command files for each monitor. For example, when you boot the XM monitor, that monitor looks for the start-up command file STRTXM.COM. (The ZM monitor looks for STRTZM.COM.) The start-up command file must reside on your system (SY) device.

You can edit your start-up command file to start system jobs when you first boot your computer. You can also create your own start-up command file, but you must retain the same file name. Digital recommends that you edit the distributed start-up command file rather than create your own, especially if you are a new user of the RT–11 operating system.

The distributed start-up command files for XM and ZM contain most of the commands necessary to run two system jobs, VTCOM and SPOOL. If you want to maintain continuous communication with a host processor when you first boot your RT–11 system, you probably want to start VTCOM from your start-up command file. If you typically begin printing files shortly after you boot your RT–11 system, you probably want to start SPOOL from your start-up command file.

Because RT–11 assigns priority to system jobs in the order in which they are started (unless you specify otherwise), you should start a time-critical system job such as VTCOM, before a non-time-critical job such as SPOOL or KEX. Because of this, VTCOM is positioned in the distributed start-up command files as the first system job.

Chapter 8 contains information on editing your start-up command file.

7.2.2 Starting System Jobs from an Indirect Command File

You can create an indirect command file that runs a particular system job. Use this method to start a system job when:

- You do not require that job when you first boot your computer
- The start procedure is more than a single command line.

Such a system job may be SPOOL. You can start SPOOL from your start-up command file; the command lines to do that are included in the file. However, if you only need the SPOOL utility periodically, you can start it from an indirect command file. By only loading SPOOL in memory when needed, you preserve more computer memory when you are not printing files or using SPOOL for some other purpose.

The following procedure creates an indirect command file you can use to start the SPOOL utility. The procedure assumes you are familiar with basic KED operations as described in Chapter 4. It also assumes you are going to use the SPOOL utility to send files to a serial printer that uses the LS handler.

1. Issue the following command to open your start-up command file (STRTXM.COM or STRTZM.COM) for inspection:

   ```
   .EDIT/INSPECT SY:STRTXM.COM RET
   or
   .EDIT/INSPECT SY:STRTZM.COM RET
   ```

2. Press the PF1 PF3 keypad sequence and search for the string *spool*. The result of that search places the cursor at the following text:

   ```
   !
   ! Start the spooler on a Pro 300  or  a  PDP-11 with a serial printer.
   ! On PDP-11 systems,  the LS handler must have its CSR and vector cor-
   ! rectly selected using the  SET LS CSR=nnnnnn  and  SET LS VECTOR=nnn
   ! commands. LS is distributed with its CSR and vector preset for a Pro
   ! 300 computer. If you are a Professional 350 user and have a printer,
   ! the following commands are normally selected.
   !
   !SRUN SPOOL.SAV/PAUSE
   !LOAD LS=SPOOL
   !RESUME SPOOL
   !ASS SP0 LP
   !ASS SP0 LP0
   !ASS SP0 LS
   !ASS SP0 LS0
   ```

3. Press the PF1 7 keypad sequence.

   ```
   Command:OPEN OUTPUT SPOOL.COM ENTER
   ```

4. Scroll down through the file until your cursor rests on the exclamation point (!) before the command SRUN SPOOL.SAV/PAUSE.

5. Create a select range by pressing the . keypad key. Then repeatedly press the 0 keypad key until the cursor rests on the exclamation point (!) one line below the command

   ```
   !ASS SP0 LS0
   ```

6. Press the PF1 7 keypad sequence.

   ```
   Command:WRITE SELECT ENTER
   ```

7. Press the PF1 7 keypad sequence.

   ```
   Command:CLOSE ENTER
   ```

8. Press the PF1 7 keypad sequence.

   ```
   Command:QUIT ENTER
   .
   ```

9. Using KEX's auxiliary output file feature, you have created a file named SPOOL.COM on your data (DK) device and are back at the monitor prompt (.).

10. Issue the following command to open the file SPOOL.COM, using the KEX editor:

    ```
    .EDIT SPOOL.COM RET
    ```

11. Edit that file to remove the exclamation points (!), which act in the file as comment delimiters. After you edit it, your file should look like the following:

    ```
    ! SPOOL.COM, created dd-mmm-yy, edited dd-mmm-yy
    ! Starts SPOOL from indirect command file
    !
    SRUN SPOOL.SAV/PAUSE
    LOAD LS=SPOOL
    RESUME SPOOL
    ASS SP0 LP
    ASS SP0 LP0
    ASS SP0 LS
    ASS SP0 LS0
    ```

12. You have now created an indirect command file that, when called with the following command, starts the SPOOL utility:

    ```
    .$@SPOOL RET
    ```

13. This indirect command file, SPOOL.COM, does not contain all the commands to fully implement the spooler and configure your serial printer. Chapter 8 discusses other commands you might require to fully implement the spooler and configure your printer. After you have completed the exercises in Chapter 8, you can use the information from this section to create indirect files to start the SL editor, the VTCOM communications package, and SPOOL.

7.2.3 Starting System Jobs from the Command Line

Some system jobs, such as KEX, can be started directly at the command line using the SRUN command. For example, the following command starts the KEX editor as a system job. Issue the command:

```
.SRUN KEX.SAV RET
```

RT–11 notifies you that KEX is now running as a system job by displaying the KEX system job prompt (KEX>) and KEX utility command prompt (*), and then returns you to the background. You are in the background environment and can enter commands even though the monitor prompt (.) may not be displayed. You can press RETURN to display the monitor prompt.

You can see KEX loaded in memory by issuing the command:

```
.SHOW MEMORY RET
```

In the display on your screen, note that KEX requires approximately 28K words of extended memory and somewhat less than 1K words of low memory.

Naming System Jobs

You can use the /NAME:jobname option to assign a name to a system job. The name can be any alphanumeric string of 1 to 6 characters. RT–11 starts the system job and assigns it that name. Then, you always refer to the system job by the name you assigned it.

You can use the /NAME:jobname option to start more than one KEX editor and distinguish between them. Issue the following command to start a second KEX editor and assign it the name KEX1:

```
.SRUN KEX.SAV/NAME:KEX1 RET
```

Then, issue the command:

```
.SHOW MEMORY RET
```

Note that KEX1 is loaded in memory below KEX and requires the same amount of memory as KEX.

7.3 Calling the System Job

System jobs are of two types. One type of system job runs transparently once started. Such a system job is SPOOL. Once you start SPOOL you need not call it. This section discusses the other type of system job — the type you call to use.

You can connect to only one background job or foreground job. Therefore, to summon a job running in the background environment, press CTRL/B. Press CTRL/F to summon a job running in the foreground environment. System jobs are different from background and foreground jobs in that you have to call them by name. Press CTRL/X to summon the system job environment, then specify the system job by name because there can be up to six system jobs running at the same time.

From the Background

As a result of the previous section, you are now running two editors as system jobs: KEX and KEX1. The following exercise demonstrates how to call those jobs.

1. Press CTRL/X. RT–11 displays the system job environment prompt:

   ```
   Job?
   ```

2. Type KEX in response to the system job environment prompt and press RETURN:

   ```
   Job? KEX RET
   ```

3. RT–11 displays the system job prompt and positions the cursor below that prompt:

   ```
   KEX>
   ```

   Although the asterisk (*) is not displayed, the cursor resides at the KEX utility command prompt. You can optionally display the KEX utility command prompt by pressing RET.

4. You then type the name of the file you want to edit or create. For the purpose of this exercise, create a file on your default data device named MYFIL.TXT:

```
KEX>
MYFIL.TXT  RET
```

RT–11 searches your directory for that file, does not find it (you are creating it), and returns the following prompt in reverse video:

```
?KEX-W-File not found - Create it (Y,N)?
```

Type Y to create that file. The KEX editor then opens that file for you to edit. For the purpose of this exercise, press RETURN a couple of times and then type the following text:

```
This is file MYFIL.TXT created using KEX.
```

5. Press CTRL/B. The background environment prompt (B>) appears on your terminal screen; you are now in the background. Optionally, you can press RET to display the keyboard monitor prompt (.). You can now enter a command to start another system job, run a foreground job or a background job, or perform an operation such as copying or printing a file. The file MYFIL.TXT is still open on the KEX editor.

6. Do the following to return to your file MYFIL.TXT (open on KEX):

   a. Press CTRL/X to display the system job environment prompt.

   b. Specify the system job KEX. Press RETURN.

   c. Press CTRL/W to repaint your screen.

   You are back in your file MYFIL.TXT and the cursor is in the same position as you left it.

In that manner, you can move between a system job and the background environment. You move between a system job and the foreground environment in the same manner, except you use CTRL/F to summon the foreground job.

From Another System Job

You can also move from one system job to another. You started another KEX editor in the previous section and named that second KEX editor KEX1. You are now going to open a file with KEX1 and move back and forth between the two editors.

1. While still in the file MYFIL.TXT, press CTRL/X. RT–11 displays the system job environment prompt:

   ```
   Job?
   ```

2. Type the system job KEX1. Press RETURN.

   ```
   Job? KEX1  RET
   ```

3. RT–11 displays the system job prompt:

   ```
   KEX1>
   ```

4. Type the file name MYFIL1.TXT and press RETURN.

   ```
   KEX1>
   MYFIL1.TXT  RET
   ```

RT–11 searches your default data device directory for that file, does not find it (you are creating it), and returns the following prompt in reverse video:

```
?KEX-W-File not found - Create it (Y,N)?
```

Type Y to create that file. KEX1 then opens that file for you to edit. For the purpose of this exercise, press RETURN a couple of times and then type the following text:

```
This is file MYFIL1.TXT created using KEX1.

This sentence is going to be copied over to MYFIL.TXT further along
in this section.
```

5. Press CTRL/B to return to the background environment. You can then press RETURN a few times to make the display easier to read.

You now have files open on two different editors and are in the background environment. You can enter a command to start another system job, start a program in the foreground environment or the background environment, or perform other operations.

You can summon either editor by pressing CTRL/X, supplying the editor name, and pressing RETURN. Pressing CTRL/W repaints your terminal screen with the file opened on that editor. The cursor is positioned in each file where you left it — each editor is currently running, contains an opened file, and is waiting for input.

Example Interaction Between System Jobs

You can use KEX's auxiliary file feature to transfer any part of a file between two (or more) KEX editors you are running as system jobs. The following is the general procedure, followed by an exercise that illustrates the procedure. You should read the procedure and perform the exercise.

1. Open an output file from the editor that contains the part of a file you want to transfer.

   You should use a consistent file extension when you are creating an auxiliary output file for this purpose, such as .AUX.

2. Write the part of the file you want to transfer to the auxiliary output file. Then close that auxiliary output file.

3. Call the editor which has the file opened that is to receive the contents of the auxiliary file. Using your keypad, move to the point in that file where you want the auxiliary file written.

4. Open that auxiliary file as an input file. Then, include the contents of the auxiliary file in the file you are currently editing.

5. Optionally, you can then go to the background environment and delete the auxiliary file.

The following exercise demonstrates the procedure described above. You currently have files opened on two KEX editors: MYFIL1.TXT on KEX1 and MYFIL.TXT on KEX. You are going to transfer the following line from MYFIL1.TXT to MYFIL.TXT:

```
This sentence is going to be copied over to MYFIL.TXT further along in this
section.
```

Do the following:

1. Call the editor which has the file opened that contains the part you want to transfer:
   a. Press CTRL/X

      `Job? KEX1` RET

   b. Press CTRL/W to repaint your screen with the contents of MYFIL1.TXT.

2. Open the auxiliary output file:

   Press PF1 7

   `Command:OPEN OUTPUT MYFIL1.AUX` ENTER

3. Write the file part to the auxiliary file:
   a. Using the keypad, move the cursor to the beginning of the target, (`This sentence is going ...`), and press . to initiate a select range.
   b. Move the cursor to the end of the target and press PF1 7.

      `Command:WRITE SELECT` ENTER

      This exercise involves only one line. In practice, you can transfer virtually any amount of text or data in this manner. The size of your cut and paste buffer does not limit auxiliary files.

      Note that the line still remains in the file; you have not deleted it but rather copied it.

   c. Press PF1 7

      `Command:CLOSE` ENTER

4. Call the editor containing the file to which you want to write the auxiliary file:
   a. Press CTRL/X

      `Job? KEX` RET

   b. Press CTRL/W to repaint your screen with the contents of MYFIL.TXT.

5. Open the auxiliary file as an input file:

   Press PF1 7

   `Command:OPEN INPUT MYFIL1.AUX` ENTER

6. Include the auxiliary file:
   a. Using the keypad, move the cursor to where you want to write the target line.
   b. Press PF1 7

      `Command:INCLUDE REST` ENTER

You will see the contents of the auxiliary file MYFIL1.AUX written into your file.

7. Delete the auxiliary file:

   Under different circumstances, you may want to save the auxiliary file. However, as this is an exercise:

   a. Press CTRL/B

   b. Issue the following command:

   ```
   .DELETE MYFIL1.AUX  RET
   ```

In that manner, you can transfer text or data between files that are open on editors running as system jobs.

For the purpose of this exercise, you are now going to make permanent the file MYFIL.TXT. Any time you open a file on an editor, you open a temporary file. If you open a file but do not close it, no directory entry is created for that file. You are going to close the file MYFIL.TXT and make it permanent with the EXIT command. You will then display the file's directory entry, type the file, and delete the file.

1. Press CTRL/X to display the system job environment prompt.

2. Notify RT–11 that you want to summon the KEX editor by typing KEX in response to that prompt and pressing RETURN:

   ```
   Job? KEX  RET
   KEX>
   ```

3. Press CTRL/W to repaint your terminal screen. KEX displays the contents of MYFIL.TXT.

4. Press the PF1 7 keypad keys and issue the command EXIT:

   ```
   Command:EXIT  ENTER
   ```

5. KEX closes the file MYFIL.TXT and displays its utility command (*) prompt. You could at that point create another file or open an existing file for editing on KEX by typing a file name. Instead, press CTRL/B to return to the background environment.

6. Issue the following command to display the directory entry for MYFIL.TXT:

   ```
   .DIRECTORY MYFIL.TXT  RET
   ```

   RT–11 displays the directory entry for that file.

7. Issue the following command to type that file on your terminal screen:

   ```
   .TYPE MYFIL.TXT  RET
   ```

   RT–11 types that file on your terminal screen.

8. Issue the following command to delete MYFIL.TXT from your directory. You have not protected that file by issuing the PROTECT command, so you can delete that file without issuing the UNPROTECT command:

```
.DELETE MYFIL.TXT  RET
```

RT–11 deletes that file from your directory.

For the purpose of this exercise, you are now going to eliminate the temporary file MYFIL1.TXT, currently opened on KEX1. No permanent directory entry exists for MYFIL1.TXT because you have opened it on KEX1 for the first time. The QUIT command stops the editing of the file MYFIL1.TXT without closing it. Therefore, no directory entry is created for MYFIL1.TXT; it is not made permanent.

1. Press CTRL/X to display the system job environment prompt.
2. Notify RT–11 that you want to summon the KEX1 editor by typing KEX1 in response to that prompt and pressing RETURN:

   ```
   Job? KEX1  RET
   KEX1>
   ```

3. Press CTRL/W to repaint your terminal screen. KEX1 displays the contents of MYFIL1.TXT.
4. Press the PF1 7 keypad keys and issue the command QUIT:

   ```
   Command:QUIT  ENTER
   ```

5. KEX1 removes the file MYFIL1.TXT, does not save it, and displays its utility command (*) prompt. You could at that point create another file or open an existing file for editing on KEX1 by typing the file name. Instead, press CTRL/B to return to the background environment.

7.4 Aborting System Jobs

You probably do not want to abort (stop) a system job that you will require again unless you will also unload it to regain the system memory that job is using.

Because of the exercises in previous sections, you are currently running two system jobs: KEX and KEX1. Issue the following command to verify that:

```
.SHOW JOBS  RET
```

RT–11 displays data about the jobs that are currently loaded. (The RESORC utility is loaded because the command SHOW JOBS calls the RESORC utility.) Most important for this exercise, notice the State of jobs KEX and KEX1 is Run.

You can abort a system job by using either of two methods:

- You can press CTRL/C at the utility input prompt (*) for the system job you want to abort. You commonly use this method if you are in the system job environment at the system job you want to abort.
- You can issue the ABORT command at the monitor prompt (.) and specify the system job you want to abort. You commonly use this procedure when you are in the background environment.

The following exercise aborts the two system jobs you are currently running. It illustrates both methods.

CTRL/C Method

The following procedure uses the CTRL/C method to abort the system job KEX1. You are currently in the background environment, so to illustrate the procedure, you must first summon KEX1 and then abort it.

1. Press CTRL/X. In response to the system job environment prompt, type KEX1 and press RETURN:

   ```
   Job? KEX1  RET
   ```

   You are now at the KEX1 command prompt, although it is not displayed.

2. Press CTRL/C to abort KEX1.

RT–11 returns you to the background environment and displays the background environment prompt (B>). If your terminal beeps when you press CTRL/C, that indicates a file is currently open in that editor. You should then press CTRL/W to repaint your terminal screen with that file, examine the file, exit or quit that file, and then again press CTRL/C.

ABORT Command Method

The following procedure uses the ABORT command to abort the system job KEX.

1. Issue the following command to abort the system job KEX by using the ABORT command:

   ```
   .ABORT KEX  RET
   ```

2. Optionally, you can then issue the SHOW JOBS command to verify that you aborted KEX. If you successfully aborted KEX, the State of the job KEX is Done. If a file was currently open on KEX, KEX would refuse the abort, and the State of the job KEX would still be Run.

Aborting a job by using either method removes the job from your computer's extended memory. If you now issue the command SHOW MEMORY, you will see that KEX and KEX1 no longer reside in extended memory, thereby freeing over 50K words of that memory. However, portions of KEX and KEX1 still reside in low memory. You must unload them to free that memory. That procedure is described in the next section.

7.5 Unloading System Jobs

You cannot unload a job or its assigned device handlers unless the job exits or you abort it. Once a system job exits or you have aborted it, you should unload it (and any device handlers assigned to only it) to free the low memory allocated to that job. Use the UNLOAD command to remove a job from the computer's low memory.

Section 7.1 states that jobs are loaded in memory in the order that you run them. If the job you are unloading is the lowest job in your computer's memory, unloading that job (and any device handlers assigned it) consolidates that job's memory with the rest of the computer's free low memory. If the job you are unloading is not the lowest in your computer's memory, you must also abort and unload any jobs below it to consolidate free low memory. Once you have consolidated computer memory in that manner, you can restart any jobs you require.

You should currently have the jobs KEX and KEX1 loaded in low memory. To verify that, issue the following command:

```
.SHOW MEMORY RET
```

To demonstrate memory fragmentation, issue the following command that unloads the job KEX:

```
.UNLOAD KEX RET
```

Issue again the following command to display memory:

```
.SHOW MEMORY RET
```

Note in the display above the job KEX1, which is still loaded in memory, you have a fragmented memory area in which KEX resided. The amount of fragmented memory is shown in decimal notation. That memory is not consolidated with the main free low memory area (..BG..). Your computer can use that fragmented memory only to contain something that fits in that space. To consolidate the memory you freed by unloading KEX, you must unload KEX1:

```
.UNLOAD KEX1 RET
```

Issue again the following command to display memory:

```
.SHOW MEMORY RET
```

Note in that display that the memory occupied by KEX and KEX1 is now consolidated with your computer's main free memory.

The KEX editor does not have any device handlers assigned to it, so there are none to unload. Other system jobs, such as VTCOM and SPOOL, do have device handlers assigned to them when you start them. When you abort and unload those jobs, you should probably then unload those device handlers. That procedure is discussed in the following section.

7.6 Aborting and Unloading System Jobs Using Indirect Files

In Section 7.2.2, you learned a method of starting system jobs using an indirect command file. This section discusses creating indirect command files that, when run, abort and unload a system job.

Consider the following points when creating such indirect command files:

- Some system jobs are simple enough to directly abort and unload. You do not require an indirect command file for them.

  Section 7.4 and Section 7.5 showed that system jobs started from a single command can be aborted and unloaded using single commands. Such system jobs do not own device handlers; you do not assign specific device handlers to those jobs. KEX is such a system job.

- Some system jobs require you to perform operations at the keyboard to abort them. VTCOM is such a system job. You cannot abort VTCOM from an indirect command file.
- You should only abort a system job by using an indirect command file when that system job is in the state where it would accept a direct command to abort.

  Section 7.4 stated that you should only abort a system job when that job is idle. If the system job is performing an operation, you should end that operation before you can abort that job. For example, you cannot abort the KEX editor when there is a file open on that editor. You should not abort the SPOOL utility when there is output in the SPOOL queue.
- You cannot unload a device handler that is assigned to a system job without first aborting that system job.

The commands necessary to abort and unload a system job are particular to that system job. However, some general rules and order of procedure apply in all cases. Use the following as a general guide for creating indirect command files that abort and unload system jobs:

1. Examine the commands you used to start the system job. Those commands will be located in either an indirect command file you created specifically to start that job or in your start-up command file. It is helpful to have a listing; print the file containing the commands.
2. Issue the SHOW command. Any handler listed in the SHOW command display that is followed by the text Loaded=system job is coupled to the system job indicated by that text.

   Once you have made idle, aborted, and unloaded that system job, you can unload that handler without affecting any other job on your system. The only job on your system using that handler is the indicated system job.
3. Determine if there is a SET command condition that will idle the system job. If there is such a condition, include that command condition first in your file. For example, you can make the SPOOL utility idle by issuing the following command:

   ```
   SET SP KILL
   ```

   Although it is not technically a system job but rather a pseudohandler, you can treat the single-line (SL) command editor as a system job in terms of aborting and unloading it from a file. If you are creating such a file, the command to abort and unload SL is:

   ```
   SET SL OFF
   ```
4. Once you have made the system job idle, abort it, using the appropriate command. For example, you abort SPOOL by issuing:

   ```
   SET SP EXIT
   ```

   On the other hand, the editor is aborted by issuing the ABORT command, while VTCOM is aborted by explicitly exiting from the job by pressing CTRL/P and issuing the EXIT command.

5. Once you have aborted the system job, unload it by using the UNLOAD command.
6. Once you have unloaded the system job, unload any device handlers that were specifically assigned to that job.
7. You can optionally include the command SHOW MEMORY at the end of your file to check that the system job is unloaded. You can also check that your computer memory is not fragmented; that you can use the memory retrieved by unloading the job and handler.

   Any jobs or handlers loaded in memory below the system job you unloaded must be unloaded to consolidate the free low memory on your system.

The following example indirect command file, LOOPS.COM, makes idle, aborts, and unloads the SPOOL utility. It illustrates the information discussed above. You can use the following procedure to create LOOPS.COM:

1. Issue the following command:

   ```
   .EDIT/CREATE LOOPS.COM  RET
   ```

2. Enter the following command lines (or ones you prefer) in that file:

   ```
   ! LOOPS.COM, created dd-mmm-yy, edited dd-mmm-yy
   ! Closes down the print spooler
   !
   SET SP KILL
   SET SP EXIT
   UNLOAD SPOOL
   UNLOAD SP
   UNLOAD LS
   DEASSIGN LP
   DEASSIGN LP0
   SHOW
   SHOW MEMORY
   ```

3. Exit from that file by issuing the following commands:

   ```
   PF1  7
   Command:EXIT  ENTER
   ```

4. You can then run the indirect command file, LOOPS.COM, which idles, aborts, and unloads SPOOL, by issuing the following command:

   ```
   .$@LOOPS  RET
   ```

You will find it helpful to create indirect command files which start each system job that you use an indirect command file to abort and unload. Creating such a file lets you restart a system job without having to issue a lot of commands or reboot your system. This is true even if you typically start a system job from your start-up command file.

7.7 Chapter Summary

The following summary is the order of operations for using system jobs.

Starting system jobs
All device handlers used by a system job must reside on the system (SY) device and be loaded in memory.

System jobs can be started three ways:

- At system boot (start-up command file)
- From an indirect file ($@filespec.COM)
- From the keyboard (SRUN command)

Calling system jobs
Once a system job is started, it must be called by name from the system job environment prompt, *Job?*. You summon the system job environment prompt by pressing CTRL/X.

Aborting system jobs
When a system job is no longer required, you can abort it. How a system job is aborted is determined by a number of factors described in the chapter, but, in general there are three ways:

- Press CTRL/C at the CSI prompt.
- Issue the ABORT command and specify the job name.
- Call the job and issue the EXIT command:

  SET SP EXIT to abort SPOOL

  CTRL/P and EXIT to abort VTCOM

Unloading system jobs
Aborting a system job unloads the portion of the job that resides in extended memory. It does not, however, unload the low-memory portion. You must explicitly unload the job, using the UNLOAD command, to reclaim use of the low memory. So as to not fragment memory, you should unload in *last loaded/first unloaded* order ("unwinding").

Chapter 8

Using the Start-Up Command File to Configure Your System

This chapter specifically discusses the start-up command file for the XM and ZM monitors. However, the general information in this section applies also to the FB monitor.

NOTE

The start-up command files for the XM and ZM monitors are identical. Rather than continuously mention each monitor file and the appropriate start-up command file separately, this chapter will use the construction *x*M to represent both monitors, the construction RT11*x*M.SYS to represent both monitor files, and STRT*x*M.COM to represent both start-up command files. In each procedure, example, or illustration, use the form that is correct for the mapped monitor you want to support.

This chapter contains the following information:

- Issues to consider before you begin
- Editing your start-up command file
- Running your start-up command file

Use the on-line index utility, INDEX, and see the on-line help utility, HELP, for more information on the topics described in this chapter.

A start-up command file is an indirect command file. It can contain commands that:

- Display a file on your terminal screen, such as V5USER.TXT
- Set up and run utilities, such as SL, SPOOL, and VTCOM
- Change device handler characteristics
- Load device handlers
- Modify hardware characteristics
- Return information about your computer's configuration
- Run other indirect command files or IND control files

The RT–11 monitor you boot (or your computer automatically boots) looks for and attempts to run a particular start-up command file. For example, the ZM monitor looks for only the start-up command file STRTZM.COM; the XM monitor for only

STRTXM.COM, and the FB monitor looks for only STRTFB.COM. To be found, a start-up command file must reside on the system (SY) device. If your monitor cannot find its start-up command file, it displays an error message.

RT–11 is distributed with a start-up command file for each of the distributed monitors. As distributed, each start-up command file only displays the file V5USER.TXT on your terminal and runs the TMSCP controller checking program, MSCPCK. It is up to you to edit your start-up command file because only you know what functionality you require from that file.

8.1 Before You Edit Your Start-Up Command File

Before you begin to edit your start-up command file, you should and consider the following issues:

- What devices are part of your computer configuration?

  Your configuration can include a printer, a modem or serial line connection to another computer, disk or magtape drives, and so on.

  – What handlers do those devices require?

    Make a list of the device handlers required by those devices.

  – Do those handlers require you to set certain characteristics?

    Device handler characteristics must match those required by each device. Use SET command conditions to modify the characteristics of your device handlers to match those required by each device. Often, the default conditions for a device handler match those required by the device. However, devices such as terminals and printers may require you to configure your device handler differently than the default configuration. Also, some devices such as printers and modems may require that you set the device handler's baud rate.

    Consult your device documentation for that device's required characteristics. See the SET command information in the *RT–11 Commands Manual* for the SET command conditions available for each handler.

    To include those SET commands in your start-up command file is the only way to reference those characteristics in the future; they are not displayed by any SHOW command.

  – Are there optional characteristics you may prefer for those handlers?

    You can tailor your device handler to control a device in the manner you prefer by using SET command conditions.

- Are there utilities you want to run at system boot?

  RT–11 includes utilities you may want to run as you boot your system. For example, if you intend to use the SL editor, you probably want to turn on SL from your start-up command file. The distributed start-up command files also include commands you can enable to turn on VTCOM and SPOOL.

  – Do you want to run those utilities as system jobs? If so, are the device handlers for those utilities loaded?

    You must load all device handlers used by system jobs. At least, you probably want to load the handler for your default data (DK) device if that device is different than your system (SY) device.

  – If a utility requires a particular handler, does that handler require you to set certain characteristics? Are there optional characteristics for that handler you may prefer?

    For example, if you are going to run the single-line editor, SL, you may want to SET SL KED to enable keypad editing of your keyboard commands.

- Do you want to assign a particular device as your default data (DK) device? If you are using a system job, is DK loaded?

  At system boot, the default data device is automatically the same as the system device. If you want to assign a different device as the default data device, you probably want to do that from your start-up command file.

- Is there information about your computer configuration, such as memory allocation, that you are going to want displayed each time you boot?

  For example, to display system memory each time you boot, include the SHOW MEMORY command. To examine the status of the device handlers in your system configuration, include the SHOW command.

- Are there other indirect command files or IND control files you want to run each time you boot your system?

  You may have another indirect command file or IND control file on your system you want to run each time you boot. You can include a command to run such a file as the last command in your start-up command file. However, if you include such a command, that file will be run every time you boot your system (including a soft boot).

  You should be careful about running any files that initialize or configure a device from your start-up command file. You probably want to initialize or configure a device only at initial boot and not when you soft boot your system. In that case, manually run such an indirect file after your start-up command file has run.

- Is everything you reference in your start-up command file located on your system (SY) device? If not, you have to make provision for that.

  Your start-up command file looks for everything (all utilities, device handlers, other files, and so forth.) on the device your are booting—your system device. If anything you reference is not on your system device, you must explicitly indicate the device on which it resides. All device handlers used by system jobs must be on your system device.

8.2 Editing the Start-Up Command File

After you have decided what functionality you require when you boot your system, you can edit your start-up command file. Use the following information as a general guide to that editing process. You should have considered the points discussed in Section 8.1 and know what functionality you want at system boot.

1. Open your *RT–11 Commands Manual* to the SET commands section. If you have not already looked up the SET commands for the handlers you will be using, look them up as you read the following information.
2. Print your start-up command file.

   If your system includes a printer, issue the following command:

   ```
   .PRINT SY:STRTxM.COM RET
   ```
3. Look at your printed copy of the file.

   Note that all commands except TYPE V5USER.TXT at the beginning of the file and R MSCPCK at the end are preceded by an exclamation point (!). Those exclamation points are comment characters. When RT–11 runs an indirect command file (such as a start-up command file), commands that are preceded by comment characters are not executed. You enable commands by removing comment characters from the commands you want executed. You disable commands by adding comment characters to commands you do not want executed.
4. If you have not done so already, preserve your distributed start-up command file.

   ```
   .COPY SY:STRTxM.COM STRTxM.DIS RET
   ```
5. Issue the following command to remove the file protection:

   ```
   .UNPROTECT SY:STRTxM.COM RET
   ```
6. Start the KEX editor as a system job:

   ```
   .SRUN KEX.SAV RET
   ```
7. Issue the following commands to open STRT*x*M.COM for editing:

   ```
   CTRL/X
   Job?KEX RET
   KEX>
   SY:STRTxM.COM RET
   ```

   The file STRT*x*M.COM is displayed on your terminal screen.
8. Assuming you have read V5USER.TXT and do not want it displayed whenever you boot your system, add a (!) to the beginning of that command line:

   ```
   !TYPE V5USER.TXT
   ```

   The monitor that processes your start-up command file when you boot your system stops parsing a line when it encounters the comment character (!). Therefore, you achieve no real increase in speed by deleting the line.

9. Following the module identifier and the TYPE V5USER.TXT command is mixed-case text. That text is explanation of the commands that follow the text. If the size of the file is a concern to you, you can delete explanatory text after reading and understanding it. However, Digital recommends you keep the text until you are sure the file is executing the way you want. In any event, never delete just the comment character (!) in front of explanatory text. Your monitor would then attempt to execute that text as a command.

10. Do not enable the command SET TT QUIET. That command stops the display of the file as it executes. If there is a problem with the file's execution, you would not know where the problem exists if you cannot see the file execute.

    After you have edited and run the file and it executes in the manner you want, you can enable the command SET TT QUIET. However, seeing the display of commands each time you boot your system is a good reminder of how the file is configuring your system.

11. Because you are using KED or KEX to edit this file, you do not want the following two commands. The commands SET TT NOSCOPE,NOTAB,FORM,NOCRLF and SET EDIT EDIT are applicable only if you are using a hard-copy terminal, and you cannot run KEX on a hard-copy terminal.

12. The following two command lines turn on the SL (single-line) command editor. They also turn on the unsupported LET utility. For the purposes of this exercise, it is assumed you do not want the LET utility. You can find out about LET in the *RT–11 System Utilities Manual*.

    Assuming you do want SL, you probably also want to use the keypad to edit command lines. To turn on SL and the keypad editing functionality, edit those two command lines to appear as follows:

    ```
    SET SL RECALL,KED
    SET SL KMON
    ```

    To ensure that the characteristics of the SL handler match those of your monitor and to ensure that SL is installed on your system, go to the background environment and issue the following commands:

    ```
    CTRL/B
    SET SL SYSGEN  RET
    INSTALL SL  RET
    ```

    Then, return to editing the file:

    ```
    CTRL/X
    Job?KEX  RET
    CTRL/W
    ```

13. The next command is SET RUN VBGEXE. Remove the comment character (!), as this command enables automatic job loading in the completely virtual environment. You can see the *RT–11 System Internals Manual* for information on the completely virtual environment.

14. The following set of command lines turns on the virtual terminal communications package, VTCOM. The command lines use the *FRUN* command to run VTCOM

as the foreground job. If you want to run VTCOM as a system job to keep the foreground environment open for other uses, change the *FRUN* command to an *SRUN* command.

Note that all SET commands are issued before the XL handler is loaded.

The following example command lines turn on VTCOM as a system job for the PDP–11 computer:

```
SRUN SY:VTCOM.SAV/PAUSE
LOAD XL=VTCOM
RESUME VTCOM
```

15. The following two sets of command lines turn on the transparent spooler, SPOOL, and configure your printer. The first set is for serial interface printers supported by the LS handler. The second set is for parallel interface printers supported by the LP handler. If you do not know what type of printer you have, you can probably use the first set. The following discussion illustrates the first set of commands and the LS handler; the discussion also applies generally to the second set and the LP handler.

    Various SET commands may be required to run your printer correctly.

    - You must set the CSR and vector addresses if you are using a PDP–11 computer. The correct addresses are located in the *RT–11 System Generation Guide*.
    - Your printer may require that you change some default LS handler SET conditions; consult your printer documentation and the SET commands section for the LS handler in the *RT–11 Commands Manual*. You should include any LS handler SET conditions that configure your printer to perform as you want.
    - You can configure the SPOOL utility handler (SP) for some characteristics, such as flagpage and endpag support.
    - Some RT–11 utilities, such as DUMP, send list files by default to the LP handler. To prevent any potential problems, you should assign the printer handler (LS or SP) to the logical name LP.

    The following example command lines include the set commands and device assignments described above. In the example, *nnnnnn* represents the correct CSR and *nnn* represents the correct vector for your LS handler. The LS handler is set to not process form-feeds. The SP handler is set to not issue a flagpage and to perform a form-feed after printing each file.

    Note that all SET commands are issued before the LS handler is loaded.

```
SRUN SPOOL.SAV/PAUSE
SET LS CSR=nnnnnn
SET LS VECTOR=nnn
SET LS NOFORM0,ENDPAG=0
SET SP FLAG=0,ENDPAG=1
LOAD LS=SPOOL
RESUME SPOOL
ASS SP0 LP
ASS SP0 LP0
ASS SP0 LS
ASS SP0 LS0
```

16. Enable the command LOAD DU (remove the exclamation point (!)) if your system includes a DU device.

    If you are unsure, examine the current status of your device handlers to determine if DU resides on your system device.

    1. Go to the background and issue the SHOW command:

       ```
       CTRL/B
       SHOW  RET
       ```

    2. See if the SHOW command displays the existence of DU. Return to editing STRT*x*M.COM:

       ```
       CTRL/X
       Job?KEX  RET
       CTRL/W
       ```

17. Enable the next command, LOAD LD (remove the exclamation point (!)). You will be using the LD handler in further exercises. As mentioned previously, a device handler must be installed before it can be loaded. Use the following procedure to determine if LD is installed:

    1. Go to the background and issue the SHOW command:

       ```
       CTRL/B
       SHOW  RET
       ```

    2. If the LD handler is listed in the SHOW command display, it is installed on your system. If the LD handler is not listed, copy the LD handler (LDX.SYS) to your system (SY) device and then issue the following command:

       ```
       .INSTALL LD  RET
       ```

    3. Return to editing STRT*x*M.COM:

       ```
       CTRL/X
       Job?KEX  RET
       CTRL/W
       ```

18. Do not enable the following command lines:

    ```
    !INITIALIZE/NOQUERY VM:
    !ASS VM0 CF
    !ASS VM0 WF
    ```

Those commands apply to VM. You will learn about using the VM device in Chapter 11.

19. The following command line, SETUP VT100,JUMP,DARK,CLEAR, configures your terminal and clears the screen. Remove the exclamation point (!). See the *RT–11 Commands Manual* for more information on SETUP commands.

20. The next command, SHOW MEMORY, displays your system memory each time you boot. Remove the exclamation point (!). There are a number of other SHOW commands you can include. See the SHOW commands section of the *RT–11 Commands Manual* for more information.

21. Enable the command DATE (remove the exclamation point (!)), if you want to be reminded of the current date.

    If your computer requires you to enter the date each time you turn it on, including this command will remind you to do so.

22. The next command, SET TT NOQUIET, need only be turned on if you turned on the command, SET TT QUIET, at the beginning of your start-up command file.

23. The last command, R MSCPCK, runs a program that checks the controller revision level for TMSCP (MU) magtape devices. It is assumed that if your hardware configuration includes a TMSCP magtape device, you have already received any report produced by that command. Disable this command by adding the exclamation point (!) before it.

24. Close STRT*x*M.COM and abort KEX.

    ```
    PF1 7
    Command:Exit  Enter
    * CTRL/C
    ```

25. If your system has a printer, issue the following command to print your edited start-up command file:

    ```
    .PRINT SY:STRTxM.COM  RET
    ```

8.3 Running Your Start-Up Command File

You can now run your start-up command file by soft-booting your processor. Issue the command:

```
.BOOT SY:  RET
```

Watch your terminal screen as your file executes the commands you turned on. Your file should run to conclusion. That is, the last command you enabled in your file should execute and RT–11 should then display the monitor prompt (.).

If your file does not run to conclusion, do the following:

1. On your printed copy of the edited file, mark the point where the file stopped running.

2. If any error message was displayed where the file stopped, look up that error message in the *RT–11 System Message Manual*.
3. Compare the portion of text of your printed copy where the file stopped with the appropriate portion of text in Section 8.2. Keep any error message in mind as you compare the text.
4. If the solution is not obvious to you, reread Section 8.1. Keep any error message in mind as you read that section.
5. Edit your start-up command file to correct the error you believe halted its running. Edit your printed copy of the file to reflect that change.
6. Rerun your start-up command file by again soft-booting your processor:

   ```
   .BOOT SY: RET
   ```

   If your start-up command file previously ran to where it had started any foreground or system jobs, RT–11 displays the prompt:

   ```
   Foreground loaded; Are you sure?
   ```

   For the purpose of debugging your start-up command file, you can respond by typing Y (YES) without being concerned about the meaning of that message. The general significance of that message is explained later in this section.
7. Repeat this process, if necessary, until your start-up command file runs to completion.

Once your start-up command file runs to completion, you should consider the following information:

- If you are using VM as your system device, you must copy the file you just edited to your default system device to preserve it.
- A start-up command file that runs to completion does not guarantee that your system is now configured as you desire. (The commands it executed can be syntactically correct but not perform the operations you expect or desire.) You should test your system to be sure it performs as you expect before you use your system for any critical application.
- RT–11 does not accept a command to run a program that is already running. Once you have edited your start-up command file to start at least one foreground or system job and then booted your processor, RT–11 is then running at least that program. Therefore, you cannot use the standard indirect command file start construction ($@STRT*x*M) to run STRT*x*M.COM unless you first abort all programs you started from STRT*x*M.COM. Typically, rather than abort programs, you soft-boot your processor.

  If you are currently running any foreground or system jobs and issue the command to soft-boot your processor (BOOT SY:), RT–11 displays the following prompt:

  ```
  Foreground loaded; Are you sure?
  ```

That prompt informs you that at least one foreground or system job is currently loaded in your processor's memory. Soft-booting your processor could corrupt a file opened by a job or interrupt an operation being performed by a job. When you see that prompt, you should ask yourself:

- What foreground or system jobs are currently loaded?

  If you are unsure about what job or jobs are currently loaded and perhaps running on your system, type NO (N) and issue the following command to display those jobs:

  ```
  .SHOW JOBS RET
  ```

  You can disregard the job RESORC. That job is called by the SHOW JOBS command.

- Will interrupting the running of those jobs cause any problems?

  At least you will want to close any file open on an editor before you soft-boot your processor. You should also consider the effect a memory rearrangement or interruption may have on any job currently loaded and perhaps running on your processor.

  Soft-booting your processor can rearrange the contents of memory in your processor. That rearrangement can lose the contents of any file open on a job such as an editor. Soft-booting your processor can also interrupt the operations of a job that is maintaining communications, performing input /output operations, or monitoring a device.

8.4 Chapter Summary

This chapter describes editing a start-up command file to configure your computer system when you boot. Important points to remember about using a start-up command file are:

- Each monitor looks for a specific start-up command file. The FB monitor requires STRTFB.COM, the XM monitor requires STRTXM.COM, and the ZM monitor requires STRTZM.COM.
- The start-up command file must reside on the system (SY) device.
- The exclamation point (!) is the comment delimiter. You remove the exclamation point to enable a command and add one to the beginning of a command line to disable a command. Do not remove the exclamation point from comment (explanation text) lines.
- Although a start-up command file may run to completion (execute all commands), that does not necessarily mean that the system is configured as you expect. You should verify that the system is configured as expected before using the system.
- The start-up command file typically starts at least one job. RT–11 forbids running a job that is already running. Therefore, you generally must soft-boot your

system to rerun the start-up command file, rather than rerunning it directly, using the $@STRTFB, $@STRTXM, or $@STRTZM command.

Part III
Using RT–11 Features

Part III describes RT–11 features available under all distributed monitors. It is divided into the following chapters.

- Chapter 9, Using the Logical Disk (LD) Utility, describes using the logical disk (LD) utility. You can use the distributed LD to divide part of a large disk into smaller logical disks. Depending on the size of your hard disk, each of the logical disks can contain up to 64K blocks.
- Chapter 10, Using the Single-Line (SL) Command Editor, describes using the single-line (SL) editor. You use SL to edit your command line and recall previous commands. Using SL can save you time because SL streamlines the command-issuing process.
- Chapter 11, Using the Virtual Memory (VM) Device As the System Device, describes creating and using the virtual memory (VM) device as your system device. Using VM as your system device can greatly increase the performance of your system.
- Chapter 12, Using the RT–11 Communications Facilities, describes using the communications facilities available with RT–11. Those facilities include the VTCOM and TRANSF utilities. You use VTCOM to initiate and maintain communications with another host processor by using a modem or serial line. That processor can be non-DIGITAL. You can install and run TRANSF from a DIGITAL host PDP–11 or VAX processor and perform optimized file transfers between your processor running RT–11 and the host processor running RT–11, RSX–11, or VMS. TRANSF can transfer any type of file and performs complete error checking.
- Chapter 13, Defining Your Own Commands, describes defining your own commands by creating new commands and redefining existing commands. Also described is a method you can use to change the meaning of an existing command by removing support for it and then defining it as a new command.
- Chapter 14, Using the RT–11 Backup Facilities, describes the facilities distributed with RT–11 you can use to back up your data. Backing-up data to diskettes, hard disks, and magtapes is explained.

- Chapter 15, Using Indirect Command Files, describes creating, running, and nesting indirect command files. A comparison with BATCH and IND is provided.

Chapter summaries are not provided in this part because of the amount and complexity of the information presented in each chapter.

Chapter 9

Using the Logical Disk (LD) Utility

This chapter describes using the logical disk subsetting utility (LD) that is distributed with RT–11. You can use LD to divide a portion of a large disk into smaller logical disks. Each logical disk is a file; a logical disk file. Once a logical disk file is mounted and given a file structure, it becomes a logical disk unit and can be treated like any data device. As LD is distributed, you can access up to eight logical disk units at one time. You can also perform a system generation (SYSGEN) and request extended device-unit support, letting you access up to 32_{10} logical disk units at the same time.

This chapter provides the following information:

- Reasons for using LD
- Creating logical disk files
- Mounting and initializing logical disk units
- Nesting logical disk units
- Using logical disk units (an exercise)
- Enlarging logical disks

Use the on-line index utility, INDEX, and the on-line help utility, HELP, to find more information on topics described in this chapter.

9.1 Should You Use Logical Disks?

You might create logical disks on your computer system for any of the following reasons:

- You are working with files that can be logically divided into groups and you want to keep those groups separate.

  You can use logical disks to store sets of files and assign a logical disk as your default data device. Storing all files of a group on one or more logical disks can help you keep track of those files.

  You can nest logical disks within another logical disk. Thus, you can organize sets of files, each set in its own logical disk, within a larger group of files on a logical disk.

- You want to create logical disks of the same size as another type of device.

  If your computer system is made up of a large fixed disk drive and a smaller removable diskette drive, you can divide a portion of the large disk into a series of logical disks of the smaller drive size. Copy and backup operations between the fixed and removable disks can then be done as device-to-device rather than file-to-file operations.

- You have exceeded the number of directory entries supported by your physical disk.

  Each type of physical disk, by default, supports a certain number of directory segments and therefore directory entries. With a large number of small files, you can exceed that number of directory entries while still having free space on the disk. Because each logical disk has its own directory, dividing a physical disk into logical disks creates more directory entry space on the physical disk.

9.2 Distributed or Extended Device-Unit Support?

As distributed, RT–11 supports access of up to eight logical disk units at one time. If you perform a system generation (SYSGEN) and request extended device-unit support, you can access up to 32_{10} logical disk units at one time. Information on generating support for extended device units can be found in the *RT–11 Master Index*.

This chapter illustrates using LD with the distributed support. However, the procedures for creating and using logical disks are the same for the distributed support and extended device-unit support. You can use the information and exercises in this chapter with either support.

9.3 Creating Logical Disk Files

You can create any number of logical disk files on a device. As LD is distributed, however, you can access only eight of them at one time.

The following information pertains to creating logical disk files:

- You use the CREATE command to create a logical disk file on a device. The handler for the device on which you create a logical disk file does not need to be loaded.
- You use the /ALLOCATE:size option with the CREATE command to set the number of decimal blocks you want for the logical disk file.

  The usable size of the logical disk file will be smaller than the size you allocate, because the logical disk must also contain the file directory. The number of blocks used for the file directory structure is determined by the size of the file. For example, a 100-block logical disk file uses an 8-block file directory structure; a 6000-block file uses a 38-block file directory structure.
- Although not a requirement, all logical disk files should be created with the file type .DSK. RT–11 assumes the file type .DSK if it is not specified in a command. Also, using the .DSK file type lets you easily find all the logical disks files on a physical disk.
- You should assign a mnemonic file name to the logical disk that reminds you what that logical disk file contains.

The following command syntax creates a logical disk file. In the command, *ddn* represents the device and unit number on which you are creating the logical disk file, *file* is the name, and *nnnnnn* is the file size in decimal blocks.

```
.CREATE  ddn:file.DSK/ALLOCATE:nnnnnn  RET
```

9.4 Associating the Logical Disk File with a Logical Disk Unit

The logical disk utilities (LD.SYS or LDX.SYS) each contain a set of tables that associate logical disk files with up to eight logical device unit numbers, LD0 through LD7. You can also optionally assign a logical device name of 1–3 characters to a logical disk unit. That name must begin with an alphabetic character. The association (but not any name assignment) is maintained until you change it; the tables are not destroyed even when you shut off your computer.

The MOUNT command associates a logical disk file with one of the eight logical disk units. The file is protected from deletion once you associate it with a unit. The handler for the device containing the file must be loaded. If the handler is not loaded, the MOUNT command loads the handler. However, if you explicitly unload that device handler or reboot your processor and do not again load that handler, the association is lost. You can regain that association by again loading that device handler.

The following command syntax associates a logical disk file with a logical disk unit. In the command, *LDn* represents a logical disk unit between LD0 and LD7, *ddn* represents the device containing the logical disk file, and *nam* represents a logical device name you can optionally assign to that logical disk unit.

```
.MOUNT  LDn:  ddn:file.DSK  [nam]  RET
```

You can make a read-only logical disk unit, using the MOUNT command /READONLY option. If you make a logical disk unit read only and later decide to allow writing to it, you can issue the SET command WRITE condition for that unit (SET LDn WRITE) or mount it again and include the /WRITE option.

Once you have associated a logical disk file with a logical disk unit, you direct all commands and operations to that unit. If you have assigned a logical device name to that unit, you can direct commands and operations to that name.

9.5 Making the Logical Disk Usable

The first association of a logical disk file with a unit does not make that logical disk usable. You must create a file directory structure on that logical disk before you can use it. You use the INITIALIZE command to create a file directory structure on a logical disk, just as you do on a physical disk. As a safety measure, you should always perform a directory operation on a logical disk before initializing it to be sure it does not already contain a directory structure and files. Performing a directory operation on an uninitialized logical disk returns the error message, ?DIR-F-Invalid directory.

The following command syntax initializes a logical disk. In the command syntax, *LDn* represents the logical disk unit you associated with a logical disk file. You can optionally use the INITIALIZE command /NOQUERY option to suppress the RT–11 `Foreground loaded; Are you sure?` prompt.

```
.INITIALIZE/NOQUERY LDn: RET
```

Once you create a file directory structure on the logical disk file associated with unit LDn, you can use it as though it were a physical disk. If you break the association between that logical disk file and unit LDn (perhaps to associate a different logical disk file with unit LDn), you can use that logical disk file again by associating it with unit LDn or a different unit.

9.6 Changing and Displaying the Logical Disk Association

As distributed, RT–11 allows you eight logical disk associations at one time, though you can have far more than eight logical disks on your system. Therefore, as you work with your logical disks, you can periodically change the logical disks you have associated with your available eight logical disk units. If you find you must repeatedly change logical disk associations, you should investigate generating support for extended device units for LD.

Use the following command to determine the status of your logical disk unit associations:

```
.SHOW SUBSET RET
```

Just as you use the MOUNT command to associate a logical disk with a logical disk unit, you can use the DISMOUNT command to break the association between a logical disk and a logical disk unit. Use the following command syntax, where LDn represents that logical disk unit:

```
.DISMOUNT LDn: RET
```

9.7 Nesting Logical Disks

You can include (nest) one or more logical disks within a larger logical disk. You may, for example, have one or more subclasses of files that can be logically grouped within a larger class of files. You can create a primary logical disk for the entire class of files and nest a logical disk for each of the subclasses of files.

Think of a logical disk as a box that can contain up to 65K blocks. That box can contain files and other nested logical disks. Those nested logical disks are also boxes that can contain files and other nested logical disks, and so on. The total size of all files and nested logical disks in the primary logical disk cannot exceed 65K blocks.

You must associate a higher unit number with a nested logical disk than that associated with the primary logical disk. For example, assume you intend to mount a primary logical disk file and three logical disk files that are nested within it. Assume also that you have already mounted two logical disk files and associated them with unit numbers LD0 and LD1. You would probably want to associate the primary logical disk file with unit number LD2 and the three nested logical disk files with unit numbers LD3, LD4, and LD5.

A method for nesting logical disk files follows. The commands used are explained in the previous sections.

1. Create a primary logical disk file large enough to hold the files and nested logical disks you foresee it containing. Use the CREATE command with the /ALLOCATE option.
2. Use the MOUNT command to associate that logical disk file with a logical disk unit. Assume you are associating that file with unit LD0.
3. Use the INITIALIZE command to initialize that logical disk unit.
4. The two ways to nest logical disks are:
   - You can nest logical disks that already exist. Use the COPY command to copy those logical disk files you want to nest to the primary logical disk unit. Then, do a directory operation on the primary logical disk unit to verify the presence of the nested logical disk files and that the number of free blocks is as you expect.
   - You can also use the CREATE command and the /ALLOCATE option to create nested logical disk files in the primary logical disk unit.

     Assume you are creating three logical disk files to nest in the primary logical disk unit. You can create and nest any number of logical disk files, limited only by the device size of the primary logical disk.

     Recall that the primary logical disk file was mounted as LD0. The following series of commands creates three nested logical disk files, FIRST.DSK, SECND.DSK, and THIRD.DSK, on the primary logical disk you associated with LD0. The size you allocate to each file is represented by *nnnnnn*:

     ```
     .CREATE LD0:FIRST.DSK/ALL:nnnnnn RET
     .CREATE LD0:SECND.DSK/ALL:nnnnnn RET
     .CREATE LD0:THIRD.DSK/ALL:nnnnnn RET
     ```
5. Associate each of the nested logical disk files in the primary logical disk unit with a different unit number. Each nested logical disk unit number must be higher than the primary logical disk unit number. Recall that the nested logical disk files were created as FIRST.DSK, SECND.DSK, and THIRD.DSK, and that the primary logical disk file was mounted as unit LD0. Given that, the series of commands to mount the nested logical disk files could be:

   ```
   .MOUNT LD1: LD0:FIRST.DSK RET
   .MOUNT LD2: LD0:SECND.DSK RET
   .MOUNT LD3: LD0:THIRD.DSK RET
   ```

   The first association of each logical disk file protects it from deletion. Therefore, FIRST.DSK, SECND.DSK, and THIRD.DSK are now protected from deletion.

6. Initialize logical disks LD1, LD2, and LD3:

```
.INITIALIZE/NOQUERY LD1: RET
.INITIALIZE/NOQUERY LD2: RET
.INITIALIZE/NOQUERY LD3: RET
```

7. Issue the following command to verify the status of the logical disks:

```
.SHOW SUBSET RET
```

8. Do a directory operation on the physical device containing logical disk LD0. Note that only the primary logical disk file is displayed. You need to mount that logical disk file and do a directory operation to display the nested logical disk files. Therefore, you cannot access a mounted nested logical disk unless the primary logical disk is also mounted.

9. If you dismount (break the association between) the primary logical disk file and its unit without first dismounting nested logical disk unitss, you should issue the following command to free the nested logical disk units:

```
.SET LD CLEAN RET
```

9.8 Using the Logical Disk

Once you create a logical disk file, associate that file with a logical disk unit, and initialize it, you can use resulting logical disk like a physical disk. You can write to it, read from it, and use it as your default data (DK) device.

In the following exercises, you create a logical disk file and associate it with a logical disk unit. You initialize that logical disk, examine its directory size, and perform directory operations to it. You create a nested logical disk within the primary logical disk and examine their relationship. Finally, you dismount and delete both logical disks. This exercise uses your default data (DK) device; you can create and use logical disks with any disk on your system.

1. Ensure that LD is installed on your system.

2. Determine the decimal number of free blocks on your current default data (DK) device by issuing the following command:

```
.DIR/FREE RET
```

3. The following command creates a 100-block logical disk file on your DK device. If the previous command did not display any free block region of at least 100 blocks, use the largest displayed free block region rather than *100* in the following command:

```
.CREATE INTRO.DSK/ALLOCATE:100 RET
```

   RT–11 creates the logical disk file INTRO.DSK.

4. Display all file entries in your DK device of type .DSK:

```
.DIRECTORY *.DSK RET
```

   That command should display the file entry INTRO.DSK.

5. Associate the logical disk file INTRO.DSK with logical disk unit LD0:

   ```
   .MOUNT LD0: DK:INTRO.DSK RET
   ```

   The first association of any logical disk file with a logical disk unit causes RT–11 to protect that disk from deletion. Also, if not already loaded, the MOUNT command causes RT–11 to load the handler for the device containing the logical disk. (For the logical disk to associate with the logical disk unit, the handler for the device containing the logical disk must be loaded.) Once you have associated a logical disk file with a logical disk unit, you direct all commands and operations to the logical disk unit.

6. Initialize logical disk unit LD0:

   ```
   .INITIALIZE/NOQUERY LD0: RET
   ```

7. Examine the directory of logical disk unit LD0:

   ```
   .DIRECTORY LD0: RET
   ```

   Note that although you allocated 100 blocks for LD0, the directory command displays a 92-block device. The 8-block difference is the file directory structure for the logical disk file mounted on unit LD0.

8. Examine the status of your logical disk units:

   ```
   .SHOW SUBSET RET
   ```

9. Create a 30-block nested logical disk file, NEST.DSK, in the logical disk mounted on LD0. Mount NEST.DSK on logical disk unit LD1 and initialize it.

   ```
   .CREATE LD0:NEST.DSK/ALLOCATE:30 RET
   .MOUNT LD1: LD0:NEST.DSK RET
   .INITIALIZE/NOQUERY LD1: RET
   ```

   Do a directory operation on your DK device for files of the type .DSK and note that NEST.DSK does not appear. Do a directory operation on LD0 to display NEST.DSK. Do a directory operation on LD1 and note its size.

   ```
   .DIRECTORY *.DSK RET
   .DIRECTORY LD0: RET
   .DIRECTORY LD1: RET
   ```

   Issue the following command to display the relationship between the nested and primary logical disk units:

   ```
   .SHOW SUBSET RET
   ```

10. Break the association between nested logical disk file NEST.DSK and logical disk unit LD1 and between INTRO.DSK and LD0:

    ```
    .DISMOUNT LD1: RET
    .DISMOUNT LD0: RET
    ```

11. Unprotect logical disk file INTRO.DSK:

    ```
    .UNPROTECT INTRO.DSK RET
    ```

12. Delete logical disk files NEST.DSK and INTRO.DSK. Since NEST.DSK is nested in INTRO.DSK, deleting INTRO.DSK also deletes NEST.DSK.

    ```
    .DELETE INTRO.DSK RET
    ```

9.9 Enlarging Logical Disks

Do not worry that a primary logical disk may become too small to contain further files and nested logical disks (if you have not already allocated it the 65K-block limit). Assuming you have enough space on a physical disk, you can always make that primary logical disk larger. Take the following steps:

1. Use the MOUNT command to associate the smaller logical disk file (the one you are enlarging) with a logical disk unit.
2. Use the CREATE command to create a larger logical disk file, TEMP.DSK. Use the /ALLOCATE option to specify the new size. This larger logical disk file does not need to be on the same physical device as the smaller logical disk file. Use the MOUNT command to associate the larger logical disk file with a logical disk unit. Initialize that larger logical disk unit.
3. Copy the files and nested logical disks from the smaller logical disk to the larger logical disk. If the smaller logical disk is mounted as LD0 and the larger is mounted as LD1, the command is:

   ```
   .COPY/SYSTEM/VERIFY LD0:*.* LD1:*.* RET
   ```

   The /VERIFY option causes RT–11 to perform error checking during the copy operation. Depending on the size of the logical disks and the type of physical device involved, this operation can take some time.
4. When the copy operation completes, you might back up the contents of the smaller logical disk file. You can then dismount it, unprotect it, and delete it.

   You can then dismount TEMP.DSK from LD1, rename TEMP.DSK to the name you had assigned the smaller logical disk file, and perhaps finally mount the renamed file on whatever logical disk unit you previously associated with the smaller logical disk file.

Chapter 10

Using the Single-Line (SL) Command Editor

This chapter describes how to use the Single-Line (SL) command editor. In general, SL lets you store, recall, manipulate, and edit command lines. Using SL saves you time, because SL streamlines the command-issuing process.

You can use most SL functionality with any distributed RT–11 monitor. However, you must be running a mapped monitor to use the powerful RECALL command function, and using SL under a mapped monitor requires the least amount of limited low memory.

This chapter contains the following information:

- Reasons for using SL
- Turning on SL
- Editing the command line
- Storing and recalling commands
- Turning on SLMIN

Use the on-line index utility, INDEX, and see the on-line help utility, HELP, for more information on topics described in this chapter.

10.1 Why You Should Use SL

Use SL for any of the following reasons:

1. Computers are often used for repetitive tasks. Likewise, you are often required to repeat tasks when you use your computer. For example, suppose you must initialize a dozen floppy diskettes and examine those diskettes for bad blocks. Without SL, you have to type the following command a dozen times (where *ddn:* represents your diskette drive and unit number):

   ```
   .INITIALIZE/BADBLOCKS/NOQUERY ddn: RET
   ```

   Using SL, you type (issue) that command once. Then, when it is time to reissue that command, you press ↑ and then press RETURN. Pressing ↑ recalls that command and pressing RETURN issues it.

2. Suppose you mistype a command. RT–11 returns an error message indicating an invalid command or option to that command. Without SL, you retype that command (and perhaps mistype it again). Using SL, you press ↑ to recall that command. You then examine the command, determine your mistake, correct your mistake using SL, and press RETURN.

3. You often reissue a command after modifying just a small fragment of that command. Without SL, you retype the entire command. Using SL, you recall the command, edit the small fragment using SL, and reissue the command.
4. SL stores previous commands in a 512-byte buffer. Under any RT–11 monitor, you can continuously press ↑ to scroll through those previous commands. You then press RETURN to issue any of those previous commands.

   Under a mapped monitor, you can use the SL command RECALL/ALL to display the stored commands. You can then recall a specific command by using either the number displayed next to each command or the shortest unambiguous abbreviation for that command. You then press RETURN to issue that recalled command.
5. You often repeat a set of commands. You might repeatedly edit, compile, link, and run a program. Without SL, you type the same set of commands repeatedly as you debug your code. Using the functionality, *set cycle* and *cycle*, you use just three keystrokes to sequentially display and repeat each command in the set.
6. You may find that you often issue a particular command. SL lets you save one command that you can issue whenever you need it.

This chapter describes and provides examples of how to perform those functions. You can also use the distributed on-line index program, INDEX, to find all references to SL in the RT–11 documentation.

10.2 Turning on SL

The RT–11 monitor you are using determines which SL functionality is available. You enable different SL functionality by specifying various conditions with the SET SL command. All SL set conditions are described in the *RT–11 Commands Manual*. The SL conditions described in this chapter are:

KED	Turns on keypad editing functionality. Keypad editing is available under all RT–11 monitors.
KMON	Turns on SL and sets it to edit only keyboard monitor commands. The KMON condition can be used under all RT–11 monitors.
OFF	Turns off SL under all monitors.
ON	Turns on SL and sets it to edit both keyboard monitor and CSI commands. If you use the CSI command syntax, you should turn on SL, using the ON condition. Otherwise, use the KMON condition. Because this chapter uses only keyboard monitor commands, the KMON condition is shown.
RECALL	Lets you use the RECALL keyboard command. RECALL can be used with only a mapped monitor.

SYSGEN	Matches certain SL characteristics with any RT–11 monitor you may have built, using the SYSGEN procedure. Setting the SYSGEN condition is described further in this section.
VT102	Matches certain SL characteristics with the VT200 and VT300 series video terminals, making SL more efficient with those terminals.

Use one of the following two series of commands to turn on SL:

If you are not running a mapped monitor:

Issue the following command (omit the VT102 conditional if not using VT200 or VT300 series terminals):

```
.SET SL VT102,KED,KMON  RET
```

If RT–11 returns an error message indicating an invalid command, a conflict may exist between SYSGEN conditions for your monitor and SL. RT–11 cannot install SL. Issue the following commands to match SL with your RT–11 monitor's characteristics:

```
.SET SL SYSGEN  RET
.INSTALL SL  RET
.SET SL KED,KMON  RET
```

If you are running a mapped monitor:

Issue the following command (omit the VT102 conditional if not using VT200 or VT300 series terminals):

```
.SET SL VT102,RECALL,KED,KMON  RET
```

If RT–11 returns an error message indicating an invalid command, a conflict may exist between SYSGEN conditions for your monitor and SL. RT–11 cannot install SL. Issue the following commands to match SL with your RT–11 monitor's characteristics:

```
.SET SL SYSGEN  RET
.INSTALL SL  RET
.SET SL RECALL,KED,KMON  RET
```

10.3 Editing the Command Line

One of the major SL functions is command line editing. Without command line editing, any change you want to make to a command line requires that you delete all characters from your current cursor position to the character you want to change or add. Because you turned on SL in the previous section, you can now move the cursor directly to that part of the command line you want to change without having to delete any intervening characters.

Two levels of editing functionality are available to you. The first level, the basic level, is available when you turn on SL. An enhanced level, keypad editing, is available when you set the SL condition KED. No memory penalty occurs with turning on keypad editing; the keypad editing functionality uses no extra computer memory.

Because you set the SL condition KED when you turned on SL in Section 10.2, you have both levels of functionality available and you can combine the features of both levels.

You should understand basic command line editing before you examine the enhanced functionality available with keypad command line editing.

10.3.1 Basic Editing of the Command Line

Basic command line editing generally involves two types of operations:

1. First, you use the arrow keys to move your cursor through the command line and into and out of the stack of previous commands. You use ← and → to move through the command line. If you need to recall a command before editing, you use ↑ and ↓ to move through the command stack to recall that command.
2. Once you have placed your cursor at the desired location, you use other keypad and keyboard keys to delete, add, and manipulate characters within the command line.

   As SL is distributed, any character you type within a command line is added; that functionality is called INSERT mode. You can change that functionality while editing a command line so that any character you type replaces the character located at the cursor position; that is called REPLACE mode. You alternate between INSERT and REPLACE mode by pressing CTRL/A. SL returns to INSERT mode each time you press RETURN to issue a command.

The following table lists the keypad and keyboard keys you can use with basic command line editing and their functions. Following the table is an exercise that demonstrates using those keys.

Key	Function
PF1	Changes the function of other keys when used in combination with them. Also referred to as the GOLD key.
PF1 S	Saves the displayed command. The S key is located on your main keyboard and is not case sensitive.
PF1 X	Displays the saved command. The X key is located on your main keyboard and is not case sensitive.
PF2	Displays a screen of helpful SL information on the top half of your screen (above the command line). If you first issue the command, SET SL LEARN, subsequently pressing PF2 causes the SL help screen to be locked onto the display. Issue the command SET SL NOLEARN to remove the help screen.

Key	Function
$\boxed{\text{PF4}}$	Delete line function key. Deletes command line from cursor position to end of line. The $\boxed{\text{PF1}}$ $\boxed{\text{PF4}}$ combination restores the deleted line.
$\boxed{\uparrow}$ or $\boxed{\text{CTRL/E}}$	Moves the cursor into the command stack, displaying previous commands. The $\boxed{\text{PF1}}$ $\boxed{\uparrow}$ combination displays the next command in the current cycle of commands. This cycle functionality is described in Section 10.4.3.
$\boxed{\downarrow}$ or $\boxed{\text{CTRL/V}}$	Moves the cursor out of the command stack, displaying previous commands. The $\boxed{\text{PF1}}$ $\boxed{\downarrow}$ combination establishes an offset into the stack of previous commands. This set cycle functionality is described in Section 10.4.3.2.
$\boxed{\leftarrow}$ or $\boxed{\text{CTRL/D}}$	Moves the cursor toward the beginning of the command line. The $\boxed{\text{PF1}}$ $\boxed{\leftarrow}$ combination moves the cursor to the beginning of the command line.
$\boxed{\rightarrow}$	Moves the cursor toward the end of a command line. The $\boxed{\text{PF1}}$ $\boxed{\rightarrow}$ combination moves the cursor to the end of the command line.
$\boxed{\text{LINE FEED}}$ or $\boxed{\text{CTRL/J}}$	Deletes the element of command syntax to the left of the cursor. An element of syntax can be the device handler name and unit number, file name, file type or extension, command, any options, and any arguments to those options.
$\boxed{\text{DELETE}}$	Deletes the character to the left of the cursor. The $\boxed{\text{PF1}}$ $\boxed{\text{DELETE}}$ combination restores that deleted character or character position.
$\boxed{\text{BACKSPACE}}$or $\boxed{\text{CTRL/H}}$	Switches the character at the current cursor position with the character to the right of the cursor and moves the cursor to the right. The $\boxed{\text{PF1}}$ $\boxed{\text{BACKSPACE}}$ combination switches the character at the current cursor position with the character to the left of the cursor and moves the cursor to the left.
$\boxed{\text{CTRL/B}}$	Requires that TT is set to condition NOFB. Same as $\boxed{\uparrow}$
$\boxed{\text{CTRL/F}}$	Requires that TT is set to condition NOFB. Same as $\boxed{\rightarrow}$
$\boxed{\text{CTRL/R}}$ or $\boxed{\text{CTRL/W}}$	Redisplays the current command line. Use $\boxed{\text{CTRL/R}}$ or $\boxed{\text{CTRL/W}}$ to refresh the command line on your terminal screen if the command line becomes garbled or corrupted.

Key	Function
CTRL/U	Deletes all characters from the current cursor position to the beginning of the command line. The PF1 CTRL/U combination restores those deleted characters.
RETURN	Issues the currently displayed command. You can press RETURN and issue a command no matter where the current cursor position in that command. The PF1 RETURN combination truncates the command (deletes all characters from the cursor position to the end of the command line) and issues the truncated command.

The following exercise shows basic command line editing:

1. Type the following command but do not press RETURN:

   ```
   .DIRECTORY/FULL SY:
   ```

2. Press PF1 and S to save that command.
3. You could press RETURN and issue that command. However, for the purpose of this exercise, press CTRL/U to delete the command line.
4. Issue the following misspelled command exactly as shown:

   ```
   .DILECTORY/PROTECT/ODRER:SIZE SY:  RET
   ```

   RT–11 returns an error message indicating an invalid command or that the command does not exist.

5. Press the ↑ key.
6. SL recalls the previous command:

   ```
   .DILECTORY/PROTECT/ODRER:SIZE SY:
   ```

7. Examine that command line and note that the command DIRECTORY is misspelled as DI*L*ECTORY and the option /ORDER:SIZE is misspelled as /O*DR*ER-:SIZE.
8. Press PF1 and ←. The cursor is then located at the beginning of the command line. Press → until your cursor is positioned at the E character in DIL*E*CTORY:

   ```
   .DILECTORY/PROTECT/ODRER:SIZE SY:
   ```

   (If you go too far, you can return using ←.)

9. Press DELETE.
10. Type the letter R.
11. Correct the misspelled option /ODRER/SIZE. Press → until the cursor rests on the letter D:

    ```
    .DIRECTORY/PROTECT/ODRER:SIZE SY:
    ```

12. Press CTLR/H to switch the letter D with the letter to the right of it (R). Press RETURN.

    You have corrected the command and it executes.

13. Press PF1 and X to display the command you saved at the beginning of this exercise. Press RETURN to issue that command.

14. Continuously press ↑ to verify you have stored the commands you issued in this exercise on your stack of previous commands. After you display all the commands on your stack of previous commands, you are returned to the monitor prompt (.).

15. Press and hold down the ↓ key. You scroll back out through your stack of previous commands to the monitor prompt (.).

10.3.2 Keypad Editing of the Command Line

The keypad method lets you use your keypad editing keys to recall command lines, move the cursor, and edit the command line. This method requires that you know the function of the keypad keys.

In keypad mode, you use the keypad to edit the command line as you would use the keypad from within the KED editor. Use the KED keypad functions appropriate for editing text within a single line. The following table lists those keypad keys you use to recall commands, move the cursor, and edit command lines. Following the table is a series of exercises that demonstrate using those keys.

Key	**Function**
PF1	Changes the function of other keys when used in combination with them. Also referred to as the GOLD key.
PF2	Displays a HELP frame on your terminal screen.
PF4	Deletes the command line from the cursor position to the end of the command line. The PF1 PF4 combination restores that deleted command line.
0	Moves the cursor to the beginning of the command line. If the cursor is at the beginning of the command line, 0 moves the cursor to the beginning of next previous or next most recent command line. The PF1 0 combination is the open-line function. Produces a blank line for entering a command.
1	Causes the cursor to move across one element of syntax. An element of syntax can be the device handler name and unit number, file name, file type or extension, command, any options, and any arguments to those options.

Key	Function
$\boxed{\text{2}}$	Moves the cursor to the end of the command line. If the cursor is at the end of the command line, $\boxed{\text{2}}$ moves the cursor to the end of the next previous or next recent command line. The $\boxed{\text{PF1}}$ $\boxed{\text{2}}$ combination deletes all characters from the cursor position to the end of line.
$\boxed{\text{3}}$	Moves the cursor one character position.
$\boxed{\text{4}}$	Causes subsequent cursor movement to be forward.
$\boxed{\text{5}}$	Causes subsequent cursor movement to be in reverse.
$\boxed{\text{,}}$	Deletes one character at the cursor position. The $\boxed{\text{PF1}}$ $\boxed{\text{,}}$ combination restores that deleted character.
$\boxed{\text{-}}$	Deletes one element of command syntax at the cursor position. An element of syntax can be the device handler name and unit number, file name, file type or extension, command, any options, and any arguments to those options. The $\boxed{\text{PF1}}$ $\boxed{\text{-}}$ combination restores that element of command syntax.
$\boxed{\text{ENTER}}$	Issues the displayed command. Equivalent to $\boxed{\text{RETURN}}$ key.

The keypad keys $\boxed{\text{.}}$ $\boxed{\text{6}}$ $\boxed{\text{7}}$ $\boxed{\text{8}}$ $\boxed{\text{9}}$ and $\boxed{\text{PF3}}$ are not implemented.

Exercises

The following exercises demonstrate keypad command line editing:

Exercise 1

This exercise recalls a command you typed previously and corrects the misspelling in that command.

1. Press the $\boxed{\text{5}}$ $\boxed{\text{0}}$ keypad keys until the following misspelled command is displayed:

   ```
   .DILECTORY/PROTECT/ODRER:SIZE SY:
   ```

   Your cursor is resting at the beginning of the command line.

2. Press the $\boxed{\text{4}}$ keypad key to direct cursor motion forward.

3. Press the $\boxed{\text{3}}$ keypad key twice so the cursor rests on the letter L:

   ```
   .DILECTORY/PROTECT/ODRER:SIZE SY:
   ```

4. Press the $\boxed{\text{,}}$ keypad key to delete the letter L.

5. Type the letter R.

6. Press the $\boxed{1}$ keypad key twice and the $\boxed{3}$ keypad key once so the cursor rests on the D:

   ```
   .DIRECTORY/PROTECT/ODRER:SIZE SY:
   ```

7. Press the $\boxed{,}$ keypad key to delete the letter D. Press the $\boxed{3}$ keypad key to advance the cursor one character and type D. Press $\boxed{\text{RETURN}}$.

   The command executes.

Exercise 2

This exercise recalls a previous command and changes an option argument. You also change the device for which you want the directory from your default data (DK) to system (SY) device.

1. Press the $\boxed{5}$ $\boxed{0}$ keypad keys to recall the command:

   ```
   .DIRECTORY/PROTECT/ORDER:SIZE SY:
   ```

2. Press the $\boxed{4}$ keypad key to change the cursor motion to forward. Press the $\boxed{1}$ keypad key three times to move forward three syntax elements:

   ```
   .DIRECTORY/PROTECT/ORDER:SIZE SY:
   ```

3. Press the $\boxed{-}$ keypad key to delete SIZE. Type DATE and press the space bar.
4. Press the $\boxed{-}$ keypad key to delete SY:.
5. Type DK: and press $\boxed{\text{ENTER}}$.

   The edited command now executes and gives you different information.

Exercise 3

This exercise recalls the previous command and changes it to return the full directory of your default data (DK) device.

1. Press the $\boxed{5}$ $\boxed{0}$ keypad keys to recall the last command:

   ```
   .DIRECTORY/PROTECT/ORDER:DATE DK:
   ```

2. Press the $\boxed{4}$ $\boxed{1}$ keypad keys:

   ```
   .DIRECTORY/PROTECT/ORDER:DATE DK:
   ```

3. Press the $\boxed{-}$ keypad key three times to delete the command options PROTECT /ORDER:DATE:

   ```
   .DIRECTORY/DK:
   ```

4. Type FULL, press the space bar, and press $\boxed{\text{ENTER}}$.

   The command executes and displays the full directory of your default data device.

10.4 Storing and Recalling Commands

Another major SL function is storing and recalling previous command lines.

Storing Commands

SL stores previous commands on its 512-byte command storage stack. Because each character and space you type in a command uses 1 byte of stack storage, you can store more commands by abbreviating commands and options to their shortest unambiguous length.

You can also store a command by explicitly saving it. SL lets you save one command that you can recall whenever you need it.

Recalling Commands

SL provides five methods for recalling stored commands. If you are using a mapped monitor and have set the SL conditions RECALL and KED (SET SL RECALL,KED), you can use all methods. Otherwise, the indicated restrictions apply.

- Arrow key method

  Under all monitors, you can use ↑ and ↓ to scroll into and out of the command storage stack.

- Keypad method

  Under all monitors with the KED condition in effect, you can use the keypad advance and backup motion function keys with the beginning-of-line or end-of-line function keys to move in and out of the command storage stack.

- Cycling method

  Under all monitors, you can use the cycling functionality (set cycle and cycle) to recall continuously a sequence of commands.

- RECALL command method

  To use the RECALL command, you must be using a mapped monitor with the RECALL condition in effect. The RECALL command provides the most direct method for displaying and recalling previous commands. The RECALL command displays a numbered list of previous commands. You can recall any command from that list by using the command number or the shortest unambiguous abbreviation for that command.

- Save command method

  Under all monitors, you can save a displayed command by pressing PF1 and S . You can then recall that saved command whenever you need it by pressing PF1 and X.

You issue any recalled command by pressing RETURN regardless of the cursor position within that command.

Exercise some caution when issuing a recalled command. Be sure to read carefully the recalled command before pressing RETURN.

The following sections describe using those methods for recalling stored commands.

10.4.1 Using Arrow Keys to Recall Commands

Use $\boxed{\uparrow}$ and $\boxed{\downarrow}$ to recall (display) previous commands.

Use $\boxed{\uparrow}$ to move into your stack of previous commands. Press and release $\boxed{\uparrow}$ to singly recall previous commands or press and hold $\boxed{\uparrow}$ to scroll into your stack of commands.

Use $\boxed{\downarrow}$ to move back out through the stack. Press and release $\boxed{\downarrow}$ to singly display commands or press and hold $\boxed{\downarrow}$ to scroll out through the command stack to the current (empty) command line.

SL returns to the monitor prompt (.) when you exceed either command stack limit; when you scroll into the stack beyond the first stored command, or scroll out of the stack beyond the last stored command. You can then issue a command in response to that monitor prompt or use the arrow keys to reenter the command stack to recall and issue a previous command.

10.4.2 Using the Keypad to Recall Commands

If you have enabled keypad editing, using the command SET SL KED, you can use the keypad to scroll into and out of your storage stack of previous commands. Use a combination of four keypad keys to do that.

You use the advance function key, $\boxed{4}$, or the backup function key, $\boxed{5}$, to set the direction of motion through the stack. Pressing $\boxed{5}$ moves you into the stack. Pressing $\boxed{4}$ moves you back out of the stack. You need to set the direction of motion only once, until you want to change it.

The beginning-of-line, $\boxed{0}$, and end-of-line, $\boxed{2}$, function keys determine your cursor placement on the command line, as you move in and out of the command stack.

For example, pressing the keypad sequence, $\boxed{5}$ $\boxed{0}$ $\boxed{0}$, places your cursor at the beginning of the command that resides two entries into the stack. Pressing the keypad key $\boxed{5}$ and then pressing and holding the keypad key $\boxed{0}$ causes SL to scroll through your stack of previous commands with the cursor at the beginning of each command. Pressing $\boxed{4}$ and $\boxed{0}$ in the same manner reverses the process.

SL returns to the monitor prompt (.) when you exceed either command stack limit; when you scroll into the stack beyond the first stored command, or scroll out of the stack beyond the last stored command. You can then issue a command in response to that monitor prompt or use the keypad to reenter the command stack to recall and issue a previous command.

10.4.3 Using the Cycling Method to Recall a Sequence of Commands

SL provides the *set cycle* and *cycle* functionality for recalling a sequence of commands. That method is useful when you are repeating an operation that requires a command sequence.

By default, SL lets you continuously repeat the previous two commands as a sequence. SL also lets you determine the number of commands you want to continuously repeat as a sequence.

10.4.3.1 Default Cycling Functionality

You can continuously repeat the previous two commands in your command storage stack. That is, the default SL cycle functionality contains an offset of two commands. You access and repeat those two commands as a sequence by pressing $\boxed{\text{PF1}}$ and $\boxed{\uparrow}$ to access each command and pressing $\boxed{\text{RETURN}}$ to repeat that command.

The default cycling functionality is useful when you want to continuously repeat the previous two commands. How to change the number of commands you continuously repeat is described in the next section.

10.4.3.2 Changing the Cycling Functionality

You can change the cycling functionality to store an offset into your storage stack of previous commands. After you have issued a series of commands that you then want to repeat, you establish the offset (set the cycle) for the first command in that sequence. You then cycle through that command sequence, return to that first command, and repeat the sequence.

Command sequence cycling is helpful, for example, when debugging source code. Assume you are debugging MACRO source code. That operation may require that you repeatedly edit, assemble, link, and run the code. You may repeatedly issue the commands EDIT filename, MACRO filename, LINK filename, and RUN filename.

After you have issued that sequence of commands, RUN filename is the first stored command on your stack, and EDIT filename is the fourth. Rather than scroll into your stack four positions to repeat each command in the sequence, you can use the set cycle and cycle functionality as follows:

1. Use the arrow keys, the keypad, or the RECALL command to recall the first command in the sequence (EDIT filename). Press $\boxed{\text{PF1}}$ and $\boxed{\downarrow}$ to store that offset (set the cycle) into your command stack. Press $\boxed{\text{RETURN}}$ to issue that command.
2. When you are ready to recall the next command in the sequence, press $\boxed{\text{PF1}}$ and $\boxed{\uparrow}$. SL recalls the next command in the sequence. Press $\boxed{\text{RETURN}}$ to issue that command.
3. Continue to press $\boxed{\text{PF1}}$ and $\boxed{\uparrow}$ to recall successive commands in the sequence. Press $\boxed{\text{RETURN}}$ to issue each of those commands.
4. When you have issued the last command (RUN filename) and want to repeat the sequence, press $\boxed{\text{PF1}}$ and $\boxed{\uparrow}$ to recall the command EDIT filename. Press $\boxed{\text{RETURN}}$ to issue that command.

   You can continue to use this command sequence cycle until you change the offset into your stack. You change that offset by issuing, rather than recalling, any command. Therefore, you should think of the set cycle and cycle functionality as a temporary method of recalling a particular command sequence.

10.4.4 Using RECALL to Display and Recall Commands

You issue the RECALL command in response to the keyboard monitor prompt (.). You can use the RECALL command to:

- Display all previous commands stored in your command storage stack.

  The following command displays all stored previous commands:

  ```
  .RECALL/ALL  RET
  ```

  SL displays the list of previous commands, as in the following example:

  ```
  .RECALL/ALL  RET
         1 SRUN KEX.SAV
       * 2 MOUNT LD2: DK:INTRO.DSK
         3 SHOW SUBSET
  .
  ```

  That display includes all previous commands stored in your command stack. In this example display, SRUN KEX.SAV is the most recent command you issued, indicated by the number 1 next to that command. The asterisk (*) next to the command, MOUNT LD2: DK:INTRO.DSK, indicates that command is first in the current cycle of commands.

- Recall the previous command.

  The RECALL command with no option recalls the most recent previous command:

  ```
  .RECALL  RET
  ```

- Recall a command, using the number displayed next to that command.

  Assuming that issuing the RECALL/ALL command returns the following example display:

  ```
  .RECALL/ALL  RET
         1 SRUN KEX.SAV
       * 2 MOUNT LD2: DK:INTRO.DSK
         3 SHOW SUBSET
  .
  ```

  You can recall the command, SHOW SUBSET, by issuing the following command:

  ```
  .RECALL 3  RET
  ```

- Recall a command, using the shortest unambiguous abbreviation for that command.

  Assuming that issuing the RECALL/ALL command returns the following example display:

  ```
  .RECALL/ALL  RET
        1 SRUN KEX.SAV
      * 2 MOUNT LD2: DK:INTRO.DSK
        3 SHOW SUBSET
  .
  ```

You can recall the command, SHOW SUBSET, by issuing the following command:

```
.RECALL SH  RET
```

SL does not keep any RECALL command in its stack of previous commands. Therefore, you cannot recall the RECALL command.

10.4.5 Saving a Command

SL lets you save one command outside the stack of previous commands. You can then recall that command whenever you need it. You save a command, using the following procedure:

1. The command you want to save must be displayed. You display a command by either typing it or recalling it from your stack of previous commands.

   Do not issue the command; do not press RETURN.

2. Once the command is displayed, press PF1 and S.

   You can then press RETURN to issue that command. However, you do not need to issue the command to save it.

You can now recall the saved command at any time by pressing PF1 and X.

10.5 SLMIN — The Smallest Single-Line Command Editor

RT–11 distributes a minimal single-line command editor that contains a subset of the basic command line editing features described in Section 10.3.1. SLMIN needs less memory than SL. SLMIN can be an appropriate command line editor when you are not using a mapped monitor and you are concerned about low-memory availability. You cannot use SLMIN if you are using a mapped monitor.

Enhanced SL features, such as keypad editing and the RECALL command, are not available under SLMIN.

SLMIN provides all the basic command line editing features described in Section 10.3.1, except:

- There is no help screen. The PF2 keypad key has no meaning under SLMIN.
- The stack of previous commands contains only two commands.
- Pressing LINE FEED or CTRL/J does not delete previous commands, but rather is equivalent to pressing RETURN.

All other basic command line editing features are available.

Issue the following commands to preserve the distributed SL and copy SLMIN to SL:

```
.SET SL OFF  RET
.RENAME SL.SYS SL.DIS  RET
.COPY SLMIN.SYS SL.SYS  RET
.REMOVE SL  RET
.INSTALL SL  RET
```

You then turn on SLMIN as SL by issuing the following command:

```
.SET SL KMON RET
```

If you want to return to using the distributed SL, issue the following commands:

```
.SET SL OFF RET
.UNPROTECT SL.SYS RET
.RENAME SL.SYS SLMIN.SYS RET
.RENAME SL.DIS  SL.SYS RET
.REMOVE SL RET
.INSTALL SL RET
.SET SL KED,KMON RET
```

Chapter 11

Using the Virtual Memory (VM) Device As the System Device

This chapter shows how to create and use a virtual memory (VM) device as your system (SY) device. All currently sold PDP–11 computers are capable of supporting a VM device. You use the VM region in the extended memory in your computer to create a VM device. If your computer hardware is capable of supporting a VM device, you can increase your computer's speed by running RT–11 from such a device.

This chapter is divided into the following sections:

- Computer hardware requirements for supporting a VM device as the system device
- The advantages in creating and using a VM system device
- Configuring a VM device in your computer's extended memory
- Determining a working system
- Creating a model working system
- Running RT–11 from a VM device
- Creating other working systems for the VM device

Use the on-line index utility, INDEX, and see the on-line help utility, HELP, for more information on topics described in this chapter.

11.1 Computer Hardware Requirements

Your computer must contain at least 512K bytes of memory and a memory management unit for you to use VM as your system (SY) device. All current and most recent PDP–11 and CTI Bus-based computers meet those requirements.

Issue the following command to determine how much memory your computer contains. Let the display produced by the command scroll to the end.

```
.SHOW CONFIGURATION RET
```

The information you are looking for is in the next-to-last block of lines.

The second line of that block reads *nnnnKB of Memory*, where *nnnn* is the memory in bytes contained in your computer. The number displayed should be at least 512, and more is better. If the number displayed is less than 512, you probably cannot create a VM device and should disregard the rest of this chapter.

11.2 VM Device

The VM device is an area in extended memory you configure as a disk. Once you configure VM as a disk, you copy a working RT–11 operating system to VM and boot VM. You are then running your computer from the memory contained in your computer. You can also copy application programs you often run into VM and run those programs from VM.

When you run your computer from a physical disk, your computer periodically must read and write operating system code between the computer's memory and that physical disk. When you run your computer from VM, your computer reads and writes operating system code between segments of memory, which is much faster. When you copy application programs into VM and run them from VM, those application programs also run much faster, especially if they contain low-memory overlays.

CAUTION

The VM device uses extended virtual memory as a disk; that is what gives VM its speed. VM uses volatile memory for data storage, which is not permanent. Any files on your VM device are destroyed when you hard boot your system or if power is disrupted to your system. Therefore, a good practice is to copy only RT–11 system and utility files or your programs to VM. Create and edit files and data on permanent media, such as your hard disk.

The following sections describe how you can configure and use a VM dsystem device and realize the increase in computer speed.

11.3 Configuring a VM Device

How you configure your VM device is determined by which RT–11 monitor you are going to run from VM. All distributed RT–11 monitors can be run from VM.

11.3.1 Configuring a VM Device for an Unmapped (SB or FB) Monitor

The unmapped monitors cannot access any extended memory other than that configured as VM. Therefore, if you intend to run an unmapped monitor from VM, you should configure VM for all extended memory.

CAUTION

If you are using FORTRAN IV, do not use virtual arrays when using VM. FORTRAN IV can write virtual arrays in the VM region, thereby corrupting VM and crashing the system.

First, determine if the VM device handler is installed on your system. Issue the following command and see if VM is listed in the display:

```
.SHOW RET
```

If VM is not listed in the display, issue the following command:

```
.INSTALL VM RET
```

Issue the following commands to assign all extended memory to VM:

```
.UNLOAD VM RET
.REMOVE VM RET
.SET VM BASE=1600 RET
.SET VM SIZE=0 RET
.INSTALL VM RET
```

After you configure VM, using the previous commands, you can initialize VM and examine its directory to determine its size:

```
.INITIALIZE/NOQUERY VM: RET
.DIRECTORY VM: RET
```

The number of free blocks indicated by the DIRECTORY command are available for your working system. You can now proceed to Section 11.4.

11.3.2 Configuring a VM Device for the Mapped Monitors

This section assumes you are running a mapped monitor.

NOTE

Functionally, configuring a VM device for the XM or ZM monitor is the same. Rather than continuously mention each monitor file and the appropriate start-up command file separately, this chapter will use the construction *x*M to represent both monitors, the construction RT11*x*M.SYS to represent both monitor files, and STRT*x*M.COM to represent both start-up command files. In each procedure, example, or illustration, use the form that is correct for the mapped monitor you want to support.

If you are not running the *x*M monitor, close any files you have open and issue the following commands:

```
.COPY/BOOT SY:RT11xM.SYS SY: RET
.BOOT SY: RET
```

The *x*M monitor automatically assigns a region in extended memory to VM. You can see that region by issuing the following command. Be ready to press the NO SCROLL or HOLD SCREEN key to stop the display as it scrolls up your terminal screen.

```
.SHOW MEMORY RET
```

Once you have seen that VM occupies a region in extended memory, again press the NO SCROLL or HOLD SCREEN key to let the display scroll to the end.

To determine the number of blocks available with the default VM region size, issue the following commands:

```
.INITIALIZE/NOQUERY VM: RET
.DIRECTORY VM: RET
```

The size in blocks of the default VM device is determined by the amount of extended memory in your computer. Unless your computer contains significantly more than 512K bytes of memory, you must lower the base address of the VM region in extended memory to enlarge VM sufficiently to create enough blocks to hold your working system.

You should also leave enough extended memory to run any jobs that require it. Using the following values with a computer containing 512K bytes of memory will create a VM device of 686 blocks and leave approximately 50K words of unused extended memory. That VM device is large enough to hold a functional working system and leaves enough unused extended memory to run at least KEX, VTCOM, and SPOOL as system jobs.

To configure the base address of the VM device for those characteristics, issue the following commands:

```
.UNLOAD VM RET
.REMOVE VM RET
.SET VM SIZE=0 RET
.SET VM BASE=5000 RET
.INSTALL VM RET
```

Those commands set the size of the VM device to its default placement in memory; that is, from its base address to the top of available memory. Those commands also set the base address of VM at 500000(octal), and VM resides at that address when it is loaded in extended memory.

To initialize VM at its new size and examine its directory, issue the following commands:

```
.INITIALIZE/NOQUERY VM: RET
.DIRECTORY VM: RET
```

11.4 Determining Your Working System

The VM device is generally smaller (contains far fewer blocks) than the physical disk you use as your system device. Therefore, you must be selective about what files you include in the working system you copy to VM.

You can think of your working system as containing two types of files: mandatory files and optional files. The mandatory files are the minimum files required to run your computer. Optional files are those files you use to do your work; they can change as the nature of your work changes.

Your working system must include at least the following mandatory files:

- SWAP.SYS
- An RT–11 monitor (RT11FB.SYS, RT11XM.SYS, or RT11ZM.SYS)

- A VM handler (VM.SYS for unmapped, or VMX.SYS for mapped monitors)
- The disk device handler or handlers for your system, which include the device handler for your current system (SY) device and the default data (DK) device (those could be the same device handler).

  If you do not know the names of your system and default data device handlers, issue the following command to see which device handler is assigned SY and DK:

  ```
  .SHOW RET
  ```

  Throughout the rest of this chapter, *ddn* represents the disk device handler and unit number from which and to which you are copying files
- PIP.SAV
- DUP.SAV
- DIR.SAV

Your working system also contains optional files. They include any utilities you typically use and any device handlers used by those utilities, along with application programs. A useful exercise is to copy the required RT–11 files to your VM device and reexamine its directory. Note the number of free blocks. You can then better determine how many other utilities and device handlers VM will hold.

The following section describes how to create a particular working system. You should create this working system as an example — you may also find it a useful working system. Section 11.7 describes how to create your own working systems. Other working system examples are shown with the procedure for creating them.

11.5 Creating a Working System for the VM Device

This section details the steps necessary to create a model working system. The model assumes a PDP–11 computer system with the following characteristics:

- Contains 512K bytes of memory.
- Is running under the *x*M monitor.
- Is using *ddn* as the system (SY) and default data (DK) disk device handlers. (You should know your actual system and default data disk device handlers from the SHOW command you issued in the previous section.)
- Contains the utilities KEX, RESORC, SETUP, SL, SPOOL, VBGEXE, and VTCOM.

You may have to make some necessary device handler substitutions in the model, depending on your particular processor, such as PIX.SYS, DWX.SYS, and DZX.SYS instead of ddX.SYS, and XCX.SYS instead of XLX.SYS).

Digital recommends that you create and run the following VM working system as practice before you modify it (or create another) for your requirements.

Perform the following steps to create a working system for VM (each step is described further in this section):

1. Create a minimum functionality start-up command file (STRT*x*M.COM) for the device your computer first boots.
2. Create a start-up command file that configures your working system on VM. Name that file STRTVM.COM.
3. Create an indirect command file that initializes VM and copies your working system to VM. Name that file VM.COM.

11.5.1 Create the Initial Start-Up Command File

STRT*x*M.COM is the file your computer uses when you first boot it. Assuming you will be running your computer from VM, you should edit the start-up file, STRT*x*M.COM, for minimum functionality. The following procedure saves your current STRT*x*M.COM file by copying it to a file named STRT*x*M.DIS, so you will not destroy the contents of that file if you have already edited it and want to keep that edited file:

1. Issue the following commands to save your current STRT*x*M.COM file and to make another copy you will use in the following section:

   ```
   .COPY SY:STRTxM.COM STRTxM.DIS  RET
   .COPY SY:STRTxM.COM STARTY.COM  RET
   ```

2. Edit STRT*x*M.COM to resemble the following example. The example assumes your hardware configuration includes a serial interface printer using the LS handler.

   ```
   ASS LS LP
   SET LS ENDPAG=1,NOFORM0
   RUN SETUP
   VT100,JUMP,LIGHT,CLEAR
   ```

   That example sets some printer and terminal characteristics. Those characteristics may not be the ones you use. Also, you may have to include others that set, for example, the CSR and vector addresses for your printer. You can set any printer and terminal characteristics you want. See the *RT–11 Commands Manual* for an explanation of SET and SETUP commands. Include those commands you want.

11.5.2 Create a VM Start-Up File (STRTVM.COM)

You have already edited the start-up command file for the device your computer initially boots—STRT*x*M.COM. You must now create a start-up command file, STRT-VM.COM, for the VM device. STRTVM.COM will be copied to VM and renamed STRT*x*M.COM by the indirect command file VM.COM, as explained in the next section.

RT–11 runs the file STRTVM.COM when you boot VM. Therefore, STRTVM.COM should contain the start-up commands you normally use in your working system. This section includes a start-up command file suitable for this working system.

You create STRTVM.COM by using the following command:

```
.COPY STARTY.COM STRTVM.COM RET
```

When editing STRTVM.COM:

- Be sure to delete the command, INITIALIZE/NOQUERY VM:, from STRTVM.COM.
- Your physical disk is no longer your system device and, therefore, is no longer automatically loaded. Any device handler used by a system job (such as KEX, VTCOM, or SPOOL) must be loaded. Be sure to load your physical disk device handler if you are going to be using system jobs.
- You should assign a default data device (probably your physical disk).

Provision for those points is made in the following sample VM start-up command file. This file performs the following functions:

- Turns on the SL editor.
- Runs VTCOM as a system job.
- Loads *dd*. The symbol *dd* represents the handler for your default data device.
- Assigns your default data device (*ddn*) as the SPOOL work file (SFD) device.

  If you do not assign SFD to your default data device, SPOOL uses up to 1000 blocks of VM for SFD. As a result, although a DIRECTORY command displays free blocks on VM, those free blocks are not available — they are assigned to SFD.
- Runs SPOOL as a system job. Flagpage support is disabled, and the spooler is set to add a form feed to the final page of each file you spool.
- Sets terminal hardware characteristics.
- Assigns *ddn* as the default data device.
- Shows the memory listing and the date.

Use the KEX editor to open the file STRTVM.COM, and edit it to appear as follows or as you prefer:

```
! STRTVM.COM, created dd-mmm-yy, edited dd-mmm-yy
! Start-up command file for VM for Intro general example
!
SET SL RECALL,KED,KMON
!
SRUN SY:VTCOM.SAV/PAUSE
LOAD XL=VTCOM
RESUME VTCOM
!
LOAD dd
ASS ddn SFD
SRUN SPOOL.SAV/PAUSE
SET SP FLAG=0,ENDPAG=1
LOAD LS=SPOOL
```

```
RESUME SPOOL
ASS SP0 LP
ASS SP0 LP0
ASS SP0 LS
ASS SP0 LS0
!
SETUP VT100,JUMP,LIGHT,CLEAR
ASS ddn DK
SHOW MEMORY
DATE
```

11.5.3 Create VM.COM to Initialize and Set Up VM

Use the KEX editor to create an indirect command file named VM.COM. VM.COM initializes the VM device and copies your working system to that device. The mandatory files for this working system are (*ddX.SYS* is required to communicate with disk device *ddn*):

```
SWAP.SYS
RT11xM.SYS
VMX.SYS
ddX.SYS
PIP.SAV
DUP.SAV
DIR.SAV
```

The optional files for this working system are:

```
LDX.SYS
LSX.SYS
SLX.SYS
SPX.SYS
XLX.SYS
KEX.SAV
RESORC.SAV
SETUP.SAV
SPOOL.SAV
STRTVM.COM
VBGEXE.SAV
VTCOM.SAV
```

The following indirect command file copies those system (.SYS) files and utilities to VM. In the following commands, *ddn* represents the physical device and unit number from which you are copying the files, and *ddX* represents your disk device handler. The last line copies the VM device start-up command file, STRTVM.COM, to the VM device and renames it STRT*x*M.COM, which lets VM find its start-up command file when you boot VM.

Use the KEX editor to create the file VM.COM. Enter the following commands in that file:

```
! VM.COM, created dd-mmm-yy, edited dd-mmm-yy
! Configures VM working system for Intro general example
!
UNLOAD VM
REMOVE VM
SET VM SIZE=0
SET VM BASE=5000
INSTALL VM
LOAD VM
INIT/NOQ VM:
COPY/SYS ddn:SWAP.SYS VM:
COPY/SYS ddn:RT11xM.SYS  VM:
COPY/SYS ddn:(VMX,ddX,XLX,LSX).SYS VM:
COPY/SYS ddn:(LDX,SLX,SPX).SYS VM:
COPY ddn:(DUP,DIR,PIP,KEX,RESORC,SETUP).SAV VM:
COPY ddn:(VBGEXE,VTCOM,SPOOL).SAV VM:
COPY ddn:STRTVM.COM VM:STRTxM.COM
COPY/BOOT VM:RT11xM.SYS  VM:
```

Note that the following commands, with various base address values, are included in all the command files that copy working systems to VM. Once you have changed the base address of VM, you must boot a VM with that base address. Therefore, the commands should be included in command files that copy working systems to VM to avoid any conflict in VM base addresses.

```
UNLOAD VM
REMOVE VM
SET VM SIZE=0
SET VM BASE=5000
INSTALL VM
LOAD VM
```

11.6 Running RT–11 from the VM Device

This section assumes you have done the steps described in Section 11.5.1 through Section 11.5.3. Use the following steps to run RT–11 from your VM device:

1. If you are now running your computer, close any files you have open. Boot your computer:

   ```
   .BOOT SY: RET
   ```

   Your computer executes the commands contained in the start-up file STRT*x*M.COM.

2. After your computer has booted, issue the following command:

   ```
   .$@VM RET
   ```

   That command runs the indirect command file VM.COM. (It is recommended that you specify the dollar sign ($) because VM.COM is an indirect command file and not an IND control file.) If VM.COM does not run to completion (execute all commands in the file), repeat Steps 1 and 2. You may have previously assigned the size of the VM device (SET VM SIZE=*nnnnn* command) to some nonzero value, causing VM.COM to abort for any number of reasons. You should not have this problem again unless you assign the size of the VM device to some nonzero value at a later time. If you do that, you will have to repeat this correction.

3. When you receive the monitor prompt (.), boot the VM device, as follows:

   ```
   .BOOT VM: RET
   ```

   After you have booted VM, examine the directory, as follows:

   ```
   .DIR SY: RET
   ```

   If your computer contains 512K bytes of memory and the size of the VM directory is not 686 blocks (combined used and free blocks), you probably have some variation of the problem described in Step 2. Repeating Steps 1, 2, and 3 should correct the problem.

You are now running the *x*M monitor from your VM system device. Running your working system from VM is different than running from your hard disk in at least two ways:

- VM is faster than your hard disk.
- The system device is smaller and, therefore, contains fewer utilities. That difference is not normally a problem, as the device is large enough to contain the set of utilities you use when doing particular work, such as writing and editing or programming. If you do more than one type of work with your computer, you may want to create more than one working system for your VM device. See the following section for a discussion of multiple VM system devices.

11.7 Creating Multiple Working Systems for the VM Device

Different working systems have different uses. A working system containing the utilities KEX, VTCOM, and SPOOL is appropriate for text editing, maintaining communications with a host computer, and so forth. That working system, however, would not be appropriate for writing and debugging programs.

To take advantage of the increased performance you attain by running RT–11 from the VM device and to perform more than one type of work with your computer, create multiple working systems for the VM device, as follows:

1. Determine the functionality you require; that is, the utilities and application programs you want to run.
2. Determine how many blocks those files need on the VM device. Remember to include the mandatory set of system (.SYS) files and utilities that RT–11 requires to run, which are listed in Section 11.5.3. If you have a diskette drive as part of your hardware configuration, a useful exercise is to copy all the files to an initialized diskette, then boot that diskette and examine its size.
3. Determine the base address for the VM device for that working system. Setting the base indicated below produces a VM device with the indicated number of blocks on computers containing 512K bytes of memory.

Base	Number of Blocks
7000	558
6000	622
5000	686
4000	750
3000	814

Remember that you have a finite amount of extended memory. Therefore, do not create a VM device that is larger than you need. (The larger the VM device, the smaller the amount of remaining extended memory.) You do not need to specify the exact values given above; the value 4200 is valid, for example.

4. Create a start-up command file.
5. Create an indirect command file to set the VM device base, initialize VM, and copy your files to VM.

The following three sections describe creating and running different types of working systems for the VM device.

11.7.1 A MACRO Programmer's Working System

The following indirect command files can be appropriate for creating a working system for writing, assembling, linking, and debugging MACRO programs. The working system assumes a computer containing 512K bytes of memory. The base address of VM is changed to 4000, creating a 750-block VM device, to provide enough free blocks for the MACRO and LINK work files. To create a MACRO programmer's working system:

1. Use the KEX editor to create a start-up command file and name it STARTP.COM. Include in that file all commands you want executed when you boot this working system. The following file may be appropriate:

```
! STARTP.COM, created dd-mmm-yy, edited dd-mmm-yy
! Start-up command file for Intro Programmer's working VM system device
!
ASS LS LP
ASS VM0 CF
ASS VM0 WF
ASS ddn DK  ! ddn represents your actual disk handler and unit number
LOAD dd     ! dd represents your actual disk handler
RUN SETUP
VT100,JUMP,LIGHT,CLEAR
RUN RESORC
/X
^C
```

2. Use the KEX editor to create the following indirect command file and name it VMPRG.COM. In the following commands, *ddn* represents the disk device handler and unit number from which you are copying the files, and *ddX* represents your disk device handler. Include the following commands in that file:

   ```
   ! VMPRG.COM, created dd-mmm-yy, edited dd-mmm-yy
   ! Configures VM for Intro Programmer's example
   !
   UNLOAD VM
   REMOVE VM
   SET VM SIZE=0
   SET VM BASE=4000
   INSTALL VM
   LOAD VM
   INIT/NOQ VM:
   COPY/SYS ddn:(SWAP,RT11xM,VMX,ddX).SYS VM:
   COPY/SYS ddn:(LSX,SDSX).SYS VM:
   COPY ddn:(DUP,PIP,DIR,MACRO,LINK).SAV VM:
   COPY ddn:(KEX,CREF,DBGSYM).SAV VM:
   COPY ddn:(VDT.OBJ,SYSLIB.OBJ,SYSMAC.SML) VM:
   COPY ddn:STARTP.COM VM:STRTxM.COM
   COPY/BOOT VM:RT11xM.SYS  VM:
   ```

Then, to use the VM device for writing, assembling, linking, and debugging MACRO programs:

1. Close any files you have open.
2. If you are now running on your physical disk, issue the following commands.

   ```
   .$@VMPRG RET
   .BOOT VM: RET
   ```

   If you are now running on your VM device, issue the following commands. In the first command, *ddn* represents your disk system device handler and unit number.

   ```
   .BOOT ddn: RET
   .$@VMPRG RET
   .BOOT VM: RET
   ```

11.7.2 A Working System for Performing a SYSGEN

You can decrease the time required to perform a SYSGEN by running RT–11 from the VM device. Use the following procedure:

1. Use the KEX editor to create a start-up command file named STARTG.COM. Include the following command lines in that file:

```
! STARTG.COM, created dd-mmm-yy, edited dd-mmm-yy
! Start-up command file for VM SYSGEN working system
!
ASS LS LP
SET LS ENDPAG=1,NOFORM0
ASS ddn DK  ! ddn represents your default data disk handler and unit number
LOAD dd     ! dd represents your default data disk handler
ASS VM0 CF
ASS VM0 WF
RUN SETUP
VT100,JUMP,LIGHT,CLEAR
RUN RESORC
/X
^C
```

2. Use the KEX editor to create an indirect command file named VMGEN.COM. The base of VM is set to 4000, giving a VM device size of 750 blocks. In the following commands, *ddn* represents the disk device handler and unit number from which you are copying the files, and *ddX* represents your disk device handler. Include in that file the following command lines:

```
! VMGEN.COM, created dd-mmm-yy, edited dd-mmm-yy
! Configures VM for Intro SYSGEN working system
!
UNLOAD VM
REMOVE VM
SET VM SIZE=0
SET VM BASE=4000
INSTALL VM
LOAD VM
INIT/NOQ VM:
COPY/SYS ddn:(SWAP,RT11xM,VMX,ddX).SYS VM:
COPY/SYS ddn:LSX.SYS VM:
COPY ddn:(DUP,PIP,DIR,MACRO,LINK).SAV VM:
COPY ddn:(IND,MONMRG).SAV VM:
COPY ddn:(SYSMAC.SML,SYSLIB.OBJ) VM:
COPY ddn:STARTG.COM VM:STRTxM.COM
COPY/BOOT VM:RT11xM.SYS  VM:
```

To perform the SYSGEN procedure while running RT–11 from your VM device:

1. Close any files you have open.
2. If you are now running on your physical disk, issue the following commands:

```
.$@VMGEN  RET
.BOOT VM:  RET
```

If you are now running on your VM device, issue the following commands. In the first command, *ddn* represents your system disk device handler and unit number.

```
.BOOT ddn:  RET
.$@VMGEN  RET
.BOOT VM:  RET
```

3. Once you have booted the VM device, start the SYSGEN procedure:

   ```
   .RUN SY:IND.SAV ddn:SYSGEN.COM RET
   ```

   Perform the procedure as documented in the *RT–11 System Generation Guide*. At the end of the SYSGEN dialog, be sure to specify your physical disk (*ddn*) as the device that contains the source input files and as the device to which you want to send all output files.

11.7.3 A Minimal Working System

The following working system contains only those files necessary to boot the VM device. The base address of VM is set to 5000, giving a device size of 686 blocks. The procedure uses the minimum functionality STRT*x*M.COM start-up command file you created in Section 11.5.1.

You then copy programs to the VM device and thereby increase the performance attainable by those programs. To create a general-purpose working system:

1. You have already created the minimum functionality start-up command file, STRT*x*M.COM.
2. Use the KEX editor to create the indirect command file, VMMIN.COM. In the following commands, *ddn* represents the disk device handler and unit number from which you are copying the files, and *ddX* represents your disk device handler. Include the following commands in that file:

   ```
   ! VMMIN.COM, created dd-mmm-yy, edited dd-mmm-yy
   ! Configures VM for Intro minimal working system
   !
   UNLOAD VM
   REMOVE VM
   SET VM SIZE=0
   SET VM BASE=5000
   INSTALL VM
   LOAD VM
   INIT/NOQ VM:
   COPY/SYS ddn:(SWAP,RT11xM,VMX,ddX).SYS VM:
   COPY ddn:(DUP,PIP,DIR).SAV VM:
   COPY ddn:STRTxM.COM  VM:
   COPY/BOOT VM:RT11xM.SYS  VM:
   ```

Then, to use the VM device as a high-speed general-purpose working system:

1. Close any files you have open.
2. If you are now running on your physical disk, issue the following commands:

   ```
   .$@VMMIN RET
   .BOOT VM: RET
   ```

If you are now running on your VM device, issue the following commands. In the first command, *ddn* represents your disk system device handler and unit number.

```
.BOOT ddn: RET
.$@VMMIN RET
.BOOT VM: RET
```

3. Once you have booted the VM device, copy any programs you want to run to it. Then, run those programs from the VM device.

Chapter 12

Using the RT–11 Communications Facilities

This chapter describes using the communications facilities distributed with RT–11. As distributed, RT–11 provides the software necessary for establishing and maintaining communications between one computer system running RT–11 and another running RT–11, RSX–11, or VMS.

This chapter describes:

- What constitutes communications between computers and the fundamental hardware required. It then describes the software components distributed with RT–11 that support that fundamental hardware.
- How to configure the components to form a successful communications chain.
- How to establish communications between two computers.
- How to use the communications functionality. It describes, for example, how to capture (log) information from the host computer and how to transfer files.

If you intend to use the RT–11 communications facilities, you should read all the first three sections and those parts of the last section that apply to your particular communications configuration.

Use the on-line index utility, INDEX, and see the on-line help utility, HELP, for more information on topics described in this chapter.

NOTE

In this chapter, the phrase *transfer files between* one computer and another means to transfer files *in either direction*.

The term *communications port* means the port on a computer to which you attach the transmission medium for connection with another computer. That port may or may not be marked as a communications port and may in fact be a console port or other serial port.

The term *data* can mean code, control sequences, or text data. Functionally, in this chapter, these terms are interchangeable.

12.1 Components

Communication between two computers occurs whenever data is transferred between them. Data transfer occurs when information located at one computer is accessed by or transferred to another computer.

For communications to occur successfully, the computers must be connected using a consistent hardware and software chain so that data passing between them is understood. That requires compatible hardware and software components.

It is beyond the scope of this chapter to discuss specific communications hardware. However, in general, the computer communications hardware chain consists of:

- Two computers at opposite ends of the communications chain.
- A communications port controller on each computer that converts data for transmission and reception.
- A transmission medium that passes serial data between the communications ports.
- Other optional hardware that manages the transmission medium, such as a modem or switching device.

The relationship of those hardware components to communications software components is described in the following sections.

12.1.1 Local and Host Computers

In software terms, the two computers that communicate with each other are called the *local* computer and the *host* computer.

Local computer

The local computer is that computer to which your console terminal is attached; the computer that is local to you and running RT–11. The RT–11 software component that manages communications on your local computer is VTCOM. VTCOM is the RT–11 *virtual terminal communications* utility.

VTCOM is supported under only the mapped and FB monitors; you cannot run VTCOM with the distributed SB monitor. If, for some reason, you require support for VTCOM under an SB monitor, you must perform a system generation procedure and build an SB monitor with timer support. You also must use a VTCOM.REL file under SB that has never been run under FB.

VTCOM provides you with the following functionality:

- Lets your local computer behave as a terminal that is directly connected to the host computer. As such, VTCOM lets you log onto the host computer and use all the software and hardware facilities available on that computer.
- Manages all file operations on your local computer when you transfer files between your local computer and the host.

- Lets you open a logging file on your local computer. That logging file records everything transmitted from the host computer to your local computer's terminal screen.
- Supports some commands and customizations that let you send, for example, a particular dial string to your modem.

Host computer

The host computer is that computer with which you want to communicate; the computer at the other end of the transmission medium. The host computer can be running RT–11, RSX–11, or VMS.

If possible, you should install the appropriate version of the RT–11 distributed component, TRANSF, on the host computer. TRANSF is the *native mode transfer* utility. You can then run TRANSF from the host computer to transfer files between the local and host computers.

TRANSF has the following capabilities:

- Lets you specify options to send files formatted as ASCII, BINARY, or FORTRAN between the hosts and local computers.
- Performs error checking on all file transfers; you are ensured of a reliable transfer.
- Supports command options that can produce a log of statistical information at the completion of the transfer. You can also specify a /PROGRESS option that displays transfer statistics as the transfer occurs.
- Decreases the size of each transfer segment automatically when it encounters line-noise or other interference, and increases the segment size automatically when the line clears. Therefore, the speed of the transmission is optimized.

RT–11 distributes specific versions of TRANSF that run in native mode on supported host computers running the RT–11, RSX–11, or VMS operating systems.

12.1.2 Communications Port Controllers

RT–11 is distributed with device handlers that manage the communications port controller on the local computer. For CTI Bus-based local computers, that device handler is XC. For Q-bus and UNIBUS local computers, that device handler is XL which supports the DL11 and DVL11 port controllers.

The communications port controller on the host computer is managed by software installed on that computer.

A communications port controller converts data that is contained within the transmitting computer to a form that is suitable for transmission. At the other end of the transmission medium, a communications port controller converts data from the form in which it was transmitted to a form that is usable by the computer receiving that data.

Port Addressing The local computer communications port controller and communications device handler pass information to each other using two addresses, the *CSR* and *vector* addresses.

Baud Rate The *baud rate* is the speed at which the communications ports send and receive data. The baud rate must be compatible throughout the communications chain. If one link in the chain can function only at a particular baud rate, all other links in the chain must function at, or support, that baud rate.

12.1.3 Transmission Medium

The transmission medium is not controlled by any software distributed with RT–11. Using distributed RT–11 software, you can only transmit data by using one type of transmission medium—the serial line. However, a serial line can connect the communications ports of two computers in two ways:

- **Unconverted Signal —- Hardwired Serial Line Connections**

  When computers are relatively close together, they can communicate through a hardwire connection between the communications port of the local computer and a console port on the host, using a single section of serial line cabling. That type of connection is the fastest way of transmitting data between computers when using the distributed RT–11 software. Depending on the length of the serial line cabling, the power of the computers involved, and the current work load of those computers, you can often transmit data at the highest baud rates supported by the computer port controllers.

- **Converted Signal —- Telephone Connections Using a Modem**

  When computers are far enough from each other that hardwiring is not possible, they communicate using a modem and telephone lines. That type of connection is considerably slower than hardwiring because of the conversion performed by the modem and the nature of telephone lines in general. With this type of connection, the speed of the transmission medium usually determines the baud rate for the communications chain.

12.1.4 Optional Hardware

Other optional hardware can manage the transmission medium. Such hardware could be a modem or switching device.

RT–11 distributes no software component that directly controls modems or switching devices. RT–11 does provide some commands and customizations that let you pass information, such as a dial string, to some modems.

12.2 Configuring the Components

The two computers for which you want to establish communications must be connected so that both the hardware and software components form a consistent chain.

12.2.1 Local Computer

Make sure that the correct version of VTCOM is located on your local computer's system (SY) device. RT–11 distributes two versions of VTCOM. Which version of VTCOM you use is determined by the RT–11 monitor you are running at your local computer:

RT–11 Monitor	Version of VTCOM
XM or ZM	VTCOM.SAV
FB	VTCOM.REL

12.2.2 Host Computer

Install the correct version of TRANSF on the host computer. RT–11 distributes the following versions of TRANSF for the following operating systems:

Version	Host Computer Operating System
TRANSF.SAV	RT–11
TRANSF.TSK	RSX–11
TRANSF.EXE	VMS

If your host computer is a PDP–11 running RT–11, make sure TRANSF.SAV is located on the host system (SY) device.

If your host computer is a PDP–11 running RSX–11, have the host computer system manager install TRANSF.TSK. Have your host system manager read the TRANSFER/TRANSF chapter of the *RT–11 System Utilities Manual*.

If your host computer is a VAX running VMS, have the host computer system manager install TRANSF.EXE. Have your host system manager read the TRANSFER/TRANSF chapter of the *RT–11 System Utilities Manual*.

NOTE

You do not need to install TRANSF on the host to initiate and maintain communications with that host. You install TRANSF on the host to perform error-free file transfers between the host and your local computer. If it is not possible for you to install TRANSF on a host computer, you can still communicate with that host computer and use its facilities.

12.2.3 Local Computer Communications Port Controller

Because the communications port controller provides the connection between the computer and the transmission medium, you must configure the port controller with both the computer's communications device handler and the transmission medium.

Communications Port Controller — Communications Device Handler

Perform the following steps to configure the communications port controller with the communications device handler:

- Make sure that the correct communications device handler is located on your local computer's system (SY) device. The following table shows the correct communications device handler file for each computer and monitor combination:

Computer (Monitor)	Required Communications Device Handler on SY
CTI Bus-based	XCX.SYS
CTI Bus-based (FB)	XC.SYS
PDP–11 (XM or ZM)	XLX.SYS
PDP–11 (FB)	XL.SYS

- Make sure the CSR and vector addresses used by the local computer's communications port are known to the communications device handler.

  You configure the port controller with the computer by setting the port controller device handler to recognize the addresses that controller uses to receive data from and send data to that computer.

  The CSR and vector address are fixed and known on CTI Bus-based computers and do not require your attention.

  If your computer is a Q-bus or UNIBUS PDP–11, do the following:

  1. Find out which CSR and vector address your communications port controller uses. You can find that information in the User Guide supplied with that controller or with your computer system documentation.
  2. Issue the SHOW DEVICE command:

     ```
     .SHOW DEVICE  RET
     ```

     That command displays the CSR and vector addresses for various device handlers on your computer system. Verify that the displayed CSR and vector address for XL match those used by the communications port controller.
  3. If either or both of the displayed addresses do not match those listed for your communications port controller, issue the following commands to match the device handler addresses with the listed port controller addresses. In the commands, the symbols *nnnnnn* and *nnn* represent respectively the correct CSR and vector addresses listed in the communications port documentation:

```
    .UNLOAD XL RET
    .SET XL CSR=nnnnnn RET
    .SET XL VECTOR=nnn RET
```

Communications Port Controller — Transmission Medium

You configure the port controller with the transmission medium by specifying the speed (baud rate) the controller should use to send data over the medium.

Make sure the transmission speed (baud rate) you set for the communications device handler matches the transmission speed of the entire communications chain.

On CTI Bus-based computers, use the following SET command, where *n* represents the baud rate, to specify the transmission speed:

```
.SET XC SPEED=n RET
```

On Q-bus and UNIBUS PDP–11 computers, set the baud rate directly on the communications port controller.

12.2.4 Transmission Medium

If the transmission medium is a hardwired serial line, connect that line between a communications port on your local computer and a console port on the host computer.

If the transmission medium is a telephone line using a modem or routed through a switching device, be sure the modem or switching device is configured for the baud rate you set for your local computer's communications handler. (In fact, a telephone transmission line will probably limit the baud rate.)

12.3 Establishing Communications

Once you have configured the communications chain, you can establish communications between the computers. Establishing communications involves starting VTCOM on your local computer and from your local computer, logging in to the host computer.

12.3.1 Starting VTCOM

The first step in establishing communications between your local computer and the host is starting VTCOM on your local computer. The procedure you use to start VTCOM is determined by which RT–11 monitor you are running. If you are running a mapped monitor, you can start VTCOM as a system job or the foreground job. Typically, you would start VTCOM as a system job. If you are running the FB monitor, start VTCOM as the foreground job.

As a System Job under Mapped Monitors

Running VTCOM as a system job lets you keep access open to the system, foreground, and background environments and to perform other tasks while maintaining communications with the host. To start VTCOM as a system job, issue the following commands. To start VTCOM as a system job each time you boot your system, include the following commands in your start-up command file. (For local CTI Bus-based computers, replace XL with XC in the second command line.)

```
.SRUN VTCOM.SAV/PAUSE  RET
.LOAD XL=VTCOM  RET
.RESUME VTCOM  RET
```

As the Foreground Job under FB

Running VTCOM as the foreground job lets you keep access open to the background environment and perform other tasks while maintaining communications with the host. To start VTCOM as the foreground job, issue the following commands. To start VTCOM as the foreground job each time you boot your system, include the following commands in your start-up command file, STRTFB.COM. (For local CTI Bus-based computers, replace XL with XC in the second command line.)

```
.FRUN VTCOM.REL/PAUSE  RET
.LOAD XL=VTCOM  RET
.RESUME  RET
```

12.3.2 Connecting with the Host

Once you have started VTCOM on your local computer, how you make connection with the host computer is determined by the manner you used to start VTCOM.

VTCOM as a System Job

Perform the following procedure to connect with the host after starting VTCOM as a system job:

1. Press CTRL/X
2. In response to the system job environment prompt, type VTCOM and press RETURN:

   ```
   Job?VTCOM  RET
   ```

3. Log in to the host computer as you would from a directly attached console terminal.

VTCOM as the Foreground Job

Perform the following procedure to connect with the host after starting VTCOM as the foreground job:

1. Press CTRL/F
2. In response to the foreground job environment prompt, press RETURN:

   ```
   F>  RET
   ```

3. Log in to the host computer as you would from a directly attached console terminal.

12.4 Using Communications

Assuming you have performed the operations described in the previous two sections, you are now connected to a host computer. Connection to a host computer from your local computer can provide you the following functionality:

- You have access to all the facilities available on the host.

If the host is connected to other computers through a network system, you, too, have access to those other computers through your connection to the host.

- You can open a logging file on your local computer and preserve in that file everything that is displayed on your terminal screen that originates from the host.
- You can transfer files between your local computer and the host. If TRANSF is installed on the host computer, you can transfer any files reliably because of error checking. If TRANSF is not installed on the host, you can transfer only ASCII files, and those transfers are performed without error checking.

12.4.1 Host Facilities

You have access to all host facilities once you establish connection. There is no functional difference, other than perhaps speed, between being connected to the host through VTCOM or through a console terminal located at the host.

12.4.2 Capturing Information from the Host — The Logging Facility

VTCOM provides a logging file facility. You can open a logging file on your local computer and preserve in that file everything that is displayed on your terminal screen that originates from the host.

For example, you could log in to a timesharing host system that provides weather information. You could open a logging file on your local system, display the weather information on the host system, and then close (preserve) the logging file for examination after you logged out of the timesharing host system.

CAUTION

VTCOM does not check for the existence of files on your system that might match the logging file name.

Therefore, if a current file with the same name already exists, VTCOM writes over (destroys) the current file without warning. You must be careful to specify either a unique (new) logging file name or a current file that contains information you no longer want.

You will find it helpful to use a particular file type for all logging files you create to capture information from the host, such as .LOG. You can then easily obtain a directory of all logging files (DIR *.LOG) and then determine which file name you want to use.

Example

The following procedure shows how you might capture and preserve weather information located on a timesharing host system. You can use this example as a general procedure for logging other kinds of information from a host computer.

1. Establish connection with the host computer.

2. Decide on a name for the logging file. For the purpose of this example, assume you want to open the logging file WEATHR.LOG. Go to the background environment and display the directory of all files of type .LOG. If WEATHR.LOG exists, determine if you want the contents of that file.

   ```
   CTRL/B
   .DIR *.LOG  RET
   ```

   For the purpose of this example, assume WEATHR.LOG does not exist in your directory.

   Reestablish connection with VTCOM.

3. Open a logging file on your local computer:

   1. Press CTRL/P.

      VTCOM displays its command prompt:

      ```
      TT::VTCOM>
      ```

   2. Type OPENLOG and press RETURN.

      ```
      TT::VTCOM>OPENLOG  RET
      ```

   3. VTCOM prompts for the logging file name:

      ```
      TT::VTCOM> Log file?
      ```

   4. Type WEATHR.LOG and press RETURN.

      ```
      TT::VTCOM> Log file?WEATHR.LOG  RET
      ```

      From this point until you close the logging file, everything displayed on your local computer terminal screen that originates from the host is logged in the file WEATHR.LOG.

   5. Press RETURN to display the host computer command prompt.

4. Access the weather information. You log only information that is displayed on your terminal screen.

5. Close the logging file:

   a. Press CTRL/P.

      VTCOM displays its command prompt:

      ```
      TT::VTCOM>
      ```

   b. Type CLOSELOG and press RETURN.

      ```
      TT::VTCOM>CLOSELOG  RET
      ```

6. You can then continue connection with the timesharing host system, open another logging file, or disconnect from the host.

7. You now have a logging file named WEATHR.LOG on your local computer. You can edit that file, type it, or print it.

In this manner, you can transfer any ASCII information located on a host computer to your local computer.

Although it is possible to transfer all or a portion of an ASCII file from a host to your local computer by using this procedure, you should, if possible, use TRANSF to transfer files. The VTCOM logging file functionality performs no error checking. Use the VTCOM logging file functionality only when the information you want to capture is not a file or you cannot install TRANSF on the host.

12.4.3 Transferring Files

As described in the previous section, you can transfer the contents of any file between your local computer and the host computer if the TRANSF utility is installed on the host computer.

If it is not possible to install TRANSF on the host, you can transfer only ASCII files by using the VTCOM logging file functionality. That functionality does not provide error checking and should be used only if you cannot install TRANSF on the host computer.

The following two sections describe transferring files when TRANSF is installed on the host computer and when it is not.

12.4.3.1 With TRANSF Installed on the Host

With TRANSF installed on the host computer, you can transfer any kind of file between your local computer and the host. Although you use TRANSF to transfer files in either direction, you always run TRANSF from the host computer.

Each version of TRANSF is written to run in native mode on the host computer's operating sytem. Therefore, this section is divided into descriptions of using TRANSF with the three supported operating system combinations.

12.4.3.1.1 With a VMS Host

A VMS host has a different file structure than your local computer. The difference in file structure characteristics has two ramifications:

- A VMS host supports a file specification with file names longer than six characters and file types longer than three characters. When you transfer a file that has an invalid (too large) file specification from the host to your local computer, you must supply a valid local computer file specification. Otherwise, TRANSF truncates the host file specification to make it valid on your local computer.
- The difference in file structures requires TRANSF to perform file format translations. TRANSF provides options you use to specify the type of file you are transferring.

The following is the general command syntax you issue from a VMS host to transfer files:

```
$ TRANSF infile.ext/options outfil.ext/options RET
```

In general, you should ask the following four questions to determine which options to use:

- Where is the file I want to transfer; where is *infile.ext*?

  If the file is currently on your local computer, you must indicate that by including the /TERMINAL option with *infile.ext*. The /TERMINAL option tells TRANSF which file, *infile.ext* or *outfil.ext*, resides on the local computer file.

- Where is the file to be transferred; where is *outfil.ext*?

  If the file is to be transferred to your local computer, you must indicate that by including the /TERMINAL option with *outfil.ext*.

- What kind of file am I transferring?

  TRANSF recognizes a number of standard file types and performs the file transfer according to those types. However, you should generally specify the type of file format you are transferring. You tell TRANSF what format to use by specifying one of the following options:

Option	Meaning
/ASCII	This is an ASCII file transfer.
/BINARY	This is a BINARY object (.OBJ) code file transfer.
/FORTRAN	This file has FORTRAN carriage control.
/IMAGE	Transfer this file in IMAGE mode. IMAGE mode transfers are bit-for-bit image copies.

  Any file type not recognized by TRANSF is transferred in IMAGE mode, possibly producing unexpected results. Use the /HELP qualifier to display those file types recognized by TRANSF.

- Do I want any information about this transfer during or after the transfer completes?

  TRANSF provides options you can specify that cause additional information concerning the transfer to be displayed before the transfer, during the transfer operation or after the transfer completes. The following table lists useful information options:

Option	Meaning
/HELP	Displays limited information about TRANSF, including the default transfer mode for various file types.
/LOG	Displays a log of the transferred files at the end of the transfer. Very useful.

Option	Meaning
/PROGRESS[:n]	Displays the progress of the transfer as the transfer takes place. The value *n* specifies the message display interval. By default, the interval is every 10 blocks or records transferred. Useful if you have questions about the quality of the transmission medium or have experienced other recent problems with transfers.
/PROMPT	Places TRANSF in interactive mode. TRANSF then displays questions about the transfer, displays defaults, and prompts for your response. Useful as a beginning aid.
/SPOOL	Directs transfer output to the default host printer queue.
/STATISTICS	Displays useful information at the end of the transfer.

The following examples demonstrate types of transfers:

ASCII Transfers

The following command line, issued at the host, transfers the ASCII file LOCAL.TXT from your local computer to a file named HOST.TXT on the host. Note the placement of the /TERMINAL option. The command requests logging and statistical information at the end of the transfer:

```
$ TRANSF LOCAL.TXT/TERMINAL/ASCII/LOG/STATISTICS HOST.TXT RET
```

The following command line, issued at the host, transfers the ASCII file HOST.TXT from the host computer to a file named LOCAL.TXT on your local computer. Note the placement of the /TERMINAL option. The command requests logging and statistical information at the end of the transfer:

```
$ TRANSF HOST.TXT/ASCII/LOG/STATISTICS LOCAL.TXT/TERMINAL RET
```

The following command line, issued at the host, transfers the ASCII file LOCAL.TXT from your local computer to a file of the same name (LOCAL.TXT) on the host. Note the placement of the /TERMINAL option and the absence of *outfil.ext*. The command requests logging and statistical information at the end of the transfer:

```
$ TRANSF LOCAL.TXT/TERMINAL/ASCII/LOG/STATISTICS RET
```

BINARY and FORTRAN-Formatted Transfers

BINARY and FORTRAN-formatted files are transferred in the same manner as ASCII files, except you use the /BINARY or /FORTRAN format option in place of the /ASCII option.

IMAGE Transfers

The following command line, issued at the host, performs a bit-for-bit transfer of the file LOCAL.FOO from your local computer to a file named HOST.FOO on the host. Note the placement of the /TERMINAL option and the absence of any formatting option; an IMAGE mode transfer is done because the FOO file type is not recognized by TRANSF. The command requests logging and statistical information at the end of the transfer:

```
$ TRANSF LOCAL.FOO/TERMINAL/LOG/STATISTICS HOST.FOO RET
```

The following command line, issued at the host, performs a bit-for-bit transfer of the file HOST.FOO from the host computer to a file named LOCAL.FOO on your local computer. Note the placement of the /TERMINAL option and the absence of any formatting option; an IMAGE mode transfer is again done because the FOO file type is not recognized by TRANSF. The command requests logging and statistical information at the end of the transfer:

```
$ TRANSF HOST.FOO/LOG/STATISTICS LOCAL.FOO/TERMINAL RET
```

The *RT–11 System Release Notes* contains a complete description of the TRANSF utility for VMS host computers. You should also use the RT–11 on-line index facility, INDEX, to find the latest information about TRANSF.

12.4.3.1.2 With an RSX–11 Host

An RSX–11 host has a different file structure than your local computer. The difference in file structure characteristics produces two ramifications:

- An RSX–11 host supports a file specification with file names longer than six characters. When you transfer a file from the host to your local computer that has an invalid (too large) file specification, you must supply a valid local computer file specification. Otherwise, TRANSF truncates the host file specification to make it valid on your local computer.
- The difference in file structures requires TRANSF to perform file format translations. TRANSF provides options you use to specify the type of file you are transferring.

The following is the general command syntax you issue from an RSX–11 host to transfer files:

```
$ TRANSF infile.ext/options outfil.ext/options RET
```

In general, you should ask the following four questions to determine which options to use:

- Where is the file I want to transfer; where is *infile.ext*?

  If the file is currently on your local computer, you must indicate that by including the /TERMINAL option with *infile.ext*. The /TERMINAL option tells TRANSF which file, *infile.ext* or *outfil.ext*, resides on the local computer file.
- Where is the file to be transferred; where is *outfil.ext*?

  If the file is to be transferred to your local computer, you must indicate that by including the /TERMINAL option with *outfil.ext*.
- What kind of file am I transferring?

  TRANSF recognizes a number of standard file types and performs the file transfer according to those types. However, you should generally specify the type of file format you are transferring. You tell TRANSF what format to use by specifying one of the following options:

Option	Meaning
/ASCII	This is an ASCII file transfer.
/BINARY	This is a BINARY object (.OBJ) code file transfer.
/FORTRAN	This file has FORTRAN carriage control.
/IMAGE	Transfer this file in IMAGE mode. IMAGE mode transfers are bit-for-bit image copies.

Any file type not recognized by TRANSF is transferred in IMAGE mode, possibly producing unexpected results. Use the /HELP qualifier to display those file types recognized by TRANSF.

- Do I want any information about this transfer during or after the transfer completes?

  TRANSF provides options you can specify that cause additional information concerning the transfer to be displayed during the transfer operation or after the transfer completes. The following table lists useful information options:

Option	Meaning
/HELP	Displays limited information about TRANSF, including the default transfer mode for various file types.
/LOG	Displays a log of the transferred files at the end of the transfer. Very useful.
/PROGRESS[:n]	Displays the progress of the transfer as the transfer takes place. The value *n* specifies the message display interval. By default, the interval is every 10 blocks or records transferred. Useful if you have questions about the quality of the transmission medium or have experienced other recent problems with transfers.
/PROMPT	Places TRANSF in interactive mode. TRANSF then displays questions about the transfer, displays defaults, and prompts for your response. Useful as a beginning aid.
/STATISTICS	Displays useful information at the end of the transfer.

The following examples demonstrate types of transfers:

ASCII Transfers

The following command line, issued at the host, transfers the ASCII file LOCAL.TXT from your local computer to a file named HOST.TXT on the host. Note the placement of the /TERMINAL option. The command requests logging and statistical information at the end of the transfer:

```
$ TRANSF LOCAL.TXT/TERMINAL/ASCII/LOG/STATISTICS HOST.TXT  RET
```

The following command line, issued at the host, transfers the ASCII file HOST.TXT from the host computer to a file named LOCAL.TXT on your local computer. Note the placement of the /TERMINAL option. The command requests logging and statistical information at the end of the transfer:

```
$ TRANSF HOST.TXT/ASCII/LOG/STATISTICS LOCAL.TXT/TERMINAL  RET
```

The following command line, issued at the host, transfers the ASCII file LOCAL.TXT from your local computer to a file of the same name (LOCAL.TXT) on the host. Note the placement of the /TERMINAL option and the absence of *outfil.ext*. The command requests logging and statistical information at the end of the transfer:

```
$ TRANSF LOCAL.TXT/TERMINAL/ASCII/LOG/STATISTICS  RET
```

BINARY and FORTRAN-Formatted Transfers

You transfer BINARY and FORTRAN-formatted files in the same manner as ASCII files, except you use the /BINARY or /FORTRAN format option in place of the /ASCII option.

IMAGE Transfers

The following command line, issued at the host, performs a bit-for-bit transfer of the file LOCAL.FOO from your local computer to a file named HOST.FOO on the host. Note the placement of the /TERMINAL option and the absence of any formatting option; an IMAGE mode transfer is done because the FOO file type is not recognized by TRANSF. The command requests logging and statistical information at the end of the transfer:

```
$ TRANSF LOCAL.FOO/TERMINAL/LOG/STATISTICS HOST.FOO  RET
```

The following command line, issued at the host, performs a bit-for-bit transfer of the file HOST.FOO from the host computer to a file named LOCAL.FOO on your local computer. Note the placement of the /TERMINAL option and the absence of any formatting option; an IMAGE mode transfer is done because the FOO file type is not recognized by TRANSF. The command requests logging and statistical information at the end of the transfer:

```
$ TRANSF HOST.FOO/LOG/STATISTICS LOCAL.FOO/TERMINAL  RET
```

The *RT–11 System Release Notes* contains a complete description of the TRANSF utility for RSX–11 host computers. You should also use the RT–11 on-line index facility, INDEX, to find the latest information about TRANSF.

12.4.3.1.3 With an RT–11 Host

RT–11 host computers have the same file structure as your local computer. Therefore, TRANSF performs no format translation and you do not need to specify format options. All file transfers take place in image mode.

The following is the general command syntax you issue from an RT–11 host to transfer files:

```
.TRANSF infile.ext/options outfil.ext/options RET
```

In general, you should ask the following three questions to determine which options to use:

- Where is the file I want to transfer; where is *infile.ext*?

  If the file resides on your local computer, you must indicate that by including the /T option with *infile.ext*. The /T option tells TRANSF which file, *infile.ext* or *outfil.ext*, is the local computer file.

- Where is the file to be transferred; where is *outfil.ext*?

  If the file is to be transferred to your local computer, you must indicate that by including the /T option with *outfil.ext*.

- Do I want a log of statistical information after completion of the transfer?

  If you want statistical information, include the /W option.

Example Transfers

The following two examples illustrate transferring files between the host and your local computer.

The following command line, issued at the host, performs a bit-for-bit transfer of the file LOCAL.FOO from your local computer to a file named HOST.FOO on the host. Note the placement of the /T option. The command requests logging and statistical information at the end of the transfer:

```
.TRANSF LOCAL.FOO/T/W HOST.FOO RET
```

The following command line, issued at the host, performs a bit-for-bit transfer of the file HOST.FOO from the host computer to a file named LOCAL.FOO on your local computer. Note the placement of the /T option. The command requests logging and statistical information at the end of the transfer:

```
.TRANSF HOST.FOO/W LOCAL.FOO/T RET
```

The *RT–11 System Utilities Manual* contains a complete description of the TRANSF utility for RT–11 host computers. You should also use the RT–11 on-line index facility, INDEX, to find the latest information about TRANSF.

12.4.3.2 Without TRANSF Installed on the Host

When you cannot install TRANSF on the host computer, you can still transfer the contents of ASCII files between the local and host computers by using the VTCOM logging file and SEND. You use the logging file functionality to transfer ASCII files from the host to your local computer. You use SEND to transfer ASCII files from your local computer to the host.

CAUTION

The VTCOM logging file functionality and SEND command perform no error checking. You should verify the contents of any file transferred using this functionality; that file could contain errors.

Do not use this functionality to transfer critical information.

Transfers from the Host to the Local Computer

You use the VTCOM logging file functionality to transfer an ASCII file between the host and your local computer. The following example illustrates the general procedure you can use. The example assumes you want to transfer the contents of a ASCII file named HOST.TXT from the host to a file named LOCAL.TXT on your local computer.

1. Establish connection with the host computer.
2. At the host computer, type the following command (do not press RETURN):

   ```
   $ TYPE HOST.TXT
   ```

3. Press CTRL/P and open the logging file, LOCAL.TXT, on your local computer:

   ```
   TT::VTCOM>OPENLOG  RET
   TT::VTCOM> Log file?LOCAL.TXT  RET
   RET
   ```

   The last RETURN you press issues the command, TYPE HOST.TXT, on the host computer. That displays the contents of HOST.TXT on your terminal screen. Therefore, VTCOM logs the contents of HOST.TXT.

   Let the contents of HOST.TXT scroll on your screen.

4. When HOST.TXT has finished scrolling on your screen and you have been returned to the host monitor prompt, press CTRL/P and close the logging file:

   ```
   TT::VTCOM>CLOSELOG  RET
   RET
   ```

   The last pressing of RETURN returns you to the host monitor prompt.

5. You can now return to your local computer and edit the file, LOCAL.TXT, to remove any characters captured by the logging file that were not part of the file HOST.TXT.

If you want to abort the VTCOM logging file operation:

1. While the file you are typing is scrolling on the terminal screen, press the CTRL/C sequence, to abort this operation from the host.
2. Press CTRL/P.
3. In response to the VTCOM command prompt, type CLOSELOG:

   ```
   TT::VTCOM>CLOSELOG  RET
   ```

4. Optionally delete the partially transferred file on your local computer.

From the Local to the Host Computer

You use the VTCOM SEND facility to transfer an ASCII file from your local computer to the host. The VTCOM SEND facility performs no error checking on the file transfer.

By default, the VTCOM SEND facility transfers the contents of a local file to the host in SLOW mode. SLOW mode transfers the contents of a file very slowly.

VTCOM also supports FAST mode for the SEND facility. FAST mode transfers characters as quickly as possible. However, the host computer terminal service may not be able to correctly process characters in FAST mode. Symptoms of an overloaded terminal service are a beeping terminal bell and/or displayed error messages. If you attempt FAST mode and your terminal beeps and/or you receive error messages from the host, return to SLOW mode.

The following example illustrates the general procedure you can use to transfer an ASCII file from your local computer to the host by using the VTCOM SEND facility. The example procedure uses SLOW mode. The example assumes you want to transfer the contents of an ASCII file named LOCAL.TXT from your local computer to a file named HOST.TXT on the host.

1. Establish connection with the host computer.
2. At a VMS or RT–11 host computer, issue the following command:

   ```
   $ COPY TT: HOST.TXT RET
   ```

   At an RSX host computer, issue the following command:

   ```
   $ COPY TI: HOST.TXT RET
   ```

   In either case, the host cursor rests on the next line, waiting for input.
3. Press CTRL/P and respond in the following manner to the VTCOM prompts:

   ```
   TT::VTCOM>SEND RET
   TT::VTCOM> Send file?LOCAL.TXT RET
   ```

   You see the contents of LOCAL.TXT scroll slowly across your terminal screen.
4. When the complete file has scrolled on your terminal screen, press CTRL/Z to close the file on the host computer.

If you want to abort the file send operation, you must abort it from your local computer. You cannot use the CTRL/C sequence to abort this operation from the host. Use the following procedure:

1. Press CTRL/P.
2. In response to the VTCOM command prompt, type RESET:

   ```
   TT::VTCOM>RESET RET
   ```

3. Press CTRL/Z, and optionally delete the partially transferred file on the host computer.

Chapter 13
Defining Your Own Commands

RT–11 lets you define your own commands. You can define two basic categories of commands:

- You can define new commands.
- You can change the definition of existing commands.

RT–11 provides a command processor that can be used for both categories. You use the distributed utility, UCL.SAV, to define new commands. To change the definition of existing commands, you use the utility, UCF.SAV, which you create from UCL.

This chapter describes:

- The three methods you can use to define commands.
- How to define new commands with UCL.
- How to redefine existing commands with SYSGEN and UCL.
- How to redefine existing commands with UCF.

Defining new commands and changing the definition of (redefining) existing commands are different. It is a lot simpler to define new commands than it is to change the definition of existing commands. Also, because of the command processing order, changing the definition of existing commands imposes an increase in the time required to execute all commands. Therefore, if possible, you should define new commands to meet your needs rather than change existing command definitions.

Use the on-line index utility, INDEX, and see the on-line help utility, HELP, for more information on topics described in this chapter.

13.1 Available Methods

This chapter describes three methods you can use to define commands:

1. Define new commands by using the UCL command processor.

   UCL stands for *user command linkage*, which you can think of as user commands *last*. RT–11 checks that a command has valid UCL syntax only after checking first for DCL, CCL, and CSI valid syntax. Therefore, defining new commands as UCL commands imposes no overhead on processing other commands. Also, RT–11 distributes UCL.SAV; UCL is the simplest to implement and use. You should try to use this method first. Use one of the following two methods only if necessary.

2. Remove support for an existing DCL command using the system generation (SYSGEN) procedure and then use UCL to define that command as a new command.

   Once support is removed for a DCL command, that command can be treated as a new command and defined using UCL. This method only applies to redefining DCL commands and requires a SYSGEN. However, using this method imposes no overhead on command processing speed and is therefore preferable to the next method.

3. Change the definition of an existing command by creating and implementing the UCF command processor.

   The UCF command processor is a preprocessor that checks all commands you enter for valid user-defined syntax before checking for valid DCL, CCL, or UCL syntax. UCF is a utility that must be called into memory and run each time you enter any command. Therefore, creating and implementing UCF imposes noticeable overhead on command processing speed. You should create and implement UCF only if the first two methods cannot meet your requirements.

The remainder of this section describes in more detail what new and changed commands are and lists reasons why you might want to use UCL or UCF.

13.1.1 New Commands — UCL

You use UCL with commands you invent. That is, you cannot define a UCL command that conflicts with a command that is recognized by the RT–11 DCL or CCL command processors. For example, with any distributed RT–11 monitor, you cannot use UCL to define the command PRINT. PRINT is supported by the distributed DCL command processor. However, you can perform a SYSGEN and build a monitor without support for the PRINT command and then define PRINT as a UCL command.

You can invent commands that:

- Perform some function that is specific to an application program. You can invent a command set that is "customized" for an application.
- Issue a string of commands. You can invent a command that then issues a defined sequential group of commands.
- Execute one or more indirect command files or IND control files. You can invent a command that executes a specified file.

Using the distributed UCL is described in Section 13.2. Included in Section 13.2 is a procedure to optimize the speed of the UCL command processor.

Using the SYSGEN procedure to remove support for a DCL command and then using UCL to define that DCL command as a new command is described in Section 13.3.

13.1.2 Changed Commands — UCF

You can change the meaning of a command. If you create and enable UCF, RT–11 checks all commands for valid UCF syntax before checking the other command processors for valid syntax. Therefore, you can change the meaning of a distributed command by including the command in UCF.

You can use the UCF facility to:

- Process a standard keyboard command in a nonstandard manner.
- Process a standard keyboard command in a standard manner.
- Cause a standard keyboard command to be ignored by RT–11.
- Cause a nonstandard (invalid) keyboard command to be processed by RT–11 in the manner you want.

UCF command processing imposes overhead on other command processing because UCF is the first command processor. Any DCL, CCL, or UCL command you issue must first be checked for UCF syntax. Also, unlike the DCL and CCL command processors, the UCF command processor you create is not resident in the monitor and must be called by the monitor each time you issue a command. Thus, UCF imposes a further execution speed penalty on all non-UCF commands. Therefore, you should not create and enable UCF command processing unless a command you must define conflicts with other command processor syntax.

Creating and enabling UCF is described in Section 13.4. Included in Section 13.4 is a procedure to optimize the speed of UCF. You should optimize UCF to minimize the overhead that UCF imposes on other command processing.

13.2 Defining New Commands with UCL

You use the UCL command processor to define new commands. The two components to UCL are the utility UCL.SAV and the data file UCL.DAT. RT–11 distributes UCL.SAV and all distributed RT–11 monitors support UCL. The UCL data file, UCL.DAT, is created automatically by UCL.SAV as a result of defining the first UCL command.

The file UCL.SAV must reside on your system (SY) device. Issue the following command to determine if UCL resides on SY:

```
.DIR SY:UCL.SAV RET
```

If the file UCL.SAV does not reside on your system device, you must add it with the COPY command.

13.2.1 UCL Command Syntax

Once you have made sure the file UCL.SAV resides on your system device, you can create UCL commands. The first UCL command you create in turn creates the UCL data file. Use the following syntax to create UCL commands:

newcommand:==command[/options[\command/options...\\]]

where:

newcommand	Represents the UCL command you are defining. The UCL command can include up to 16 letters and numbers. Newcommand cannot contain the slash character (/); you cannot include command options in newcommand. You can define an abbreviation for newcommand by including an asterisk character (*). For example, the following UCL command line defines the newcommand STATUS and specifies the shortest valid abbreviation for that command as STAT: `.STAT*US:==SHOW ALL\SHOW QUEUE\\`
:==	Is the command definition separator, which separates the UCL command from those commands that define the UCL command.
command	Represents the UCL command definition. You can define a UCL command as: • One or more valid RT–11 commands (including valid options and option parameters) • One or more previously defined UCL commands • A modified IND or indirect command file calling syntax. After entering the UCL command to call the modified IND or indirect command file, you must then edit the UCL data file to change the command line modification. See Section 13.2.5 for information on using UCL to call IND or indirect command files.
\	Is the command separator you use if you include more than one command as the definition of the UCL command.
\\	Is the terminator you use if you define the UCL command as a string of more than one command.

You can include up to 128 characters in each UCL command definition.

13.2.2 Adding Information from the Command Line

Some information that you include in a command, such as a file specification, can change with each command. You plan for this kind of information change by including the UCL append symbol (^) in command definitions. The append symbol (^) lets you show UCL where to add information you supply in the command line.

For example, the following UCL command definition creates a command, KILL, that unprotects and deletes without confirmation the file specification you supply for the two append symbols:

```
.KIL*L:==UNPRO ^\DEL/NOQ ^\\ RET
```

When you then issue the KILL command, any text following that command is inserted in place of the append symbols when UCL processes the command. For example, the following UCL command would unprotect and then delete, without confirmation, all files on diskette DU2:

```
.KIL DU2:*.* RET
```

13.2.3 Displaying Command Definitions

The SHOW COMMANDS command displays UCL command definitions. The default output device for the SHOW COMMANDS display is your terminal. The following example shows a sample list of UCL commands:

```
.SHOW COMMANDS RET
!                          User Command Linkage (UCL)

      vm               :== $@vm\boot vm:\\
      vmprg            :== $@vmprg\boot vm:\\
      vmgen            :== $@vmgen\boot vm:\\
      vmmin            :== $@vmmin\boot vm:\\
      stat*us          :== show all\show queue\\
      kil*l            :== unpro ^\del/noq ^\\
```

You can also send the display to a file you specify, using the /OUTPUT:filespec option, or to your printer, using the /PRINTER option.

13.2.4 Redefining and Deleting UCL Commands

You redefine an existing UCL command by issuing a new command definition. After UCL processes and stores the new command definition, RT–11 returns the monitor prompt (.). You can then issue SHOW COMMANDS to verify the command redefinition.

You delete an existing UCL command by issuing a null definition, which consists of only the existing UCL command and command definition separator with no definition. For example, assume the existence of the following UCL command:

```
kil*l            :== unpro ^\del/noq ^\\
```

You could remove the UCL command, KILL, by issuing the following command:

```
.KILL:== RET
```

You use the SHOW COMMANDS command display to verify the removal of a UCL command.

13.2.5 Using UCL Commands to Run IND or Indirect Command Files

You cannot directly define a UCL command that runs one or more IND or indirect command files. For example, the following UCL command line fails to run the XM monitor start-up command file, STRTXM.COM, because the keyboard command processor (KMON) attempts to parse the $@ construction:

```
BEG*IN:==$@STRTXM
```

To run an indirect file from a UCL command, you must use a modified UCL command syntax and then edit the UCL data file. Rather than use the $@ construction, use the $$ construction:

```
BEG*IN:==$$STRTXM
```

Then, edit the UCL data file to change the second dollar sign ($) to the correct at sign (@).

Use the following practice exercise to go through the complete procedure:

1. Create an indirect command file named TEST1.COM:

   ```
   .EDIT TEST1.COM  RET
   ?KEX-W-File not found - Create it (Y,N)? Y  RET
   ```

2. Enter the following command lines in TEST1.COM :

   ```
   SHOW MEMORY  RET
   SHOW JOBS  RET
   SHOW CONFIG  RET
   ```

3. Close TEST1.COM with the EXIT command.

4. Enter the following UCL command line:

   ```
   .MULE:==$$TEST1  RET
   ```

5. Edit the UCL data file to change the $$TEST1 construction to $@TEST1:

   a. Open the UCL data file for editing:

      ```
      .EDIT SY:UCL.DAT  RET
      ```

   b. Press the following key sequence to search for $$, replace the second $ with the correct @, and exit from the editor:

      ```
      PF1  PF3
      Model:$$  4  3  ,  @
      PF1  7 Command: EXIT  Enter
      .
      ```

   c. You have now edited the UCL data file so that the command MULE runs the indirect command file TEST1.COM:

      ```
      .MULE  RET
      ```

      Assuming the utility RESORC.SAV resides on your system (SY) device, the command MULE runs TEST1.COM.

You can run an IND control file from a UCL command. The KMON SET conditional IND (SET KMON IND) must be in effect before that UCL command is parsed, but that conditional can be set from within UCL too. Instead of the $$ prefix before the IND control file, use the $ prefix in the UCL command definition, and then, edit the UCL data file to change the $ prefix to @. For example, the following example command sets the KMON conditional to IND and then runs (after UCL.DAT is edited) the IND control file WOOPS.COM:

```
.MUNGA:==SET KMON IND\$WOOPS\\  RET
```

You can run a string of indirect files with a single UCL command. Separate each indirect file by using the command separator (\ ):

```
.WOOGA:==$$LOOPS\$$SPOOL\\  RET
```

Then, edit the UCL data file to change the second dollar sign ($) prefix at each indirect command file to an at sign (@).

You can run indirect command files and an IND control file from the same UCL command so long as the IND control file is the last file specified in the UCL command definition.

13.2.6 Changing the Size and the Name of the UCL Data File

As distributed, the UCL data file created by the UCL utility can contain up to 31_{10} commands. The maximum number of UCL commands can be increased or decreased to a value you specify. You can also change the name of the UCL data file and the default device on which the file resides.

See the *RT–11 Installation Guide* for information on changing the size and name of the UCL data file.

13.2.7 Optimizing UCL

You can increase the speed with which UCL processes commands by combining the save image (.SAV) program and the data (.DAT) file. If you intend to increase or decrease the size of the UCL data file, you should do that before you optimize UCL.

Preserve the distributed UCL file:

```
.COPY SY:UCL.SAV UCL.DIS  RET
```

To optimize UCL, issue the following command:

```
.UNPROTECT SY:UCL.SAV  RET
.COPY SY:UCL.(SAV+DAT) SY:UCL.SAV  RET
```

If you determine you must change the size of the UCL data file after you have optimized UCL, perform the following procedure:

1. Create a file of the contents of the UCL data file by issuing the following command. In the command, *filespec* represents the name of the file, and must be of type .COM:

   ```
   .SHOW COMMANDS/OUTPUT:filespec  RET
   ```

   Verify that the file created by that command contains the information you expect by displaying its contents on your terminal screen:

   ```
   .TYPE filespec  RET
   ```

2. Delete the optimized UCL.SAV file.
3. Copy the UCL.DIS file from your default data (DK) device (or UCL.SAV from your software distribution kit) to your working system (SY) volume.
4. Perform the customization patch described in Section 13.2.6 to change the size of the UCL data file that you create in the next step.

5. Issue the following command that runs the file you created in Step 1 as an indirect command file. This command creates a new UCL data file and enters the commands in that file you previously defined:

   ```
   .$@filespec  RET
   ```

6. Optimize the UCL save image program you previously copied from your software distribution kit:

   ```
   .UNPROTECT SY:UCL.SAV  RET
   .COPY SY:UCL.(SAV+DAT) SY:UCL.SAV  RET
   ```

7. Issue SHOW COMMANDS to verify that the new optimized UCL utility contains the commands you expect.

13.3 Redefining DCL Commands with SYSGEN and UCL

If your software configuration (system and/or application software) requires RT–11 to execute a DCL command in a nonstandard manner, you can remove support for the DCL command by using the system generation (SYSGEN) procedure. You can then define the command as a new command by using the UCL command processor. You do not need to create and enable UCF command processing. This method can be preferable to using UCF if you need to change only the way a few DCL commands execute. Also, this method applies only to DCL commands; you cannot change the way CCL commands execute.

Redefining DCL commands, using this method, involves the following steps:

1. Examine the list of individual keyboard monitor (DCL) command SYSGEN conditionals in the *RT–11 System Generation Guide*.
2. Edit the answer (.ANS) file for your RT–11 monitor.
3. Perform a system generation using the edited answer file.
4. Exchange monitors.
5. Define as UCL commands the command or commands for which you removed support.

The following sections describe those steps in detail. The EXECUTE DCL command is used to illustrate the process of removing support for a DCL command and then defining it as a UCL command.

13.3.1 SYSGEN Command Conditionals

The *RT–11 System Generation Guide* contains a listing of the individual keyboard monitor command (DCL) conditionals. DCL commands for RT–11 are divided into three subsets, as shown in the list. To remove support for individual DCL commands, you set each individual command conditional to zero.

For example, the EXECUTE command is located in the LANGUAGE subset of DCL commands. As distributed, RT–11 monitors support the entire LANGUAGE subset of commands (L$ANG=1). Therefore, individual command conditionals are not specified. The conditional for the individual command EXECUTE is EXEC$$=n. To remove monitor support for only EXECUTE, you set the EXECUTE command conditional to zero (L$ANG=1 and EXEC$$=0). That combination of conditionals causes monitor support for all LANGUAGE subset commands except EXECUTE.

13.3.2 Edit the SYSGEN Answer (.ANS) File for Your Monitor

An answer file is included in the RT–11 software distribution kit for each distributed RT–11 monitor. Each answer file can be used to build a corresponding distributed monitor. If your monitor is not a distributed monitor, if it was built using the SYSGEN procedure, that procedure can create an answer file. In other words, each answer file is produced by building a monitor and each can be used to reproduce that monitor.

To remove support for one or more DCL commands, you edit the answer file for the monitor you are using to change only command support. Use the following procedure:

1. Determine which answer file built your monitor. If you are using a distributed unmapped monitor, that file is SbFB.ANS. If you are using a distributed mapped monitor, that file is can be XB.ANS, XM.ANS, ZB.ANS, or ZM.ANS. If you are using a monitor built by SYSGEN, that file will be of type .ANS and have a name assigned it during the SYSGEN procedure.
2. Preserve a copy of the answer file for your monitor by copying it to a file of the same name with file type .DIS.
3. Remove the file protection from the answer (.ANS) file and open that file for editing.
4. Do a search in the answer file for the appropriate language subset conditional. In the distributed answer files, all language subset conditionals are set to 1; all DCL commands are supported. Also, in distributed answer files, no individual DCL command conditionals are listed because all DCL commands are supported. For this example, because we are removing support for the EXECUTE command, the appropriate language subset conditional is L$ANG=1.
5. Edit the answer file to place EXEC$$=0 after L$ANG=1:

   ```
   L$ANG=1
   EXEC$$=0
   ```
6. Close the answer file.

13.3.3 Build a New Monitor

Do a SYSGEN and specify the answer file you just edited to build a new monitor. For the purpose of this procedure, the symbol *monitr* represents the monitor name, and *monitr*.ANS represents the answer file. Do the following:

1. Close any files you have open and stop and unload any jobs you have running. The SYSGEN procedure requires most of your computer's memory.
2. Turn to the beginning of the SYSGEN dialog, Studying the Sysgen Dialog, in Chapter 1 of the *RT–11 System Generation Guide*.

3. Issue the following command to start the SYSGEN procedure:

   ```
   .RUN IND.SAV SYSGEN.COM RET
   ```

   The SYSGEN procedure starts and displays prompts on your terminal screen. Respond to the prompts in the following manner:

   ```
   Do you want an introduction to system generation? (N)? N RET
   Do you want to use a previously created answer file? (N)? Y RET
   What answer file do you want to use (SYSGEN.ANS)? monitr.ANS RET
   Do you want to create an answer file (N)? N RET
   ```

   SYSGEN then checks for the presence of various protected files.

4. Refer now to the last part of the SYSGEN dialog in Chapter 1 of *RT–11 System Generation Guide*, DEVICE ASSIGNMENTS AND SYSGEN CLEANUP.

   SYSGEN requests the names of the input and output devices. SYSGEN then displays the file names it has created and the commands you use to build a monitor, supported devices, or both. Finally, SYSGEN exits and you receive the monitor prompt.

5. You are only interested in building a monitor. Therefore, issue the command to run the indirect command file of type .MON:

   ```
   .$@monitr.MON RET
   ```

   The indirect command file runs and displays command lines on your terminal screen. The result is a monitor of the appropriate name with the file type .SYG or *monitr*.SYG.

13.3.4 Exchange Monitors

Replace the monitor you are currently running with the monitor you have just built:

1. Preserve your current monitor by copying it to a file of the same name with file type .DIS:

   ```
   .COPY SY:monitr.SYS DK:monitr.DIS RET
   ```

2. Use the RENAME command and /NOREPLACE option to change the monitor you just built from file type .SYG to .SYS:

   ```
   .RENAME/NOREPLACE SY:monitr.SYG SY:NEWMON.SYS RET
   ```

3. Boot your new monitor:

   ```
   .BOOT SY:NEWMON.SYS RET
   ```

4. The monitor you are now running (NEWMON.SYS) does not support the EXECUTE command. Therefore, you can define EXECUTE as a UCL command.

5. If NEWMON.SYS functions correctly, use the following command to make it the default bootable monitor for the device:

   ```
   .COPY/BOOT SY:NEWMON.SYS SY: RET
   ```

13.3.5 Define the UCL Command

If you are already using the UCL command processor, you can add the EXECUTE command to the UCL data file. If you are not using the UCL command processor:

1. Make sure UCL.SAV resides on your system device:

   ```
   .DIR SY:UCL.SAV RET
   ```

   If UCL.SAV does not reside on your system device, copy it to there.

2. Issue the command:

   ```
   .EXECUTE:==how_you_want_EXECUTE_to_now_work
   ```

13.4 Changing Commands — Creating and Implementing UCF

You should read Section 13.2 before you decide to create and implement a UCF facility. Because you create UCF from the distributed UCL, they appear functionally similar. However, you should note the following two major differences:

- Unlike the UCL command processor, UCF imposes a noticeable increase in the processing time for other command processors. Therefore, unless you need to, you should not create and implement UCF.

  If you want to use UCF to change only the definition of one or more DCL commands, you should read Section 13.3 before creating UCF. You may be able to use UCL, and, if possible, you should.

- The following DCL commands do not function when UCF is enabled:

  B
  CLOSE
  D
  E
  GET
  REENTER
  SAVE
  START

  Also, to use UCF to define the Single-Line Command Editor's RECALL command, you must first SET SL NORECALL.

If you decide that you need UCF command processing, use the following procedure to create and implement UCF. You should proceed through the following sections in order.

13.4.1 Prepare UCL Before Creating UCF

You create the UCF.SAV utility by using the distributed UCL utility, UCL.SAV. You also create the UCF data file from a data file you create by using the distributed UCL. Therefore, the procedure you use to create the UCF utility and data file is determined by what, if anything, you have previously done with UCL.

13.4.1.1 If You Have Not Previously Used UCL

Issue the following command to determine if the file UCL.SAV is on your system (SY) device:

```
.DIR SY:UCL.SAV RET
```

If that command displays the file UCL.SAV on your system device, preserve a copy of the distributed UCL by using the following command, and then proceed to Section 13.4.3.

```
.COPY SY:UCL.SAV UCL.DIS RET
```

If that command does not display UCL.SAV on your system device, copy it to there by using the following command. In the command, the symbol *ddn* represents the device on which UCL.SAV resides:

```
.COPY ddn:UCL.SAV SY:UCL.SAV RET
```

Preserve a copy of the distributed UCL by using the following command and then proceed to Section 13.4.3.

```
.COPY SY:UCL.SAV UCL.DIS RET
```

13.4.1.2 If You Have Previously Used But Not Optimized UCL

You must rename (but preserve) the UCL data file, UCL.DAT. If you have been using UCL, the UCL utility and data file reside on your system (SY) device. Issue the following command to rename the UCL data file:

```
.RENAME SY:UCL.DAT SY:UCLDAT.DIS RET
```

Issue the following command to verify that the files UCL.SAV and UCLDAT.DIS reside on your system (SY) device:

```
.DIR SY:UCL*.* RET
```

Preserve the distributed UCL, using the following command, and then proceed to Section 13.4.3.

```
.RENAME SY:UCL.SAV SY:UCL.DIS RET
```

13.4.1.3 If You Have Used and Optimized UCL

You create the UCF utility from the distributed UCL utility, UCL.SAV. If you previously enabled and optimized UCL by issuing the following command, the UCL.SAV file on your system (SY) device is no longer as distributed:

```
COPY SY:UCL.(SAV+DAT) UCL.SAV
```

Use the information under the first following subheading if you followed the instructions in Section 13.2.7, and preserved the distributed UCL.SAV file as UCL.DIS before optimizing UCL. Use the information under the second following subheading if you did not preserve the distributed UCL.SAV as UCL.DIS.

If You Preserved the Distributed UCL.SAV

Rename (but preserve) the UCL file, UCL.SAV. If you have been using UCL, UCL.SAV resides on your system (SY) device. Then, copy the file UCL.DIS from the device on which it resides (*ddn*) to your system device:

```
.RENAME SY:UCL.SAV SY:UCLSAV.DIS  RET
.COPY ddn:UCL.DIS SY:UCL.SAV  RET
```

You should also perform a directory operation on both your system (SY) and default data (DK) device to determine if the file UCL.DAT exists on your system. If it does, issue the following command, where *ddn* represents that device on which UCL.DAT resides, to rename (but preserve) that file:

```
.RENAME ddn:UCL.DAT ddn:UCLDAT.DIS  RET
```

You should now have the distributed file UCL.SAV on your system (SY) device, and any previous UCL data file should be renamed to type .DIS.

If You Did Not Preserve the Distributed UCL.SAV

Rename (but preserve) the UCL file, UCL.SAV. If you have been using UCL, UCL.SAV resides on your system (SY) device. Then, copy the distributed file UCL.SAV from your RT–11 software distribution kit (on device *ddn*) to your system device:

```
.RENAME SY:UCL.SAV SY:UCLSAV.DIS  RET
.COPY ddn:UCL.SAV SY:UCL.SAV  RET
```

You should also perform a directory operation on both your system (SY) and default data (DK) device to determine if the file UCL.DAT exists on your system. If it does, issue the following command where *ddn* represents that device on which UCL.DAT resides, to rename (but preserve) that file:

```
.RENAME ddn:UCL.DAT ddn:UCLDAT.DIS  RET
```

You should now have the distributed file UCL.SAV on your system (SY) device, and any previous UCL data file should be renamed to type .DIS.

13.4.2 Create the Data File for UCF

Create a data file for UCF on your system device before proceding further. Create the data file by defining at least one command, such as:

```
.FOO:==DIR DK:  RET
```

You will remove support for that command further in the procedure.

13.4.3 Create UCF.SAV from UCL.SAV

Issue the following command to create the file UCF.SAV from UCL.SAV:

```
.RENAME SY:UCL.SAV SY:UCF.SAV
```

13.4.4 Enable UCF Command Processing

You must enable UCF command processing before you enter UCF commands and command definitions in the UCF data file. The only reason to enable UCF command processing is to change the definition of commands supported by distributed command processors. Therefore, if you attempt to redefine supported commands for the UCF data file before enabling UCF command processing, the distributed command processors parse those commands and return errors.

Use the SET CLI command, described in the *RT–11 Commands Manual* to enable UCF. Include that command in your start-up command file near the end of the file. Place it near the end so that if, for some reason, you do not want to turn on UCF processing, you can stop the execution of the commnd file before reaching the SET CLI command by pressing CTRL/C twice.

13.4.5 How Many Commands to Support?

You are going to store the UCF commands in a new UCL data file and then optimize UCF by combining the UCF utility with the UCL data file. You cannot easily change the size of the data file once you have combined it with the UCF utility. Therefore, you should first determine how many commands UCF supports. Make a list of the commands you require. By default, you can include up to 31_{10} UCF commands in the data file. If the number of commands you need is significantly more or less than 31, you should customize the data file for the size you need.

In the customization, replace the symbol ..CMDS with the value for that symbol located at Section 2.6.57 in the file CUSTOM.TXT. Replace the value *nnnnn* with the octal number of commands you will include in the UCF data file. (You may want to add support for a few more commands than you now expect to include.)

```
.RUN SIPP RET
*SY:UCF.SAV/A RET
Base?     0 RET
Offset?   ..CMDS RET

Base    Offset         Old   New?
 000000  ..CMDS         37   nnnnn RET
 000000  ..CMDS+2  ???????    CTRL/Y  RET
* CTRL/C
.
```

13.4.6 Boot Your System Device

Once you have enabled UCF command processing and created the files UCF.SAV and UCL.DAT on your system device, boot your system device to load into memory the RT–11 monitor you have modified to support UCF. Close any files you have open and issue the following command:

```
.BOOT SY: RET
```

Your system boots and your RT–11 monitor now first checks all commands for valid UCF command syntax.

13.4.7 Define the UCF Commands in the Data File

First, delete the UCF command you defined in Section 13.4.2:

```
.FOO:== RET
```

Then, using the rules described in Section 13.2.1, enter the UCF commands and command definitions. You can use the list of UCL commands as syntax examples. You use the following commands to display commands and definitions after you enter them:

```
.UCF RET
*TT:= RET
```

The header displayed by those commands reads User Command Linkage (UCL), although this is the UCF data file. You use a UCL data file as the UCF data file, which is not a problem because you will combine the UCF utility with this data file in the next section.

You do, however, want any error messages generated by the UCF facility to display the UCF name (?UCF-) rather than the default UCL name. Therefore, after you have entered the UCF commands and definitions into the data file, you should perform the following customization to change the name displayed by UCF data file error messages.

In the customization, replace the symbol ..ERR with the value for that symbol located at Section 2.6.55 in the file CUSTOM.TXT.

```
.RUN SIPP RET
*SY:UCF.SAV/A RET
Base?     0 RET
Offset?   ..ERR RET

Base    Offset       Old   New?
 000000  ..ERR    ??????   ;A RET
 000000  ..ERR       <?>   RET
 000000  ..ERR+1     <U>   RET
 000000  ..ERR+2     <C>   RET
 000000  ..ERR+3     <L>   ;AF RET
 000000  ..ERR+4     <->   CTRL/Y RET
* CTRL/C
.
```

13.4.8 Optimize the UCF Command Processor

Once you have created the UCF data file, you should optimize the UCF command processor by combining the UCF utility with the UCF data file. Issue the following command to optimize UCF:

```
.COPY SY:(UCF.SAV+UCL.DAT) SY:UCF.SAV RET
.DELETE SY:UCL.DAT RET
```

13.4.9 Restore the UCL Command Processor

The monitor can support both UCL and UCF command processors. You can, if you wish, restore the UCL command processor and use UCL with UCF. The procedure to create UCF required that you rename UCL. How you restore UCL is determined by how you used UCL before creating UCF:

- If you have not previously used UCL, you can restore UCL by issuing the following command and then read Section 13.2 for information on using UCL. In the command, *ddn* represents the device on which UCL.DIS resides.

  ```
  .RENAME ddn:UCL.DIS SY:UCL.SAV RET
  ```

- If you had previously used UCL by not optimized it, you restore UCL by issuing the following command:

  ```
  .RENAME SY:UCL.DIS SY:UCL.SAV RET
  .RENAME SY:UCLDAT.DIS SY:UCL.DAT RET
  ```

- If you had previously used and optimized UCL, you restore UCL by issuing the following command:

  ```
  .RENAME SY:UCLSAV.DIS SY:UCL.SAV RET
  ```

13.4.10 Optimize Your Configuration for UCF

Because you have created and enabled a UCF command processor, the RT–11 monitor must now read the file UCF.SAV from your system (SY) device each time you enter a command. Therefore, the faster the monitor can read files from the system device, the faster commands get processed.

The fastest system device available is the virtual memory VM device. Because you are using a UCF command processor, you notice an obvious increase in command processing speed if you configure a VM device. See Chapter 11 for information on configuring a VM device for your computer.

Whatever type of system device you use, the UCF command processor should reside on that device near the directory (near the beginning of the device).

13.4.11 Changing the Size of the UCF Data File

You can use the following procedure to change the size of the UCF data file if you determine you need more UCF commands. The procedure lets you use some UCL functionality to make changing the size of the UCF data file easier.

1. If you are using UCL, use the RENAME command to preserve UCL files as type .DIS files of the same name.
2. Use the RENAME command to change UCF to UCL as follows:

   ```
   .RENAME SY:UCF.SAV SY:UCL.SAV RET
   ```

3. Your UCF command processor is now named UCL. Therefore, you can take advantage of UCL functionality that lets you write commands and definitions to a file. Using this functionality means you do not need to retype all your UCF

commands into the new data file. Issue the following command to create the file of commands:

```
.SHOW COMMANDS/OUTPUT:UCF.COM RET
```

That command creates an indirect command file, UCF.COM, that contains all the commands and command definitions residing in UCF data file.

4. Unprotect, if protected, and delete the file UCL.SAV that resides on your system device, as follows:

```
.UNPROTECT SY:UCL.SAV RET
.DELETE SY:UCL.SAV RET
```

5. From your RT–11 software distribution kit, copy the distributed file UCL.SAV to your system device. If one of the UCL files you renamed to type .DIS as part of Step 1 is the distributed UCL.SAV, you can copy that UCL.DIS to UCL.SAV.
6. Use the RENAME command to change the distributed UCL.SAV on your system device to UCF.SAV, as follows:

```
.RENAME SY:UCL.SAV SY:UCF.SAV RET
```

7. Perform the customization patch described in Section 13.4.5 to change the size of the UCF data file to the value you want.
8. Issue the following command to run the indirect command file you created in Step 3:

```
.$@UCF RET
```

That command creates a new UCF data file (UCL.DAT) which contains the commands and definitions from your previous UCF data file. Define the other UCF commands you want to add to your UCF command processor.

9. Optimize the UCF command processor by combining the UCF utility with the UCF data file, as follows:

```
.COPY SY:(UCF.SAV+UCL.DAT) SY:UCF.SAV RET
.DELETE SY:UCL.DAT RET
```

10. If you are using UCL, restore it by renaming the type .DIS file or files you created in Step 1 to their correct file type.

Chapter 14
Using the RT–11 Backup Facilities

RT–11 provides complete software facilities to back up and preserve data.

This chapter describes:

- Why you should periodically back up your data.
- The software and hardware requirements for various types of backup operations.
- The facilities RT–11 provides to perform backup operations.
- How to perform backup operations to diskettes, removable disks, and magtapes. Backup operations are making the backup, obtaining a directory of the backup, and optionally restoring data from the backup volume.

Anyone using the RT–11 backup facilities should read Section 14.1 through Section 14.6, as those sections contain general information. After reading those sections, you then only need to read the subsequent section that pertains to your particular backup medium. You should look at the various ways you can back up data to your particular medium because each way has specific features.

Use the on-line index utility, INDEX, and see the on-line help utility, HELP, for more information on topics described in this chapter.

14.1 Why Back Up Data?

There are at least two reasons to make backup copies of your data:

- Data Protection

  You should back up your data in a methodical and periodic manner. Discovering you have no data backup when you need one causes pain and costs time and money. The minimal effort required to maintain good backups can be its own reward. Think how well you will sleep knowing you are free of that subliminal doubt. Think too of the small fortunes made by those wizards schooled in restoring data from trashed media, producing amazing results often months after the individual responsible for making backups has been fired. Think of the humiliation of explaining to your client/professor/boss that it's *gone*. Think of the last time you got away with telling your teacher, perhaps in the second grade, that the dog ate your homework. You are nodding your head. Good. Back up your data.

  How often you back up your data is determined by how much of that data you could afford to lose. If, for example, you daily create critical data that cannot be duplicated, you should habitually back up that data on a daily basis.

- Record Keeping

  Making periodic backup copies of your data maintains an exact record of your data at the point of each backup. Each backup can be thought of as a *snapshot* of the state of the data. This can be especially helpful in determining when changes or errors were introduced into data.

14.2 Software Requirements

You can perform the backup operations described in this chapter by using any RT–11 monitor. The different backup operations are performed by the BACKUP, COPY, and COPY/DEVICE commands. Each command in turn runs a different RT–11 utility:

Command	**Required Utility**
BACKUP	BUP
COPY	PIP
COPY/DEVICE	DUP

Enter the following command to verify that all three utilities reside on your system (SY) device:

```
.DIRECTORY SY:(BUP.SAV,DUP.SAV,PIP.SAV) RET
```

If that command does not display all three utilities as residing on your system device, copy any missing utility to SY.

DUP and PIP have no memory requirements. However, BUP has memory requirements that are determined by the RT–11 monitor you are using and whether backup operations are performed to a disk or magtape device.

Under the mapped monitors, BUP buffers are created in extended memory. BUP under mapped monitors requires about 8K words of low memory for backup operations to both disk and magtape.

Under the unmapped monitors, BUP buffers are created in low memory. Therefore, you should unload all unnecessary foreground jobs to free as much low memory as possible. For backup operations to disk devices, BUP requires at least 8.5K words of low memory. For backup operations to magtape devices, BUP requires at least 11.5K words of low memory. For backup operations to both disk and magtape devices, more than the minimum required memory increases BUP performance because BUP can create larger buffers.

Note that with operations to magtape devices, BUP does not use the file structure module (FSM), which is included in the distributed magtape handlers. You can decrease the memory size of magtape handlers for use with BUP by performing a system generation and building such handlers without FSM support. See the *RT–11 System Generation Guide* for information on building magtape handlers with only hardware support. Such handlers do not support RT–11 COPY or DIRECTORY operations and should be substituted for the distributed magtape handlers only for BUP operations.

14.3 Hardware Requirements

The only hardware requirement for performing backup operations is a backup device that supports removable media. Such devices are diskette drives, removable disk drives, and magtape drives.

14.4 How Is the Data Organized?

Data that you want to back up can be organized on a device volume in a file-structured or non-file-structured manner. The following lists which backup procedures are appropriate for each:

- **File-Structured Data**

  File-structured data can be backed up as individual specified files, files of a particular catagory or categories, or all files residing on a device. The backup procedure is determined by:

  - The file size

    Small files can be backed up using BACKUP or COPY. Large files are best backed up using BACKUP. If any of the files to be backed up is larger than the capacity of the backup volume, BACKUP must be used.

  - The backup device

    BACKUP or COPY can be used if the backup device is a diskette or removable disk drive. BACKUP should be used if the backup device is a magtape drive.

- **Non-File-Structured Data**

  Non-file-structured data must be backed up by a device image backup operation, using BACKUP or COPY/DEVICE:

  - BACKUP lets you back up a device image across more than one output volume.
  - COPY/DEVICE requires that the device image fit on a single output volume. However, COPY/DEVICE supports block selection options /START and /END, that let you specify which blocks to back up.

  Select which backup method to use based on your requirements. Section 14.5 describes backing up device images by using both methods.

14.5 Backup Facilities

RT–11 provides three facilities you can use to back up data.

BACKUP
COPY
COPY/DEVICE

14.5.1 BACKUP

You can use the BACKUP commands to make file or device image backups from any media to any media, except magtape to magtape. Unlike COPY and COPY/DEVICE, BACKUP is used only to make backups and, therefore, is the most efficient backup facility and provides the most features:

- Only facility that lets you back up files that are larger than a single volume of the backup media. Therefore, it is the facility of choice in backing up variable-size files to diskette.
- Groups backed-up files into *savesets* on the backup volume for logical preservation and recovery. A directory can be requested for the savesets and the files within savesets. Individual files can be restored from savesets.
- Most efficient facility with magtape backup media.
- Most efficient verification procedure. Verifies backed-up data in a separate pass, rather than in single blocks during backup operation.
- Supports wildcards in file specifications.
- Can produce directories of, and retrieve files from, unmounted logical disks.
- Can create savesets on magtape that are easily transportable to VAX processors running the VMS operating system. Once the files on saveset are transported to a VAX, those files are easily read and manipulated.
- Can back up files to automatically created logical disks that contain exactly enough disk space to hold the files. Files can be specified with wildcards. No calculations are required. The resulting logical disk is identical to one created manually.

14.5.2 COPY

You can use the COPY command, with various options, to back up small files to a diskette or to a logical disk residing on a removable disk. COPY cannot back up a file to more than one output volume and is not very efficient with magtape backup devices.

The COPY command is particularly useful when making incremental backups of the same small files to a particular diskette or logical disk. In this instance, COPY can be more useful than BACKUP because BACKUP does not allow updating of saveset images. If, for example, you make daily changes to varying modules in a set of modules, you can use the COPY command with the /PREDELETE and /DATE options to back up only those modules you modified that day.

14.5.3 COPY/DEVICE

You can use the COPY command with the /DEVICE and other options to make device image backups from files or devices to files or devices. COPY/DEVICE is most appropriate when backing up the complete contents of one device volume to a device volume of the same size or to a magtape. That is, COPY/DEVICE is most appropriate for volume duplication. COPY/DEVICE cannot back up files or a device to more than one output volume.

14.6 What Is the Backup Device?

Not all backup facilities are appropriate for all backup devices. For example, if your backup device is a diskette drive and the files you want to back up reside on a large fixed disk, you probably want to make file-oriented backups and not back up the entire disk. If, however, your backup device is a cartridge magtape, which functions poorly in start/stop file-oriented mode, you may want to make device-oriented or saveset-oriented backups. In short, you want to back up your data in a manner that best utilizes your backup device.

The following sections describe backup operations to specific backup devices. As you read the section for your backup device, think of the backup facilities that best utilize that device.

14.7 Diskette Backup Media

Diskettes are often used as backup media because they are found on many RT–11 computer systems and are removable, reasonably inexpensive, and easily stored.

14.7.1 BACKUP Command with Diskettes

The BACKUP command runs the BUP utility.

The BACKUP command is very useful with diskette backup media and its operation is quite simple, because BUP provides informational prompts as necessary. Using the BACKUP command, you can:

- Back up files or a device of any size. As BUP fills each diskette, it prompts you to mount another, initializes it, and proceeds with the backup.
- Group the backed-up files or device into a saveset and assign the saveset a name. With savesets, you need not worry about multiple copies of the same file on the same diskette overwriting each other; individual files are encased in savesets.
- Use wildcards with file specifications to back up only files of a specified name or type.
- Obtain a directory of the savesets located on a series of backup diskettes.
- Obtain a directory of the files residing on each saveset.
- Restore an individual file from a saveset or use wildcards to restore all files of a particular name or type.

This section describes procedures you can use to:

- Create device image diskette backups of complete volumes or logical disks.
- Back up all files on a volume.
- Back up selected files on a volume.
- Obtain directories of the savesets on a series of diskette backups and directories of individual files on each saveset.
- Restore data from diskette backups.

14.7.1.1 Device Image

You can use the following procedure to back up an entire device image to diskettes. The example procedure assumes there is a disk device, *ddn*, that is being backed up to diskettes on device DU1.

1. Determine how many blocks of data you are backing up:

   ```
   .DIRECTORY dnn: RET
   ```

   You are backing up the entire volume, including data and free blocks.

2. Obtain enough diskettes to hold the total (written and free) blocks displayed by the DIRECTORY command. The diskettes need not be new or initialized, but must not contain any information you want to preserve. The backup operation destroys any data residing on the diskettes.

3. Assume the current date is May 27, 1989. Insert a diskette into drive DU1 and enter the following command, which assigns this backup the saveset name 27MAY.BUP:

   ```
   .BACKUP/INITIALIZE/VERIFY/DEVICE ddn: DU1:27MAY RET
   ```

   If you want BUP to omit the *are you sure?* queries, include the /NOQUERY option in the command line. BUP initializes the diskette as a backup medium and, by default, performs a bad-block scan. If you are completely sure all the diskettes contain no bad blocks, you can include the /NOSCAN option that prevents the bad-block scan of each diskette.

   If you omit the saveset name, BUP assigns the saveset the device name with type .BUP (*ddn*.BUP) because this is a device image backup.

4. BUP begins the backup operation on the diskette. When the diskette is full, BUP verifies the diskette against the input (*ddn*) volume. After verification, you are prompted to remove it and insert another. You repeat this until you have backed up device *ddn*.

5. Label the backup diskettes and store them in a safe place.

14.7.1.2 All Files

You can use the following procedure to back up all files residing on a device. The procedure backs up only files and not empty blocks. The example procedure assumes files on a disk device, *ddn*, being backed up to diskettes on device DU1.

1. Determine how many blocks of data you are backing up:

   ```
   .DIRECTORY dnn: RET
   ```

   You are backing up the data blocks; free blocks are not backed up.

2. Obtain enough diskettes to hold the blocks displayed by the DIRECTORY command. The diskettes need not be new or initialized but must not contain any information you want to preserve. The backup operation destroys any data residing on the diskettes.

3. Assume the current date is May 27, 1989. Insert a diskette into drive DU1 and enter the following command, which assigns this backup the saveset name 27MAY.BUP:

   ```
   .BACKUP/INITIALIZE/VERIFY ddn:*.* DU1:27MAY RET
   ```

   If you want BUP to omit the *are you sure?* queries, include the /NOQUERY option in the command line. BUP initializes the diskette as a backup medium and, by default, performs a bad-block scan. If you are completely sure all the diskettes contain no bad blocks, you can include the /NOSCAN option that prevents the bad-block scan.

   If you omit the saveset name, BUP assigns the saveset the name BACKUP.BUP because this is a file backup.

4. BUP begins the backup operation on the diskette. When the diskette is full, BUP verifies the diskette against the input (*ddn*) volume. After verification, you are prompted to remove it and insert another. You repeat this until you have backed up all the files residing on device *ddn*.

5. Label the backup diskettes and store them in a safe place.

14.7.1.3 Selected Files

You can use the following procedure to back up only selected files residing on a volume. You use wildcards to select the files. Files that are not selected and empty blocks are not backed up. The example procedure assumes you are backing up all files matching the wildcard file specifications T*.FOR, *.MAC, and T*.SAV on disk device, *ddn*, to diskettes on device DU1.

1. Determine how many blocks of data you are backing up:

   ```
   .DIRECTORY dnn:(T*.FOR,*.MAC,T*.SAV) RET
   ```

   Note the number of blocks returned by the display.

2. Obtain enough diskettes to hold the blocks displayed by the DIRECTORY command. The diskettes need not be new or initialized but must not contain

any information you want to preserve. The backup operation destroys any data residing on the diskettes.

3. Insert a diskette into drive DU1 and enter the following command, which assigns this backup the saveset name WRK.BUP:

   ```
   .BACKUP/INITIALIZE/VERIFY dnn:(T*.FOR,*.MAC,T*.SAV) DU1:WRK RET
   ```

   If you want BUP to omit the *are you sure?* queries, include the /NOQUERY option in the command line. BUP initializes the diskette as a backup medium and, by default, performs a bad-block scan. If you are completely sure all the diskettes contain no bad blocks, you can include the /NOSCAN option that prevents the bad-block scan.

   If you omit the saveset name, BUP assigns the saveset the name BACKUP.BUP.

4. BUP begins the backup operation on the diskette. When the diskette is full, BUP verifies the diskette against the input (*ddn*) volume. After verification, you are prompted to remove it and insert another. You repeat this until you have backed up all files matching the wildcard file specifications that reside on device *ddn*.

5. Label the backup diskettes and store them in a safe place.

14.7.1.4 Directory Operations

You can obtain a directory of the savesets on a series of diskette backups and a directory of files residing on each saveset. You can display the directories on your terminal, write them to a file, or print them.

You should create and save a directory of all savesets on each series of diskette backups and a directory of all files on each saveset. You should create those directories after performing the backup operation.

A saveset is the result of a single backup operation and can be written on one or more backup volumes. Savesets can be added to a backup volume that already contains one or more savesets, so a single diskette could contain the last section of a multivolume saveset series, one or more complete single-volume savesets, and the first section of the next multivolume saveset series.

You should obtain a directory of each diskette that contains a complete saveset and the first diskette section of each multivolume saveset. Those diskettes contain the section information and the directory of individual files in each saveset.

Obtaining Saveset Directories

Mount the diskette that contains either the entire saveset or the first section of a multivolume saveset. In the following example, DU1 is assumed to contain such a diskette. The following command returns the directory of the backup volume in device DU1:

```
.BACKUP/DIRECTORY DU1: RET
```

Saveset Directory Format

The following directory is an example of the first volume in a series of five diskette backups that contains three savesets:

```
 RT-11 BACKUP
 05-May-88 09:28
 Volume 1

 Saveset      Section     Blocks       Date

OBJ   .BUP    1/1         474/474      05-May-88
TEMP  .BUP    1/1         14/14        05-May-88
RUNOFF.BUP    1/5         304/3114     05-May-88

 3 Saveset sections, 792 Blocks
 0 Free blocks
```

Obtaining File Directories from Savesets

You obtain a directory of the files residing on a saveset from the diskette that contains either the entire saveset or the first section of a multivolume saveset. You then specify in the command line the saveset for which you want a file directory. For example, the following command displays the files backed up in saveset TEMP.BUP, residing on DU1:

```
.BACKUP/DIRECTORY DU1:TEMP.BUP/SAVESET RET
```

File Directory Format

The following example directory displays the files residing on saveset TEMP.BUP:

```
 RT-11 BACKUP
 05-May-88 10:54
 Saveset: DU1:TEMP.BUP
 Created: Thursday   05-May-88 09:23

 File         Blocks  Volume          Date

TEMP  .TMP    2       1        Thursday   05-May-88
CACHE .TMP    3       1        Thursday   05-May-88

 2 Files, 5 Blocks
```

Directory Destination

You can display directories on your terminal (the default), send directories to a specified file, or print them.

The following command displays the directory for the saveset TEMP.BUP, which resides on device DU1:

```
.BACKUP/DIRECTORY DU1:TEMP.BUP/SAVESET RET
```

The following command writes that directory to the file specification DU0:TEMP.DIR:

```
.BACKUP/DIRECTORY/OUTPUT:DU0:TEMP.DIR DU1:TEMP.BUP/SAVESET RET
```

The following command prints the directory:

```
.BACKUP/DIRECTORY/PRINTER DU1:TEMP.BUP/SAVESET RET
```

14.7.1.5 Restoration Operations

You can restore a complete saveset or selected files that reside within a saveset. If the complete saveset resides on a single diskette, mount that diskette. If the complete saveset or selected files within a saveset reside on more than one diskette, mount the first backup diskette containing the appropriate section.

A directory of the savesets on a series of backup diskettes points you to the correct diskette to mount. A directory of the files on a saveset points you to the backup volume containing a file you want to restore.

For example, to restore the complete device image saveset INT.BUP residing completely on device DU1 to device LD5:

```
.BACKUP/RESTORE/DEVICE/VERIFY DU1:INT/SAVESET LD5: RET
```

If INT.BUP resided on more than one diskette, you would mount the diskette containing the first section, and BUP would prompt you to mount successive diskettes until the entire saveset was restored.

To restore selected files from a saveset residing completely on a single diskette, mount the diskette and specify the selected files. For example, the following command restores to device LD5 all files of type RNO from saveset INT.BUP residing on device DU1:

```
.BACKUP/RESTORE/VERIFY DU1:INT/SAVESET,*.RNO LD5: RET
```

If INT.BUP resided on more than one diskette, you would mount the diskette containing the first section, and BUP would prompt you to mount successive diskettes until all files of type RNO had been restored.

To restore a single selected file from a saveset when the file is split across two or more diskette volumes, mount the first diskette volume, and BUP prompts you to mount successive diskettes until the entire file is restored.

14.7.2 COPY Command with Diskettes

Consider the following points before using the COPY command to back up data to diskettes:

- COPY lets you selectively update backup copies of files that already reside on a backup volume.
- COPY can be used to back up small files. Files are considered small when they each contain significantly fewer blocks than the backup media. Using COPY to back up large files can be inefficient because a large file cannot be copied across volumes, often leaving unused blocks on the backup volume. Also, COPY cannot back up any file that is larger than the backup volume. In general, you should use the BACKUP command described in Section 14.7.1 to back up large files.
- A diskette (or any device volume) can contain only one version of any file. Therefore, to avoid the possibility of the current backup operation overwriting a file from a previous backup operation, keep the diskettes separate from each backup operation.

You can use the procedure in this section to maintain a periodic backup system that is appropriate for backing up small files to one or more diskettes. This procedure backs up files according to creation or modification date. You can select additional criteria.

The following example illustrates the procedure and assumes the following:

- A hard disk volume, DU0, contains small files that were created since the last backup.
- The backup device is a diskette on drive DU1.
- The last backup was done on May 10, 1989.

Given those assumptions, the procedure to back up all files that were created or modified since May 10, 1989 is as follows:

1. Determine if any files reside on DU0 that you do not want to keep or preserve. You may not, for example, want to keep previous (.BAK) versions of some files. Delete any files you do not want before starting the backup operation.
2. Determine how many blocks are to be backed up:

   ```
   .DIRECTORY/SINCE:10:MAY:89 DU0: RET
   ```

3. Initialize and examine for bad blocks enough diskettes to contain the number of blocks displayed by the previous command:

   ```
   .INITIALIZE/BADBLOCKS DU1: RET
   ```

4. Back up the files, using the following command. If the number of file blocks to be backed up is greater than the capacity of one diskette, the system prompts you to insert further diskettes as each is filled:

   ```
   .COPY/MULTIVOLUME/VERIFY/SINCE:10:MAY:89 DU0: DU1: RET
   ```

5. After the backup operation is completed, label each diskette with enough information to identify its contents. Be sure to include the date you made the backup. Store the backup diskettes in a safe place, preferably in a location separate from the source files.

Restoring Backed-Up Files

If you want to restore a specific version of a file from diskette backup media, obtain the set of backup diskettes that contains the file version and, specifying the file name with the DIRECTORY command, locate the file.

To restore a backed-up file, you copy that file from the backup volume on which it resides (input volume) to the volume on which you want to restore the file (output volume). If a file of the same name and type resides on the output volume, determine if you want to keep both files. If you want to preserve both files, you must change the file name and/or type of one of the files. Otherwise, the output volume file will be written over or, if that file is protected, the restoration operation will fail because of a protection violation.

14.7.3 COPY/DEVICE Command with Diskettes

COPY/DEVICE should be used when you want to create duplicates of diskette volumes. COPY/DEVICE copies all blocks from one diskette to another, thereby preserving the home and boot blocks. Because the input and backup diskettes are identical, no special directory or restoration operations are necessary.

For example, the following command duplicates diskette DU1 on diskette DU2:

```
.COPY/DEVICE/VERIFY/NOQUERY DU1: DU2: RET
```

14.8 Removable Disk Backup Media

Removable disks are often used as backup media. Depending on the capacity of the disk, you can often use the same removable disk to store the output of multiple periodic backups.

14.8.1 BACKUP Command with Removable Disks

The BACKUP command runs the BUP utility.

BUP provides two methods to back up data to a removable disk:

- You can back up data (files or a device) to a *saveset*. A saveset can be thought of as a container that holds one or more files or devices from a single backup operation. Each saveset is the result of a single backup operation, and you can create one or more savesets on the disk.

  You can obtain a directory of the savesets on a disk and the individual files within savesets. You can restore entire savesets or specified individual files residing on a saveset.

- You can back up data to an automatically created logical disk on a standard RT–11 volume. The BACKUP command with the /SUBSET option automatically creates a logical disk of exactly enough capacity to contain the files you want to back up. You need not manually create, initialize, or mount the logical disk; BUP does that for you.

  You can obtain a directory of the files located on the logical disks without having to mount them. You can also restore specified files from unmounted logical disks.

14.8.1.1 Saveset Backup Operations

You can perform multiple backup operations to the same backup volume or series of volumes. The result of each volume or file backup operation is a saveset. You should name each saveset to identify the saveset with the device or files it contains. Each saveset name must be unique, or BUP returns a fatal level error message. If you omit the saveset name with device image backups, BUP by default assigns the name of the device containing the data being backed up; for file backups, BUP assigns the name BACKUP.BUP.

You can back up an entire volume, including empty blocks; all files residing on a volume; or selected files. In any case, BUP first determines if the output (backup) device is a valid BUP-initialized volume. If not, BUP initializes the volume and examines it for bad blocks. In all cases, BUP provides informational messages and prompts you for any required responses.

Backing Up an Entire Volume

The following example illustrates backing up the volume located on device DY1. In the command, DY1 is the input device, DL0 is the device containing the backup volume, and FOO is the saveset name with the default file type of .BUP. The success of the operation is verified:

```
.BACKUP/DEVICE/VERIFY DY1: DL0:FOO RET
```

If you do not supply a saveset name, BUP uses the input device name with file type .BUP as that saveset name (DY1.BUP).

Backing Up All Files

The following example illustrates backing up all files on the volume located on device DY1. No empty blocks are backed up and the success of the operation is verified. In the command, DY1 is the input device containing the files, DL0 is the backup volume, and 28MAY is the optional saveset name. It is assumed that saveset FOO.BUP is already written on DL0; saveset 28MAY.BUP is being added to DL0:

```
.BACKUP/VERIFY DY1:*.* DL0:28MAY RET
```

BUP notifies you that this saveset is being added to a volume that contains at least one other saveset by displaying the following informational message:

```
?BUP-I-Appending to volume
```

Backing Up Selected Files

The following example illustrates backing up all files with the name, MYFILE, residing on the volume located on device DY1. The success of the operation is verified. In the command, MYFILE.*, residing on DY1, are the files being backed up, and DL0 is the backup volume. The saveset being created on DL0 is MYFILE.BUP. It is assumed that savesets FOO.BUP and 28MAY.BUP are already written on DL0; saveset MYFILE.BUP is being added to DL0:

```
.BACKUP/VERIFY DY1:MYFILE.* DL0:MYFILE RET
```

BUP notifies you that this saveset is being added to a volume that contains at least one other saveset by displaying the following informational message:

```
?BUP-I-Appending to volume
```

14.8.1.2 Saveset Directory Operations

You can obtain a directory of all savesets residing on a removable disk. You can also obtain a directory of all files residing on a specific saveset. Saveset and file directory structures are shown in Section 14.7.1.4.

You can send a directory of savesets or files residing on a saveset to your terminal, a specified output file, or your printer.

For example, the following command displays the directory of savesets for disk DL0:

```
.BACKUP/DIRECTORY DL0: RET
```

The following command displays on your terminal the files backed up to saveset MYFILE.BUP on disk DL0:

```
.BACKUP/DIRECTORY DL0:MYFILE.BUP/SAVESET RET
```

The following command copies the directory of savesets residing on DL0 to device DL1 with the file name DL0BAC and the default file type .DIR.

```
.BACKUP/DIRECTORY/OUTPUT:DL1:DL0BAC DL0: RET
```

The following command prints the directory of the files in the saveset, 28MAY.BUP, which resides on device DL0:

```
.BACKUP/DIRECTORY/PRINTER DL0:28MAY.BUP/SAVESET RET
```

14.8.1.3 Saveset Restoration Operations

You can restore a complete saveset or selected files from a saveset. You can use wildcards to restore files of only a particular name or type. As insurance, you can use the /VERIFY option in restoration operations to ensure that the restored data can be read from the destination device.

For example, the following command restores all the files in the saveset 28MAY.BUP, from device DL0 to DL1:

```
.BACKUP/RESTORE/VERIFY DL0:28MAY.BUP/SAVESET DL1: RET
```

Assuming the saveset 28MAY.BUP contains the file FOO.OBJ, you could restore that file to device DL1, using the following command:

```
.BACKUP/RESTORE/VERIFY DL0:28MAY.BUP/SAVESET,FOO.OBJ DL1: RET
```

You could also restore all files of type OBJ from saveset 28MAY.BUP, using the following command:

```
.BACKUP/RESTORE/VERIFY DL0:28MAY.BUP/SAVESET,*.OBJ DL1: RET
```

The restoration of an unspecified device image saveset causes BUP to look for a saveset matching the output device name and, if not found, to return an error message. For example, the following command causes BUP to look for saveset DL1.BUP on device DL0:

```
.BACKUP/DEVICE/RESTORE/VERIFY DL0: DL1: RET
```

The restoration of an unspecified file image saveset causes BUP to look for a saveset named BACKUP.BUP and, if not found, to return an error message. For example, the following command causes BUP to look for saveset BACKUP.BUP on device DL0:

```
.BACKUP/RESTORE/VERIFY DL0: DL1: RET
```

14.8.1.4 Logical Disk Backup Operations

Using the /SUBSET option with the BACKUP command automatically creates logical disk images of the files you want to back up. You perform no CREATE, INITIALIZE, or MOUNT operations; the /SUBSET option performs those operations for you. The /SUBSET option creates logical disks on only standard RT–11 disks and not on BUP-formatted disks or any magtape volume. The standard RT–11 disk format is not changed and the disk can continue to be used normally.

The logical disk created by the /SUBSET option is identical to one you create manually, except no free blocks are allocated and the number of directory segments is only enough to contain the files being backed up.

The /SUBSET option is an alternative to the /SAVESET option and, unlike savesets, logical disk images created by the /SUBSET option must reside on a single backup volume. Also, SUBSET operations produce no mount prompts or bad-block scans.

The /SUBSET option, together with other BACKUP command options, lets you obtain the directory of an unmounted logical disk file and restore from that unmounted logical disk file any files you specify. You can also manually mount a logical disk file created by the /SUBSET option and perform standard logical disk operations, such as copying, printing, and deleting files.

CAUTION

Once you have created a logical disk on a volume by using the /SUBSET option, do not then perform any other type of backup operation to that volume.

Logical disks cannot be created on BUP backup volumes. You cannot initialize a backup volume as a BUP volume and then use the /SUBSET option to create logical disk images on that volume. Similarly, you should not create logical disks on a volume and subsequently attempt to perform different BUP backup operations on that volume, because BUP backup operations initialize the volume and destroy any logical disks.

The /SUBSET option is appropriate only for file operations. You should not back up entire volumes (disk images) to logical disks with /SUBSET. To back up disk images, use the BACKUP/DEVICE or the COPY/DEVICE command.

Use the following procedure to automatically create logical disks, obtain a directory, and restore files:

1. If the volume to contain the logical disk has not been initialized (if it is an unused volume), initialize it as you would normally initialize an RT–11 volume. You should also check the volume for bad blocks. The following command initializes the volume in device *ddn*:

   ```
   .INITIALIZE/BADBLOCKS ddn: RET
   ```

   Respond appropriately to any prompts. Note that the command does not initialize the disk as a BUP volume, but rather as a standard RT–11 volume.

2. Back up selected files to a logical disk. For example, the following command backs up all files on DL0 of type OBJ to a logical disk, OBJ.DSK on device DL1. The success of the operation is verified:

   ```
   .BACKUP/VERIFY DL0:*.OBJ DL1:OBJ/SUBSET RET
   ```

   Note that file type .DSK need not be included in the output specification (DL1:OBJ); .DSK is the default file type. The command displays all files backed up to DL1. If DL1 does not contain enough free blocks for all the .OBJ files, BUP returns an error message indicating insufficient space, and no files are backed up.

14.8.1.5 Logical Disk Directory Operations

A directory operation to device DL1 displays the presence of the logical disk file OBJ.DSK. To obtain a directory of the contents of OBJ.DSK, you need not mount the file, but rather issue the following command:

```
.BACKUP/DIRECTORY DL1:OBJ.DSK/SUBSET RET
```

You could also copy the directory of OBJ.DSK to the file OBJ.DIR on device DL0:

```
.BACKUP/DIRECTORY/OUTPUT:DL0:OBJ.DIR DL1:OBJ.DSK/SUBSET RET
```

You could also print the directory of OBJ.DSK:

```
.BACKUP/DIRECTORY/PRINT DL1:OBJ.DSK/SUBSET RET
```

14.8.1.6 Logical Disk Restoration Operations

You can restore one or more selected files from mounted or unmounted logical disks. For example, the following command restores and verifies the file, FOO.OBJ, previously backed up to logical disk OBJ.DSK on DL1 to device DL0:

```
.BACKUP/RESTORE/VERIFY DL1:OBJ.DSK/SUBSET,FOO.OBJ DL0: RET
```

You could also restore multiple selected files from a logical disk, using wildcards. For example, the following command restores all files of name WOOGA and verifies the operation:

```
.BACKUP/RESTORE/VERIFY DL1:OBJ.DSK/SUBSET,WOOGA.* DL0: RET
```

And the next example restores all files:

```
.BACKUP/RESTORE/VERIFY DL1:OBJ.DSK/SUBSET,*.* DL0: RET
```

14.8.2 COPY Command with Removable Disks

You can often use the same removable disk to store the output of multiple periodic backups. To keep each backup operation separate, create a different logical disk on the physical disk for each periodic backup.

There is no file size restriction that determines efficiency. You can, assuming enough disk space, create a logical disk large enough to contain the entire backup without wasting disk blocks. However, the logical disk must have enough blocks to contain the largest individual file you are backing up and should be large enough to contain all the files you are backing up.

If a removable disk does not contain enough free blocks to create such a logical disk, use a different removable disk for the backup. If any file you are backing up contains more blocks than the full capacity of the removable disk, you must use the BACKUP command described in Section 14.8.1.

The following example illustrates the backup operation to a removable disk and assumes the following:

- A hard disk volume, DL0, contains small files that were created since the last backup.
- The backup device is a removable disk on drive DL1.
- The last backup was done on May 9, 1989, and backups are performed on a weekly basis.

Given those assumptions, the procedure to back up all files that were created or modified since May 10, 1989 would be:

1. Determine if any files reside on DL0 that you do not want to keep or preserve. You may, for example, not want to keep previous (.BAK) versions of some files. Delete any files you do not want before starting the backup operation.
2. Determine how many blocks are to be backed up:

   ```
   .DIRECTORY/SINCE:10:MAY:89 DL0: RET
   ```

3. Create a logical disk on DL1 enough to contain the blocks displayed in the previous command plus a directory structure. Give the logical disk a name that associates it with this backup, such as a data abbreviation. If you do not know how to create logical disks, you should read Chapter 9, but for now assume the following command creates a logical disk large enough to hold the backup blocks and file structure:

   ```
   .CREATE DL1:16MAY.DSK/ALLOCATE:400. RET
   ```

4. Associate the logical disk with an unused LD unit number:

   ```
   .MOUNT LD5: DL1:16MAY.DSK RET
   ```

5. Initialize and examine the logical disk for bad blocks:

   ```
   .INITIALIZE/BADBLOCKS LD5: RET
   ```

6. Then, back up the appropriate files, using the following command:

   ```
   .COPY/VERIFY/SINCE:10:MAY:89 DL0: LD5: RET
   ```

7. After the backup operation is completed, store the removable disk in a safe place, preferably in a location separate from the source files.

Directory and Restoration Operations

If you want to restore a specific file from removable disk backup media, mount the removable disk that contains the logical disk backup volume. Mount the logical disk and, specifying the file name with the DIRECTORY command, locate the file.

If you are unsure which logical disk file contains the file you want to restore, the BACKUP command with /DIRECTORY and /SUBSET options can search for a file specification through each logical disk file on a physical disk. Each logical disk file must be specifed on the command line; wildcards are not permitted. The logical disk files need not be mounted. The command line does accept wildcards for file specifications and will search for all files of a given name or type on the specified logical disk file.

For example, the following command seaches the unmounted logical disk file 16MAY.DSK on physical device DL1 for all files named FILNAM:

```
.BACKUP/DIRECTORY DL1:16MAY.DSK/SUBSET,FILNAM.* RET
```

If any file of name FILNAM resides on 16MAY.DSK, BACKUP displays the logical disk name, the file specification, the file size, and most importantly the date of creation or modification. You can then mount 16MAY.DSK, and examine and manipulate the file.

14.8.3 COPY/DEVICE Command with Removable Disks

COPY/DEVICE should be used when you want to create duplicates of removable disks. COPY/DEVICE copies all appropriate blocks from one disk to another, thereby preserving the home and boot blocks. On disks that support bad-block replacement, the bad-block replacement table is correctly not duplicated. Because the input and backup disks are otherwise identical, no special directory or restoration operations are necessary.

For example, the following command duplicates disk DL0 on disk DL1:

```
.COPY/DEVICE/VERIFY DL0: DL1: RET
```

14.9 Magtape Backup Media

Magtapes are the least expensive backup media. They hold a large amount of data and are easy to store.

14.9.1 BACKUP Command with Magtapes

The BACKUP command runs the BUP utility.

Backup operations involve initializing the backup magtape, making the backup, optionally obtaining a directory, and perhaps restoring what was backed up.

You can perform multiple backup operations to the same magtape or a series of magtapes. Each backup operation creates a saveset on the magtape or series of magtapes. A *saveset* can be thought of as a *container* that holds one or more files or devices from a single backup operation.

This section contains the following information:

- Initializing magtape backup volumes
- Creating magtape backup volumes
- Magtape backup directories
- Restoring volumes and files from magtape backup volumes
- Transporting BUP-written magtape backup volumes to VAX processors running the VMS operating system and manipulating files on those magtapes.

14.9.1.1 Initializing Volumes

All magtapes must be initialized. The initialization process rewinds the magtape and writes volume and header labels at the beginning of the magtape.

You must explicitly initialize any magtape volume before or during the first backup operation to that magtape. When you back up files or volumes to a series of magtapes as part of a single backup operation, BUP implicitly initializes all subsequent magtapes in the series.

Do not initialize any magtape after you have performed the first backup operation or you will destroy the data residing on that magtape.

You initialize the magtape volume during the first backup operation by including the /INITIALIZE option with the BACKUP command line. For example, the following command initializes magtape MS0 and backs up the device DL1 to MS0. The result of this operation is a saveset named FOO.BUP on device MS0.

```
.BACKUP/INITIALIZE/DEVICE DL1: MS0:FOO RET
```

14.9.1.2 Creating Backup Savesets

You can perform multiple backup operations to the same magtape or series of magtapes. The result of each volume or file backup operation is a saveset.

Use the following information to create savesets:

- You should name each saveset as shown in the following example. Each saveset name must be unique, or BUP returns a fatal level error message. The following example illustrates backing up the volume located on device DL1 with verification. In the command, DL1 is the input device, MS0 is the backup magtape volume, and FOO is the saveset name with the default file type of .BUP:

  ```
  .BACKUP/DEVICE/VERIFY DL1: MS0:FOO RET
  ```

  With device image backups, if you do not supply a saveset name, BUP uses the input device name with file type .BUP as that saveset name (DL1.BUP).

- You can back up files that are located on a volume to a saveset on magtape by including file names or wildcards in the command line. For example, the following command backs up files, MYFILE.*, located on device DL1, to magtape MS0, with the saveset name MYFILE.BUP. The operation is verified. Assuming that saveset FOO.BUP is already written on magtape MS0; saveset MYFILE.BUP is added to magtape MS0.

  ```
  .BACKUP/VERIFY DL1:MYFILE.* MS0:MYFILE RET
  ```

  BUP notifies you that this saveset is being added to the magtape MS0 by displaying the following informational message:

  ```
  ?BUP-I-Appending to volume
  ```

  With file backups, if you do not supply a saveset name, BUP assigns the saveset name BACKUP.BUP.

- BUP normally rewinds magtapes before each backup operation. If you intend to create a number of savesets on a magtape, you can inhibit the magtape rewinding by including the /NOREWIND option in the backup command line. However, BUP cannot check that the saveset name you use is unique, unless the tape rewinds before each backup operation. Therefore, DIGITAL recommends you explicitly assign unique saveset names, especially when you use the /NOREWIND option.

- If BUP detects a write-locked magtape volume while writing the first record to that volume, BUP prompts you to remount that magtape volume. You can then write-enable that magtape volume and remount it. This is especially helpful when making multivolume backups; when a saveset is being written to more than one magtape volume.

  Attempting to write other than the first record to a write-locked magtape returns a fatal level error message.

14.9.1.3 Directory Operations

You should create and save a directory of all savesets on each series of magtape backups and a directory of all files on each saveset. You should create those directories after performing the backup operation.

You can restore the first section of a saveset or a file located on a saveset by mounting the backup magtape containing the saveset section. A directory of the savesets on a series of backup magtapes can point you to the correct magtape to mount. A directory of the files on a saveset can point you to the saveset containing a file you want to restore.

Because magtapes can contains multiple savesets, BUP must read a volume directory to the logical end of magtape volumes before completing the directory. For certain magtape devices, this process can take some time.

Directory Format

BUP magtape directories include the following information:

- The saveset name.
- The section of that saveset residing on the magtape volume. If a saveset is spread across more than one magtape volume, the magtape volume containing the first section of the saveset is identified as section 1, that containing the second section as section 2, and so on.
- The size of that saveset in blocks on the magtape volume followed by the total size of the saveset. If the two numbers are the same, the entire saveset is on this magtape volume.
- The date on which that saveset was backed up to the magtape volume.

The following display is an example of the BUP magtape directory. The magtape backup volume contains the second section (2348 blocks) of a 5400-block saveset named BIGDSK.BUP, the complete savesets FIRST.TXT and SECOND.BUP, and the first (408-block) section of a 988-block saveset THIRD.BUP.

```
 Saveset       Section     Blocks        Date

BIGDSK.BUP        2      2348/5400      20-MAR-88
FIRST .TXT        1       800/800       20-MAR-88
SECOND.BUP        1      5400/5400      21-MAR-88
THIRD .BUP        1       408/988       26-MAR-88
```

You would need to mount a previous magtape in this series to restore the saveset BIGDSK.BUP, because the first section of that saveset is not located on this volume. A previously created printed directory of savesets for each magtape backup volume would direct you to the correct volume to mount. You can restore the savesets FIRST.TXT and SECOND.BUP from this volume. Proceed to the next magtape volume of this series to restore the second section of THIRD.BUP.

Directory Destination

You can send a magtape backup volume directory to your terminal, a specified output file, or your printer.

You can display a directory of the savesets residing on a specified magtape to your terminal with the BACKUP/DIRECTORY command. For example, the following command displays the directory of savesets for magtape MS0:

```
.BACKUP/DIRECTORY MS0: RET
```

You can send the directory of the savesets on a magtape to the device and file you specify by appending the /OUTPUT[:filespec] option. The default file type is .DIR. For example, the following command stores the directory of magtape MS0 on device DL1 with the file name MAG1.DIR:

```
.BACKUP/DIRECTORY/OUTPUT:DL1:MAG1 MS0: RET
```

You can send the directory of the savesets on a magtape to your printer by appending the /PRINTER option. For example, the following command sends a directory of the savesets on magtape MS0 to your printer:

```
.BACKUP/DIRECTORY/PRINTER MS0: RET
```

14.9.1.4 Restoration Operations

From a magtape backup volume, you can restore a device image saveset or one or more files residing on a device image saveset. The BACKUP command /VERIFY option verifies that the contents of a device or file image are the same as the contents of the saveset. As insurance, you can use the /VERIFY option in restoration operations to ensure that the restored data can be read from the destination device.

Restoring a Device Image Saveset

You restore a device image saveset from a magtape or series of magtapes, using the following command. Because you are performing a device restoration to a disk, the disk is initialized as part of the operation. In the command, FOO is the named saveset contained on magtape MS0. You are restoring FOO to disk device DL1:

```
.BACKUP/RESTORE/DEVICE/VERIFY MS0:FOO DL1: RET
```

A device image restoration from a magtape backup volume where no saveset is specified causes the first saveset on the magtape to be restored.

Restoring a File Located Within a Device Saveset

You use the /SAVESET option to specify which device image saveset from which to restore one or more files. Wildcards are permitted. Because you are performing a file restoration to a disk, BUP does not initialize that disk as part of the operation. In the command line, you couple the /SAVESET option with the saveset name and supply one or more file names (with or without wildcards).

The following example restores all files of type TXT, located within saveset FOO.BUP contained on magtape MS0. This example assumes that magtape MS0 contains all requested files; if not, you would be prompted to mount the next magtape input volume.

```
.BACKUP/RESTORE/VERIFY MS0:FOO/SAVESET,*.TXT DL1: RET
Mount input volume 1 in MS0: Continue? Y RET
?BUP-I-Restore operation started on volume 1
?BUP-I-Copy operation is complete
.
```

If you do not specify the saveset name in the command line, BUP attempts to restore the specified file or files from the first saveset encountered on the magtape.

Restoring a File Image Created Before RT–11 Version 5.5

Versions of RT–11 before Version 5.5 let you create a file image backup that was not contained within a saveset. Such a file image has a different format than a saveset. You restore such a file image from a magtape or series of magtapes by including the /FILE option and omitting the /DEVICE option. Because you are performing a file restoration to a disk, BUP does not initialize that disk as part of the operation. For example, the following command restores, with verification, the file image FIRST.TXT from magtape MS0 to device DL1:

```
.BACKUP/RESTORE/FILE MS0:FIRST.TXT DL1: RET
```

14.9.1.5 Transporting Backed-Up Magtapes to VMS

You can transport BUP-written magtapes to a VMS system and extract files from those magtapes. Use the following procedure:

1. Back up an RT–11 device or file image to a magtape. The following example command backs up the device image LD3 to magtape *Mdn*, assigns the saveset name MYDISK to that backup image, and verifies the operation:

   ```
   .BACKUP/DEVICE/VERIFY LD3: Mdn:MYDISK RET
   ```

2. Mount that backup magtape on a VMS system. All magtapes created by BUP have the volume label, RTBUP. Therefore, use the following command, where *mddn* represents the magtape drive and RTBUP is the volume label:

   ```
   $ MOUNT mddn: RTBUP RET
   ```

3. Copy the BUP backup image to disk, using the following command, where MYDISK is the saveset name for the RT–11 device you backed up. This command produces a virtual disk image file:

   ```
   $ COPY mddn:MYDISK.BUP * RET
   ```

4. Use the VMS EXCHANGE utility or RTEM to manipulate files on that virtual disk file. See the VMS EXCHANGE utility documentation for information on using EXCHANGE. The following command is shown as an example. In the example, *vdn* is a virtual disk name:

   ```
   $ EXCHANGE  RET
   EXCHANGE>MOUNT/VIRTUAL vdn: MYDISK.BUP  RET
   EXCHANGE>DIR vdn:  RET
   ```

14.9.2 COPY/DEVICE Command with Magtapes

COPY/DEVICE should be used with the /FILE option when you want to create duplicates of disk volumes as files on magtape volumes.

For example, the following command duplicates disk volume DL0 as file DL0.DSK on magtape MS0:

```
.COPY/DEVICE/FILE DL0: MS0:DL0.DSK  RET
```

The number of device images you can store on a magtape is limited only by the magtape storage capacity. However, this functionality is provided only for compatibility with previous versions of RT–11 and should not be used as the normal disk volume to magtape volume backup facility. Use BACKUP/DEVICE instead.

Chapter 15

Using Indirect Command Files

This chapter describes how to create and use indirect command files. An indirect command file is composed of a series of operating system commands that are executed by RT–11 in sequential order.

Each indirect command file typically performs a single general task, such as configuring the operating system, starting or stopping a job, configuring a virtual memory (VM) device, or performing backup operations. Also, indirect command files can be used to perform operations that involve much computer processing but do not require your supervision or intervention. For example, multiple assemblies, compilations, and data transfer (copy) operations are ideal operations for indirect command files.

You can include in the indirect command file a command to run a second indirect command file, which is called *nesting* indirect command files. You can also create indirect command files that contain only commands to run any number of other indirect command files.

This chapter contains the following information:

- Comparison between indirect command file functionality and IND (indirect control file processor) and BATCH
- Creating indirect command files
- Running indirect command files
- Nesting indirect command files

Use the on-line index utility, INDEX, and see the on-line help utility, HELP, for more information on topics described in this chapter.

15.1 Comparison with IND and BATCH

RT–11 supports three methods for creating and running indirect files:

- Indirect command files, described in this chapter and illustrated in various chapters in this manual
- IND indirect control files, described in the *RT–11 Commands Manual*
- BATCH files, described in the *RT–11 System Utilities Manual*

Each method has its own characteristics. While IND and BATCH provide the most functionality, they are more complex to use and are not discussed in this chapter. Indirect command files are simple to use, but contain the following limitations:

- Process commands in only sequential order.
- Do not support direct user intervention once started.
- Do not support conditional (if, else) logic.

Those limitations do not exist with IND and BATCH. If you require indirect files without those limitations, you should also investigate IND and BATCH and in that order. The distributed file that runs automatic installation, STARTA.COM, and the system generation procedure file, SYSGEN.COM, are examples of IND files. RT–11 distributes no BATCH files.

15.2 Creating an Indirect Command File

Use the editor to create an indirect file as a text file. You can call the file by any file name you wish, but you should give it a file type of .COM, since this file type is the default used by the monitor to locate the file. You should choose a name for the file that suggests the task that the file performs. You should preface the commands in a file with a comment header containing the name, creation and edit dates, and a short description of the file's function.

You enter commands into the indirect file as you would on the terminal. You do not include the monitor prompt. You can include comment lines in the file, so long as each comment line begins with an exclamation mark (!).

The following rules and information apply to creating indirect command files:

- You can use any valid command (DCL, CCL, or CSI) syntax. You can also include commands you have defined, using UCL or UCF, so long as the command is complete and requires no additional information from the terminal.
- You should use the /NOQUERY option (or CSI equivalent) with commands that would otherwise require a response from the terminal.

  Some commands, such as BOOT, do not accept a /NOQUERY option. You cannot, for example, execute a BOOT command from within an indirect command file if the foreground is loaded. RT–11 will always request confirmation of the operation. You should, therefore, unload any foreground jobs before running the indirect command file or abort any jobs and unload the foreground from within the file before issuing the BOOT command.
- You can factor commands, and all rules applying to factoring commands apply.
- You can use wildcards, but you should be careful that wildcard operations execute in the manner you expect each time you run the indirect command file.

  Remember that the system configuration and the contents of storage volumes will change, and a wildcard operation might produce unexpected results. Commands in indirect command files should be as specific as possible, so that the operation performed by the command is completely predictable.

- You should specify multiple operations of the same type on the same command line, allowing for command syntax restrictions. For example, the following command line:

  ```
  COPY DU0:(RT11XM.SYS,SWAP.SYS,DUX.SYS,VMX.SYS,MSX.SYS) VM:
  ```

  executes faster than:

  ```
  COPY DU0:(RT11XM.SYS) VM:
  COPY DU0:(SWAP.SYS) VM:
  COPY DU0:(DUX.SYS) VM:
  COPY DU0:(VMX.SYS) VM:
  COPY DU0:(MSX.SYS) VM:
  ```

  because the first command line calls the PIP utility only once to perform the COPY operation, where the second set of commands must call PIP five times.
- Each command in an indirect command file is placed in a memory buffer before it is executed. The indirect command file aborts if the memory buffer overflows. You can flush the buffer by including a CTRL/C periodically after a number of commands. In practice, overflowing the buffer is not a problem unless you run a large indirect command file, nest files together, or run an indirect command file that in turn calls multiple files.
- You can include a command to run another indirect command file. Nesting indirect command files is described, with examples, in Section 15.4.
- You can include a command to run an IND control file. However, such a command must be the last command in the indirect command file because you cannot return to an indirect command file from an IND control file. Also, you cannot call an indirect command file from within that IND control file.

As an exercise, use the following procedure to create an example indirect command file, EXAMP.COM. The example file does the following:

- Creates a logical disk on your default data (DK) volume named EXAMP.DSK, containing 50 blocks.
- Verifies that EXAMP.DSK resides on DK.
- Displays the status of logical disk units.
- Mounts EXAMP.DSK on logical disk unit LD7.
- Initializes LD7.
- Displays a directory of LD7.
- Dismounts LD7.
- Removes the file protection for EXAMP.DSK.
- Deletes EXAMP.DSK.
- Verifies that EXAMP.DSK is deleted from DK.

Note that the file has no lasting effect on the system or your storage disk; it cleans up after itself. The procedure to create the file is:

1. Create EXAMP.COM:

   ```
   .EDIT/CREATE EXAMP.COM RET
   ```

2. Enter the following:

   ```
   !EXAMP.COM, created dd-mmm-yy, edited dd-mmm-yy
   ! Example indirect command file for Intro to RT

   CREATE EXAMP.DSK/ALL:50.
   DIRECTORY EXAMP.DSK
   SHOW SUBSET
   MOUNT LD7: EXAMP.DSK
   INITIALIZE/NOQUERY LD7:
   DIRECTORY LD7:
   DISMOUNT LD7:
   UNPROTECT EXAMP.DSK
   DELETE EXAMP.DSK
   DIRECTORY EXAMP.DSK
   ```

3. Close the file by exiting from the editor.

This manual contains many examples of indirect command files. If you have worked through the chapters in the Part II, and particularly Section 7.2.2 and Section 7.6, you have created indirect command files to start, abort, and unload a system job. If you have worked through the exercises in Chapter 11, you have created indirect command files to create and use a virtual memory (VM) system device. Such applications are excellent uses for indirect command files.

15.3 Running an Indirect Command File

You can run an indirect command file by preceding the file name (the file type .COM is assumed unless you indicate otherwise) with an at sign (@) and pressing RETURN. The following example command line runs the indirect command file INDFIL.COM:

```
.@INDFIL RET
```

That syntax is valid so long as the keyboard monitor (KMON) remains set to interpret the *@filename* construction as being an indirect command file. That syntax becomes invalid if KMON is set using SET KMON IND to interpret the *@filename* construction as an indirect *control* file. If you precede the at sign with a dollar sign (*$@filename*), KMON always interprets the file as an indirect command file. Therefore, the following construction is always valid for running indirect command files and is the recommended construction:

```
.$@INDFIL RET
```

As an exercise, now run the indirect command file, EXAMP.COM:

```
.$@EXAMP RET
```

Execution starts immediately, and the system processes commands in consecutive order. By default, each command is echoed on the terminal as it is processed.

Terminal echo can be suppressed by including the SET TT QUIET command at the beginning of the file.

You can temporarily suspend the file execution by pressing HOLD SCREEN or by pressing CTRL/S, and resume execution by pressing HOLD SCREEN again or pressing CTRL/Q. You can abort the file execution by pressing CTRL/C twice.

If an error occurs during command processing, the system prints a message on the terminal and stops execution of the file from the point at which the error occurred. Therefore, it is important that you check your indirect command file for errors before you start it if you are going to run it and leave the area.

You can change the level of error that aborts an indirect command file, using the SET ERROR command. By default, an ERROR level error aborts a file. The level of error that causes an abort can be reduced to WARNING. If you are creating and testing an indirect command file and want to ensure that the file executes completely as expected, you can change the error level to WARNING:

```
.SET ERROR WARNING  RET
```

Once you have run the file and verified it executes as expected, you can, at your option, change the abort error level back to ERROR.

15.4 Nesting Indirect Command Files

As mentioned previously, indirect command files typically perform a single task. They are a modular tool. You can combine the tasks performed by an indirect command file by nesting one indirect command file within another. You enter the command in the file just as you would from the keyboard, using the ($@filename) construction.

RT–11 lets you nest indirect command files to three levels; you can run an indirect command file that calls another file, which in turn calls another file. Attempting to nest files beyond the third level returns a fatal level error message and aborts the third level nested file.

Assume that you have created an indirect command file, PRINT.COM, and you wish to run PRINT.COM after EXAMP.COM. You nest PRINT.COM in EXAMP.COM as follows:

```
!EXAMP.COM, created dd-mmm-yy, edited dd-mmm-yy
! Contains nested PRINT.COM
! Example indirect command file for Intro to RT

CREATE EXAMP.DSK/ALL:50.
DIRECTORY EXAMP.DSK
SHOW SUBSET
MOUNT LD7: EXAMP.DSK
INITIALIZE/NOQUERY LD7:
DIRECTORY LD7:
DISMOUNT LD7:
UNPROTECT EXAMP.DSK
DELETE EXAMP.DSK
DIRECTORY EXAMP.DSK
$@PRINT
```

There can be times when such an indirect command file is useful. You have, however, made EXAMP.COM unusable unless you also want to run PRINT.COM.

In keeping with the concept of modular indirect command files, it may be more useful to create a third file that calls both EXAMP.COM and PRINT.COM, thus keeping those files modular and usable by themselves. For example, you could create an indirect command file named WHOLE.COM and enter the following:

```
!WHOLE.COM, created dd-mmm-yy, edited dd-mmm-yy

$@EXAMP
$@PRINT
```

Running WHOLE.COM would then run EXAMP.COM, followed by PRINT.COM.

Finally, as described in Section 13.2.5, you can define a command that calls an indirect command file. That file can contain commands and can call other indirect command files.

Part IV
Programming with RT–11

Part IV describes RT–11 support for developing programs in the FORTRAN IV, FORTRAN–77, BASIC–PLUS, and MACRO–11 programming languages. Part IV is separated into the following seven chapters:

- Chapter 16, Choosing a Programming Language, provides introductory material on choosing a programming language. High-level and machine-level languages are discussed. This chapter lists the files required to develop programs in the languages presented in this part.
- Chapter 17, Running a FORTRAN Program, describes developing programs in the FORTRAN IV and FORTRAN–77 programming languages.
- Chapter 18, Running a BASIC–PLUS Program, describes developing programs in the BASIC–PLUS programming language.
- Chapter 19, Running a MACRO–11 Assembly-Language Program, describes developing programs in the MACRO–11 programming language.
- Chapter 20, Linking Object Programs, describes using the RT–11 linker.
- Chapter 21, Constructing Library Files, describes creating and using object module libraries.
- Chapter 22, Debugging a User Program, describes the debugging process in general and specifically describes using the distributed symbolic debugger, DBG–11, to debug MACRO–11 programs.

Chapter 16
Choosing a Programming Language

Whenever you plan to write a program, you must first decide on the programming language that you will use, since most computer systems support several. After you have chosen the language, you must design and code your program using appropriate language statements, carefully following formatting rules and restrictions. Finally, you must use the corresponding language processor to convert your program statements into a format suitable for execution.

16.1 High-Level Versus Machine-Level Languages

Hundreds of programming languages have been developed for computer systems. Some languages can be used only for specific applications or with a particular computer system. Other languages are general purpose; they are suitable for a variety of problem-solving situations and, in addition, are easier to learn and use. The languages demonstrated in this manual include two well-known and widely used high-level programming languages (BASIC–PLUS and FORTRAN) and one RT–11 system-specific machine-level programming language (MACRO–11).

NOTE

Throughout the *Introduction to RT–11*, FORTRAN means FORTRAN IV and FORTRAN–77. Information that applies to both compilers uses the term FORTRAN. Information specific to either compiler uses the specific compiler name.

Also, references to FORTRAN–77 mean FORTRAN–77 /RT–11 and references to BASIC–PLUS mean BASIC–PLUS/RT–11.

High-level languages, like BASIC–PLUS and FORTRAN, are easier to learn and use. You write programs by using language statements that need not deal with the specifics of the computer system. The language processor—and perhaps other utility programs as well—handles all conversions that are necessary for program execution. Since a single high-level language statement may perform several computer operations and, since you need not be concerned or familiar with the structure of the computer and peripheral devices, you can concentrate solely on solving the problem at hand. The language processor takes care of translating the statements into computer information.

Thus, high-level languages are considered machine-independent because a program written in the language can usually be compiled, linked, and executed on a different computer system (that supports the language) with few, if any, modifications.

Machine-level languages, on the other hand, such as the assembly language MACRO–11, require that you know about the computer and the peripheral devices and how they work together. You write programs in formats that are closer to those required for execution. Since a single machine-level language statement usually performs only one computer operation, you must account in your program for each computer operation that will be required.

For this reason, machine-level languages are machine-dependent languages. The program is coded in a format that is not usually interchangeable among systems. Machine-level language programs can be more efficient because the knowledgeable programmer can choose the fastest and most precise instructions for getting a job done.

Table 16–1 lists a comparison of high-level versus machine-level languages.

Table 16–1: Language Comparisons

High-Level	**Machine-Level**
Easy to learn and use; no experience required	More difficult to learn and use; familiarity with the computer system required
Machine-independent	Machine-dependent
Many hidden conversions necessary for program execution; more computer memory is used	Only direct translation is necessary for program execution; less computer memory is used
Slower execution time	Faster execution time
Less efficient; the system makes decisions concerning computer operations	More efficient; the programmer makes decisions concerning computer operations
Easier to debug (find and fix errors)	Harder to debug (find and fix errors)
Easier to understand programs; functions added with less difficulty	Harder to understand programs; functions added with greater difficulty

Beginning programmers, students, commercial applications programmers, and the casual computer user prefer high-level languages because they are less difficult to learn and to use, and they produce fast results. System programmers, on the other hand, may prefer machine-level languages for writing programs (those that make up an operating system, for example) that must often be as fast and efficient as possible.

The designers of a computer system generally select programming languages that will satisfy and suit the current (or perhaps potential) system user environment. The RT–11 computer system is used in many environments: education, business, laboratory, and so on. Some of its applications include data acquisition and analysis, record keeping, control systems, and learning through computer simulation. RT–11 programmers and users include both the very knowledgeable and the student /beginner.

To satisfy the varied requirements of these environments, several programming languages run under RT–11:

High-Level	Machine-Level
BASIC–PLUS	MACRO–11
C	
DIBOL	
FORTRAN IV	
FORTRAN–77	
PASCAL	

Whenever you choose one or more of these programming languages for your own use, consider the following criteria:

- What is your programming experience? What languages do you already know?
- How much time do you have to learn a new language?
- For what applications will you use the language? How important are program speed and efficiency?
- Will you use your program on any other computer systems?

If you are already familiar with a language supported by the system, you may prefer to continue using that language rather than spend time learning a new one. However, if you want to learn a language, consider your application. High-level languages handle most programming jobs. Unless you plan to write extremely detailed or time-critical programs, you should select a high-level language.

If you are a beginning programmer, you may prefer to start with a language like BASIC–PLUS, which is a conversational, interactive language. Language statements are simple, English-like words and common mathematical expressions. You can request immediate answers to problems by using the immediate modes of the language or you can create detailed programs by combining single language statements into larger segments. BASIC–PLUS is a superset of the industry-standard BASIC developed at Dartmouth College. Chapter 18 of this manual describes BASIC–PLUS in more detail.

FORTRAN has long been recognized for its use in the scientific field; in addition, it is one of the most commonly supported languages across systems. You may decide to choose FORTRAN because it is a more powerful language than BASIC–PLUS. Chapter 17 describes FORTRAN in more detail.

Finally, if you are an experienced user, you may select the machine-level programming language MACRO–11. This powerful language lets user programs access and utilize every feature available on the RT–11 computer system. The language requires a considerable amount of computer experience and knowledge to be used effectively, however. The MACRO–11 language is best for you if you are a system programmer or an experienced high-level language programmer. It is described in more detail in Chapter 19.

16.2 Choosing a Language for the Demonstration

Three RT–11 programming languages are demonstrated in the following chapters: FORTRAN, BASIC–PLUS, and MACRO–11. Each language requires that one or more files reside on your system. Only the files required to demonstrate MACRO–11 are distributed with RT–11. The other language processors are optional, and you should be sure your system includes the appropriate files before attempting the demonstration for a programming language.

The required files for each programming language are:

Language	**Required Files**	**Description**
FORTRAN IV	FORTRA.SAV	Compiler for FORTRAN IV
	FORLIB.OBJ	Routine library for FORTRAN IV
	SYSLIB.OBJ	RT–11 subroutine library
	LINK.SAV	RT–11 linker
FORTRAN–77	F77XM.SAV	Compiler for FORTRAN–77 under mapped monitors
	F77.SAV	Compiler for FORTRAN–77 under unmapped monitors
	F77COM.MSG	Error message file
	F77OTS.OBJ	Routine library for FORTRAN–77
	SYSLIB.OBJ	RT–11 subroutine library
	LINK.SAV	RT–11 linker
BASIC–PLUS	BASIC.SAV	Interpreter (compiler) for BASIC–PLUS
MACRO–11	MACRO.SAV	Assembler for MACRO–11
	SYSMAC.SML	Routine library for MACRO–11
	LINK.SAV	RT–11 linker
	CREF.SAV	Cross-reference table utility

Select one or more languages to continue the demonstration. If you choose FORTRAN, see Chapter 17. If you choose BASIC–PLUS, see Chapter 18. If you choose MACRO–11, see Chapter 19.

Chapter 17
Running a FORTRAN Program

The FORTRAN programming languages, FORTRAN IV and FORTRAN–77, are machine-independent programming languages that are useful with a variety of applications.

You do not need to understand FORTRAN to successfully complete the exercises in this chapter.

NOTE

Throughout the *Introduction to RT–11*, the term FORTRAN means FORTRAN IV and FORTRAN–77. Information that applies to both programming languages uses the term FORTRAN. Information specific to a particular version of FORTRAN uses the specific language name.

17.1 Developing an Executable FORTRAN Program

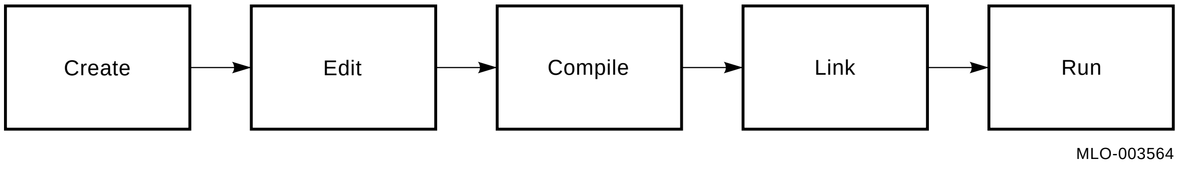
FORTRAN (FORmula TRANslation) is an algebraically oriented language. You write a FORTRAN program as a sequence of language statements that combine common English words with quasi-algebraic formulas. You use tabs and spaces to format each line properly. You should insert comment statements throughout the source code to explain what various parts of the program are doing. You then arrange groups of the language statements into logical units called program units. One or more program units make up an entire executable FORTRAN source program.

When you have finished creating the program as a complete, edited file, you next enter it as input to the appropriate FORTRAN compiler. The FORTRAN compiler processes the language statements, converting them into internal machine-language code called object code. This code is next processed by the system linker, which combines your program units and necessary system-supplied routines to make your program suitable for execution. The development of an executable FORTRAN program is represented in Figure 17–1.

17.2 Establishing the Default FORTRAN Compiler

RT–11 supports the FORTRAN IV and FORTRAN–77 versions of FORTRAN. From looking at Section 16.2, you should know which version of FORTRAN resides on your system. If you have not already determined which version of FORTRAN resides on your system, do that now by going back to Chapter 16.

Figure 17–1: Evolution of a FORTRAN IV Program



RT–11 provides SET commands that let you specify which FORTRAN compiler is called. Once you have issued the appropriate SET command, RT–11 automatically calls the correct FORTRAN compiler. Make sure the compiler is on your system (SY) volume and do the following.

- If the FORTRAN IV compiler (FORTRA.SAV) resides on your system volume, issue the following command:

  ```
  .SET FORTRA F4 RET
  ```

- If the FORTRAN–77 compiler (F77XM.SAV or F77.SAV) resides on your system volume, issue the following command:

  ```
  .SET FORTRA F77 RET
  ```

  In addition, be sure the error message file, F77COM.MSG, resides on your system volume:

  ```
  .DIRECTORY SY:F77COM.MSG RET
  ```

  If F77COM.MSG does not reside on your system volume, copy it to there.

17.3 Using the FORTRAN Compiler

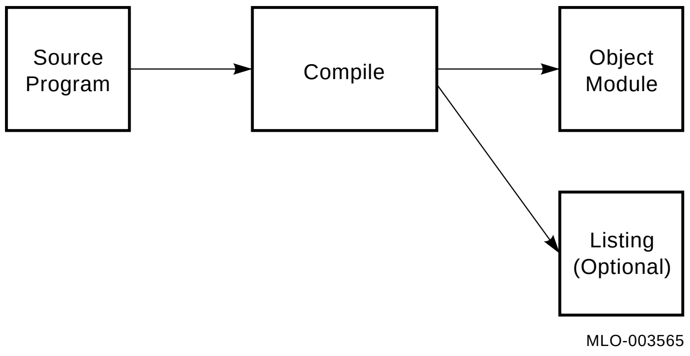
The FORTRAN compiler translates your source program into a machine language program.

Since you create a FORTRAN source program in ASCII format, you must next translate the program into a machine format that the computer can use. The FORTRAN compiler performs the translation, producing as output a new version of the program called an object module. You may instruct the FORTRAN compiler to produce a listing of the source program at the same time. Figure 17–2 is a diagram of the compiler's function.

17.4 Using the Library and Error Message Modules

FORTRAN programs often require similar operations. Many programs, for example, use routines and instructions that calculate square roots, exponentials, and other arithmetic functions; handle input and output operations; detect certain kinds of error conditions; test values; calculate subscripts; and perform conversions. These

Figure 17–2: Function of a FORTRAN Compiler



commonly used operations are gathered into library module files. Two library files are used by FORTRAN programs:

- FORLIB.OBJ (for FORTRAN IV) or F77OTS.OBJ (for FORTRAN–77)

  FORLIB.OBJ and F77OTS.OBJ contain library object modules that perform FORTRAN operations for the appropriate compiler. The appropriate library file is distributed with the compiler. The object modules (routines) located within the FORTRAN library file are described in the appropriate FORTRAN documentation.

- SYSLIB.OBJ

  SYSLIB.OBJ contains library modules (system routines) that let a FORTRAN program use RT–11 system services. SYSLIB.OBJ is distributed with RT–11 and described in the *RT–11 Programmer's Reference Manual*.

During the processing of your source program, the FORTRAN compiler examines each language statement in the program. The compiler translates all the information gathered during processing (your converted language statements and the references to library object modules) into numerical data called object code that the system linker can use. The result of the compilation, therefore, is an object format file, called an object module, which is then joined with the called library object modules at link time. Linking all the necessary object modules together produces a complete, workable FORTRAN program.

Operations that are provided by FORTRAN library modules are performed internally and not noted. If you use operations that are provided in SYSLIB, the compiler notes them and makes references to SYSLIB.

FORTRAN IV and FORTRAN–77 both produce diagnostic messages when they discover syntax errors in a source program. The means by which FORTRAN IV interprets and reports diagnostic messages is internal to the compiler; no other file is required. FORTRAN–77 uses a separate file, F77COM.MSG, to interpret and report diagnostic messages. Without error interpretation, FORTRAN–77 does not produce a complete listing if compilation errors are encountered. F77COM.MSG must reside on your system volume.

17.5 Demonstration FORTRAN Program

RT–11 distributes a FORTRAN demonstration source file, DEMOF3.FOR, for this chapter. DEMOF3.FOR can be compiled by the FORTRAN IV and FORTRAN–77 compilers and contains intentional errors to illustrate compiler diagnostic message displays.

Copy DEMOF3.FOR to GRAPH.FOR, print GRAPH.FOR, and then look at your printed copy while reading the following explanation of the program logic.

```
.COPY DEMOF3.FOR GRAPH.FOR RET
.PRINT GRAPH.FOR RET
```

The following is a reproduction of the demonstration program. Following the reproduction is a description of the program logic.

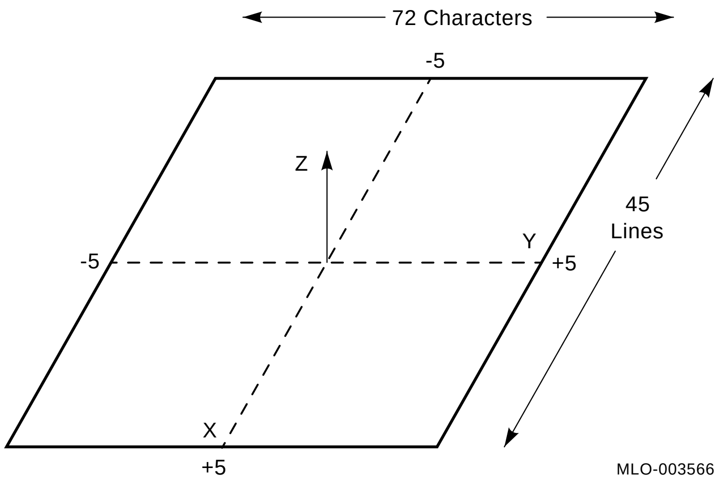
```
C DEMOF3.FOR     VERSION PROVIDED
C THIS PROGRAM PRODUCES A PLOT ON THE TERMINAL
C OF AN EXTERNAL FUNCTION, FUN(X,Y)
C THE LIMITS OF THE PLOT ARE DETERMINED BY THE DATA STATEMENTS
C "STAB" IS FILLED WITH A TABLE OF HEIGHT FLAGS
C "STRING" IS USED TO BUILD A LINE OF GRAPH FOR PRINTING
C
        LOGICAL*1 STRING(13,3),STAB(100)
        DATA XMIN,XMAX,MAXX/ -5.0, 5.0, 45/
        DATA YMIN,YMAX,MAXY/ -5.0 ,5.0, 72/
        DATA FMIN,FMAX/0.0,1.0/
C
        SCAL(ZMIN,ZMAX,MAXZ,K)=ZMIN+FLOAT(K-1)*(ZMAX-ZMIN)/FLOAT(MAXZ-1)
C
        CALL SCOPY('- 1 2 3 4 5 6 7 8 9 +',STAB)
        MAXF=LEN(STAB)
        DO 20 IX=1,MAXX
          X=SCAL(XMIN,XMAX,MAXX,IX)
          CALL REPEAT('*',STRING,MAXY)
          IF(IX.EQ.1 .OR. IX.EQ.MAXX) GOTO 20
            DO 10 IY=2,MAXY-1
              Y=SCAL(YMIN,YMAX,MAXY,IY)
              IFUN = 2 + INT(FLOAT(MAXF-3)*(FUN(X,Y)-FMIN)/(FMAX-FMIN))
10            STRING(IY)=STAB(MIN0(MAXF,MAX0(1,IFUN)))
30        WRITE (7, 1001) (STRING(I),I=1,MAXY)
1001      FORMAT (' ', 72A1)
        CALL EXIT
        END
C
        FUNCTION FUN(X,Y)
        R=SQRT(X**2+Y**2)
        FUN=(X*Y*R*EXP(-R))**2
        RETURN
        END
```

The first few lines of this program are user comment lines that briefly describe the program. Essentially, this program produces on the terminal a graph of a "3-dimensional" function, FUN(X,Y). The graph is plotted using 45 lines down and 72 characters across the terminal page. The limits of the X and Y axes are +5.0 and –5.0. The third dimension, height, is a real number within the range 0 to 1 and is represented in the listing as a number within a scale of 1 to 9. These dimensions are illustrated in Figure 17–3.

The SCAL function determines the value of the next coordinate on the graph. The statements within the DO loops calculate the coordinates by using the SCAL function and determine the height value. This is done for an entire line of coordinates across the terminal page. The entire line is then printed on the terminal, using the WRITE statement; the number 7 in this statement is the FORTRAN method of naming the terminal as the output device. This procedure is repeated until all 45 lines of the graph have been printed.

The function to be plotted is shown in the last few lines of the program. It is compiled as a separate program unit, and you can edit these lines to plot any function of your choice (several alternate functions are suggested later in the chapter).

Figure 17–3: Dimensions of FUN(X,Y)



17.6 Compiling the FORTRAN Demonstration Program

Since a source program is in ASCII text format, the next step is to use a FORTRAN compiler to convert it to object code. As a result of the SET FORTRA command you issued at the beginning of this chapter, RT–11 recognizes the correct FORTRAN compiler when you use the FORTRAN command to compile the source program.

Use the FORTRAN command with its /LIST option to compile your program and produce a listing. Note the /LIST option is coupled with the file specification; coupling the /LIST option with the FORTRAN command sends the listing to your printer. The system assigns the name GRAPH.OBJ to the object module and GRAPH.LST to the listing file and stores both newly created files on your default data (DK) volume, which is the default storage volume for input/output operations.

```
.FORTRAN GRAPH.FOR/LIST RET
```

Compilation begins. If the compiler discovers an error during processing, it prints a message.[1] Because errors were intentionally entered into the source code, diagnostic messages are displayed on your screen.

If the compiler is FORTRAN IV, the following diagnostic messages are displayed:

```
.MAIN.
?FORTRAN-I-[.MAIN.] Errors: 4, Warnings: 0
FUN
?FORTRAN-I-[FUN   ] Errors: 1, Warnings: 0
```

If the compiler is FORTRAN–77, the following diagnostic messages are displayed:

```
F77 -- ERROR 72-E Arguments incompatible with function, assumed user supplied
                  [MAXF=LEN(STAB)] in module .MAIN. at line 7
F77 -- ERROR 42-F Number of subscripts does not match array declaration
                  [    STRING(IY)=] in module .MAIN. at line 15
F77 -- ERROR 42-F Number of subscripts does not match array declaration
                  [01) (STRING(I),] in module .MAIN. at line 16
F77 -- ERROR 54-F Unclosed DO loops or IF block
                  [END] in module .MAIN. at line 18
F77 -- ERROR 50-F Undefined statement label
                  [ 20] in module .MAIN]
F77 -- ERROR  8-E Extra characters following a valid statement
                  [=X*Y*R*EXP(-R))] in module FUN at line 3
```

The diagnostic messages, along with other information, are recorded in the listing you created.

17.7 Examining the Compiler Listing

The format and information in the compilation listing is determined by the compiler. You should at this point look at the listing produced by the compiler. Print the listing on either the printer or terminal.

To print the listing on your line printer:

```
.PRINT GRAPH.LST RET
```

To display the listing on your terminal:

```
.TYPE GRAPH.LST RET
```

[1] See the *RT–11 System Message Manual* for greater detail about any system messages printed.

17.7.1 FORTRAN IV Compiler Listing

Your listing should look like the following example. (To save space, the listing has been compressed by eliminating the page separations.)

```
FORTRAN IV     V02.8      Wed 11-Sep-91 14:00:31          PAGE 001

      C DEMOF3.FOR     VERSION PROVIDED
      C THIS PROGRAM PRODUCES A PLOT ON THE TERMINAL
      C OF AN EXTERNAL FUNCTION, FUN(X,Y)
      C THE LIMITS OF THE PLOT ARE DETERMINED BY THE DATA STATEMENTS
      C "STAB" IS FILLED WITH A TABLE OF HEIGHT FLAGS
      C "STRING" IS USED TO BUILD A LINE OF GRAPH FOR PRINTING
      C
0001          LOGICAL*1 STRING(13,3),STAB(100)
0002          DATA XMIN,XMAX,MAXX/ -5.0, 5.0, 45/
0003          DATA YMIN,YMAX,MAXY/ -5.0 ,5.0, 72/
0004          DATA FMIN,FMAX/0.0,1.0/
      C
0005          SCAL(ZMIN,ZMAX,MAXZ,K)=ZMIN+FLOAT(K-1)*(ZMAX-ZMIN)/FLOAT(MAXZ-1)
      C
0006          CALL SCOPY('- 1 2 3 4 5 6 7 8 9 +',STAB)
0007          MAXF=LEN(STAB)
0008          DO 20 IX=1,MAXX
0009            X=SCAL(XMIN,XMAX,MAXX,IX)
0010            CALL REPEAT('*',STRING,MAXY)
0011            IF(IX.EQ.1 .OR. IX.EQ.MAXX) GOTO 20
0013              DO 10 IY=2,MAXY-1
0014                Y=SCAL(YMIN,YMAX,MAXY,IY)
0015                IFUN = 2 + INT(FLOAT(MAXF-3)*(FUN(X,Y)-FMIN)/(FMAX-FMIN))
0016  10            STRING(IY)=STAB(MIN0(MAXF,MAX0(1,IFUN)))
0017  30        WRITE (7, 1001) (STRING(I),I=1,MAXY)
0018  1001      FORMAT (' ', 72A1)
0019          CALL EXIT
0020          END

FORTRAN IV      Diagnostics for Program Unit .MAIN.

In line 0008,  Error:   Reference to undefined statement label
In line 0012,  Error:   Reference to undefined statement label
In line 0016,  Error:   Wrong number of subscripts for array "STRING"
In line 0017,  Error:   Wrong number of subscripts for array "STRING"

FORTRAN IV      Storage Map for Program Unit .MAIN.

Local Variables, .PSECT $DATA, Size = 000336 (  111. words)

Name   Type  Offset     Name   Type  Offset     Name   Type  Offset
FMAX   R*4   000244     FMIN   R*4   000240     I      I*2   000320
IFUN   I*2   000312     IX     I*2   000302     IY     I*2   000304
K      I*2   000272     MAXF   I*2   000274     MAXX   I*2   000224
MAXY   I*2   000236     MAXZ   I*2   000270     MAX0   I*2   000316
MIN0   I*2   000314     X      R*4   000276     XMAX   R*4   000220
XMIN   R*4   000214     Y      R*4   000306     YMAX   R*4   000232
YMIN   R*4   000226     ZMAX   R*4   000264     ZMIN   R*4   000260

Local and COMMON Arrays:

Name     Type   Section Offset  ------Size----- Dimensions
STAB   L*1      $DATA   000047  000144 (   50.) (100)
STRING L*1 Vec  $DATA   000000  000047 (   20.) (13,3)

Subroutines, Functions, Statement and Processor-Defined Functions:

Name   Type   Name   Type   Name   Type   Name   Type   Name   Type
EXIT    R*4   FLOAT   R*4   FUN     R*4   INT     I*2   LEN     I*2
REPEAT  R*4   SCAL    R*4   SCOPY   R*4

FORTRAN IV      V02.5      Wed 11-Sep-91 14:00:35                PAGE 001

      C
0001          FUNCTION FUN(X,Y)
0002          R=SQRT(X**2+Y**2)
0003          FUN=X*Y*R*EXP(-R))**2
***** P
0004          RETURN
0005          END

FORTRAN IV      Diagnostics for Program Unit FUN
```

```
In line 0003,  Error:     [see source listing]

FORTRAN IV      Storage Map for Program Unit FUN

Local Variables, .PSECT $DATA, Size = 000024 (   10. words)

Name   Type  Offset     Name   Type  Offset     Name   Type  Offset
FUN    R*4   000004 Eqv R      R*4   000010     X      R*4 @ 000000
Y      R*4 @ 000002

Subroutines, Functions, Statement and Processor-Defined Functions:

Name   Type   Name   Type   Name   Type   Name   Type   Name   Type
SQRT    R*4
```

The first part of the listing shows the main program unit and consists of the language statements up to, but not including, the function. This is followed by a diagnostics list, then by a storage map. Next the language statements composing the function program unit are listed, again followed by a diagnostics list and a storage map.

This program as it stands contains errors. The compiler detected certain error conditions during processing that prevent the program from working properly. The compiler printed appropriate messages in the diagnostics sections of the program listing. Look first at the messages following the main program unit. Errors were discovered in lines 8, 12, 16, and 17.

The first two messages in the diagnostics section show that reference has been made from both lines 8 and 12 to an undefined label. (Line 12 is actually the second portion of line 11, the GO TO statement.) Statement label 20 is referenced in each case, but only labels 10 and 30 are shown in the program. This indicates either that a statement is missing or that a typing error exists. Examination of the program logic shows a typing error in line 17. Label 30 should actually be 20. Correct line 17 as follows:

```
20      WRITE (7,1001) (STRING(I),I=1,MAXY)
```

The last two messages in this diagnostics section state that an incorrect number of subscripts was given for the array "STRING". Again, examination of program logic shows that the error is actually in line 1. "STRING" is really a vector (a 1-dimension array), not a matrix (a multi-dimensioned array). Thus, the comma is a typing error and line 2 should be changed as follows:

```
LOGICAL*1 STRING(133),STAB(100)
```

Go next to the diagnostics section for the FUN program unit. The message printed there refers you to the source listing, to line 3. A letter "P" appears next to this line. The documentation distributed with FORTRAN IV describes a P error as an indication of unbalanced parentheses. Note that the parentheses are not properly matched in this line. Thus, line 3 should be corrected as follows:

```
FUN=(X*Y*R*EXP(-R))**2
```

This explains the errors flagged by the compiler in the diagnostics sections. Other sections of the program listing (storage map, for example) provide additional information that is helpful for programmers who wish to check the data types of various symbols and later make sure that object modules have been appropriately linked.

Before you can continue the exercises in this chapter, you must edit, in the source program, those statements that contain errors. If necessary, review the editing commands in Chapter 4. Then, use the KED editor to edit the file GRAPH.FOR so that the program is error free. Do not rename the file. When you are ready, recompile the program, using the FORTRAN command, and obtain a new object module and a new listing. This time the program should compile without error (that is, no diagnostics should list). If diagnostics occur, you have not edited the program correctly and should try again.

17.7.2 The FORTRAN–77 Compiler Listing

Your listing should look like the following example. (To save space, the listing has been compressed by eliminating the page separations.)

```
PDP-11 FORTRAN-77 V5.0A        14:18:04     11-Sep-91           Page 1
DK:GRAPH.FOR      /C:19/F:128/L:2/N:6/R:136/S:BLO

        C DEMOF3.FOR     VERSION PROVIDED
        C THIS PROGRAM PRODUCES A PLOT ON THE TERMINAL
        C OF AN EXTERNAL FUNCTION, FUN(X,Y)
        C THE LIMITS OF THE PLOT ARE DETERMINED BY THE DATA STATEMENTS
        C "STAB" IS FILLED WITH A TABLE OF HEIGHT FLAGS
        C "STRING" IS USED TO BUILD A LINE OF GRAPH FOR PRINTING
        C
0001            LOGICAL*1 STRING(13,3),STAB(100)
0002            DATA XMIN,XMAX,MAXX/ -5.0, 5.0, 45/
0003            DATA YMIN,YMAX,MAXY/ -5.0 ,5.0, 72/
0004            DATA FMIN,FMAX/0.0,1.0/
        C
0005            SCAL(ZMIN,ZMAX,MAXZ,K)=ZMIN+FLOAT(K-1)*(ZMAX-ZMIN)/FLOAT(MAXZ-1)
        C
0006            CALL SCOPY(’- 1 2 3 4 5 6 7 8 9 +’,STAB)
0007            MAXF=LEN(STAB)
F77 -- ERROR 72-E Arguments incompatible with function, assumed user supplied
                [ MAXF=LEN(STAB)] in module .MAIN. at line 7

0008            DO 20 IX=1,MAXX
0009              X=SCAL(XMIN,XMAX,MAXX,IX)
0010              CALL REPEAT(’*’,STRING,MAXY)
0011              IF(IX.EQ.1 .OR. IX.EQ.MAXX) GOTO 20
0012                DO 10 IY=2,MAXY-1
0013                  Y=SCAL(YMIN,YMAX,MAXY,IY)
0014                  IFUN = 2 + INT(FLOAT(MAXF-3)*(FUN(X,Y)-FMIN)/(FMAX-FMIN))
0015    10            STRING(IY)=STAB(MIN0(MAXF,MAX0(1,IFUN)))
F77 -- ERROR 42-F Number of subscripts does not match array declaration
                 [    STRING(IY)=] in module .MAIN. at line 15

0016    30        WRITE (7, 1001) (STRING(I),I=1,MAXY)
F77 -- ERROR 42-F Number of subscripts does not match array declaration
                  [01) (STRING(I),] in module .MAIN. at line 16

0017    1001      FORMAT (’ ’, 72A1)
0018            CALL EXIT
0019            END
F77 -- ERROR 54-F Unclosed DO loops or IF block
                  [  END] in module .MAIN. at line 19

F77 -- ERROR 50-F Undefined statement label
                  [ 20] in module .MAIN.


F77 -- 5 Errors  DK:GRAPH.FOR

PDP-11 FORTRAN-77 V5.0A        14:18:08    11-Sep-91          Page 2
DK:GRAPH.FOR      /C:19/F:128/L:2/N:6/R:136/S:BLO

        C
0001            FUNCTION FUN(X,Y)
0002            R=SQRT(X**2+Y**2)
0003            FUN=X*Y*R*EXP(-R))**2
F77 -- ERROR  8-E Extra characters following a valid statement
                  [=X*Y*R*EXP(-R))] in module FUN at line 3
```

```
0004           RETURN
0005           END

PDP-11 FORTRAN-77 V5.0A       14:18:08    11-Sep-91          Page 3
DK:GRAPH.FOR      /C:19/F:128/L:2/N:6/R:136/S:BLO

PROGRAM SECTIONS

Number   Name       Size               Attributes

  1     $CODE1  000104    34            RW,I,CON,LCL
  3     $IDATA  000004     2            RW,D,CON,LCL
  4     $VARS   000004     2            RW,D,CON,LCL


ENTRY POINTS

 Name   Type  Address    Name   Type  Address    Name   Type  Address

 FUN R*4   3-000000


VARIABLES

 Name   Type  Address    Name   Type  Address    Name   Type  Address

 R      R*4   4-000000   X      R*4   F-000002*  Y      R*4   F-000004*


FUNCTIONS AND SUBROUTINES REFERENCED

 $EXP  $SQRT


Total Space Allocated = 000114    38

F77 -- 1 Errors  DK:GRAPH.FOR
```

The first part of the listing shows the main program unit and consists of the language statements and main program diagnostics, followed by the function and function diagnostics. Finally, the program sections, entry points, variables, and referenced functions and subroutines are listed.

This program as it stands contains errors. The compiler detected error conditions during processing that prevent the program from working properly and printed appropriate messages. Look first at the messages in the main program unit. Errors were discovered in lines 7, 15, 16, and 19.

The diagnostic message at line 7 indicates that the standard FORTRAN–77 LEN function is incompatible with the LOGICAL*1 STAB data structure. FORTRAN–77 assumes that an external function is supplied, and the LEN function in SYSLIB is used. This diagnostic message is actually a warning, is repeated at each compilation of GRAPH.FOR, and can be ignored because the program is written to use the SYSLIB LEN function.

The next two diagnostic messages state that an incorrect number of subscripts was given for the array "STRING". Examination of program logic shows that the error is actually in line 1. "STRING" is really a vector (a 1-dimension array), not a matrix (a multi-dimensioned array). Thus, the comma is a typing error and line 2 should be changed as follows:

```
LOGICAL*1 STRING(133),STAB(100)
```

The last two diagnostic messages in the main program section are connected to a single error. The reference in lines 8 and 11 to label 20 are not satisfied. Statement label 20 is referenced in each case, but only labels 10 and 30 are shown in the program. This indicates either that a statement is missing or that a typing error exists. Examination of the program logic shows a typing error in line 16. Label 30 should actually be 20.

Correct line 16 as follows:

```
20     WRITE (7,1001) (STRING(I),I=1,MAXY)
```

Go to the next section for the FUN program unit. The diagnostic message printed there indicates that the parentheses are not properly matched in this line. Thus, line 3 should be corrected as follows:

```
FUN=(X*Y*R*EXP(-R))**2
```

This explains the errors flagged by the compiler. Other sections of the program listing provide additional information that is helpful for programmers who wish to check the data types of various symbols and later make sure that object modules have been appropriately linked.

Before you can continue the exercises in this chapter, you must edit, in the source program, those statements that contain errors. If necessary, review the editing commands in Chapter 4. Use the KED editor to edit the file GRAPH.FOR. Do not rename the file. When you are ready, recompile the program, using the FORTRAN command, and obtain a new object module and a new listing. This time the program should compile with the single error, previously described, that you can ignore. If other diagnostics occur, you have not edited the program correctly and should try again.

17.8 Linking Object Modules

The object module produced by the FORTRAN command is in itself incomplete. As mentioned earlier, it needs parts of the system library, SYSLIB, and perhaps other object modules and libraries as well, to form a complete functioning program.[1] All required object modules must be joined or linked before the program can work.

Even if your program does not require any other object modules, you must still link it. In addition to joining object modules, the link operation adjusts the object code to account for many program units being placed one after the other. The result of the link operation is a memory image load module, which is actually a picture of what computer memory looks like just before program execution. Figure 17–4 is a diagram of the link operation.

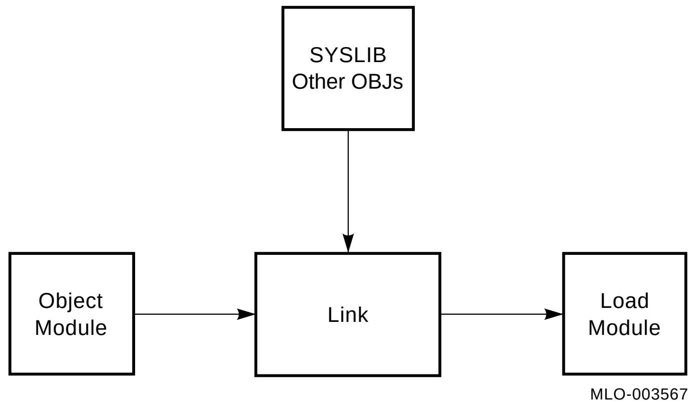
To link the object modules, use the monitor LINK command and enter the names of the input modules and any libraries that contain modules, which are called by the input modules. Generally, this means you link the input modules, the default system libary (SYSLIB.OBJ), and the compiler subroutine library (FORLIB.OBJ or F77OTS.OBJ).

The following rules apply to the LINK command with linking operations:

- LINK.SAV must reside on the system (SY) volume.
- The system assumes the input file or files (object modules) reside on the default data (DK) volume; otherwise, the volume on which they reside must be specified.

[1] For more information on linking files and using library files, see Chapter 20 and Chapter 21, respectively.

Figure 17–4: Link Operation



- You can omit typing the .OBJ file types in the command line, since the LINK command assumes this file type for input.
- The system automatically assigns as the output file specification the file name of the first input file and a file type of .SAV.
- If the default system library (SYSLIB.OBJ) resides on the system (SY) volume, it need not be specified in the command line. Otherwise, SYSLIB.OBJ and the volume containing it must be specified.
- Object module libraries other than SYSLIB.OBJ must be specified in the command line. If a library does not reside on the default data (DK) volume, the volume on which it resides must be specified.

You can see that knowing the location of the program source files and object module libraries is important. You should also know if SYSLIB.OBJ and the compiler library (FORLIB.OBJ or F77OTS.OBJ) have been merged. If you do not know, you can find out by performing a directory operation for SYSLIB.OBJ and looking at the file size. The libraries have not been merged if the displayed file size is approximately 50_{10} blocks. The libraries have been merged if SYSLIB.OBJ is considerably more than twice that size.

17.9 Linking the Demonstration Program

The following LINK commands to create GRAPH.SAV assume that GRAPH.OBJ resides on the default data (DK) volume and that all object module libraries reside on the system (SY) volume. Therefore, SYSLIB is automatically searched and need not be specified.

FORTRAN IV

Issue the following command if FORLIB is not included in SYSLIB:

```
.LINK GRAPH,SY:FORLIB  RET
```

Issue the following command if FORLIB is included in SYSLIB:

```
.LINK GRAPH  RET
```

FORTRAN–77

Issue the following command if F77OTS is not included in SYSLIB:

```
.LINK GRAPH,SY:F77OTS  RET
```

Issue the following LINK command if F77OTS is included in SYSLIB:

```
.LINK GRAPH RET
```

Any messages printed on the terminal identify error conditions discovered by the system during the link operation (for example, you may not have specified all the object modules that are needed as input). However, assuming that you edited your source program correctly and that it compiled without error, it should also now link without error.

A load module is one that you can run on the system. Unless your program contains logic errors that prevent it from running properly (errors that the system generally cannot detect), running the .SAV version of your file should produce the results you intended. However, if logic errors exist within your program, running the program will produce either erroneous results or none at all. If this is the case, you must study the source program, rework it, reedit it, and perform the compile and link operations again.

17.10 Running the Demonstation Program

If your FORTRAN program is error free, running the .SAV version should produce the expected results. In this demonstration, running the GRAPH.SAV file should produce a graph on the terminal screen.

Before you run GRAPH.SAV, you have the option of changing the output device from the terminal screen to the printer by using the monitor ASSIGN command to assign device names (see Chapter 2). If you prefer to print the graph on the printer, assign the logical device name 7 (which is the FORTRAN logical unit number for the terminal) to the printer handler. You have designated a new output device without altering the source program.

To change the device assignment to a serial interface printer (LS), type:

```
.ASSIGN LS: 7 RET
```

To change the device assignment to a parallel interface printer (LP), type:

```
.ASSIGN LP: 7 RET
```

This assignment remains in effect until you deassign the names or reboot the monitor.

Now, to execute the FORTRAN demonstration program, use the monitor RUN command. You can omit typing the .SAV file type since it is assumed within the RUN command.

```
.RUN GRAPH RET
```

After a brief pause, the graph begins to print on the terminal (or printer) and should look like the graph shown in Figure 17–5.

Figure 17–5: Result of GRAPH.SAV

```
************************************************************************
*       111111111111111111                    111111111111111111       *
*     111111111111111111111                  111111111111111111111     *
*   11111111            11111              11111            11111111   *
*  1111111                1111            1111                1111111  *
* 111111     22222222222   111            111   22222222222     111111 *
*11111     22222      2222  111          111  2222      22222     11111*
*1111    2222      3     22  11          11  22     3      2222    1111*
*1111   222    333333333  22  11        11  22  333333333    222   1111*
*111    22   333       333 22 11        11 22 333       333   22    111*
*111   222  333   4444   33 2  1        1  2 33   4444   333  222   111*
*111   222  33   4444444  3  2 11      11 2  3  4444444   33  222   111*
*111   222  33  4444  444 33 2 11      11 2 33 444  4444  33  222   111*
*111   222  33  4444  444  3 2 11      11 2 3  444  4444  33  222   111*
*1111   222  33  44444444 33 2 11      11 2 33 44444444  33  222   1111*
*11111   222  33    444   3  2 11      11 2  3   444    33  222   11111*
*  1111    22   3333   333  2  1        1  2  333   3333   22    1111  *
*   11111    222          22  11        11  22          222    11111   *
*      11111     222222222  111          111  222222222     11111      *
*          11111111      1111              1111      11111111          *
*                   1111                        1111                   *
*                                                                      *
*                                                                      *
*                                                                      *
*                   1111                        1111                   *
*          11111111      1111              1111      11111111          *
*      11111     222222222  111          111  222222222     11111      *
*   11111    222          22  11        11  22          222    11111   *
*  1111    22   3333   333  2  1        1  2  333   3333   22    1111  *
*11111   222  33    444   3  2 11      11 2  3   444    33  222   11111*
*1111   222  33  44444444 33 2 11      11 2 33 44444444  33  222   1111*
*111   222  33  4444  444  3 2 11      11 2 3  444  4444  33  222   111*
*111   222  33  4444  444 33 2 11      11 2 33 444  4444  33  222   111*
*111   222  33   4444444  3  2 11      11 2  3  4444444   33  222   111*
*111   222  333   4444   33 2  1        1  2 33   4444   333  222   111*
*111    22   333       333 22 11        11 22 333       333   22    111*
*1111   222    333333333  22  11        11  22  333333333    222   1111*
*1111    2222      3     22  11          11  22     3      2222    1111*
*11111     22222      2222  111          111  2222      22222     11111*
* 111111     22222222222   111            111   22222222222     111111 *
*  1111111                1111            1111                1111111  *
*   11111111            11111              11111            11111111   *
*     111111111111111111111                  111111111111111111111     *
*       111111111111111111                    111111111111111111       *
************************************************************************
```

MLO-003568

17.11 Combining Operations

To produce these results, you first compiled the FORTRAN source program (GRAPH.FOR), then linked it with the default library (SYSLIB.OBJ), and finally ran the resulting .SAV file (GRAPH.SAV). You can combine these three operations by using one monitor command, the EXECUTE command.

The use of the EXECUTE command requires that the following files reside on your system volume:

- The correct FORTRAN compiler:

  FORTRA.SAV (FORTRAN IV)

  F77XM.SAV (FORTRAN–77 under mapped monitors) or F77.SAV (FORTRAN–77 under unmapped monitors)
- LINK.SAV
- The correct FORTRAN object module library, if not merged with SYSLIB:

  FORLIB.OBJ (FORTRAN IV)

  F77OTS.OBJ (FORTRAN–77)
- F77COM.MSG (FORTRAN–77)
- SYSLIB.OBJ

If the FORTRAN source file, GRAPH.FOR, resides on your default data (DK) device, the device need not be specified in the EXECUTE command line. If the file resides on a different device, that device must be specified. In any case, the handler for the device must be loaded in memory when the EXECUTE command is issued.

The EXECUTE command instructs the system to select the compiler, then process, link, and run the program. Because of SET commands you issued in Section 17.2, the EXECUTE command automatically calls the correct FORTRAN compiler. The EXECUTE command can be issued in the following ways:

- Use the /FORTRAN option and specify only the file name (omit the .FOR file type). The system assumes the .FOR file type because of the /FORTRAN option.
- Omit the /FORTRAN option and include the .FOR file type. The system assumes a FORTRAN operation because of the .FOR file type.
- Omit the /FORTRAN option and the .FOR file type and let the system choose a file type of .MAC, .DBL, or .FOR for the source file you name. If, for example, you specify the file GRAPH, the monitor searches DK (your default data volume) for the files GRAPH.MAC, GRAPH.DBL, and GRAPH.FOR, in that order. If the monitor finds neither GRAPH.MAC nor GRAPH.DBL, it invokes the FORTRAN compiler to compile GRAPH.FOR.

  Be careful about using this way to issue the EXECUTE command.

Examine the following command that combines the compile-link-run operations that you performed in this chapter. In the command, *librar* represents the correct FORTRAN object module library for your compiler (FORLIB or F77OTS). Do not issue this command until you have read Section 17.12:

If the FORTRAN object module library is not included in SYSLIB:

```
.EXECUTE/FORTRAN GRAPH/LIST/LINKLIB:librar RET
```

If the FORTRAN object module is included in SYSLIB:

```
.EXECUTE/FORTRAN GRAPH/LIST RET
```

17.12 Alternate Functions

You can substitute the following alternate functions in your FORTRAN source program to produce different graphs. Reedit the program (GRAPH.FOR) so that lines 1–5 in the function portion at the end contain one of the following alternate functions. Then, compile, link, and run the programs as described in the previous sections. If the necessary files are available on your system volume (see Section 17.11), use the EXECUTE command to run the program. The source program compiles, links, and runs, and the new graph prints on the terminal or printer.

Function 1

```
FUNCTION FUN(X,Y)
FUN=EXP(-SQRT(X**2+Y**2))
RETURN
END
```

Function 2

```
FUNCTION FUN(X,Y)
R=SQRT(X**2+Y**2)
FUN=X*Y*(R-3.)/(1.+EXP(3.*(R-3.5)))
RETURN
END
```

Function 3

```
FUNCTION FUN(X,Y)
FUN=EXP(+SQRT(X**2+Y**2))/1177.4
RETURN
END
```

17.13 Summary of Commands to Run FORTRAN Programs

EXECUTE

Combines the compile-link-run operations into one command.

EXECUTE/LINKLIB:library

Combines the compile-link-run operations into one command. Specifies the libraries to be used during linking.

EXECUTE/FORTRAN *file*

Combines the compile-link-run operations into one command and specifies the input file to be a FORTRAN file.

EXECUTE/LIST

Combines the compile-link-run operations into one command. Obtains a listing file of the source program and prints on line printer.

FORTRAN

Compiles the FORTRAN source program and produces an object module.

FORTRAN/LIST

Compiles the FORTRAN source program and produces both an object module and a listing file.

LINK

Links individual object modules to form a program and produces a load module.

RUN

Runs the indicated load module.

SET FORTRA F4

Makes FORTRA.SAV (for FORTRAN IV) the default compiler.

SET FORTRA F77

Makes F77XM.SAV or F77.SAV (for FORTRAN–77) the default compiler. The monitor determines which FORTRAN–77 compiler is appropriate and uses the correct compiler.

Chapter 18

Running a BASIC–PLUS Program

The BASIC–PLUS/RT–11 program language[1] is a machine-independent programming language that is one of the easiest languages for the beginning programmer to learn. It has both elementary language features that you use to write simple programs, and more advanced operations that let you produce complex and efficient programs. In addition, a special "immediate mode" lets you use BASIC–PLUS like a calculator to obtain instant answers to mathematical problems.

You do not need to understand the BASIC–PLUS language or the way the examples work to successfully perform the exercises in this chapter.

NOTE

In this chapter, the term BASIC–PLUS means BASIC–PLUS/RT–11.

18.1 Developing a BASIC–PLUS Program

BASIC (Beginner's All-purpose Symbolic Instruction Code)–PLUS is conversational in nature. It uses simple English keywords and common mathematical expressions to form easily understood language statements.

You write a BASIC–PLUS program as a series of one or more program lines. You begin each program line with a number that both identifies the line and indicates the order in which the line will be processed. Individual program lines contain one or more BASIC–PLUS language statements that define the operations to be performed.

When you are satisfied with the logic of your BASIC–PLUS source program, you create it as a file. However, unlike your methods under other programming languages, you can create the file under the control of the BASIC–PLUS interpreter. Thus, you can use commands that are part of the BASIC–PLUS language processor to create and edit the program, list it, run it, and save it for later use.

18.2 Using the BASIC–PLUS Interpreter

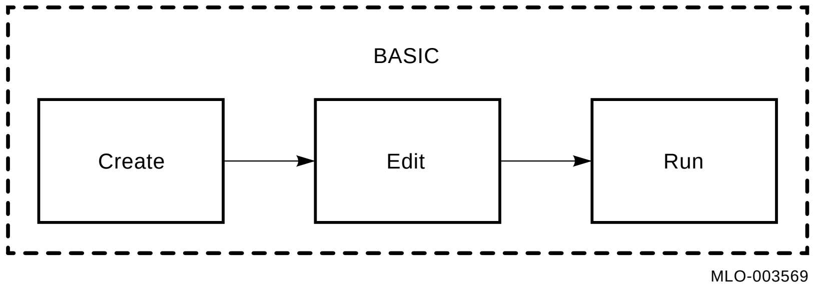
The BASIC–PLUS program, BASIC.SAV, is an interactive interpreter. The interpreter lets you create and execute a program in its entirety or a few lines at a time. The interpreter examines each program language statement, interprets it, and executes it before going on to the next.

If BASIC–PLUS discovers an error that prevents further processing, it calls the error file BASIC.ERR. BASIC–PLUS retrieves and prints on the terminal a message

[1] BASIC-PLUS is a superset of the standard BASIC language developed at Dartmouth College.

informing you of the error condition and stops. You correct the error so that execution can continue past that point and then rerun the program.

Figure 18–1: Functions of the BASIC–PLUS Interpreter



The functions of program creation, editing, processing, and execution are all handled by the BASIC–PLUS interpreter. Some RT–11 systems store the BASIC–PLUS interpreter on a volume apart from the system volume. You can quickly determine whether the BASIC–PLUS interpreter is on your system volume by typing the monitor DIRECTORY command and specifying the BASIC.SAV program. You should also verify that the error message file, BASIC.ERR, resides on your system volume:

```
.DIRECTORY SY:BASIC.* RET
```

In the directory listing that results, if the directory entries for BASIC.SAV and BASIC.ERR are listed on your terminal, then the required BASIC–PLUS files are on your system volume and you are ready to use the interpreter. However, if they did not appear in your listing, then you should copy any missing file to your system volume.

If you are running under a mapped monitor, issue the following commands to activate the BASIC–PLUS interpreter:

```
.V BASIC RET

BASIC-PLUS/RT-11 V3.x-xx

Ready
```

If you are not running under a mapped monitor, issue the following command to activate the BASIC–PLUS interpreter:

```
.R BASIC RET

BASIC-PLUS/RT-11 V3.x-xx

Ready
```

BASIC–PLUS prints the Ready message to indicate that it is ready to accept a BASIC–PLUS command. Any text that you type that is not preceded by a BASIC–PLUS command is accepted as program (or immediate mode) input. If at any time you wish to return to the monitor command mode, type the EXIT command following the Ready message. The Ready message appears after any BASIC–PLUS execution that is completed or interrupted by pressing CTRL/C twice, or after any BASIC–PLUS wait condition that is terminated by pressing CTRL/C once.

18.2.1 Immediate Mode

Immediate mode lets you use the BASIC–PLUS interpreter like a calculator to obtain immediate answers to arithmetic problems. You enter the appropriate BASIC–PLUS statement keyword and any necessary mathematical formula. When you press RETURN, BASIC–PLUS immediately calculates and prints the results. (Press DELETE or CTRL/U to correct any typing errors.)

```
PRINT (128+75)*3 RET
 609

Ready
```

BASIC–PLUS adds the two numbers in parentheses, multiplies them by 3, and prints the answer. The PRINT statement causes the answer to be printed on the terminal. The following command provides another example:

```
PRINT INT(34.67) RET
 34

Ready
```

The greatest integer less than or equal to 34.67 is printed.

You can combine several statements, including variable names, arithmetic equations, and data. BASIC–PLUS considers all the information, calculates the answer and prints it on the terminal, as illustrated in the following example:

```
A=5  RET

Ready

B=14  RET

Ready

C=.3729  RET

Ready

PRINT "THE HEIGHT IS";A*SIN(C)+B;"METERS"  RET
THE HEIGHT IS 15.8216 METERS

Ready
```

The first three statements equate variable names with values; the final statement introduces a formula for calculating a result and prints it.

You can use immediate mode to solve fairly lengthy and complicated mathematical problems. However, immediate mode information is temporary; you cannot save it. You can recall and change statements, if you have previously issued the RT–11 command SET SL ON, to enable the single-line command editor (SL). See Chapter 10 for information about SL.

If your needs are more complex, or if you want to save your statements, you should create a BASIC–PLUS program.

18.2.2 Creating and Editing a BASIC–PLUS Program

You use the BASIC–PLUS NEW command to create a program. The NEW command erases any contents in memory and asks you for a new program name:

```
NEW  RET
New File Name -
```

Supplying a file name is optional. Do not supply a file name if you are creating a temporary BASIC–PLUS program. Instead, press RETURN, creating a temporary program named NONAME. Specify a file name if you want to preserve the program. BASIC–PLUS automatically uses the file type .BAS.

To create a BASIC–PLUS program, assign line numbers to the language statements you type on the terminal keyboard. Your program lines are saved in memory and you can transfer program control to specific lines within the program, repeat parts of the program any number of times, store the entire program for later use, and perform other similar operations that are not possible in immediate mode.

Once you have created the program, you use BASIC–PLUS editing commands to list lines, change lines, add and erase lines, and correct typing errors. Create the following example program:

```
NEW  RET
New File Name -  RET

Ready
 3 LET T=0  RET
 5 FOR I=1 TO 10  RET
20 INPUT J  RET
25 LET T=T+J  RET
50 NEXT I  RET
55 PRINT "THE TOTAL IS";T  RET
88 END  RET
```

Those program lines are now in memory.

You could at this point preserve this program by issuing the BASIC–PLUS SAVE command. The SAVE command copies the program to the specified storage volume and gives the program the file name and file type that you indicate in the command line. The SAVE command does not alter the current contents of memory. You could, for example, preserve variations of the same program by saving the program at different stages of development to different file names. If you specify no device name or file specification, the SAVE command creates a file with the program name and of type .BAS on the default data (DK) volume.

You can list individual lines by specifying the line number. For example:

```
LIST 5  RET

NONAME     23-SEP-89      11:49 AM

5 FOR I=1 TO 10

Ready
```

Note that BASIC–PLUS prints a header line. Since you have not yet assigned a name to your program, BASIC–PLUS assigns it the name NONAME and prints this name, along with the date (which is only correct if previously entered by way of the DATE monitor command) and the time when you use the LIST command. You can omit the header line by using the LISTNH command instead of the LIST command.

You can change the contents of a program line by reentering the line. For example, to change the contents of line 25:

```
LISTNH 25 RET

25 LET T=T+J

Ready
25 LET T=T+I RET
LISTNH 25 RET

25 LET T=T+I

Ready
```

You can display a range of program lines by specifying the line range in the LIST command:

```
LISTNH 50-88 RET

50 NEXT I
55 PRINT "THE TOTAL IS";T
88 END

Ready
```

You can also list all the program lines by specifying no line range in the LIST command:

```
LISTNH RET

 3 LET T=0
 5 FOR I=1 TO 10
20 INPUT J
25 LET T=T+I
50 NEXT I
55 PRINT "THE TOTAL IS";T
88 END

Ready
```

You can delete a single program line by specifying the line number and pushing RETURN:

```
20 RET
LISTNH RET

 3 LET T=0
 5 FOR I=1 TO 10
25 LET T=T+I
50 NEXT I
55 PRINT "THE TOTAL IS";T
88 END

Ready
```

You can delete a range of program lines by specifying the line range in the DELETE[1]

[1] Do not confuse the BASIC–PLUS DELETE command with the DELETE key on the terminal keyboard.

command:

```
DELETE 25-50 RET
LISTNH RET

 3 LET T=0
 5 FOR I=1 TO 10
20 INPUT J
55 PRINT "THE TOTAL IS";T
88 END

Ready
```

Finally, you can delete the entire program, using the SCRATCH command:

```
SCRATCH RET
LISTNH RET

Ready
```

All program lines are erased from memory.

18.3 BASIC–PLUS Demonstration Program

RT–11 distributes a BASIC–PLUS demonstration program, DEMOB1.BAS, for this chapter. Ensure that DEMOB1.BAS resides on your default data (DK) volume:

```
.DIRECTORY DEMOB1.BAS RET
```

If the directory entry for DEMOB1.BAS is not displayed, copy the file from your software distribution kit to your DK volume. Once you have copied the file to your DK volume, issue the following command to copy the file DEMOB1.BAS to the demonstration program file MATCH.BAS, thereby preserving DEMOB1.BAS in its original state:

```
.COPY DEMOB1.BAS MATCH.BAS RET
```

MATCH.BAS contains the demonstration program 23 Matches.[2] The following is a reproduction of the program:

```
100 REM THE PROGRAM 23 MATCHES
101 REM
110 PRINT "WE BEGIN WITH 23 MATCHES. YOU MOVE FIRST. YOU"
115 PRINT "MAY TAKE 1, 2, OR 3 MATCHES. TYPE YOUR CHOICE"
120 PRINT "FOLLOWED BY A CARRIAGE RETURN. THEN THE COM-"
125 PRINT "PUTER CHOOSES 1, 2, OR 3 MATCHES. YOU CHOOSE"
130 PRINT "AGAIN, AND SO ON. WHOEVER MUST TAKE THE LAST"
135 PRINT "MATCH, LOSES."
140 PRINT \ LET M=23
200 REM THE HUMAN MOVES
201 REM
210 PRINT \ PRINT "THERE ARE NOW";M;"MATCHES."
215 PRINT \ PRINT "HOW MANY DO YOU TAKE";
230 INPUT H
240 IF H>M THEN 510
250 IF H<>INT(H THEN 510
260 IF H<=0 THEN 510
```

[2] 23 Matches, *101 BASIC Computer Games*, Maynard, Mass.: Digital Equipment Corporation, 1975.

```
270 IF H>=4 THEN 510
280 LET M=M-H
290 IF M=0 THEN 410
300 REM THE COMPUTER MOVES
301 REM
305 IF M=1 THEN 440
310 LET R=M-4*INT(M/4)
320 IF R<>1 THEN 350
330 LET C=INT(3*RND)+1 \ GO TO 360
350 LET C=(R+3)-4*INT((R+3)/4)
360 LET M=M-C
370 IF M=0 THEN 440
380 PRINT \ PRINT "THE COMPUTER TOOK";C;"....";
390 GO TO 310
400 REM SOMEBODY WON
401 REM
410 PRINT \ PRINT "THE COMPUTER WON." \ GO TO 999
440 PRINT \ PRINT "YOU WON." \ GO TO 999
500 REM BAD INPUT
501 REM
510 PRINT "ENTER ONLY 1, 2, OR 3." \ GO TO 215
999 END
```

As you can see from the first few lines of the listing, this program is a mathematical game where you match your logic against the program logic. The PRINT statements in the program print messages, game instructions, results, and so forth, on the terminal. The REM statements identify comment lines—remarks that provide general information about the program, but that are ignored by BASIC–PLUS during processing. The INPUT statement in line 230 lets you supply data to the terminal. Depending on the value you enter, program control transfers to various other parts of the program. For example, if you type an invalid value, program control skips ahead to a PRINT statement in line 510 informing you of your mistake and then returns to line 215 to ask for a value again. The mathematical algorithms of this program are in lines 310 through 350, which determine the number of matches the computer will select based on your choice.

18.4 Running a BASIC–PLUS Program

Once you have ensured that MATCH.BAS resides on your DK volume, you read MATCH.BAS into memory, using the BASIC–PLUS OLD command:

```
.R BASIC RET
OLD RET
Old file name-MATCH.BAS RET
Error - Invalid expression at line 250

Ready
```

As BASIC–PLUS reads MATCH.BAS into memory, it checks the syntax of MATCH.BAS and finds an error on program line 250. Although an error is found, the program is in memory. You can now display the program line that contains an error:

```
LISTNH 250 RET
```

```
?250 IF H<>INT(H THEN 510)
?
?Invalid expression at or preceding "THEN"
?
Ready
```

Note that the right parenthesis is missing after the second H in the line. Correct the line by retyping it:

```
250 IF H<>INT(H) THEN 510 RET
```

You are now ready to run the program. The BASIC–PLUS RUN command initiates program execution. This command prints a header that includes the program name, date, and time. If you want to omit the header line, type the RUNNH command instead:

```
RUNNH  RET
```

You see this text print on your terminal:

```
WE BEGIN WITH 23 MATCHES. YOU MOVE FIRST. YOU
MAY TAKE 1, 2, OR 3 MATCHES. TYPE YOUR CHOICE
FOLLOWED BY A CARRIAGE RETURN. THEN THE COM-
PUTER CHOOSES 1, 2, OR 3 MATCHES. YOU CHOOSE
AGAIN, AND SO ON. WHOEVER MUST TAKE THE LAST
MATCH, LOSES.

THERE ARE NOW 23 MATCHES.

HOW MANY DO YOU TAKE?
```

When the program pauses and asks you a question, you must supply data, in this case a 1, 2, or 3. Enter your choice (represented here by *n*):

```
n  RET

Ready
```

BASIC–PLUS begins processing and enters an infinite loop, a series of commands that it repeats endlessly. After several lines have printed, press CTRL/C CTRL/C; this interrupts execution and returns control to BASIC–PLUS command mode, as follows:

```
n RET

THE COMPUTER TOOK 1 ....
THE COMPUTER TOOK 1 ....
THE COMPUTER TOOK 3 ....
THE COMPUTER TOOK 2 ....
THE COMPUTER TOOK 2 ....
THE COMPUTER TOOK 3 ....
THE COMPUTER TOOK 1 ....
THE COMPUTER TOOK 1 ....
THE COMPUTER TOOK 3 ....
THE COMPUTER TOOK 1 ....
THE COMPUTER TOOK 3 ....
 CTRL/C  CTRL/C

Ready
```

An infinite loop is a programming logic error. However, since the error does not prevent processing, BASIC–PLUS does not print an error message. Instead BASIC–PLUS is caught in a loop of instructions and executes them endlessly. This particular loop is obvious because it prints a line of text; other kinds of loops may not be so evident. At this point, you must examine the program logic to determine why these instructions are being repeated.

Look at the program listing. The problem in this case is at line 390. That line instructs program control to return to line 310; therefore lines 310 through 390 are repeated endlessly without ever obtaining your next value choice. Program control should really return to line 210. Correct line 390 as follows by retyping it:

```
390 GO TO 210  RET
```

Now, you are ready to run the program again. This time the entire program should execute without error. Enter your value choices when requested. (A hint to playing the game: your first value choice determines whether you can win; if your first choice is wrong, the program has the advantage throughout.) A sample run follows.

```
RUNNH  RET

WE BEGIN WITH 23 MATCHES. YOU MOVE FIRST. YOU
MAY TAKE 1, 2, OR 3 MATCHES. TYPE YOUR CHOICE
FOLLOWED BY A CARRIAGE RETURN. THEN THE COM-
PUTER CHOOSES 1, 2, OR 3 MATCHES. YOU CHOOSE
AGAIN, AND SO ON. WHOEVER MUST TAKE THE LAST
MATCH, LOSES.

THERE ARE NOW 23 MATCHES.

HOW MANY DO YOU TAKE? 1 RET

THE COMPUTER TOOK 1 ....
THERE ARE NOW 21 MATCHES.

HOW MANY DO YOU TAKE? 1 RET

THE COMPUTER TOOK 3 ....
THERE ARE NOW 17 MATCHES.

HOW MANY DO YOU TAKE? 2 RET

THE COMPUTER TOOK 2 ....
THERE ARE NOW 13 MATCHES.

HOW MANY DO YOU TAKE? 1 RET

THE COMPUTER TOOK 3 ....
THERE ARE NOW 9 MATCHES.

HOW MANY DO YOU TAKE? 1 RET

THE COMPUTER TOOK 3 ....
THERE ARE NOW 5 MATCHES.

HOW MANY DO YOU TAKE? 3 RET

THE COMPUTER TOOK 1 ....
THERE ARE NOW 1 MATCHES.

HOW MANY DO YOU TAKE? 0 RET
ENTER ONLY 1, 2, OR 3.
```

```
HOW MANY DO YOU TAKE? 1 RET
THE COMPUTER WON.
Ready
```

You can repeat the program as often as you like, using the BASIC–PLUS commands RUN or RUNNH.

You do not now want to exit from BASIC–PLUS as you would not preserve the corrected MATCH program. If the original version of the MATCH program resided in memory, you would preserve it with the SAVE command. However, because MATCH.BAS already exists on a storage (the DK) volume, you preserve the corrected version with the REPLACE command:

```
REPLACE  RET
Ready
```

The current version of MATCH.BAS is written to DK and the MATCH program remains in memory.

18.5 Using the KED Editor with BASIC–PLUS Programs

BASIC–PLUS provides commands you can use to perform file operations outside the BASIC–PLUS interpreter. You can create and edit BASIC–PLUS programs by using the KED editor.

The OLD command reads an existing BASIC–PLUS program from an ASCII text file into computer memory. As this command reads in the program, BASIC–PLUS converts the program from ASCII text format to an intermediate format used by BASIC–PLUS.

The SAVE and REPLACE commands copy a BASIC–PLUS program from computer memory to a storage volume. As these commands copy the program, they convert it from the intermediate format used by BASIC–PLUS to ASCII text format.

Thus, you can, if you prefer, use the KED editor to create and edit BASIC–PLUS programs, since the editor also uses ASCII text format. However, many users would rather use BASIC–PLUS to create and edit a BASIC–PLUS program, since they can then run the program, reedit it, rerun it, and save the new version—all in BASIC–PLUS command mode—rather than perform the several corresponding monitor commands.

18.6 Preserving the Intermediate Translated Program

When you create, read into memory, or edit a BASIC–PLUS program, the interpreter creates an intermediate translation of each command line. This intermediate translation process lets you run a BASIC–PLUS program as soon as you create it or read it into memory. Once you have finalized a BASIC–PLUS program, you can preserve the intermediate translation in a file by issuing the BASIC–PLUS COMPILE command. Preserving the translated program version eliminates the need for BASIC–PLUS to translate the ASCII source program each time you run it.

A translated BASIC–PLUS program has the file type .BAC, and BASIC–PLUS looks first for the .BAC file type when you issue the command to run (RUN or RUNNH) a program name. BASIC–PLUS looks for the ASCII source (.BAS) version of a program only when it cannot find the intermediate translated (.BAC) version.

Although translation is most effective with larger BASIC–PLUS programs, for practice, preserve the intermediate translation of the program MATCH, by creating MATCH.BAC on your DK device:

```
COMPILE MATCH  RET
```

```
Ready
```

You cannot use the OLD command to read a translated (.BAC) program into memory; you must run the program, using the RUN or RUNNH command. Also, the .BAC version of a program is not a source file and cannot be edited. You should, therefore, always preserve the .BAS version of each BASIC–PLUS program in case you want to edit the program in the future.

18.7 Stopping the BASIC–PLUS Interpreter

You stop the BASIC–PLUS interpreter and return to the RT–11 monitor by issuing the EXIT command:

```
EXIT  RET
```

```
.
```

18.8 Summary of Commands

The following BASIC–PLUS commands are described and illustrated in this chapter. If the meaning of any command is unclear, review the example in this chapter that illustrates the command.

line

Erases the indicated program lines.

COMPILE

Copies the translated version of the BASIC–PLUS program currently in memory to the specified volume (default volume is DK).

CTRL/C

Under control of BASIC–PLUS only, interrupts execution of the BASIC–PLUS program and returns control to BASIC–PLUS command mode.

DELETE *line #*

Erases the indicated program lines.

EXIT

Returns control to monitor command mode (only when using BASIC–PLUS).

LIST

Lists the entire program and prints a header that includes the program name, date, and time.

LIST *line #*

Lists the indicated lines and prints a header that includes the program name, date, and time.

LISTNH

Lists the entire program but does not print a header.

LISTNH line #

Lists the indicated lines but does not print a header.

NEW

Creates a new BASIC–PLUS program and assigns the indicated file name.

OLD

Copies into memory an existing BASIC–PLUS program (for use under BASIC–PLUS).

REPLACE

Copies the BASIC–PLUS program currently in memory to the indicated storage volume and replaces the version that already exists on that volume.

RUN

Executes the BASIC–PLUS program currently in memory; prints a header line that includes the program name, date, and version number.

RUNNH

Executes the BASIC–PLUS program currently in memory; omits the header line.

SAVE

Copies the BASIC–PLUS program currently in memory to the indicated storage volume.

SCRATCH

Erases all program lines from memory and changes the program name to (the default) NONAME.

Chapter 19

Running a MACRO–11 Assembly-Language Program

The MACRO–11 programming language is a machine-dependent programming language developed for the PDP–11 programmer, or for the FORTRAN programmer who intends to combine assembly-language routines and FORTRAN routines. The MACRO–11 language enables the knowledgeable programmer to access all the features of the RT–11 computer system by using a precise and efficient programming code.

You do not need to understand the MACRO–11 language to successfully complete the exercises in this chapter.

19.1 A Sample MACRO–11 Program

RT–11 distributes a demonstration MACRO–11 source program, DEMOM1.MAC, to be used with this chapter.

To preserve the original file, copy DEMOM1.MAC to file SUM.MAC and unprotect SUM.MAC. The following commands assume the file DEMOM1.MAC resides on your default data (DK) volume. If the first command fails with a message indicating the file is not found, copy DEMOM1.MAC from the volume on which it resides to DK and retry the command.

```
.COPY DEMOM1.MAC SUM.MAC  RET
.UNPROTECT SUM.MAC  RET
```

To see what the program is, type the following command:

```
.TYPE SUM.MAC  RET
```

The following is the MACRO–11 code in the file SUM.MAC. Note the use of the semicolon (;) in the code to indicate the programmer's comments.

The first lines of comment indicate the purpose of the program, which is to figure the sum of the reciprocals of the factorial of 70 (1/0 + 1/1 .... + 1/70 —A factorial is the product of all positive integers from 1 to any given number. A reciprocal of a number is that number divided into 1):

```
        .TITLE SUM.MAC

        .MCALL .TTYOUT, .EXIT, .PRINT

        N = 70.         ;NO. OF DIGITS OF 'E' TO CALCULATE
;       'E' = THE SUM OF THE RECIPROCALS OF THE FACTORIALS
;       1/0! + 1/1! + 1/2! + 1/3! + 1/4! + 1/5! + ...

        .DSABL GBL
```

```
EXP:    .PRINT  #MESSAG         ;PRINT INTRODUCTORY TEXT
        MOV     #N,R5           ;NO. OF CHARS OF 'E' TO PRINT
FIRST:  MOV     #N+1,R0         ;NO. OF DIGITS OF ACCURACY
        MOV     #A,R1           ;ADDRESS OF DIGIT VECTOR
SECOND: ASL     @R1             ;DO MULTIPLY BY 10 (DECIMAL)
        MOV     @R1,-(SP)       ;SAVE *2
        ASL     @R1             ;*4
        ASL     @R1             ;*8
        ADD     (SP)+,(R1)+     ;NOW *10, POINT TO NEXT DIGIT
        DEC     R0              ;AT END OF DIGITS?
        BNE     SECOND          ;BRANCH IF NOT
        MOV     #N,R0           ;GO THRU ALL PLACES, DIVIDING
THIRD:  MOV     -(R1),R3        ;BY THE PLACES INDEX
        MOV     #-1,R2          ;INIT QUOTIENT REGISTER
FOURTH: INC     R2              ;BUMP QUOTIENT
        SUB     R0,R3           ;SUBTRACT LOOP ISN'T BAD
        BCC     FOURTH          ;NUMERATOR IS ALWAYS < 10*N
        ADD     R0,R3           ;FIX REMAINDER
        MOV     R3,@R1          ;SAVE REMAINDER AS BASIS
                                ;FOR NEXT DIGIT
        ADD     R2-2(R1)        ;GREATEST INTEGER CARRIES
                                ;TO GIVE DIGIT
        DEC     R0              ;AT END OF DIGIT VECTOR?
        BNE     THIRD           ;BRANCH IF NOT
        MOV     -(R1),R0        ;GET DIGIT TO OUTPUT
FIFTH:  SUB     #10.,R0         ;FIX THE 2.7 TO .7 SO
                                ;THAT IT IS ONLY 1 DIGIT
        BCC     FIFTH           ;(REALLY DIVIDE BY 10)
        ADD     #10+'0,R0       ;MAKE DIGIT ASCII
        .TTYON                  ;OUTPUT THE DIGIT
        CLR     @R1             ;CLEAR NEXT DIGIT LOCATION
        DEC     R5              ;MORE DIGITS TO PRINT?
        BNE     FIRST           ;BRANCH IF YES
        .EXIT                   ;WE ARE DONE

   EXP: .REPT    N+1
        .WORD   1               ;INIT VECTOR TO ALL ONES
        .ENDR

MESSAG: .ASCII /THE VALUE OF E IS:/ <15><12> /2./ <200>
        .EVEN

        .END    EXP
```

To help you understand what the MACRO–11 assembler does with source code, look at this sample MACRO–11 program. Note the 4-column structure of most of the program.

```
LABEL    OPERATOR    OPERAND(S)    COMMENT

EXP:     .PRINT      #MESSAG       ;PRINT INTRODUCTORY TEXT
```

The following summary describes each column in this program:

- **LABEL**

  The first column contains only labels, which, however, are optional. Note that each label in this column ends with a colon (:). You can also place labels (without the colon) in column three. The optional label identifies a line of code so that you can refer to the instructions or data on that line from other parts of the program.

- **OPERATOR**

  The second column contains operators, which are short mnemonic commands such as MOV (for MOVE), ADD, DEC (for DECREMENT), and so on.

- **OPERAND(S)**

  The third column contains operands, the arguments to (or objects of) the commands.

  The operator and/or operand are instructions selected from the PDP–11 instruction set, data needed by the instructions, or assembler directives (instructions to the assembler to guide the assembly process).

  The MACRO–11 assembly language uses the PDP–11 instruction set, a list of mnemonic instructions that corresponds to PDP–11 computer operations. These instructions let you add, compare, increment, complement, and perform many other manipulations on numerical data. By choosing the appropriate instructions and by providing any additional data needed, you can create a program.

  The instructions are summarized in a pocket-sized folding card, called the PDP–11 Programming Card (Figure 19–1), and are described in detail in the *PDP–11 Processor Handbook*. The *PDP–11 MACRO–11 Language Reference Manual*, which is for both new and advanced MACRO–11 programmers, presumes knowledge of the PDP–11 instruction set.

- **COMMENT**

  The final column is reserved for optional comments, beginning with a semicolon, which describe the function of the code.

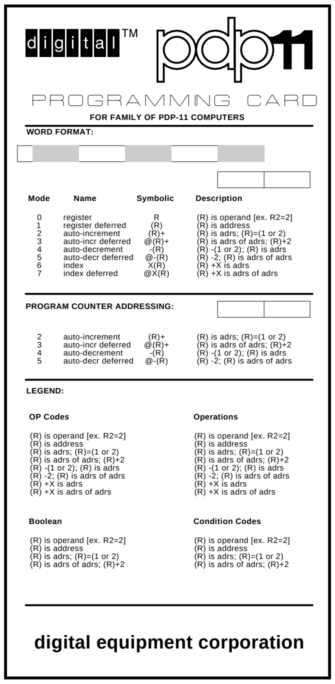
Sequences of language statements constitute a program; that is, perform a specific function. In this case, the program sequence begins with the EXP: label and ends with .END EXP statement. The program function is to sum the reciprocal of a factorial.

19.2 Creating a MACRO–11 Program

You can use the KED editor to create MACRO–11 source programs as files. The KED editor is described in Chapter 4. You use tabs and spaces to make the program more readable so that the lines are formatted in the columns described in the previous section.

When you have finished creating the program as a complete, edited file, you next enter it as input to the MACRO–11 language processor, which is part of the RT–11 operating system and is stored on your system volume.

Figure 19–1: PDP–11 Programming Card



digital™ pdp11

PROGRAMMING CARD

FOR FAMILY OF PDP-11 COMPUTERS

WORD FORMAT:

Mode	**Name**	**Symbolic**	**Description**
0	register	R	(R) is operand [ex. R2=2]
1	register deferred	(R)	(R) is address
2	auto-increment	(R)+	(R) is adrs; (R)=(1 or 2)
3	auto-incr deferred	@(R)+	(R) is adrs of adrs; (R)+2
4	auto-decrement	-(R)	(R) -(1 or 2); (R) is adrs
5	auto-decr deferred	@-(R)	(R) -2; (R) is adrs of adrs
6	index	X(R)	(R) +X is adrs
7	index deferred	@X(R)	(R) +X is adrs of adrs

PROGRAM COUNTER ADDRESSING:

2	auto-increment	(R)+	(R) is adrs; (R)=(1 or 2)
3	auto-incr deferred	@(R)+	(R) is adrs of adrs; (R)+2
4	auto-decrement	-(R)	(R) -(1 or 2); (R) is adrs
5	auto-decr deferred	@-(R)	(R) -2; (R) is adrs of adrs

LEGEND:

OP Codes

(R) is operand [ex. R2=2]
(R) is address
(R) is adrs; (R)=(1 or 2)
(R) is adrs of adrs; (R)+2
(R) -(1 or 2); (R) is adrs
(R) -2; (R) is adrs of adrs
(R) +X is adrs
(R) +X is adrs of adrs

Operations

(R) is operand [ex. R2=2]
(R) is address
(R) is adrs; (R)=(1 or 2)
(R) is adrs of adrs; (R)+2
(R) -(1 or 2); (R) is adrs
(R) -2; (R) is adrs of adrs
(R) +X is adrs
(R) +X is adrs of adrs

Boolean

(R) is operand [ex. R2=2]
(R) is address
(R) is adrs; (R)=(1 or 2)
(R) is adrs of adrs; (R)+2

Condition Codes

(R) is operand [ex. R2=2]
(R) is address
(R) is adrs; (R)=(1 or 2)
(R) is adrs of adrs; (R)+2

digital equipment corporation

MLO-003570

19.3 Using the MACRO–11 Language Processor

The MACRO–11 language processor, called an assembler, assembles (translates) the MACRO–11 programs you code, called *source programs*, into machine-language programs called *object modules* having file types of OBJ.

When you create a program with KED, the editor stores the program in a file coded in ASCII format; that is, the file is in a form you can read. But when you assemble that file with the MACRO–11 assembler, the assembler translates that program into a binary-code format, called machine language, which the computer uses when you run your program. Binary-code format is required by the computer but it is difficult to read.

The assembler interprets and processes the assembly language lines of code, called statements, one line at a time. As it translates each line, a program counter keeps track of the addresses in computer memory where each instruction and data are stored.

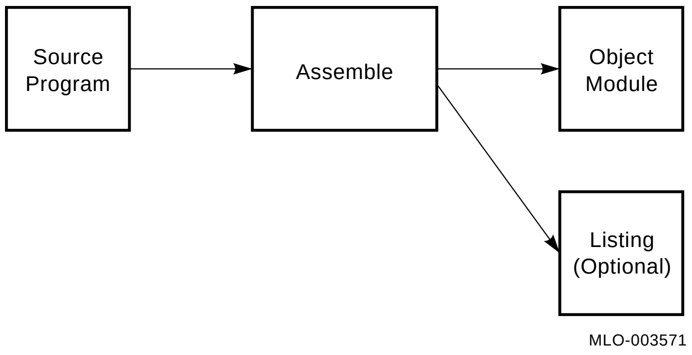
PDP–11 computer memories consist of physical storage locations that can hold numerical data. These locations are numbered consecutively from 0 up to the highest memory location, which varies according to the amount of memory acquired with the computer system.

The following are the major tasks of the MACRO–11 assembler:

- Converts assembly language mnemonics, user-defined symbols, and data values (from a file formatted in ASCII code) into their respective machine-language equivalents (to a file formatted in machine-language code). That is, if the assembler finds no errors, then it translates your source file into an object-module file with a file type of OBJ.
- Accounts for all instructions used within your source program and determines their relative positions in computer memory.
- Keeps track of all user-defined symbols and their respective values in a symbol table.
- Lists any source code errors it finds.
- Displays an optional numbered listing of your source code with octal-code translations, a list of the symbols you use in your program, an optional set of tables listing cross references within your program, and some statistical information.

The role of the assembler is summarized in Figure 19–2.

Figure 19–2: Assembler Operation



19.3.1 Assembler Listings

When you assemble a MACRO–11 program, you can request the assembler to produce a listing of the source program at the same time it translates the program. The four advantages for doing this are the following assembly-listing features that can help you debug a program.

- **Line Numbering**

  The assembler automatically numbers each line of code in your program listing so that it is easy to find lines.

- **Octal Code Listing**

  The assembler translates the binary code it produces for your program into octal code. The assembler does this to help you understand how the computer deals with each instruction and data value in your program.

- **Symbol Listing**

  The assembler places after its listing of your program a table of all the symbols you use in the program. This is a helpful reference summary of your program's symbols.

  You use symbolic names in a MACRO–11 program to refer to program variables, lines of code, and some constants. For example, the label FIRST is a symbol for the line of code following that label below:

  ```
  FIRST:  MOV     #N+1,R0         ;NO. OF DIGITS OF ACCURACY
  ```

- **Error Listing**

  The assembler lists any programming errors it finds and indicates on which lines those errors occur.

19.3.2 Assembler Cross References

Along with your program listing, it is also good to request the assembler to give you a cross-referenced listing. This type of listing can be a helpful debugging tool since it cross references all the symbols in a program. For example, the following code is on line 43 of the previous example MACRO–11 program.

```
BNE     FIRST           ;BRANCH IF YES
```

A cross-reference listing of that code would include the following line:

```
FIRST    1-13#     1-43
```

The number sign (#) after a line number in a cross-reference listing indicates where a symbol is defined. So, the preceding example means that the symbol FIRST is defined in line 13 and is referred to in line 43.

19.4 Assembling the Sample MACRO–11 Program

Now that you are familiar with what the assembler does, the next step is to use the MACRO–11 assembler, MACRO.SAV, to convert the sample source program SUM.MAC to object code. To do this, use the MACRO command. Use that command with its /LIST and /CROSSREFERENCE options to assemble your source program and produce a cross-referenced assembly listing.

You can omit typing the MAC file type for your input file, since the MACRO command assumes this file type unless you indicate otherwise. RT–11 will automatically assign the name SUM.OBJ to the object module and SUM.LST to the listing file and will store both newly created files on the default data (DK) volume.

Enter the command as follows:

```
.MACRO SUM/LIST/CROSSREFERENCE RET
```

The assembler takes a few moments to translate the file. However, this demonstration file purposely has errors in it. So, when the assembler has translated the file, it displays a message similar to the following on your terminal:

```
?MACRO-W-Errors detected:  6
DK:SUM,DK:SUM/C=DK:SUM
```

This message indicates the assembler detected errors in six lines of the source program during processing. Examine the assembly listing to see what the errors are and to correct the program. To do so, print the assembly listing:

```
.PRINT SUM.LST RET
```

It is easier to see the listing as a whole if you print it. But, if you do not have a printer available, you can display the assembly listing on your terminal. Before displaying the listing, set your terminal screen to 132-column mode to display the entire listing:

```
.SETUP 132COLUMNS  RET
.TYPE SUM.LST  RET
```

When you are done displaying the listing, you can return the terminal to 80-column mode by issuing the command:

```
.SETUP 80COLUMNS  RET
```

Your listing should look like the following. Refer to the listing as you read the rest of this section and the next four sections.

```
SUM.MAC (VERSION PROVIDED)     MACRO V05.04  Wednesday 18-Sep-91  03:59  Page 1 ❶

      1                                         .TITLE SUM.MAC  (VERSION PROVIDED)
      2
      3                                         .MCALL .TTYOUT, .EXIT, .PRINT
      4
      5        000106                           N = 70.         ;NO. OF DIGITS OF 'E' TO CALCULATE
      6                                 ;       'E' = THE SUM OF THE RECIPROCALS OF THE FACTORIALS
      7                                 ;       1/0! + 1/1! + 1/2! + 1/3! + 1/4! + 1/5! + ...
      8
      9                                         .DSABL GBL
     10
M    11 000000                          EXP:    .PRINT  #MESSAG         ;PRINT INTRODUCTORY TEXT
     12 000006  012705  000106                  MOV     #N,R5           ;NO. OF CHARS OF 'E' TO PRINT
     13 000012  012700  000107          FIRST:  MOV     #N+1,R0         ;NO. OF DIGITS OF ACCURACY
U    14 000016  012701  000000                  MOV     #A,R1           ;ADDRESS OF DIGIT VECTOR
     15 000022  006311                 SECOND:  ASL     @R1             ;DO MULTIPLY BY 10 (DECIMAL)
     16 000024  011146                          MOV     @R1,-(SP)       ;SAVE *2
     17 000026  006311                          ASL     @R1             ;*4
     18 000030  006311                          ASL     @R1             ;*8
     19 000032  062621                          ADD     (SP)+,(R1)+     ;NOW *10, POINT TO NEXT DIGIT
     20 000034  005300                          DEC     R0              ;AT END OF DIGITS?
     21 000036  001371                          BNE     SECOND          ;BRANCH IF NOT
     22 000040  012700  000106                  MOV     #N,R0           ;GO THRU ALL PLACES, DIVIDING
     23 000044  014103                   THIRD: MOV     -(R1),R3        ;BY THE PLACES INDEX
     24 000046  012702  177777                  MOV     #-1,R2          ;INIT QUOTIENT REGISTER
     25 000052  005202                  FOURTH: INC     R2              ;BUMP QUOTIENT
     26 000054  160003                          SUB     R0,R3           ;SUBTRACT LOOP ISN'T BAD
     27 000056  103375                          BCC     FOURTH          ;NUMERATOR IS ALWAYS < 10*N
     28 000060  060003                          ADD     R0,R3           ;FIX REMAINDER
     29 000062  010311                          MOV     R3,@R1          ;SAVE REMAINDER AS BASIS
     30                                                                 ;FOR NEXT DIGIT
AR   31 000064  066167  000000  000000'         ADD     R2-2(R1)        ;GREATEST INTEGER CARRIES
     32                                                                 ;TO GIVE DIGIT
     33 000072  005300                          DEC     R0              ;AT END OF DIGIT VECTOR?
     34 000074  001363                          BNE     THIRD           ;BRANCH IF NOT
     35 000076  014100                          MOV     -(R1),R0        ;GET DIGIT TO OUTPUT
     36 000100  162700  000012           FIFTH: SUB     #10.,R0         ;FIX THE 2.7 TO .7 SO
     37                                        ;THAT IT IS ONLY 1 DIGIT
     38 000104  103375                 BCC     FIFTH           ;(REALLY DIVIDE BY 10)
     39 000106  062700  000070         ADD     #10+'0,R0       ;MAKE DIGIT ASCII
U    40 000112  000000                 .TTYON                  ;OUTPUT THE DIGIT
     41 000114  005011                 CLR     @R1             ;CLEAR NEXT DIGIT LOCATION
     42 000116  005305                 DEC     R5              ;MORE DIGITS TO PRINT?
     43 000120  001334                 BNE     FIRST           ;BRANCH IF YES
     44 000122                         .EXIT                   ;WE ARE DONE
     45
M    46 000124  000107            EXP: .REPT   N+1
     47                                .WORD   1               ;INIT VECTOR TO ALL ONES
     48                                .ENDR
     49
     50 000342    124     110     105  MESSAG: .ASCII /THE VALUE OF E IS:/ <15><12> /2./ <200>
 000345    040     126     101
 000350    114     125     105
 000353    040     117     106
 000356    040     105     040
 000361    111     123     072
 000364    015     012     062
 000367    056     200

SUM.MAC (VERSION PROVIDED)     MACRO V05.04  Wednesday 18-Sep-91 03:59  Page 1-1


     51                                .EVEN
     52
D    53         000000'                .END    EXP

SUM.MAC (VERSION PROVIDED)      MACRO V05.04  Wednesday 18-Sep-91 03:59  Page 1-2
Symbol table ❷

A     = ******        FIRST   000012R      MESSAG 000342R      SECOND  000022R    .TTYON= ******
EXP     000000R       FOURTH  000052R      N     = 000106      THIRD   000044R    ...V1 = 000003
FIFTH   000100R

. ABS.  000000    000   (RW,I,GBL,ABS,OVR)
        000372    001   (RW,I,LCL,REL,CON)
Errors detected:  6

*** Assembler statistics ❸


Work  file  reads: 186
Work  file writes: 67
Size of work file: 9363 Words  ( 37 Pages)
Size of core pool: 3584 Words  ( 14 Pages)
Operating  system: RT-11

Elapsed time: 00:00:13.51
DK:SUM,DK:SUM/C=DK:SUM

SUM.MAC (VERSION PROVIDED)      MACRO V05.04  Wednesday 18-Sep-91 03:59 Page S-1
Cross reference table (CREF V05.04) ❹


...V1    1-11
.TTYON   1-40
A        1-14
EXP      1-11#     1-46#     1-53
FIFTH    1-36#     1-38
FIRST    1-13#     1-43
FOURTH   1-25#     1-27
MESSAG   1-11      1-50#
N        1-5#      1-12      1-13      1-22      1-46
SECOND   1-15#     1-21
THIRD    1-23#     1-34

SUM.MAC (VERSION PROVIDED)      MACRO V05.04  Wednesday 18-Sep-91 03:59 Page M-1
Cross reference table (CREF V05.04)


...CM5   1-11
.EXIT    1-3#      1-44
.PRINT   1-3#      1-11
.TTYOU   1-3#

SUM.MAC (VERSION PROVIDED) MACRO V05.04  Wednesday 18-Sep-91 03:59 Page E-1
Cross reference table (CREF V05.04)
```

```
A       1-31
D       1-53
M       1-11      1-46
R       1-31
U       1-14      1-40
```

Note that the preceding listing is divided into four main parts, indicated by the numbers in black circles. These parts are:

❶ Listing of the original code along with line numbers, octal code, and any error messages

❷ Symbol table

❸ Program statistics

❹ Cross-reference tables

These four parts of the listing are described in the next five sections.

19.4.1 Examining the Octal Code in the Line-Numbered Listing

Examine the line-numbered listing, the first part of the assembler's output. The assembler assigns consecutive decimal line numbers to each line of the source program, including blank lines and comment lines. These numbers help you find lines of code. Note that the program has 53 lines of code.

Octal Numbers

Next to the line numbers are the octal numbers. A MACRO–11 assembly listing shows the addresses of memory locations and their contents as octal numbers. The octal numbers represent the respective binary machine-language numbers (code) that make up the object module the assembler produces.

For example, line 13 in the assembly listing has the octal numbers 00012, 012700, and 000107. The first number shows the memory location for the instruction and data shown in the second two numbers:

```
13 000012 012700 000107   FIRST:  MOV   #N+1,R0    ;NO. OF DIGITS OF ACCURACY
```

Even-Numbered Storage Locations

The octal numbers list consecutive even-numbered storage locations, called *addresses*, within the computer's memory where each program instruction and data value are stored. In the preceding example, the octal number 00012 (the decimal number 10) is an even-numbered storage location.

A single instruction requires one or more words. So there are at least two octal numbers next to a line number; one for the address and one for the contents. Notice line 13 needs three storage locations; one for the address and two for the contents. This program has been assigned *relative* memory locations 0 through 370.

Relative Locations

These locations in memory are relative; that is, not fixed, because MACRO–11 programers often use pre written MACRO–11 subroutines supplied in the RT–11 subroutine library, SYSLIB.OBJ. These subroutines are added to a program after it is successfully assembled. With the addition of library subroutines, the program size and the location of code within memory change. So, at that time, the program locations are made fixed. Section 19.5 and Chapter 20 describe the linking procedure that joins RT–11 MACRO–11 subroutines with a program.

Binary Code

The octal code in the assembler listing is the assembler's translation of the binary code it has produced for your MACRO–11 program. The computer uses binary code when it runs your program.

Binary code is a base-2 numbering system consisting of only the two digits 0 and 1. Since these two digits can easily represent the 2-state (On and Off) electronic logic in a digital computer, machine language consists of binary code. However, for the following reasons, the assembler displays on your program listing an octal-code translation of your program's binary code:

- Octal code is easier for someone to read than the machine-readable binary code.
- Information in PDP–11 computers is stored in units of 8-bit bytes.
- Seeing how the assembler translates each instruction and data value of your program can help you make any needed improvements in that program. Of course, this presumes you are an experienced MACRO–11 programmer.

The following is what an assembled instruction in PDP–11 computer memory looks like in binary code:

```
LOCATION    LOCATION
ADDRESS     CONTENTS
01000       11000000
01001       11100101
```

In the preceding example, the binary number 01000 (in the address) equals the even decimal number 8, and the binary number 01001 equals the odd decimal number 9.

Since a single instruction requires two (or more) consecutive memory locations, this instruction is actually put together in memory in the following manner.

```
        high-order byte   low-order byte

01001   1 1 1 0 0 1 0 1   1 1 0 0 0 0 0 0   01000
```

The byte in the even-numbered memory address is called the low-order byte and is stored first. The byte in the odd-numbered memory address is called the high-order byte and is stored next. Both bytes together form the following PDP–11, 16-bit word:

```
ONE WORD

1110010111000000
```

In summary, binary code is divided into the following three units:

Bit	Each individual digit is called a bit (binary digit).
Byte	A single memory location has 8 bits and is called a byte.
Word	Two consecutive memory locations have 16 bits and is called a word.

The assembler (and computer) works in terms of these 8-bit bytes and 16-bit words of binary data.

Translating Binary Code

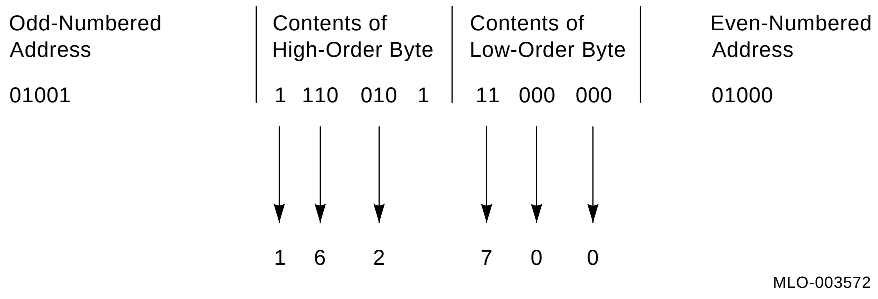
All binary numbers can be converted into decimal and octal numbers, which are easier to understand. Table 19–1 shows the decimal numbers 0 through 10 and their respective octal and binary equivalents. Tables and formulas are available to calculate higher conversions.

Table 19–1: Decimal, Octal, and Binary Equivalents

Decimal	Octal	Binary
0	0	000
1	1	001
2	2	010
3	3	011
4	4	100
5	5	101
6	6	110
7	7	111
8	10	1 000
9	11	1 001
10	12	1 010

When the assembler translates a PDP–11 word—16 bits into an octal number, it first translates the contents of the low-order byte and then the contents of the high-order byte in groups of three bits. Figure 19–3 illustrates how the assembler translates the preceding binary instruction (1110010111000000—one PDP–11 word) into one octal number (162700).

Figure 19–3: Translating Binary Code to Octal Code



19.4.2 Examining the Symbol Table in the Listing

As the assembler processes your source program, it constructs a symbol table, which is a compiled, alphabetical list of all the symbolic names and labels you have used in your program. Next to each symbol is a definition. The MACRO–11 assembler defines each symbolic name by assigning an address or data value, as appropriate, and adds the symbol definition to the symbol table.

For example, the symbolic variable name N is defined (in line 7) as 000106_8 or 70_{10} an absolute value. Labels are defined as addresses. The symbolic label FIRST is defined (in line 13) as 000012, a relocatable address (the R following 000012 in the symbol table indicates that the address will be relocated or modified during linking). A row of asterisks next to any symbolic name in the table indicates that for some reason—possibly a programming error—the assembler could not define the symbol.

19.4.3 Examining the Program Statistics

At the very end of the symbol table (where the . ABS. occurs) is the program's size information (or synopsis) in terms of the total number of octal storage locations it requires (in this case, 372). Following is the number of errors detected, and the amount of free and used memory pages (statistics provided by the assembler).

19.4.4 Examining the Cross-Reference Tables in the Listing

Following the program statistics is the cross-reference (CREF) listing. The CREF listing is optional—as is the assembly listing—but it provides you with useful reference and debugging information, especially if the program is large. The CREF listing can contain several kinds of tables of reference information, each beginning on a new page. The default tables are the three shown here.

Every reference in a CREF table shows the page number of the listing (in the preceding example, all references are on page 1), followed by the appropriate line number. A number sign (#) following a line number indicates that this line is where a label or symbol definition occurs.

The three default cross-reference tables:

- **Table of User-Defined Symbols**

The first CREF table shown here alphabetically lists all user-defined symbols and label references, that is, symbols and labels you define in your MACRO–11 program.

- **Table of Macro-Defined Symbols (Macros)**

  The second CREF table alphabetically lists all macro symbol references.

  Macros, from which the MACRO–11 language processor derives its name, are a useful feature of the MACRO–11 assembly language. You can define as a macro any recurring sequence of coding instructions. By giving the macro a name, you can thereafter call it from any other part of the program by using a single language statement.

  In addition to the macros you define yourself, RT–11 provides system macros that your program can access when it is assembled. To understand how a MACRO–11 program uses these system macros, you should know the following:

  - They can be stored in the system macro library (SYSMAC.SML) or the system definition macro library (SYSTEM.MLB)
  - They are calls to RT–11 monitor services

  A brief explanation of these two aspects of the system macros follows:

  — **Macro Library Requests**

  These system macros are defined in a special system library file called SYSMAC.SML (SML stands for System Macro Library). SYSMAC.SML is part of the RT–11 operating system and is stored on the system volume. If you request one of these system macros from your source program, the MACRO–11 assembler automatically searches SYSMAC.SML for the required information.

  The system macros defined in SYSMAC.SML are calls to services performed by the RT–11 monitor, such as terminal handling, input and output operations, program termination, file capabilities, and so on. The portion of the monitor that performs these services or that is capable of getting the necessary program code to perform these services is always in memory and is, therefore, called the resident monitor. Thus, whenever your source program is in memory and is to be executed, the resident monitor is also there with its services.

  — **Programmed Requests**

  You communicate the need for a monitor service by issuing a programmed request in your source program. A programmed request consists of a macro call to a specific macro defined in SYSMAC.SML. The macro expands into the appropriate machine-language code, which, during program execution, makes a request to the resident monitor to supply the desired service.

  You specify all programmed requests that you intend to use in your source program in an .MCALL statement, like the one shown at line 3 in the listing:

```
3         .MCALL .TTYOUT, .EXIT, .PRINT
```

For example, the programmed request .TTYOUT requests the monitor to print an ASCII character on the console terminal. During assembly, the .TTYOUT macro in SYSMAC.SML is expanded into machine-language code. During program execution, this code requests the resident monitor to take the indicated ASCII character and send it to the console terminal.

Line 11 in the program SUM uses another programmed request, .PRINT, to print a message on the terminal. Note in the macro cross-reference listing, that .PRINT first appears on line 3 (where it is defined) and next appears on line 11.

- **Table of Error-Message Symbols**

  The third CREF table alphabetically lists the codes of the errors detected during assembly. You must correct these errors before you can run the program. Section 19.4.5 explains the errors in your listing and how to correct them.

19.4.5 Finding and Correcting the Assembly-Listing Errors

To help you understand the errors, first look again through the listing of your program, note in general what the program does, and then look at the error listing.

19.4.5.1 What the Program Does

The first two comment lines (preceded by semicolons) indicate that the program calculates the value of ‘E’, which is the sum of the inverse of the factorials between 1 and infinity (reduced to 70 in this program).

The algorithm used in this program is somewhat complicated (this was necessary to keep the program reasonably short). ‘E’ is calculated one digit at a time by using a difference function between its actual value and the current approximation for each new digit. The algorithm is:

```
1 + (1 + (1 + ... + (1 + ((1 + (1/N))/(N-1))/(N-2))/.../2)/1)
```

The statements between lines 1 through 7 define initial states to the assemblers, such as the value of N, and designate the macros that will be used throughout the program.

Lines 12 through 14 are initialization instructions: they set initial values in three of the special registers. Lines 15 through 21 represent a routine that does a multiplication by 10. Lines 22 and 23 are setup instructions for the division routine of lines 25 through 27. Lines 28 through 34 save the quotient and remainder. Lines 35 through 40 print the digits of E. Lines 42 and 43 count the number of digits.

The statements at lines 46 through 48 reserve a buffer area (a series of locations in memory) to be used by the program and, therefore, not to be assigned to other instructions. The statement at line 50 provides the data for printing the ASCII text message THE VALUE OF E IS: 2.

19.4.5.2 What the Assembly-Listing Errors Are

Note the first column of capital letters in the assembler listing. These letters are the assembler’s code for errors in your program. Note that six lines in the program have errors preventing your program from assembling properly: lines 11, 14, 31, 40, 46, and 53 respectively.

Each capital letter stands for a specific type of error. Line 31, containing two capital letters (AR), has two types of errors. See the *RT–11 System Utilities Manual* for a description of these error codes. See the *RT–11 System Message Manual* for explanations of any other RT–11 system messages displayed during normal system use.

If you do not request a listing when assembling a program, and the program has errors, the assembler displays on your terminal any error codes with the related line numbers.

19.4.5.3 Correcting the Errors

To correct the assembly-listing errors, you need to do the following three things:

1. **First Correction**

   The first error occurs at line 11 and is an M error, which means means a label was defined more than once. You can refer to a label any number of times but you may define it only once. By looking at the CREF user symbol table, you can see that the label is defined at line 11 and again at line 46; one of these definitions is wrong. Examination of the program logic reveals that the definition at line 11 is correct. Before deciding how to change line 46, though, check the other errors to see if one of them indicates what should be done. In fact, the next error encountered (line 14) shows what is wrong. A U error identifies an undefined symbol. The label A is referenced in line 14 but is never defined in the program. It should be defined logically at line 46.

   Therefore, you should change line 46 to read:

   ```
   A:  .REPT   N+1
   ```

   This one change eliminates three errors flagged by the assembler: those at lines 11, 14, and 46.

   Use the KED editor to make the correction. If necessary, review the editing commands in Chapter 4 and then edit the file SUM.MAC so that line 46 is error free.

2. **Second Correction**

   The next error occurs at line 31. Actually, the assembler flagged two errors here. An A error indicates an addressing problem and an R error indicates a register error (invalid use of a register, a special PDP–11 storage feature). If you look at the language statement in line 31, you can see that the ADD operator is followed by one operand. However, ADD is an instruction that requires two operands (two values to be added together) separated by a comma. This statement simply contains a typing error, which can be corrected by inserting a comma between the R2 and the -2(R1).

   Therefore, you should change line 31 to read:

   ```
   ADD      R2,-2(R1)
   ```

   This corrects both the addressing problem and eliminates the invalid register expression.

3. **Third Correction**

   At line 40 is another undefined symbol, the macro symbol .TTYON. Since the program calls the TTYOUT macro (see line 3), this error indicates a misspelling.

   Correct line 40 to read:

   ```
   .TTYOUT
   ```

   Finally, a D error occurs in line 53. This indicates that the program defined a symbol more than once. However, you have eliminated this error by correcting line 46.

When you have finished editing (correcting) the file SUM.MAC, do not rename the file. Rather, reassemble the program, using the MACRO command, and obtain a new object module and a new listing. This time the program should assemble without error.

If errors occur, you have not edited the program correctly. Compare listings and try to correct your errors or reread this section on how to correct the program.

19.5 Linking Object Modules

You must next process the SUM object module with the LINK utility. This utility does the following:

- Links (joins) any needed object modules (system-library and user subroutines) with your program.
- Resolves symbolic and library references.
- Relocates object modules. For example, the memory addresses of two or more object modules that are joined need to be relocated to accommodate one another.
- Assigns absolute-memory addresses to the relative-memory addresses calculated by the MACRO–11 assembler.
- Creates an overlay structure if required.

Chapter 20 describes these link functions. For now, note the first and third function: linking modules and assigning memory addresses.

Linking Object Modules

A primary function of the linker is to supply your program with subroutines that your program can reference in the operating system subroutine library, SYSLIB.OBJ. By referencing these subroutines, you can save yourself a lot of time rewriting code for commonly used functions.

However, the SUM program does not reference any subroutines from SYSLIB.OBJ. The only subroutines this program references are some in the macro library file SYSMAC.SML, and these subroutines were referenced when the program was assembled. Nevertheless, you still need to link the program SUM.OBJ.

Assigning Absolute-Memory Addresses

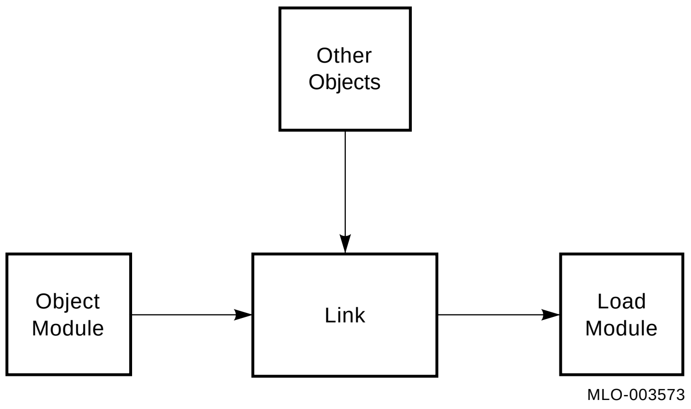
Another function of the linker is to assign the instructions in your program absolute (fixed) memory addresses. Though you do not need to link your SUM program to join subroutines to it, you do need to link that program to assign its code absolute-memory addresses.

Note that the memory addresses on the listing of SUM.MAC begin with address 0. However, a program cannot be stored and run in memory beginning at address 0, since operating system information is already stored at that address. Address 0 is thus relative, referring to the first line of code in the object module of program SUM.

When the LINK utility processes SUM.OBJ, it assigns the program absolute-memory addresses, normally beginning at address 1000, since this begins a large section of free memory space.

The result of a Link operation is a memory-image load module, with any module links resolved, all memory addresses fixed, and storage information assigned (Figure 19–4). The memory-image module, then, is actually a picture of what computer memory looks like just before program execution.

Figure 19–4: Link Operation



Use the LINK command to process your file with the linker. You can omit typing the OBJ file type in the command line, since the LINK command assumes this file type for input. For example, link your file as follows:

```
.LINK SUM RET
```

You do not have to specify an output file name, since RT–11 automatically gives the file it creates the same file name as your input file but with a SAV file type.

Any messages displayed inform you of error conditions discovered during the link operation (for example, if you fail to specify all the necessary input object modules). However, assuming you edited your source program correctly and that it assembled without error, it should also now link without error.

A load module is one that you can run on the RT–11 operating system. Unless your program contains logic errors that prevent it from running properly (errors that RT–11 cannot always detect), running the SAV version of your program should produce the results you intended.

However, if logic errors exist in your program, running the program will produce either erroneous results or none at all. If this is the case, you must study the source program, rework it, reedit it, and rerun the assembly and link operations again.

19.6 Running the MACRO–11 Program

If your MACRO–11 program is error free, running the SAV version should produce the expected results. In this demonstration, running the SUM.SAV file should produce a value on the terminal that is the constant E (2 followed by 70 digits).

To execute the MACRO–11 demonstration program, use the RUN command. You can omit typing the SAV file type, since the RUN command assumes this file type. The format is:

RUN filename

Type the following and note the results displayed on your terminal:

```
.RUN SUM RET
THE VALUE OF E IS:
2.5/606/606237.2301314.06525/130440275535025.7147773735274474540550 2.544
.
```

You can see that something is wrong. There should be only numbers after the "2.", but there are slashes and periods in the result, indicating that an error still exists somewhere in the program.

19.7 Correcting the Final Error

Programming errors, called "bugs," can be difficult to find and fix, especially if the assembler finds no errors. MACRO–11 has a debugging tool to help you find and fix programming errors. This tool is called DBG–11 and is described in Chapter 22. With this tool you can correct the program’s final error and rerun the program. For now, however, use KED to make the final correction.

Look at line 39 in the assembly listing. Note that the instruction in this line converts a digit into the appropriate ASCII code before displaying it on the terminal. To do this, the constant 10 is added into the value of the digit already stored in memory and, then, the value is converted—by the ’0, the ASCII code for 0—to an ASCII code that can be displayed.

Unless you explicitly designate a value as decimal, however, the assembler assumes all values used in the program are octal. Therefore, it interprets the constant as 10_8, that is, 8_{10}, and adds the wrong value every time. The conversion consequently causes the codes of the ASCII characters / and . to be used as results in some cases. The codes of other digits, while representing numeric values, are also off by two.

To correct this error, insert a period after the 10 in line 39 so that it reads:

```
ADD     #10.+'0,R0       ;MAKE DIGIT ASCII
```

The period instructs the assembler to accept the constant value 10 as a decimal value.

19.8 Combining Operations

To produce program results, you first assembled the MACRO–11 source program, SUM.MAC, then linked it, and finally ran the resulting SAV file, SUM.SAV. You can combine these three operations by using one command, the EXECUTE command.

Note, however, that to use the EXECUTE command, you must have the following files on your system volume:

- MACRO.SAV (and CREF.SAV because you want a cross-reference listing)
- LINK.SAV
- SYSLIB.OBJ

  This last file, SYSLIB.OBJ, is required only if the MACRO–11 program you need to link refers to routines that are contained in the system library. The program used in this demonstration, SUM.MAC, does not require SYSLIB.OBJ.

The EXECUTE command instructs the system to select the appropriate language processor, then process, link, and run the program. You have three ways to select which language processor the EXECUTE command invokes.

- You can specify a language-name option, such as /MACRO, which invokes the MACRO–11 assembler.
- You can omit the language-name option but specify the file type for the source file. The EXECUTE command then invokes the language processor that corresponds to that file type. Specifying the file SUM.MAC, for example, invokes the MACRO–11 assembler.
- You can let the system choose a file type of MAC, DBL, or FOR for the source file you name. If, for example, you specify the file SUM, the monitor searches the default data (DK) volume for the files SUM.MAC, SUM.DBL, and SUM.FOR, in that order. If it finds a file named SUM.MAC, it invokes the MACRO–11 assembler to process the file.

Choose the third alternative and type the following command to combine the ASSEMBLE, LINK, and RUN operations as one:

```
.EXECUTE SUM/LIST/CROSSREFERENCE RET
THE VALUE OF E IS:
2.7182818284590452353602874713526624977572470936999595749669676277240766
```

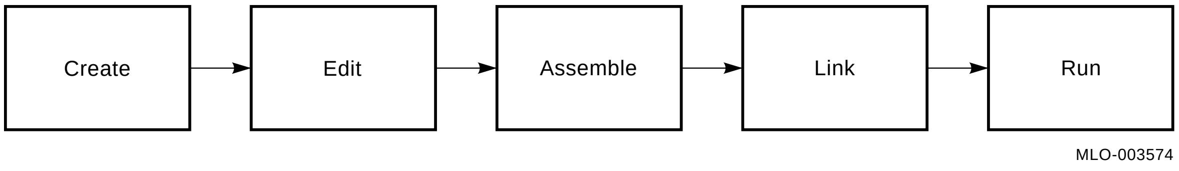
Note how you use the /LIST and /CROSSREFERENCE options following the file name to request both an assembly and a cross-referenced listing.

19.9 Chapter Summary

19.9.1 Summary of the Evolution of a MACRO–11 Program

Figure 19–5 illustrates the main steps you take to develop and run a MACRO–11 program.

Figure 19–5: Evolution of a MACRO–11 Program



19.9.2 Summary of Commands to Run MACRO–11 Programs

EXECUTE

Combines the assemble-link-run operations into one command.

EXECUTE *file/MACRO*

Combines the process-link-run operations into one command, and specifies the input file to be a MACRO–11 file.

EXECUTE/LIST/CROSSREFERENCE

Produces a cross-referenced listing file.

EXECUTE/LIST

Produces a listing file of the source program.

LINK

Links individual object modules and forms a complete program and produce a load module (a file of type SAV).

MACRO

Assembles the MACRO–11 source program and produces an object module (a file of type OBJ).

MACRO/LIST/CROSSREFERENCE

Assembles the MACRO–11 source program and produces both an object module and a cross-referenced listing file.

MACRO/LIST

Assembles the MACRO–11 source program and produces both a listing on the printer and an object module.

RUN

Runs the indicated load module.

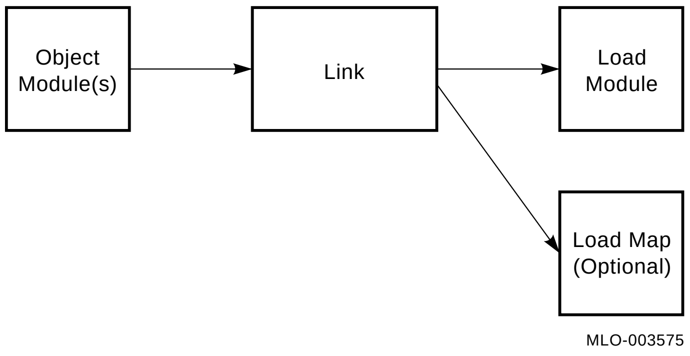
Chapter 20

Linking Object Programs

MACRO–11 and FORTRAN programs require additional processing after their conversion to object format. Before you can run these programs on the system, you must link them. The link operation:

- Joins the object modules that use a symbol with the object module that defines it.
- Relocates individual object modules as necessary and assigns absolute (permanent) memory addresses; it can also define an overlay structure.
- Produces a load module and an optional load map (Figure 20–1).

Figure 20–1: Link Functions



Program linking gives you the advantage of a module approach to programming. You can create a program as a series of smaller, independent subprograms. One of these is written as the main, or controlling, program, and the rest as subordinate subprograms and subroutines. You use a language processor to translate each part of the program into an object module. Then, you use the linker to join all the object modules into a complete, functioning unit.

Module programming makes program creation and debugging easier. For example, several programmers can simultaneously work on a single program, each creating a portion of it. The individual portions, or subprograms, can be processed and linked with test programs and debugged for logic errors separately. Then, all the object modules can be joined to form a complete program that can be tested as a whole. If errors occur at this stage, only those object modules with errors need to be debugged and changed.

In addition, modular programming lets you make use of library files. These are files containing subprograms and subroutines that have been debugged. After you join library files with your program at link-time, their routines can be used by your program as needed.

20.1 Resolving Symbolic and Library References

The linker reads through all the object modules that you indicate as input to the LINK command. It gathers and evaluates information (provided to the modules by the language processor) that is necessary for program linking. For each input module, this information includes the object code, information needed for relocation, the relative address of the first instruction, the global symbols used, and the absolute length of each program and program section.

One of the linker's first functions is to resolve all user-defined symbolic references and library references in the joined routines. The are two types of user-defined symbols are internal symbols and global symbols.

Internal symbols are limited to the object module in which they appear; thus, they cannot be referenced from or defined in any other module. A program containing only internal symbolic references—like those in the demonstration program in Chapter 19—is complete in itself and does not need to be joined with any other object programs at link-time. Thus, internal symbols are not resolved at link-time because they have already been resolved by the language processor.

Global symbols, on the other hand, are the key to modular programming. Global symbols provide the communication between object modules. Such symbols may be symbolic labels to instructions, symbolic labels to data, or symbols that are equated to a value or constant. Global symbols are defined in one object module and referenced from other object modules that have been separately assembled or compiled. Such symbols must be designated as global in the source code. The following segment of MACRO–11 assembly language code illustrates the use of global symbols:

```
.MAIN.  MACRO V05.04  Friday 18-Oct-91 14:45  Page 1

      1                                        .GLOBL  A,C,VALUE       ;DECLARE A, C, AND VALUE
      2                                                                ;AS GLOBAL SYMBOLS
      3 000000  013500                     A:  MOV     @(R5)+,R0       ;GLOBAL SYMBOL A IS DEFINED
      4                                                                ;HERE AND CAN BE REFERENCED
      5                                                                ;FROM OTHER MODULES, PROBABLY
      6                                                                ;BY A SUBROUTINE CALL
      7 000002  016701  000016                 MOV     LOCAL,R1        ;LOCAL IS AN INTERNAL SYMBOL
      8                                                                ;DEFINED AND REFERENCED ONLY
      9                                                                ;WITHIIN THIS MODULE
A    10 000006  000000G  000007  000000G       USR     PC,C            ;CALL TO GLOBAL ROUTINE C
     11                                                                ;DEFINED IN ANOTHER MODULE
     12 000014  013501                         MOV     @(R5)+,R1
     13 000016  005003                         CLR     R3
     14 000020  000207                         RTS     PC
     15 000022  000011                 VALUE:  .WORD   11              ;GLOBAL SYMBOL VALUE IS USED TO
     16                                                                ;REFERENCE THIS DATA LOCATION
     17 000024  177777                 LOCAL:  .WORD  177777           ;INTERNAL SYMBOL USED FOR DATA
     18         000001                         .END
```

While internal symbolic references, such as LOCAL in the example, can be resolved by the assembler or compiler in the single program unit, global references, such as C, cannot. They require other object modules. During translation, the language processor notes in the object module those symbols that are global. During linking, the linker keeps track of the global references and definitions found in all the object modules. As linking proceeds, it makes the appropriate correlations and modifies instructions or data as necessary. After linking, the linker prints on the terminal a list of all symbolic references that were not resolved (undefined globals), either because of a programming error or because all necessary object modules were not included in the linking process.

References to library files also involve the use of global symbols. You access the routines in a library by naming a routine as a global symbol in the source code of your program. You then link your program with the appropriate library file, and the linker resolves the library references as it does any global symbol. Library usage is discussed in greater detail in Chapter 21.

20.2 Program Relocation and Address Assignment

A second important function of the linker is to 'fix' the relative memory addresses so that they are absolute.[1] The object module represents translated source instructions that have been assigned memory addresses relative to a base address of 0.

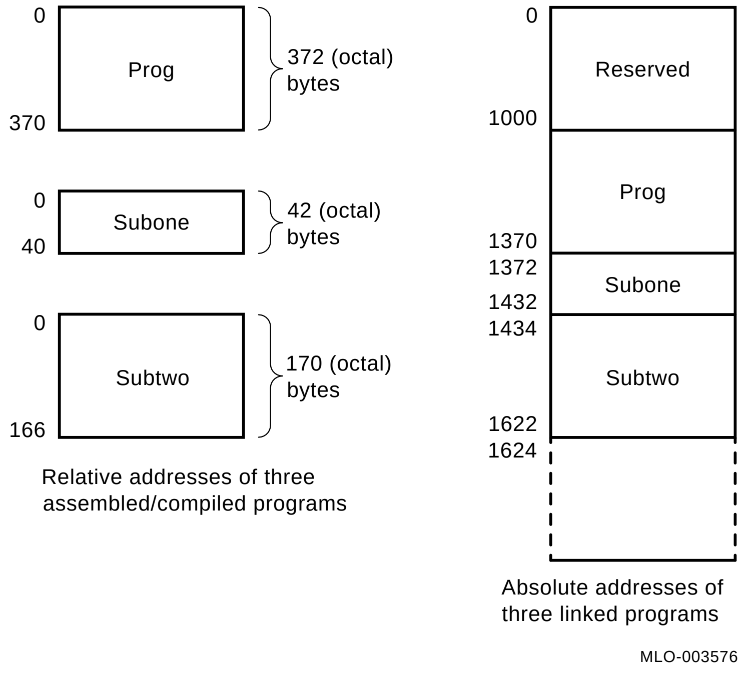
Look back at the assembly listing in Chapter 19. Note the second column; these addresses are relative to a base address of 0. Thus, the first instruction is assembled at relative address 0, the second at relative address 6, and so on. A program cannot actually be stored and run in memory by using locations relative to address 0, however, because system information is already stored in some of these locations. For example, the RT–11 operating system uses byte addresses 40 through 57 to store information about the program currently executing. In addition, the RT–11 operating system uses locations in the upper range of memory for storing the resident monitor. Thus, the linker must assign memory addresses to your program that are not already in use or that will not be used during program execution. It must, therefore, assign absolute memory addresses to the relative addresses assigned by the language processor.

The linker normally starts assigning memory addresses at address 1000_8, since this begins a large section of free memory space. So, to obtain the actual addresses used for program loading, you must add the relocation constant 1000 to each relative address shown in the assembly listing.

A conflict arises when several individually processed object modules are linked. The linker cannot assign memory addresses starting at 1000_8 to every module, since address assignments of one would then override those of another. However, part of the information that the language processor calculates and passes to the linker is the size of each program section in each module. So, the linker adds this size into the relocation constant for each module and assigns higher addresses, appropriately modifying the relative location of all instructions and data as necessary to account for the relocation of each individual module. Figure 20–2 illustrates the relocation that must occur to accommodate the object modules.

[1] FORTRAN and BASIC–PLUS users who have not performed the demonstration in Chapter 19 can read Section 19.3. That section explains the concept of converting and storing instructions in computer memory.

Figure 20–2: Object Module Relocation



20.3 Absolute and Relocatable Program Sections

Just as global symbols let you create an entire program, using several individual object modules, program sections let you create an object module as a series of individual sections. The advantages gained through the sectioning of programs relate primarily to control of memory allocation, program modularity, and more effective partitioning of memory. The linker processes the program section information in the object modules as directions on how to create the executable program image.

The FORTRAN and MACRO–11 language processors insert program sectioning information into the object module. The FORTRAN language processor does this automatically when program sectioning is implied by the source language statements in a user program. For example, FUNCTION, SUBROUTINE, and COMMON statements result in the production of program section directives. In MACRO–11 assembly language, you are responsible for explicitly directing the assembler to output program section information for the linker. You do this through the .PSECT (or .CSECT and .ASECT) MACRO–11 assembly language statement.

Some of the basic functions associated with program sections are:

1. Instructions or data can be placed in absolute locations in memory. The named absolute program section (. ABS.) lets you instruct the linker as to exactly where to place program code or data. Declaring a section as part of the absolute program section instructs the assembler or compiler to use the internal value of the program counter as the physical memory address to be assigned after

linking. This section is processed relative to absolute memory address 0 and is not relocated at link-time.

2. Named relocatable program sections are used to group data or instructions into logical portions of memory. The FORTRAN COMMON statement invokes this construct to allow access to named data areas from many separate routines. Declaring a section as part of a named relocatable program section causes the section to be processed at relocatable address 0. Such sections are relocated by the linker.
3. If you do not need exact control over where a portion (section) of a program will be placed in memory, use the blank program section—a special program section that the linker treats as relocatable. The linker decides where to place this program section in the loadable memory image. The blank program section is the default for a MACRO–11 source program and remains in effect until an explicit program section is identified (the program example in Chapter 19 used the blank program section).
4. A program section can be identified as an instruction section. The linker, using this information, can provide automatic loading of declared overlay code when needed by the executing program (this will be discussed further in more detail).

The language processor, then, maintains several program counters—one for the absolute program section, one for the unnamed relocatable program section, and as many as needed (maximum is 254) for named relocatable program sections. The assembled example that follows helps explain this concept.

```
.MAIN.  MACRO V05.04  Friday 18-Oct-91 12:09  Page 1


  1
  2                                                        ;UNNAMED RELOCATABLE PROGRAM
  3                                                        ;SECTION IS DECLARED (BY DEFAULT)
  4                                                        ;(".PSECT" IS ASSUMED)
  5  000000  005000                  START:  CLR     R0
  6  000002  012701  000034'                 MOV     #BEG,R1
  7  000006  062100                   LOOP:  ADD     (R1)+,R0
  8  000010  022701  000044'                 CMP     #BEG+10,R1
  9  000014  100374                          BPL     LOOP
  10 000016  012767  002000  000020          MOV     #2000,ADDR
  11 000024  005003                          CLR     R3
  12 000000                                  .PSECT  CLEAR    ;NAMED RELOCATION PROGRAM
  13 000000  012703  000100                  MOV     #100,R3  ;SECTION IS DECLARED (VIA ".PSECT NAME")
  14 000004  012701  000044'                 MOV     #ADDR,R1
  15 000010  005021                  AGAIN:  CLR     (R1)+
  16 000012  005303                          DEC     R3
  17 000014  001375                          BNE     AGAIN
  18 000000                                  .ASECT           ;ABSOLUTE PROGRAM SECTION
  19 000042                          .=42              ;DECLARED (VIA ".ASECT")
  20 000042  001000                          .WORD   1000     ;THE VALUE 1000 WILL BE
  21                                                          ;STORED IN ABSOLUTE MEMORY LOCATION 42
  22                                                          ;WHEN THE PROGRAM IS EXECUTED
  23 000026                                  .PSECT           ;BACK TO UNNAMED RELOCATABLE
  24 000026  005267  000012                  INC     ADDR     ;PROGRAM SECTION
  25 000032  000000                          HALT
  26 000034  000001  000002  000003  BEG:    .WORD    1,2,3,4
  27 000042  000004
  28 000044  000000                  ADDR:   .WORD   0
  29                 ;NOTE THAT YOU CAN WRITE LANGUAGE STATEMENTS THAT WILL BE LOADED
  30                 ;CONTIGUOUSLY IN MEMORY, BUT DO NOT NECESSARILY OCCUR CONTIGUOUSLY
  31                 ;IN THE SOURCE PROGRAM (I.E., THE CODE AT LINES 1-11 AND 22-29)
  32 000001                          .END
```

Since the system does not know at assembly (or compile) time into which actual memory locations each relocatable section goes, all references among sections (see line 18) are relative to the base of the section. This information is then passed to the linker so that it can make the appropriate adjustments at link-time.

The Overlay Feature

The RT–11 linker can handler the relocation and address assignments that are required whenever you indicate that an overlay structure is needed. An overlay structure is necessary when you write a program that is too large to fit in the available memory of your system. You write the program in discrete parts (some programming restrictions must be observed) so that your program can subsequently be executed in parts. Some of these parts, or segments, are allowed to share memory with other segments, thus reducing the overall memory requirements of the program. One segment of the program is called the root segment and must remain in memory at all times. The root segment contains the necessary information for use by the other segments of the program, called overlay segments. Overlay segments are stored on storage volumes and brought into memory as needed. The purpose of the overlay structure is for parts of the program to share the available memory in such a way that when one part is complete, it is overlaid (and therefore erased) by another.

You indicate how to plan to overlay your program by using the /PROMPT option in the LINK command line. The linker then creates a load module that contains the necessary information for loading the appropriate segments as needed during execution. The *RT–11 System Utilities Manual* explains the overlay feature in more detail. You need not specify an overlay structure for the examples demonstrated in this chapter.

20.4 Producing a Load Module and a Load Map

The load module is the result of the linking processes described so far: joining object modules, resolving symbolic and library references, relocating object modules, assigning absolute addresses, and creating an overlay structure if required. The load map is essentially a synopsis of the load module—that is, what memory looks like when the program is loaded and ready to be executed.

In Chapter 17 and Chapter 19, you produced load modules, but you did not request load maps. You obtain a load map by using the /MAP option with the LINK (or EXECUTE) command. At this time, relink the MACRO–11 or FORTRAN object module (SUM.OBJ or GRAPH.OBJ) and use the /MAP option to produce a load map. The load map is created as a file on the default data (DK) volume, which is the default storage volume for input/output operations. The load map has the name of the first input module and a file type of .MAP.

With MACRO–11

Issue the following command to produce a MACRO–11 load module and load map:

```
.LINK SUM/MAP RET
```

Now, list the .MAP file on either the printer or terminal, choosing the appropriate command:

To print the MACRO–11 load map:

```
.PRINT SUM.MAP RET
```

To display the MACRO–11 load map:

```
.TYPE SUM.MAP RET
```

With FORTRAN

The command you issue to produce a load map is determined by the FORTRAN compiler (FORTRAN IV or FORTRAN–77) you are using.

If you are using the FORTRAN IV compiler, issue the following command to produce a load module and load map:

```
.LINK  GRAPH/MAP,FORLIB RET
```

If you are using the FORTRAN–77 compiler, issue the following command to produce a load module and load map:

```
.LINK GRAPH/MAP,F77OTS RET
```

Now, list the .MAP file on either the printer or terminal, choosing the appropriate command:

To print the FORTRAN load map:

```
.PRINT GRAPH.MAP RET
```

To display the FORTRAN load map:

```
.TYPE GRAPH.MAP RET
```

20.5 Examining the Load Maps

Load maps for MACRO–11 and FORTRAN programs have the same general format. The first line contains header information. The second line has the name and file type of the load module created. Next, the absolute section and each named and unnamed section are listed under the SECTION column. To the right are abbreviated codes designating whether the section is Read/Write or Read Only, contains Instructions or Data, is a Local or Global section, is Relocatable or Absolute, is Concatenated or Overlaid. Below this falls a listing of all the global symbols (GLOBAL) and their values (VALUE). Finally, at the end of the map is the transfer address where the program actually starts when executed, followed by the high limit—the total number of bytes used by all the individual program sections.

The following subsections provide and briefly describe the load maps for the demonstration programs.

LOAD MAP FOR MACRO–11 (SUM.MAP)

Look first at the MACRO–11 load map. The default absolute section starts at absolute location 0; its size is 1000 bytes. Thus, it extends from absolute memory location 0 through absolute memory location 777_8. The unnamed program section (there were no named program sections in this program) starts at absolute 1000; its size is 374_8 bytes. Thus, it extends from absolute location 1000 to absolute location

1372_8. The high limit of this program (total bytes) is therefore 1372_8. Since this program is not linked to any other object modules, there are no global symbols and the rest of the map is blank.

```
RT-11 LINK  V05.15      Load Map        Friday 18-Oct-91 15:08  Page 1
SUM   .SAV      Title:  SUM.MA  Ident:

Section  Addr   Size    Global  Value   Global  Value   Global  Value

 . ABS.  000000 001000 = 256.   words  (RW,I,GBL,ABS,OVR)
         001000 000372 = 125.   words  (RW,I,LCL,REL,CON)

Transfer address = 001000, High limit = 001370 = 380.   words
```

LOAD MAP FOR FORTRAN IV (GRAPH.MAP)

Look now at the FORTRAN IV load map, remembering that it reflects the appropriate expansions into machine language code provided by the FORTRAN IV compiler. Again, the absolute section extends from absolute 0 through absolute 777_8. Globals listed in the absolute section show the global variable names that the program uses as constants throughout the program.

The unnamed relocatable program section begins at absolute location 1000_8. Some of the named relocatable sections that are declared are OTSP, SYSI, and $CODE. Global symbols and their respective addresses appear to the right of all sections. The total number of bytes used is 24474_8 or 5278_{10} words.

```
RT-11 LINK  V05.15      Load Map        Friday 18-Oct-91 15:19  Page 1
FORLIB.SAV      Title:  .MAIN.  Ident:  FORV02

Section  Addr   Size    Global  Value   Global  Value   Global  Value

 . ABS.  000000 001000 = 256.   words  (RW,I,GBL,ABS,OVR)
                        $USRSW  000000  $RF2A1  000000  $HRDWR  000000
                        .VIR    000000  .LEN    000002  .REPEA  000002
                        .SCOPY  000003  $NLCHN  000006  $SYSV$  000020
                        .SYSLB  000020  $WASIZ  000152  $LRECL  000210
                        $TRACE  004737
 OTS$I   001000 017610 = 4036.  words  (RW,I,LCL,REL,CON)
                        $$OTSI  001000  $CVTIF  001000  $CVTIC  001014
                        $CVTID  001014  CCI$    001026  CDI$    001026
                        $IC     001026  $ID     001026  CFI$    001042
                        $IR     001042  EXP     001126  ADF$IS  001466
                        ADF$PS  001474  SUF$PS  001500  SUF$MS  001504
                        ADF$MS  001516  SUF$IS  001526  $ADDF   001534
                        $SUBF   001550  SUF$SS  001562  $SBR    001562
                        ADF$SS  001566  $ADR    001566  ADD$    001602
                        DIF$PS  002226  DIF$MS  002232  DIF$IS  002242
                        $DIVF   002250  DIF$SS  002262  $DVR    002262
                        MUF$PS  002550  MUF$MS  002554  MUF$IS  002564
                        $MULF   002572  MUF$SS  002604  $MLR    002604
                        $OTI    003142  $$OTI   003144  $SETOP  003354
                        $$SET   005026  SQRT    005322  STK$L   005516
                        STK$I   005522  STK$F   005526  IOR$    005536
                        AND$    005542  EQV$    005550  XOR$    005552
                        NMI$1M  005566  NMI$1I  005600  BLE$    005610
                        BEQ$    005612  BGT$    005620  BGE$    005622
                        BRA$    005624  BNE$    005630  BLT$    005632
                        CAI$    005642  CAL$    005650  $OPNER  005700
                        $CHKER  005736  $IOEXI  005762  $EOL    006030
                        EOL$    006032  EXIT    006146  MOF$SS  006152
                        MOF$MS  006164  MOF$PS  006176  MOF$SM  006202
                        MOF$SP  006212  MOF$MM  006216  MOF$MA  006230
                        MOF$MP  006236  MOF$PM  006244  MOF$PA  006250
                        MOF$PP  006254  MOF$RS  006260  MOF$RM  006266
                        MOF$RA  006276  MOF$RP  006302  NGD$S   006306
                        NGF$S   006306  NGD$M   006320  NGF$M   006320
                        NGD$P   006334  NGF$P   006334  NGD$A   006340
                        NGF$A   006340  ADI$SS  006344  ADI$SA  006350
                        ADI$SM  006354  ADI$IS  006360  ADI$IA  006364
                        ADI$IM  006370  ADI$MS  006374  ADI$MA  006400
                        ADI$MM  006404  CMI$SS  006410  CMI$SI  006414
                        CMI$SM  006420  CMI$IS  006424  CMI$II  006430
                        CMI$IM  006434  CMI$MS  006440  CMI$MI  006444
                        CMI$MM  006450  IFW$    006454  $IFW    006460
                        $$IFW   006464  IFW$$   006522  MOI$RS  006572
                        MOL$RS  006572  MOI$RM  006576  MOI$RP  006602
                        MOI$RA  006604  MOI$SS  006610  MOL$SS  006610
                        MOI$SM  006614  MOI$SA  006620  MOI$IS  006624
                        MOL$IS  006624  REL$    006624  MOI$IM  006630
                        MOI$IA  006634  MOI$MS  006640  MOI$MM  006644
                        MOI$MA  006650  MOI$0S  006654  MOI$0M  006660
                        MOI$0A  006664  MOI$1S  006670  MOI$1M  006676
                        MOI$1A  006704  ICI$S   006712  ICI$M   006716
                        ICI$P   006722  ICI$A   006724  DCI$S   006730
                        DCI$M   006734  DCI$P   006740  DCI$A   006742
                        IDINT   006746  INT     006746  MOI$IP  006774

RT-11 LINK  V05.15      Load Map        Friday 18-Oct-91 15:19  Page 2
FORLIB.SAV      Title:  .MAIN.  Ident:  FORV02

                        MOI$SP  006776  MOI$PP  007004  MOI$MP  007010
                        MOI$PS  007020  MOI$PM  007026  MOI$PA  007034
                        MOI$0P  007042  MOI$1P  007050  ISN$    007060
                        $ISNTR  007064  LSN$    007100  $LSNTR  007104
                        SUI$SS  007240  SUI$SA  007244  SUI$SM  007250
                        SUI$IS  007254  SUI$IA  007260  SUI$IM  007264
                        SUI$MS  007270  SUI$MA  007274  SUI$MM  007300
                        MOL$SM  007304  MOL$SA  007310  MOL$MS  007314
                        MOL$MM  007324  MOL$MA  007330  MOL$SP  007334
                        MOL$PP  007342  MOL$MP  007346  MOL$PM  007356
                        MOL$PS  007364  MOL$PA  007370  MOL$IM  007376
                        MOL$IA  007404  MOL$IP  007412  LLE$    007422
                        LEQ$    007424  LGT$    007432  LGE$    007434
                        LNE$    007444  LLT$    007446  TSL$S   007452
                        TSL$M   007456  TSL$I   007462  TSL$P   007470
                        MAX0    007476  MIN0    007522  RET$L   007546
                        RET$F   007552  RET$I   007560  RET$    007562
                        $OTIS   007616  $$OTIS  007620  TVL$    007740
                        $TVL    007740  TVF$    007746  $TVF    007746
                        TVD$    007754  $TVD    007754  TVQ$    007762
                        $TVQ    007762  TVP$    007770  $TVP    007770
                        TVI$    007776  $TVI    007776  SAL$IM  010132
                        SAL$SM  010134  SVL$IM  010140  SVL$SM  010142
                        SAL$MM  010150  SVL$MM  010154  $CVTFB  010160
                        $CVTFI  010160  $CVTCB  010174  $CVTCI  010174
                        $CVTDB  010174  $CVTDI  010174  CIC$    010206
                        CID$    010206  CLC$    010206  CLD$    010206
                        $DI     010206  CIF$    010216  CLF$    010216
                        $RI     010216  CIL$    010330  CLI$    010334
                        $INITI  010336  $CLOSE  010454  $ERRTB  011232
                        $ERRS   011337  $FCHNL  015100  $FIO    015742
                        $$FIO   015746  $PUTRE  017112  $PUTBL  017420
                        $GETBL  017630  $EOFIL  020014  $EOF2   020030
                        SAVRG$  020050  THRD$   020226  $STPS   020230
                        STP$    020236  $STP    020236  FOO$    020242
                        $EXIT   020262  $WAIT   020406  $VRINT  020450
                        $DUMPL  020462
 OTS$P   020610 000054 = 22.    words  (RW,D,GBL,REL,OVR)
 SYS$I   020664 000246 = 83.    words  (RW,I,LCL,REL,CON)
                        LEN     020664  REPEAT  020702  SCOPY   021034
 USER$I  021132 000000 = 0.     words  (RW,I,LCL,REL,CON)
 $CODE   021132 001056 = 279.   words  (RW,I,LCL,REL,CON)
                        $$OTSC  021132  FUN     022042
 OTS$O   022210 001036 = 271.   words  (RW,I,LCL,REL,CON)
                        $$OTSO  022210  $OPEN   022210
 SYS$O   023246 000000 = 0.     words  (RW,I,LCL,REL,CON)
 $DATAP  023246 000072 = 29.    words  (RW,D,LCL,REL,CON)
 OTS$D   023340 000006 = 3.     words  (RW,D,LCL,REL,CON)
                        NHCLN$  023344
 OTS$S   023346 000002 = 1.     words  (RW,D,LCL,REL,CON)
                        $AOTS   023346
 SYS$S   023350 000004 = 2.     words  (RW,D,LCL,REL,CON)
                        $SYSLB  023350  $LOCK   023352  $CRASH  023353
 $DATA   023354 000520 = 168.   words  (RW,D,LCL,REL,CON)
 USER$D  024074 000000 = 0.     words  (RW,D,LCL,REL,CON)
 .$$$$.  024074 000000 = 0.     words  (RW,D,GBL,REL,OVR)

RT-11 LINK  V05.15      Load Map        Friday 18-Oct-91 15:19  Page 3
FORLIB.SAV      Title:  .MAIN.  Ident:  FORV02

Transfer address = 021132, High limit = 024072 = 5149.  words
```

LOAD MAP FOR FORTRAN–77 (GRAPH.MAP)

Examine the FORTRAN–77 load map. It too reflects the machine language code expansions. The absolute section extends from absolute 0 through absolute 777_8. The symbol names listed in the absolute section are used as constants throughout the program. Following the absolute section are local and global relocatable sections, such as $$NAM, $$OTSV, and so on. Listed under each section are the global symbols located within the section. The last line of map shows the program uses 5786_{10} words.

```
RT-11 LINK  V05.15      Load Map        Friday 18-Oct-91 15:26  Page 1
GRAPH .SAV      Title:  .MAIN.  Ident:  18OCT

Section  Addr   Size    Global  Value   Global  Value   Global  Value

 . ABS.  000000 001000 = 256.   words  (RW,I,GBL,ABS,OVR)
                        $VIRSZ  000000  $USRSW  000000  .LEN    000002
                        .REPEA  000002  .SCOPY  000003  $NLCHN  000006
                        $HRDWR  000010  $SYSV$  000020  .SYSLB  000020
                        $WASIZ  000161  $LRECL  000210
 $$NAM   001000 000002 = 1.     words  (RW,I,LCL,REL,CON)
                        $NAM$   001000
 $$OTSV  001002 000010 = 4.     words  (RW,D,GBL,REL,OVR)
                        QBLK$   001002  VIRSZ$  001004  IOBKF$  001006
                        IOBKL$  001010
 $$OTSS  001012 000024 = 10.    words  (RW,D,LCL,REL,CON)
 $$AOTS  001036 000700 = 224.   words  (RW,D,LCL,REL,CON)
                        $OTSV   001036  $AOTS   001036  $OTSVA  001040
                        $SEQC   001040  $NAMC   001066  .NLUNS  001070
                        .MOLUN  001072  $ERCNT  001220  PRINT$  001266
                        $TYPE   001270  $ACCPT  001272  $READ$  001274
                        $MXFNL  001312  $LUN0   001330  LIMIT$  001402
                        WNMAP$  001406  WNDOW$  001412
 $$OBF1  001736 000100 = 32.    words  (RW,D,LCL,REL,OVR)
 $$OBF2  002036 000000 = 0.     words  (RW,D,LCL,REL,CON)
 $$FIOS  002036 000000 = 0.     words  (RW,D,GBL,REL,OVR)
                        $LICSB  002036
 $$FIOC  002036 003546 = 947.   words  (RO,I,GBL,REL,CON)
                        $FIO    003122
 $$FIOD  005604 000366 = 123.   words  (RO,D,GBL,REL,CON)
 $$FIO2  006172 000050 = 20.    words  (RO,D,GBL,REL,OVR)
 $$FIOI  006242 000000 = 0.     words  (RO,I,GBL,REL,OVR)
 $$FIOL  006242 000000 = 0.     words  (RO,I,GBL,REL,OVR)
 $$FIOR  006242 000000 = 0.     words  (RO,I,GBL,REL,OVR)
 $$FIOZ  006242 000000 = 0.     words  (RO,I,GBL,REL,OVR)
 $$OTSD  006242 004452 = 1173.  words  (RO,D,LCL,REL,CON)
                        $ERRTB  006530  $ERRTE  006706  $FTRAP  006750
                        $ERTXT  007116
 $$OTSU  012714 001600 = 448.   words  (RO,I,LCL,REL,CON)
                        $QSET   012714  $RTOPN  012730  $CLOSE  014076
 $$OTSI  014514 010036 = 2063.  words  (RO,I,LCL,REL,CON)
```

```
                        OTI$    014514  $OTIS   015344  $POLSH  015344
                        ORGSQ$  015346  ORGIX$  015346  ORGRL$  015346
                        OSF$    015350  OSFE$   015362  $INITI  015462
                        EOLST$  016230  IOAB$   016352  IOVB$   016362
                        IOAL$   016400  IOVL$   016410  IOAM$   016426
                        IOVM$   016436  IOAI$   016454  IOVI$   016464
                        IOAJ$   016502  IOVJ$   016512  IOAR$   016530
                        IOVR$   016540  IOAD$   016556  IOVD$   016566
                        IOAC$   016604  IOVC$   016614  IOAH$   016716
                        IOACH$  017004  EXIT    017020  $EXP    017052
                        $DEXP   017074  $$EXP   017122  $$DEXP  017140
                        $SQRT   017312  $DSQRT  017334  $$SQRT  017362
                        $$DSQR  017364  $SST0   017526  $SST1   017536
                        $SST2   017550  $SST3   017556  $SST4   017564
                        $EXIT$  017614  $SST6   017642  $SST7   017704
                        $ERRAA  017712  $ERXIT  020120  $EXIT   020140
                        EXIT$   020140  $PUTS   020210  $OPEN   020766
                        $OPEN$  021106  $FLDEF  021402  $FCHNL  021512
                        $GETFI  021654  $IOEXI  021706  NAM$    021752

RT-11 LINK  V05.15      Load Map        Friday 18-Oct-91 15:26  Page 2

GRAPH .SAV      Title:  .MAIN.  Ident:  18OCT

                        $NAM    022012  $VINIT  022014  $NOLXM  022022
                        $SAVP0  022026  $SAVP1  022034  $SAVP2  022042
                        $SAVP3  022050  $SAVP4  022056  $SAVP5  022064
                        $SAVP6  022072  $SAVP7  022100  $SAVP8  022106
                        $SAVPX  022176  $SAVPC  022204  $FPERR  022214
                        $STFPP  022400  $SVFPP  022420  $ERRLG  022504
                        $ERRZA  023240  $BINAS  023246  $FILL   023322
                        .SAVR1  023432  $OTSH1  023472  $OTSH0  023502
                        $DUMPL  023504  $PUTBL  023666  $GETBL  024076
                        $EOFIL  024254  $EOF2   024266  $WAIT   024310
                        $R50    024352  $ERRNL  024472  $ERRW1  024504
                        $DET1L  024514  $DET    024520  $DETIC  024520
                        $ATT    024536  $REAMO  024550
 $SPACE  024552 000000 = 0.     words  (RO,D,LCL,REL,CON)
 $CODE1  024552 000662 = 217.   words  (RW,I,LCL,REL,CON)
                        FUN     025326
 $PDATA  025434 000114 = 38.    words  (RW,D,LCL,REL,CON)
 $VARS   025550 000434 = 142.   words  (RW,D,LCL,REL,CON)
 $TEMPS  026204 000004 = 2.     words  (RW,D,LCL,REL,CON)
 . VIR.  026210 000000 = 0.     words  (RW,D,GBL,REL,CON)
 $IDATA  026210 000004 = 2.     words  (RW,D,LCL,REL,CON)
 SYS$I   026214 000246 = 83.    words  (RW,I,LCL,REL,CON)
                        LEN     026214  REPEAT  026232  SCOPY   026364
 SYS$S   026462 000004 = 2.     words  (RW,D,LCL,REL,CON)
                        $SYSLB  026462  $LOCK   026464  $CRASH  026465

Transfer address = 024552, High limit = 026464 = 5786.  words
```

Load maps are most helpful when used in debugging to locate and correct assembly language programming errors. Load maps are not generally obtained or used for FORTRAN programs, except to determine program size. In Chapter 22, you will see how a load map is used to debug the one remaining error in the MACRO–11 demonstration program.

20.6 Summary of Commands for Linking Programs

The following commands are described and illustrated in this chapter. If the meaning of any command is unclear, review the example in this chapter that illustrates the command.

LINK

Link individual object modules together to form a complete program and to produce a load module.

LINK/MAP

Link individual object modules, and produce a load map showing all address assignments made during linking.

Chapter 21
Constructing Library Files

A library is a specially constructed file consisting of one or more programming routines. Generally, these routines provide services that you may need repeatedly or services that are related and so have been gathered for ease of use and storage. You use the routines in a library by joining the library file with your source program. Usually this occurs at link-time; but with assembly language programs, it can also occur at assembly-time.

The RT–11 operating system provides several library files; SYSMAC and SYSLIB for example. These libraries supply the monitor services, input and output routines, conversion routines, and other programming services that user programs may need. You can create other library files yourself. Thus you can construct libraries that contain routines specific to your programming needs or to the combined needs of those using your RT–11 system.

Use the on-line index utility, INDEX, and see the on-line help utility, HELP, for more information on topics described in this chapter.

21.1 Kinds of Library Files

The two kinds of library files are macro libraries and object libraries.

21.1.1 Macro Libraries

Macro libraries, such as SYSMAC.SML, are used by MACRO–11 source programs at assembly-time and consist entirely of macros. A macro is described in Chapter 19 as a recurring sequence of coding instructions, which, when defined in a .MACRO statement, can then be called and used anywhere in your program. A macro library is a collection of macro definitions gathered into a single file. To use the macros in a macro library, you name those macros you plan to use in a .MCALL statement. When the assembler encounters the .MCALL statement during processing, it searches the appropriate macro library (SYSMAC.SML is default) for the definitions. It takes the definitions from the library and inserts them in a macro symbol table where they are available for use during assembly.

21.1.2 Object Libraries

Object libraries, such as SYSLIB, are used by assembled MACRO–11 source programs and/or by compiled FORTRAN source programs at link-time. These libraries consist of object modules that contain global routines; such routines have been defined with global entry points and then named as global symbols in the source program. During the link operation, the linker searches the object libraries to determine if they provide definitions for any undefined globals. If the linker finds definitions, it takes those object modules containing the definition from the library and includes them in the link.

The global symbol table lists each global in a given object library. You can print this list on the terminal or the printer and thus keep track of an object library's current contents.

21.2 Creating and Maintaining a Library File

You create a library file by dcombining several macro routines or several object modules into a single larger file using the LIBRARY command.

To build a macro library, first use the editor to create an ASCII text file that contains all the macro definitions. Then, process this file by using the LIBRARY command in combination with its /MACRO option. To update a macro library (that is, to add or delete macro definitions), edit the ASCII text file and, then, reprocess the file with the LIBRARY command.

To build an object library, use the editor to create an ASCII text file. The file contains the routines and functions written as complete program segments in either the MACRO–11 assembly language or a FORTRAN programming language. Then process the file, producing an object module. Next, use the LIBRARY command in combination with its /CREATE option. Once the library file is created, update it (add and delete routines) by means of other options to the LIBRARY command.

In the following exercises, you create an object library that contains three input object modules. The routines in two of these modules can be used by both MACRO–11 and FORTRAN programs; the routine in the third module can be used by only FORTRAN programs.

To build the library file, first use the editor to create the three ASCII text files. Then convert the ASCII text files to object format. Finally, process the object files with the LIBRARY command. Once you create the library files, use LIBRARY command operations and options to add and delete modules and globals and to obtain a listing of the library file contents.

21.2.1 Creating Object Library Input Files

The first step in building an object library is to prepare the source code of the routines and functions that you choose to include in the library. Use the editor to create the following three text files, calling them FIRST.MAC, SECOND.MAC, and THIRD.FOR, respectively. FORTRAN users should create all three files; MACRO–11 users (who do not use FORTRAN) should create only the first two files.

FIRST.MAC

```
        .TITLE          COMB
        .MCALL          .PRINT
;       I = LEN(A)
        .GLOBL         LEN
LEN:    TST             (R5)+           ;SKIP # OF ARGS
        MOV             @R5,R0          ;GET STRING POINTER
1$:     TSTB            (R0)+           ;FIND END OF STRING
        BNE             1$              ;LOOP UNTIL NULL BYTE
        DEC             R0              ;BACK UP
        SUB             @R5,R0          ;CALC # OF CHARS IN STRING
        RETURN

;       CALL            PRINT(ISTRING)
        .GLOBL          PRINT
PRINT:  MOV             2(R5),R0        ;ADDR OF ISTRNG
        .PRINT                          ;.PRINT
        RETURN                          ;AND RETURN
        .END
```

SECOND.MAC

```
        .TITLE  ITTOUR
;       I=ITTOUR(ICHAR)
;       I=0     CHARACTER HAS BEEN OUTPUT
;        =1     RING BUFFER IS FULL
        .MCALL  .TTOUTR
        .GLOBL  ITTOUR
ITTOUR: MOVB    @2(R5),R0         ;GET CHARACTER
        .TTOUTR                   ;.TTOUTR
        BIC     R0,R0             ;CLEAR ERROR FLAG
        ADC     R0
        RETURN                    ;RETURN
        .END
```

THIRD.FOR

```
C       CALL PUTSTR(LUN,AREA,CC)
        SUBROUTINE PUTSTR(LUN,AREA,CC)
        LOGICAL*1 AREA(250),CC
        DO 10 I=1,250
        IF(AREA(I).EQ.0) GO TO 20
10      CONTINUE
20      LEN=I-1
        IF(CC.NE.0) GO TO 30
        WRITE(LUN,99)(AREA(I),I=1,LEN)
        RETURN
30      WRITE(LUN,99)CC,(AREA(I),I=1,LEN)
99      FORMAT(250A1)
        END
```

The routines in these files represent the services generally provided in a library file.

FIRST.MAC contains two global routines, LEN and PRINT. The LEN routine returns the number of characters in a string. PRINT outputs an ASCII string terminated with a zero byte to the terminal (it is the FORTRAN equivalent of the system macro

.PRINT, used in the demonstration program in Chapter 19. SECOND.MAC contains one global routine, ITTOUR, which transfers a character to the console terminal. THIRD.FOR also contains one global routine, PUTSTR. This routine can be used by only FORTRAN programs and writes a variable-length character string on a specified FORTRAN logical unit.

Once you create these text files, the next step is to convert them from ASCII format to object format. Assemble or compile the text files as appropriate, first assembling FIRST.MAC and obtaining an object module (a listing is not necessary). FORTRAN users who are not familiar with the assembly process can type the MACRO commands as shown.

```
.MACRO FIRST  RET
```

The command creates an object module called FIRST.OBJ on the default data (DK) volume. If errors occur, the assembler prints a message on the terminal, indicating the number of errors encountered during assembly. No errors should occur.

In the same way, assemble SECOND.MAC. Again, no errors should occur.

```
.MACRO SECOND  RET
```

If any errors occur during the assembly operations, you have typed the source files incorrectly. Find and correct the typing errors and reassemble.

If you are a FORTRAN IV or FORTRAN–77 user, continue by compiling THIRD.FOR. The following command calls the correct FORTRAN compiler because of the SET FORTRA F4 or SET FORTRA F77 command you issued in Chapter 17:

```
.FORTRAN THIRD  RET
```

If any errors occur during the compile operation, you have typed the source file incorrectly. Find and correct the typing errors and recompile.

Once you have produced the object modules, you are ready to build the object library file.

21.2.2 Building the Object Library

Use the LIBRARY command in combination with its /CREATE option to construct a library file. You must indicate in the command the name of the library file and the names of the input object modules. Call the library file LIBRA and specify as input the two object modules, FIRST and SECOND. The LIBRARY command assumes that the input modules have the .OBJ file type (unless you indicate otherwise) and automatically assigns .OBJ to the library file.

```
.LIBRARY/CREATE LIBRA FIRST,SECOND  RET
```

Once the library creation operation is complete, obtain a listing of the library file's contents, using the LIBRARY command with its /LIST option. The printer is the assumed output device for the list file, although you may indicate a different output device by adding a 2-letter device mnemonic to the /LIST option that follows.

To print a listing:

```
.LIBRARY/LIST LIBRA  RET
```

To display a listing:

```
.LIBRARY/LIST:TT: LIBRA RET
```

The listing produced shows the library file's current contents. This library has three entry points: LEN and PRINT in the first module and ITTOUR in the second module.

```
RT-11 LIBRARIAN V05.03  TUE 25-OCT-88 10:38:31
DK:LIBRA.OBJ            TUE 25-OCT-88 10:38:13

MODULE          GLOBALS         GLOBALS         GLOBALS

                LEN             PRINT
                ITTOUR
```

21.2.3 Updating the Object Library

Once you have created an object library, you use various LIBRARY command operations to update and maintain the library by adding and deleting modules and globals.

If you created the THIRD.OBJ object module, you can add it to the library file by using the /INSERT option. If you did not create this module, read through this section anyway; the command steps apply to any object module you wish to insert.

```
.LIBRARY/INSERT LIBRA THIRD RET
```

This operation inserts the object module contained in the file THIRD.OBJ, including all its globals, into the library file LIBRA. Obtain a listing of the library contents, using the /LIST option, to verify that the new globals have been added. The listing should look like this:

```
RT-11 LIBRARIAN V05.03  TUE 25-OCT-88 10:42:31
DK:LIBRA.OBJ            TUE 25-OCT-88 10:42:07

MODULE          GLOBALS         GLOBALS         GLOBALS

                LEN             PRINT
                ITTOUR
                PUTSTR
```

This listing shows the complete library file containing the globals from all three modules.

You can remove individual globals by using the /REMOVE option. For example, to remove the global ITTOUR, type:

```
.LIBRARY/REMOVE LIBRA RET
Global? ITTOUR RET
Global? RET
```

The library file's contents now look like this:

```
RT-11 LIBRARIAN V05.03  TUE 25-OCT-88 10:44:06
DK:LIBRA.OBJ            TUE 25-OCT-88 10:43:51

MODULE          GLOBALS         GLOBALS         GLOBALS

                LEN             PRINT
                PUTSTR
```

These are some of the library maintenance operations that you can perform by using the LIBRARY command. Other library operations are available and are explained in the *RT–11 Commands Manual*.

21.3 Summary of Commands for Maintaining Library Files

The following commands are described and illustrated in this chapter. If the meaning of any command is not clear, review the example in this chapter that illustrates the command.

LIBRARY/MACRO

Creates a macro library.

LIBRARY/CREATE

Creates an object library.

LIBRARY/INSERT

Inserts object modules into the object library.

LIBRARY/LIST[:filespec]

Lists the current contents of an object library on the printer; [:filespec] is an optional output file and/or device.

LIBRARY/REMOVE

Remove globals from the object library.

Chapter 22

Debugging a User Program

Debugging is the process of finding and fixing the programming errors that almost every program initially contains. From your experience in Chapters 17, 18, and 19, you already know about programming errors that can prevent a program from working properly when you run it on the system.

Frequently, debugging a program requires more time and persistence than writing the program code requires. Therefore, you should anticipate the debugging process throughout the entire program development cycle. That is, you should follow some common programming practices that help you to make as few programming errors as possible. When errors become apparent during the phases of development, correct them immediately. Test the validity of any algorithms used in your program. Finally, even though the program appears to be working properly, check it thoroughly with test data.

22.1 Avoiding Programming Errors

You can take several steps to decrease the likelihood of introducing errors into your program and to make debugging easier.

- Always use a high-level language if one will suit your programming needs. High-level language programs tend to use fewer statements. Descriptive words and phrases in the language statements make the program logic easier to follow.
- Design the program. The technique of flowcharting the program and then correlating it with the program coding simplifies tracking the program logic and module interrelationships.
- Use modular programming. Create the program as a series of smaller, self-contained subprograms. Debug the program in parts.
- For frequently used functions, maximize the use of subroutines, subprograms, and—in the case of assembly language programs—macros. These help to structure the program and make it easier to alter or to add features that may be required in the future.
- Make use of any software provided by the system, such as library routines and functions. System software has already been debugged and can save you the trouble of re-creating the services.
- Make the general flow of a program proceed down the page. Avoid nonstructured branching and convoluted logic, as these tend to produce programs that are difficult to debug. Finally, use comments liberally throughout the program to show what individual statements or groups of statements do. Use spaces and tabs in the program code to make it easier to read.

Following these preventive steps eliminates many common programming errors and helps to create a programming style. However, even the most careful programmer can overlook a small detail: a typing error during program creation, an undefined label in the code, or some other programming bug. When something is overlooked, debugging becomes necessary.

22.2 When Programming Errors Occur

The three general types of programming errors are syntax, clerical, and logical.

Syntax errors are errors in the physical coding, such as omitting necessary portions of the statement (delimiters, for example), reversing the order of information within the statement, or misspelling keywords or instructions.

Clerical errors are nonsyntax errors in the physical coding, such as mistyped letters or digits in data. Clerical errors may result in valid statements that do not reflect correct programming logic.

Logical errors are errors made in program design.

The translating program (compiler/assembler/interpreter) generally catches syntax errors and flags them as such in the program listing or on the terminal. On the other hand, you must locate clerical and logical errors by reexamining the program code and logic, using one or more debugging techniques.

Some debugging techniques involve insertion of special debugging code in the program. For example, one way to locate logical errors is to write out intermediate results of a program. You can insert WRITE or PRINT statements at strategic points in the program logic to show the intermediate state of values being calculated. When debugging is complete, you can remove these statements or change them to comments.

You may also find it useful to write a special debugging subroutine that writes out values, particularly if the same variables must be examined many times.

Another method for finding logic errors—unit testing—is to break the program into smaller parts and test each part separately with artificial data. After you test all parts individually, you can test routine and module linkage—system testing—to see that all related code fits together properly.

Check the program with test data. A standard method for checking out modules is to write a test program that calls the program with possible options. The test should cause the program to execute all steps in all algorithms. Check programs first with representative data, then with improper data (data that is not in the correct range or size). You should also do volume testing to see that the program works successfully with a representative amount of data.

Each programming language has special debugging aids for examining immediate states. For example, BASIC has a STOP statement that you can insert at strategic points in the program. When the program arrives at a STOP statement, it pauses so that you can use BASIC's immediate mode to examine variables, values, and so on. To continue execution, use an immediate mode GO TO statement pointing to the appropriate line number.

FORTRAN has a special DEBUG statement indicator, a D in the first column of a statement line. Operations in statements marked with a D can perform useful debugging functions, such as printing intermediate results. You can treat such statements as source tests (and thus execute them) or as comments (and thus ignore them), depending on whether you use a special compiler command option. In addition, FORTRAN has a traceback feature that locates the actual program unit and line number of a run-time error. If the program unit is a subroutine or function subprogram, the error handler traces back to the calling program unit and displays the name of that program unit and the line number where the call occurred. This process continues until the calling sequence has been traced back to a specific line number in the main program unit. Finally, FORTRAN has an optional interactive debugger called FDT (FORTRAN Debugging Technique) that can be linked with a user program.

For MACRO–11 users, RT–11 provides two on-line debugging tools: ODT (Octal Debugging Technique), and DBG–11. These tools are provided as part of the RT–11 operating system. ODT is an object program on your system volume that you must link with the .OBJ file you want to debug.

DBG–11 is an interactive debugging tool that lets you monitor program execution from your console terminal. The DBG–11 program behaves somewhat like a device handler; it appears to the monitor to be the SD handler and it handles traps through the BPT vector (memory location 14). Treating DBG–11 as a device handler is done purely for convenience. DBG–11 must interact with the monitor, the user program, and the BPT interrupt vector, and a program that looks like a device handler to the system fits that interface niche quite well. The complete DBG–11 package consists of several .SYS files and a .SAV file.

Both ODT and DBG–11 are used exclusively for debugging assembled MACRO–11 programs. ODT requires that you work exclusively in octal numbers, a tedious process, while DBG–11 allows symbolic input and output. For this reason, this chapter uses DBG–11 for its examples. You will probably want to use ODT only if DBG–11 is unsuitable for some reason, such as memory size limitations; ODT requires less memory than DBG–11.

The next section describes the use of DBG–11 for MACRO–11 users and for those FORTRAN users who combine MACRO–11 and FORTRAN program code.

22.3 Using the Debugging Technique

The MACRO–11 demonstration program in Chapter 19 still contains one error, which you can locate and correct using DBG–11. Once you are in DBG–11, you can use its special debugging commands to control the execution of your assembled machine language program from the console terminal, to examine memory locations, to change their contents, and to stop and continue program execution. Several DBG–11 debugging commands are demonstrated in the process.

DBG–11 is provided as several .SYS files on your system volume; you use one or another based on your debugging needs. For most purposes, you will probably want to use one of the following files:

SDH.SYS

Variant of DBG–11 that does not use monitor's terminal support routines; for use with unmapped monitors (for example, RT11FB).

SDHX.SYS

Variant of DBG–11 that does not use monitor's terminal support routines, for use with mapped monitors (for example, RT11XM).

To use DBG–11, you need to give a different file name to either SDH.SYS or SDHX.SYS, depending on the monitor you are using, so your monitor recognizes the file as a device handler and installs it. The files are distributed with other names so that you can choose the one you want your monitor to install.

If SD is already named correctly on your system and your monitor has installed it as a device handler, you do not need to issue these commands. You can check to see if SD is already installed by issuing the SHOW command with no arguments:

```
.SHOW
```

If SD appears in the list of handlers that is displayed on your terminal, it is installed and you can skip the following steps. If SD is not listed and you are using an unmapped monitor (SB or FB), issue this command:

```
.COPY SY:SDH.SYS SY:SD.SYS
```

If you are using a mapped monitor (XM or ZM), issue the following command:

```
.COPY SY:SDHX.SYS SY:SDX.SYS
```

You should COPY the file, rather than RENAME it, so you keep the original copies for future use with their original file names.

Then, install the handler with this command:

```
.INSTALL SD
```

Before you can use DBG–11 to debug your program, you must add a BPT instruction to your program so DBG–11 gets entered when your program begins to execute. The BPT instruction causes what appears to be an interrupt through the vector at location 14. The SD handler (the DBG–11 program) handles interrrupts through location 14, so when the BPT instruction gets executed, control transfers to DBG–11.

The first line of the sample MACRO–11 program looks like this:

```
EXP::   .PRINT  #MESSAG         ;PRINT INTRODUCTORY TEXT
          .
          .
          .
```

Use the KED editor to change the first lines of the program SUM.MAC to look like this:

```
EXP::   BPT                     ;BREAKPOINT TO ENTER DBG-11
        .PRINT  #MESSAG         ;PRINT INTRODUCTORY TEXT
          .
          .
          .
```

Also, if you corrected the final error (the 'bug') in SUM.MAC in Chapter 19, you must remove the correction to demonstrate DBG-11. To remove the correction, edit line 39 of SUM.MAC to look like this:

```
        ADD     #10+'0,R0       ;MAKE DIGIT ASCII
```

Adding the BPT instruction to the program changes the memory address of all the instructions in the program that follow it, so you must create a new program assembly listing to use while debugging. Assemble your source program and produce a cross-referenced assembly listing as you did in Chapter 19. Remember that SUM.MAC is on your default data (DK) volume:

```
.MACRO SUM/LIST/CROSSREFERENCE RET
```

Print or display the listing.

To print the listing:

```
.PRINT SUM.LST RET
```

To display the listing:

```
.TYPE SUM.LST RET
```

Figure 22–1 shows the program listing (SUM.LST). It has been edited slightly to remove the symbol table and cross-reference listing.

Figure 22–1: Listing of SUM.MAC with Added BPT Instruction

```
SUM.MAC (VERSION PROVIDED) MACRO V05.05  Tuesday  1-Oct-91 13:31  Page 1

     1                                       .TITLE SUM.MAC (VERSION PROVIDED)
     2
     3                                       .MCALL .TTYOUT, .EXIT, .PRINT
     4
     5        000106                         N = 70.         ;NO. OF DIGITS OF 'E' TO CALCULATE
     6                                       ;        'E' = THE SUM OF THE RECIPROCALS OF THE FACTORIALS
     7                               ;       1/0! + 1/1! + 1/2! + 1/3! + 1/4! + 1/5! + ...
     8
     9                                       .DSABL GBL
    10
    11 000000 000003                 EXP:    BPT                     ;BREAKPOINT TO ENTER DBG-11
    12 000002                                .PRINT  #MESSAG         ;PRINT INTRODUCTORY TEXT
    13 000010  012705  000106                MOV     #N,R5           ;NO. OF CHARS OF 'E' TO PRINT
    14 000014  012700  000107        FIRST:  MOV     #N+1,R0         ;NO. OF DIGITS OF ACCURACY
    15 000020  012701  000126'               MOV     #A,R1           ;ADDRESS OF DIGIT VECTOR
    16 000024  006311                SECOND: ASL     @R1             ;DO MULTIPLY BY 10 (DECIMAL)
    17 000026  0111P46                       MOV     @R1,-(SP)       ;SAVE *2
    18 000030  006311                        ASL     R1              ;*4
    19 000032  006311                        ASL     @R1             ;*8
    20 000034  062621                        ADD     (SP)+,(R1)+     ;NOW *10, POINT TO NEXT DIGIT
    21 000036  005300                        DEC     R0              ;AT END OF DIGITS?
```

Figure 22–1 (continued on next page)

Figure 22–1 (Cont.): Listing of SUM.MAC with Added BPT Instruction

```
000040  001371                          BNE     SECOND          ;BRANCH IF NOT
000042  012700  000106                  MOV     #N,R0           ;GO THRU ALL PLACES, DIVIDING
000046  014103                  THIRD:  MOV     -(R1),R3        ;BY THE PLACES INDEX
000050  012702  177777                  MOV     #-1,R2          ;INIT QUOTIENT REGISTER
000054  005202                  FOURTH: INC     R2              ;BUMP QUOTIENT
000056  160003                          SUB     R0,R3           ;SUBTRACT LOOP ISN'T BAD
000060  103375                          BCC     FOURTH          ;NUMERATOR IS ALWAYS < 10*N
000062  060003                          ADD     R0,R3           ;FIX REMAINDER
000064  010311                          MOV     R3,@R1          ;SAVE REMAINDER AS BASIS
                                                                ;FOR NEXT DIGIT
000066  060261  177776                  ADD     R2,-2(R1)       ;GREATEST INTEGER CARRIES
                                                                ;TO GIVE DIGIT
000072  005300                          DEC     R0              ;AT END OF DIGIT VECTOR?
000074  001364                          BNE     THIRD           ;BRANCH IF NOT
000076  014100                          MOV     -(R1),R0        ;GET DIGIT TO OUTPUT
000100  162700  000012          FIFTH:  SUB     #10.,R0         ;FIX THE 2.7 TO .7 SO
                                                                ;THAT IT IS ONLY 1 DIGIT
000104  103375                          BCC     FIFTH           ;(REALLY DIVIDE BY 10)
000106  062700  000070                  ADD     #10+'0,R0       ;MAKE DIGIT ASCII
000112                                  .TTYOUT                 ;OUTPUT THE DIGIT
000116  005011                          CLR     @R1             ;CLEAR NEXT DIGIT LOCATION
000120  005305                          DEC     R5              ;MORE DIGITS TO PRINT?
000122  001334                          BNE     FIRST           ;BRANCH IF YES
000124                                  .EXIT                   ;WE ARE DONE

000126  000107                   A:     .REPT   N+1
                                        .WORD   1               ;INIT VECTOR TO ALL ONES
                                        .ENDR

000344     124     110     105  MESSAG: .ASCII  /THE VALUE OF E IS:/ <15><12> /2./ <200>
000347     040     126     101
000352     114     125     105
000355     040     117     106
000360     040     105     040
000363     111     123     072
000366     015     012     062

SUM.MAC (VERSION PROVIDED) MACRO V05.05  Tuesday  1-Oct-91 13:31  Page 1-1

000371     056     200
                                        .EVEN

        000000'                         .END    EXP
```

Now, link the MACRO–11 program object module (SUM.OBJ) and print or display a link map.

To link and print the link map:

```
.LINK/MAP SUM RET
```

To link and display the link map:

```
.LINK/MAP:TT: SUM RET
```

Figure 22–2 is a copy of the link map for SUM.SAV.

The link map shows that the absolute p-sect (. ABS.) starts at location 0 and is 1000_8 bytes long; the unnamed p-sect, containing the binary code for the program SUM, starts at address 1000_8. Look at the listing for the MACRO–11 program. The program is 374_8 bytes long (do not forget to count word 0 too!) and is assembled starting at relative location 0000.

Figure 22–2: Link Map for SUM.SAV

```
RT-11 LINK  V05.15      Load Map       Thursday 08-Aug-91 13:28  Page 1
SUM   .SAV      Title:  SUM.MA  Ident:

Section  Addr   Size    Global  Value   Global  Value   Global  Value

 . ABS.  000000 001000 = 256.   words  (RW,I,GBL,ABS,OVR)
         001000 000374 = 126.   words  (RW,I,LCL,REL,CON)

Transfer address = 001000, High limit = 001372 = 381.   words
```

Before you run SUM.SAV, you must load the SD handler to enable DBG–11. Enter the following command at the monitor prompt:

```
.LOAD SD

DBG Vxx.xx - RT-11  ( SOFT   SD:   GRH )

.
```

When SD loads, it prints an identifying version number and configuration information; the mnemonics are explained in the *DBG–11 Symbolic Debugger User's Guide*.

If the SD handler was already loaded, this command will have no effect. If that occurs, you should unload then load the SD handler to be sure you start with a clean copy:

```
.UNLOAD SD
.LOAD SD

DBG Vxx.xx - RT-11  ( SOFT   SD:   GRH )

.
```

After you load the SD handler, you can run the test program by using the monitor RUN command. The RUN command brings the load module, called SUM.SAV, into the absolute (physical) memory locations shown in the load map and transfers control to the starting address of the program. The BPT instruction you put in as the first instruction of the program executes, and control transfers to DBG–11:

```
.RUN SUM RET

BE:(U) 1000
DBG>
```

DBG–11 first prints BE:, indicating that it was entered because of a breakpoint instruction (the BPT instruction you put as the first instruction in the program). Next, it prints (U), indicating that the program is running in user mode. Further, it prints 1000, the value of the program counter when DBG–11 was entered. Finally, on a new line, it prints an identifying message on the terminal and an angle bracket indicating that you are in DBG–11 command mode and can enter a DBG–11 command. You can now use DBG–11 to control the execution of your program.[1]

[1] You should read the *DBG–11 Symbolic Debugger User's Guide* before you use DBG–11 with any of your own programs. That manual gives a more detailed explanation of DBG–11 than the one presented here.

DBG–11 commands let you execute the entire program or just portions of it, examine individual locations, examine the contents of the computer's general registers, and change the contents of any locations in your program you wish. If you make a mistake while you are typing any commands, press the DELETE key; DBG–11 deletes the last character typed, allowing you to enter the correct character.

If you have a video terminal that supports VT–100 escape sequences, you can take advantage of DBG–11's graphics register display. To enable the display, enter:

```
DBG>_REG;T
```

If DBG–11 recognizes the type of video terminal you are using, the contents of your computer's general registers should appear at the top of your terminal's video screen. DBG–11 updates the display as the contents of the registers change during the execution of your program.

Since the MACRO–11 program was linked to begin at address 1000, you must add the constant 1000 to each address shown in the assembly listing to obtain the actual address used during loading. DBG–11 can display this symbolically for you if you define a symbol equal to 1000. In fact, you can define up to 20 different symbols, each set to a different value. Thus, if you have linked several modules together, you can set different symbols to the corresponding relocation constants of the individual modules. You can then include the symbol name in your address specification, along with a plus sign (+), and DBG–11 will automatically add the value of that symbol to the address specified in your command. For example, use the DBG–11 colon command (:) to set the symbol EXP equal to 1000:

```
DBG>1000,EXP:
```

Now, to examine the first few locations in the assembly listing, you can type:

```
DBG>EXP/
(U) EXP        /      BPT     LF
(U) EXP+2      /      MOV     #EXP+344,R0     LF
(U) EXP+6      /      EMT     351      RET
DBG>
```

By typing a location address and a slash, you open that location for examination and possible modification. DBG–11 displays the addresses as the symbolic value EXP plus an offset. DBG–11 displays the contents (if possible) as a symbolic MACRO–11 instruction and advances the program counter by one, two, or three words depending on the length of the instruction; for example, the second instruction in the short segment shown above is two words long. A line feed closes that location and opens the next sequential location for examination. Pressing RETURN closes the currently open location and returns to the DBG–11 prompt. In keeping with common MACRO–11 practice, DBG–11 uses the symbolic names R0 through R5, SP, and PC to refer to the general PDP–11 registers 0 through 7.

Program execution was interrupted because of the BPT instruction at the program's start (transfer) address at the tag EXP. You can continue execution from that point now, using the DBG–11 ;P command. The ;P command indicates that you want to continue from the current address:

```
DBG>;P
THE VALUE OF E IS:
2.5/606/606237.2301314.06525/130440275535025.7147773735274474540502.544
.
```

As you discovered in Chapter 19, the program results are incorrect. A period has printed, indicating that you are back in monitor command mode. This MACRO–11 program returns to the monitor after execution. Therefore, to continue using DBG–11, you must run the load module again:

```
.RUN SUM  RET
```

```
BE:(U) EXP
DBG>
```

As you can see, the symbol EXP is still defined. Any symbol definitions that you make with DBG–11, and any DBG–11 characteristics that you set such as enabling the automatic register display, remain in effect until you specifically change them or until you reboot or UNLOAD and LOAD the SD handler. Thus, when you run your program again, you can usually continue where you left off without having to repeat much of the setup work. However, you must reenter any changes you made with DBG–11 to the contents of locations in your program, since a fresh copy of that is brought into memory each time you run it.

To help you find programming errors, DBG–11 provides a breakpoint feature. Whenever a breakpoint is encountered in your program during its execution, control passes to DBG–11. DBG–11 then prints a short message on the terminal, informing you that a breakpoint has occurred and showing the location at which execution has stopped. By typing DBG–11 commands, you can examine and possibly modify variables or data as they exist at the time of the breakpoint. Breakpoints are numbered from 1 to 10_8, so that you can have a total of eight breakpoints set at various instructions in the program at one time.

For example, set breakpoint 1 at location EXP+24 (line 16 in the assembly listing) and breakpoint 2 at location EXP+42 (line 23):

```
DBG>EXP+24,1;B
DBG>EXP+42,2;B
```

Type ;P to proceed:

```
DBG>;P
THE VALUE OF E IS:
2.BPT1>(U) EXP+24      /      ASL    @R1      RET
```

When execution encounters a breakpoint, DBG–11 prints a breakpoint message and number (in this case, `BPT1>`), and the contents of the location. Here, the characters 2. precede the breakpoint message because of the monitor .PRINT request at line 12 of the assembly listing.

Program control has paused at location EXP+24 in the MACRO–11 program. Since the breakpoint was set at the instruction at location EXP+24, that instruction has not yet been executed, but all preceding instructions have. Look in the assembly listing at the instructions that occur there. The instruction at location EXP+20 (line 15) stores the address of the digit vector (at label A) in R1. You can examine the contents of R1 to determine what this address is,[1] then, open the address and, examine its contents and the contents of several addresses following it by using the predefined symbol name R1 and the DBG–11 commands left square bracket ([ ) and at-sign (@). The [command opens a location in symbolic data, rather than symbolic instruction mode, and displays the contents of the location as a symbolic value, if possible. The @ command uses the contents of a location as the address of the location to examine:

```
DBG>R1[
R1            [  EXP+126   @
(U) EXP+126      [         1        LF
(U) EXP+130      [         1        LF
(U) EXP+132      [         1        LF
(U) EXP+134      [         1        RET
```

The @ command uses the contents of the currently open location as an address and opens that location for examination by using the same mode (in this case, [for symbolic data mode) as that used to open the original location. The digit vector A, which begins at location EXP+126, has been initialized to the value 1, the value indicated by the comment at line 49 of the program listing. Since the contents of the locations (1) is less than any defined symbol, the [command displays the values in numeric rather than symbolic form.

If you were to continue program execution now, the branch instruction at line 22 of the assembly listing would cause program control to loop back to the instruction at line 16 where breakpoint 1 is set, again causing execution to pause. Since you want to continue to the next breakpoint (set at location EXP+42), you must first cancel the breakpoint at location EXP+24. To do this, type:

```
DBG>,1;B
```

This removes breakpoint 1 at location EXP+24. The number (in this case 1) indicates which breakpoint is to be removed. Now, continue program execution using the ;P command. Execution progresses through the loop and continues until it reaches the breakpoint set at location EXP+42:

```
DBG>;P
BPT2>(U) EXP+42       / MOV     #106,R0  RET
```

Now, reexamine the contents of several of the digit vector locations beginning at location EXP+126. You could use the [command again, but since you know there is no symbol that corresponds to the contents of the locations, there is little point in using symbolic data mode to look at them. Instead, you can use the vertical bar (|) command to open the locations in numeric mode. The | command always displays values in numeric form:

```
DBG>EXP+126|
(U) EXP+126      |       12       LF
(U) EXP+130      |       12       LF
(U) EXP+132      |       12       LF
(U) EXP+134      |       12       RET
```

[1] If your terminal supports DBG–11's graphics register display and you have enabled it by typing _REG;T, you can also see the contents of R1 displayed there.

The instructions before the breakpoint at location EXP+42 are a multiplication routine. The routine multiplies the vector contents by 10_{10} (12_8), as you have just verified.

You can see how the breakpoint feature is a very useful debugging aid. It lets you execute selected portions of a program and verify that data and variables are being used correctly during execution. You can use the breakpoint feature to locate the error that is in this program.

First, clear all previously set breakpoints (in this case, there is only the one remaining at location EXP+42) by typing the ;B command with no argument:

```
DBG>;B
```

Now, set a breakpoint at location EXP+112 (line 42 of the assembly listing). You want to verify the data that is being passed to the monitor in R0 in the ADD instruction in line 41:

```
DBG>EXP+112,1;B
DBG>;P
BPT1>(U) EXP+112      / EMT     341       RET
```

The instruction EMT 341 is the assembly code generated by the .TTYOUT programmed request. Now, examine the contents of R0:

```
DBG>R0|
R0           |  65      ′5       RET
```

At this point in execution, R0 contains 65_8. The apostrophe command (′) prints the low-order byte of the opened location on the console terminal as an ASCII character, if it is a valid ASCII code. In this case, the number 5 prints. If you look back at the program results printed earlier in this chapter, you can see that 5 is the first digit of the tabulated result (following the message `THE VALUE OF E IS 2.`). You know this result is incorrect because the approximate value of E is 2.718 . . . . And you now also know that the program error is not in the interface to the monitor service used to print the result (.TTYOUT), but that it occurs somewhere before location 110. So, the next step in debugging this program is to set a breakpoint at some earlier point in the program logic and rerun the program. Leave DBG–11 and return to monitor mode by typing 0;G. The monitor prompt (`.`) appears, indicating that you are in monitor mode.

```
DBG>0;G

.
```

Run your program again:

```
.RUN SUM RET
```

```
BE:(U) EXP
DBG>
```

Set a breakpoint at location 100 (line 38 in the assembly listing) and start program execution at the beginning:

```
DBG>EXP+100,0;B
DBG>;P
THE VALUE OF E IS:
2.BPT0>(U) EXP+100       /       SUB      #12,R0   RET
```

Again, examine R0 to verify its contents.

```
DBG>R0|
R0            |   33       RET
```

By following the program logic in the assembly listing, you know that the value in R0 should be 33_8 (2.7, previously multiplied by 10, = 27_{10} = 33_8). So, the value in R0 is correct. From this, you can deduce that the error must occur somewhere between locations 100 and 112. The proper step now is to check the assembly listing, where you find the error at line 41. The decimal point that should follow the 10, identifying it as a decimal 10, is missing. Therefore, the program treats the 10 as an octal 10 or decimal 8, making each digit in the result off by an additive factor of 2. The data in location 110 should be 72, not 70. Since this data has not yet been used, you can change it now with DBG–11 and continue program execution; if it had been used, you would need to restart the program and then change the data. To change the contents of a location, open the location, type in the new contents, and close the location by pressing RETURN:

```
DBG>EXP+110|
(U) EXP+110      |      70      72 RET
```

Now eliminate all breakpoints:

```
DBG>;B
```

Continue program execution; the correct results should print:

```
DBG>;P
718281828459045235360287471352662497757247093699959574966967627724076 6
.
```

The previous part of the message already appeared, so only the fractional part of the result prints now. It appears to be correct.

Changes you make with DBG–11 are temporary. Therefore, you should now use the editor to correct the source program SUM.MAC. You should edit line 41 so that it reads:

```
        ADD     #10.+'0,R0 ;Make digit ASCII
```

You remove the BPT instruction to make a runnable version of the program. The file SUM.MAC is currently stored on the default data (DK) volume. Edit this file to remove the BPT instruction, then reassemble, relink, and rerun it to verify that it is correct. When you have done this, store the updated version of the source file on the storage volume under the same name (SUM.MAC), including the files SUM.OBJ and SUM.SAV.

22.4 Summary of the Steps for Debugging Programs

- Using the KED editor, insert a BPT instruction at the starting address of your program.[1]
- Assemble and link the program you want to test and get a listing and a link map.
- Check to see if the SD handler is already installed and loaded on your system by typing the command SHOW DEVICE at the monitor prompt. Install and load the SD handler, if necessary.
- Run the program you want to debug using the monitor R or RUN command.

The list below summarizes the DBG–11 commands described in this chapter. Only a few of the available debugging commands have been demonstrated. See the *DBG–11 Symbolic Debugger User's Guide* for a description of all DBG–11 commands.

RET

Closes the currently open location.

LF

Closes the currently open location and opens the next sequential location in the current examination mode for examination and possible modification.

addr/

Opens the location indicated (*addr*) in symbolic instruction mode for examination and possible modification. If the contents of the location cannot be displayed in symbolic format, it will appear as numeric.

addr[

Opens the location indicated (*addr*) in symbolic data mode for examination and possible modification. If the contents of the location cannot be displayed in symbolic format, it will appear as numeric.

addr |

Opens the location indicated (*addr*) in numeric mode for examination and possible modification.

@

Uses the contents of the currently open location as an address; closes the currently open location; goes to the new address, and opens it for examination and possible modification.

'

Displays on the console terminal the low-order byte of the currently open location as an ASCII character.

[1] You can also use SIPP to put a BPT instruction into the .SAV image of your program, but editing the source file is less prone to error.

;P

Continues program execution from the current location.

addr,n;B

Sets one of the eight available breakpoints (*n*) at the indicated address (*addr*).

,n;B

Cancels the indicated breakpoint (*n*).

;B

Cancels all breakpoints.

value,name:

Defines a symbolic name (*name*) to be equal to the specified value (*value*).

R0, R1, R2, R3, R4, R5, SP, PC

Predefines symbolic names for the general registers that you can use in DBG–11 commands.

Glossary

Absolute address
The binary number that is assigned as the address of a physical memory storage location.

Absolute section
The portion of a program in which the programmer has specified physical memory locations of data items.

Access time
The interval between the instant at which data is requested from or for a storage device and the instant at which the data actually begins moving to or from the device.

ADC (Analog to Digital Converter)
A circuit that converts analog (voltage) signals to binary data.

Address
A label, name, or number that designates a location in memory where information is stored.

Algorithm
A prescribed set of well-defined rules or processes for the solution of a problem in a finite number of steps.

Alphanumeric
The subset of ASCII characters, including the 26 alphabetic characters and the 10 numeric characters.

ANSI
American National Standards Institute.

Application program (or package)
A program that uses the operating system to run but is not part of that system. Examples include games, business management programs, and graphics programs.

Argument
A variable or constant value supplied with a command that controls the command's action, specifically its location, direction, or range.

Array
An ordered arrangement of subscripted variables.

ASCII
The American Standard Code for Information Interchange; a standard code consisting of 8-bit coded characters for upper- and lowercase letters, numbers, punctuation, and special communication control characters.

Assembler
A program that translates symbolic source code into machine instructions. This program replaces symbolic operation codes with binary operation codes and symbolic addresses with absolute or relocatable addresses.

Assembly language
A symbolic programming language that can be translated directly into machine language instructions and is specific to a given type of central processing unit.

Assembly listing
A listing, produced by an assembler, that shows the symbolic code written by a programmer next to a representation of the actual machine instructions generated.

Asynchronous
The type of operation that is triggered by another event, as opposed to synchronous, or occurring at set time intervals.

Background program
A program that runs at a low priority; that is, when a higher priority (foreground) program is not using system resources.

Backup file
A copy of a file created as a precaution against loss of the primary file.

Backup volume
A volume that stores a copy of your own and/or the operating system files. This gives you a means of replacing the files if they become damaged or accidentally deleted.

Bad blocks
The blocks on a storage volume that are not usable because of some physical damage or flaw. The INITIALIZE/BADBLOCKS command finds bad blocks and calls them FILE.BAD. You can use disks or diskettes that have bad blocks, but you should make a backup copy and discard a volume that has many bad blocks.

Base address
An address used as the basis for computing the value of some other relative address; the address of the first location of a program or data area.

BASIC–PLUS (Beginner’s All-purpose Symbolic Instruction Code)
An interactive, algebraic computer language that combines English words and decimal numbers. It is a widely available, standardized, simple beginner’s language capable of handling industry and business applications.

Batch processing
A processing method in which programs are run consecutively without operator intervention.

Baud
Also called *baud rate*. The speed (in bits per second) at which data is transmitted over a data line.

Binary
The number system with a base of two; used by the internal logic of all digital computers.

Binary code
A code that uses two distinct characters, usually the numbers 0 and 1.

Bit
A binary digit. The smallest unit of information in a binary system of notation. It corresponds to a 1 or 0 and to a 1-digit position in a physical memory word.

Block
A group of physically adjacent words or bytes of a size that is specific to a device (512 contiguous bytes for most disk devices). For input/output operations, the smallest addressable unit on a mass storage device or the basic unit for storing information on a volume. A block in a text file contains about 80 words of text.

Boot (bootstrap)
To start an operating system on a computer. A *hard* (*hardware*) boot is starting the operating system when the computer has been turned off. A *soft* (*software*) boot is bootstrapping an operating system by command after the computer has been turned on.

Boot blocks
The storage area on a volume reserved for the bootstrap routine.

Bootstrap
A technique or routine whose first instructions are enough to start a system of programs that brings an operating system into memory.

BOT (Beginning Of Tape)
A marker that is applied near the beginning of a magtape and identifies the beginning of the magtape’s recordable surface.

Bottom address
The lowest memory address into which a program is loaded.

Breakpoint
A location at which program operation is suspended to allow operator investigation.

Buffer
A storage area used to temporarily hold information being transferred between two devices or between a device and memory. A buffer is often a special register or a designated area of memory.

Bug
A flaw in the design or implementation of a program; a problem that can cause erroneous results.

Bus
Interconnects computer system components to provide communication paths for addresses, data, and control information.

Byte
In a PDP–11 computer system, is the smallest memory-addressable unit of information and is equivalent to eight bits.

Cache
A very fast small memory that can be used in combination with slower, large capacity memories. It contains copies of data recently used by the processor and fetches several bytes of data from memory in anticipation that the processor will access the next sequential series of bytes.

Caching
The process of storing blocks in memory for future use; used to minimize the frequency with which the disk is accessed for read operations.

Call
A transfer from one part of a program to another with the ability to return to the original program at the point of the call.

Calling sequence
A specified arrangement of the instructions and data necessary to pass parameters and control to a given subroutine.

CCL (Concise Command Language)
A syntax for issuing commands that lets you run a program and pass it a CSI command string on a single command line.

Central processing unit (CPU)
A hardware unit of a computer that includes main memory and the registers and circuits that control the interpretation and execution of instructions. The CPU is the control center of the computer. It performs all data processing, such as comparing information, calculating a value, and printing, displaying, or storing information.

Channel
A logical path connecting a user process to a physical device unit. A user process requests the operating system to assign a channel to a device so the process can communicate with that device.

Character
A single letter, numeral, or symbol used to represent information.

Character pointer
The place where the next character typed will be entered. During editing, the character pointer indicates the place in an ASCII text file where the next character typed will be entered into the file.

Clear
To delete the contents of a storage location by replacing the contents, usually with 0s or spaces.

Clock
A device in a computer system that keeps time, counts pulses, measures frequency, or generates regular periodic signals for synchronization.

Code
A system of symbols used to represent data or instructions that are executed by a computer.

Coding
The writing of instructions for a computer, using a system of symbols that is meaningful to a computer, an assembler, a compiler, or a language processor.

Command
A word, mnemonic, or character that, by virtue of its syntax in an input line, causes a computer system to perform a predefined operation.

Command language
The vocabulary used by a program or set of programs that directs the computer system to perform predefined operations.

Command language interpreter
The program that translates a predefined set of commands into instructions that a computer system can interpret.

Command string
A line of input entered into a computer system that generally includes a command, one or more file specifications, and optional qualifiers.

Compile
To produce binary code from the symbolic instructions of a high-level source language.

Compiler
A program that translates a high-level source language into machine instructions.

Completely Virtual Environment
The program environment established when a program is loaded by VBGEXE; typically when a program is run by the V or VRUN commands.

Computer
A machine that can be programmed to execute a set of instructions.

Computer program
A plan or routine for solving a problem on a computer.

Computer system
A data processing system that consists of hardware devices, software programs, and documentation that describes the operation of the system.

Concatenation
The joining of two or more strings of characters to produce a single string.

Conditional assembly
The assembly of certain parts of a symbolic program that occurs only when certain conditions are met during the assembly process.

Configuration
A selection of hardware devices, software routines, or programs that function together.

Console terminal
A keyboard terminal that acts as the primary interface between the computer operator and the computer system. The console terminal is used to initiate and direct system operations by running software on the computer.

Constant
A value that remains the same throughout a distinct operation. (Compare with Variable.)

Context switching
The saving of key registers and other memory areas before switching between jobs with different modes of execution. An example of context switching is the use of foreground/background programming.

Conversational
See Interactive.

CPU
See Central processing unit.

Crash
A hardware crash is the failure of a device to operate; the operation of an entire computer system may be affected. A software crash is the result of an operating system malfunctioning; the system's protection mechanisms may have failed or the software may not have executed correctly.

Create
To open, write data to, and close a file for the first time.

Cross-reference listing
A printed listing that identifies all references in a program to each specific symbol in a program. It includes a list of all the symbols used in a source program and the statements where the symbols are defined or used.

CSI (Command String Interpreter)
That part of RT–11 that accepts a line of ASCII input, usually from the console terminal, and interprets it as a string of input specifications, output specifications, and options for use by a utility program.

Current location counter
A counter kept by an assembler to determine the address assigned to an instruction or constant being assembled.

Cursor
A reference point on the terminal screen that shows where the next entry will appear. The cursor can be a blinking block or a blinking underline character.

Data
A term used to denote facts, numbers, letters, and symbols. Data are the basic elements of information that can be processed by a computer.

Data base
An organized collection of interrelated data items that allows one or more applications to process the items, while disregarding physical storage locations.

Data collection
To bring data from one or more locations to a central location for eventual processing.

DCL (Digital Command Language)
The standard interface between a user and a Digital operating system. Commands in this language are English-like words that instruct the operating system to do specific tasks. The implementations of DCL on Digital operating systems are somewhat different. The implementation of DCL on RT–11 is also called the KMON (keyboard monitor) command language.

Debug
To detect, locate, and correct coding or logic errors in a computer program.

Default
The value of an argument, operand, or field assumed by a program if not specifically supplied by the user.

Default Data Device
The device on which the operating system looks for files to work on or stores files you create when you do not specify a device in a command. This device is also called the default storage device and has the logical name DK.

Define
To assign a value to a variable or constant.

Delimiter
A character that separates, terminates, or organizes elements of a character string, statement, or program.

Device
Any peripheral hardware connected to the processor having to do with input, output, or storage of information rather than the processing of information. Devices include terminals, printers, fixed disks, and diskette drives.

Device control unit
A hardware unit that electronically supervises one or more of the same type of devices. It acts as the link between the computer and the I/O devices.

Device handler
A system software component that lets you use devices, whether they are storage devices, terminals, printers, or whatever other devices you have connected to your computer. This program services and controls the hardware activities of a device connected to your CPU.

Device independence
The ability to program I/O operations independently of the device for which the I/O is intended.

Device name
A unique name that identifies each device unit on a system. It usually consists of a two-letter device mnemonic followed by an optional device unit number and a colon. For example, the common device name for an MSCP disk drive unit 1 is DU1:.

Device unit
One of a set of similar peripheral devices or partitions of a device. An example of a device unit is disk unit 0.

Diagnostics
A set of procedures used to detect and isolate malfunctions and mistakes.

Digit
A character used to represent one of the non-negative integers smaller than the radix (for example, in decimal notation, one of the characters 0 to 9; in octal notation, one of the characters 0 to 7; in binary notation, one of the characters 0 and 1).

Direct access
See Random access.

Directive
Assembler directives are mnemonics in an assembly language source program that are recognized by the assembler as commands to control a specific assembly process.

Directory
A list of files on a volume. This list is in a file in the form of a table containing the names of and pointers to files on a mass storage volume. Directories also include creation date and the number of blocks for each file.

Directory segment
The basic unit, consisting of two blocks, in which an RT–11 directory on a disk or diskette is stored. You can have a maximum of 72 files for each directory segment, and 31 directory segments for each device unit.

Directory-structured
A storage volume is directory structured if there is a directory at the beginning of the volume that contains information (file name, file type, length, and date-of-creation) about all the files on the volume. Such volumes include all initialized disks and diskettes.

Disk device
An auxiliary storage device on which information can be read or written.

Display
A peripheral device used to represent data graphically; normally refers to some type of cathode-ray tube system.

Distribution volume
The volume (or volumes) containing the operating system files as distributed by Digital. From this volume, you can create a working system volume geared to your requirements.

Downtime
The time interval during which a device or system is inoperative.

Drive Unit
The hardware that holds a volume and performs the read/write operations.

Echo
The printing of characters typed by the programmer on an I/O device, such as a terminal.

Edit
To arrange and/or modify the format of data; for example, to insert or delete characters.

Editor
A program for creating or editing text files. KED and KEX are editors. Editors are language independent and will edit anything in character representation.

EEPROM (Electrically Erasable Programmable ROM)
A chip containing a SETUP table you can program for booting an operating system on the PDP–11/84 and the MicroPDP–11.

Effective address
The address used in the execution of a computer instruction.

Emulator
A hardware device that permits a program written for a specific computer system to be run on a different type of computer system.

Entry point
A location in a subroutine to which program control is transferred when the subroutine is called.

EOF (End Of File)
A marker (character or symbol) that indicates the end of a file. The actual marker used depends on the media containing the file.

EOT (End Of Tape)
A warning marker applied near the end of a magtape, which indicates the approach of the end of the magnetic recordable surface. the end of the reel.

EPROM (Erasable Programmable ROM)
A chip in the PDP–11 and the MicroPDP–11 containing CPU boot and diagnostic ROM code.

Error
Any discrepancy between a computed, observed, or measured quantity and the specified value or condition.

Execute
To perform an instruction or run a program on the computer.

Expression
A combination of operands and operators that can be evaluated to a distinct result by a computing system.

Extended device unit
A device unit higher in value than 7.

Extended memory
Physical memory above 28K words.

Extension
The synonym used for file type.

External storage
A storage medium other than main memory for example, a disk or tape.

Factoring
A method of specifying several files at a time by enclosing in parentheses the part of a multiple file specification that differs. This part can be multiple file names, multiple sections of a file name, multiple file types, or multiple sections of a file type; for example: MEMO(1,2,30).TXT.

Field
A specified area of a record used for a particular category of data.

FIFO (First In/First Out)
A data manipulation method in which the first item stored is the first item processed.

File

Information stored on a volume is organized into units called files. A file can contain, for example, a memo, a single program, or a collection of data.

File maintenance
The activity of keeping a mass storage volume and its directory up to date by adding, changing, or deleting files.

File name
The alphanumeric character string assigned by a user to identify a file. It can be read by both an operating system and a user. A file name has a fixed maximum length that is system dependent. (The maximum length in an RT–11 operating system is six characters, the first of which must be alphanumeric. Spaces are not allowed.)

File specification (filespec)
A name that uniquely identifies a file maintained in any operating system. A file specification generally consists of at least three components: a device name, a file name, and a file type.

File-structured device
A device on which data is organized into files. The device usually contains a directory of the files stored on the volume. For example, a disk is a file-structured device, but a printer or terminal is not.

File type
An alphanumeric character string field assigned to a file either by RT–11 or the user, which can be read by both. System-recognizable file types are used to identify files having the same format. If present in a file specification, the file type follows the file name in a file specification, separated from the file name by a period. The maximum length is three characters.

Flag
A variable or register used to record the status of a program or device; the detection of errors by a translating program.

Floating point
A number system in which the position of the radix point is indicated by the exponent part of a number and another part represents the significant digits or fractional portion of a number (for example, 5.39 x 10(8)—Decimal; 137.3 x 8(4)—Octal; 101.10 x 2(13)—Binary.)

Flowchart
A graphical representation for the definition, analysis, or solution of a problem, in which symbols are used to represent operations, data, flow, and equipment.

Foreground
The area in memory designated for use by a high-priority program. The program that gains the use of machine facilities immediately on request.

Format
In RT–11, the FORMAT utility formats a disk or diskette by writing headers on each block in that volume. The header of a block contains data the device controller uses to transfer information to and from that block.

FORTRAN (FORmula TRANslation)
A language designed to permit scientists, engineers, and others to express mathematical operations in a form with which they are familiar. It is used in a variety of applications, including process control, real-time data acquisition, information retrieval, and commercial data processing.

FSM (File Structured Module)
A set of macro routines that implement a file structure for magtape handlers (MT, MM, MS, and MU). FSM routines perform functions (such as read and write) relating to file structures on magtapes.

Full duplex
In communication, pertaining to a simultaneous, 2-way, independent, asynchronous transmission.

Function
An algorithm, accessible by name and contained in the system software, that performs commonly used operations. For example, the square root calculation function.

General register
One of eight 16-bit internal registers in the PDP–11 computer. These are used for temporary storage of data.

Global
A value defined in one program module and used in others. Globals are often referred to as entry points in the module in which they are defined and as externals in the other modules that use them.

Half duplex
Pertaining to a communication system in which 2-way communication is possible, but only one way at a time.

Handler
See Device handler.

Hardware
The physical equipment components of a computer system. The computer, terminal, drive units, and printer are all hardware.

Hardware bootstrap
A bootstrap that is stored in the hardware and need only be activated by specifying the appropriate load and start address.

High-level language
A programming language whose statements are translated into more than one machine language instruction. Examples are BASIC–PLUS and FORTRAN–77.

High memory
See Extended memory.

High-order byte
The most significant byte in a word. The high-order byte occupies bit positions 8 through 15 of a PDP–11 word and is always an odd address.

Image mode
A mode of data transfer in which each byte of data is transferred without any interpretation or data changes.

Indirect address
An address that specifies a storage location containing either a direct (effective) address or another indirect (pointer) address.

Indirect command file
A file containing commands that are processed sequentially.

Indirect control file
A file that can be read and processed by the indirect control (IND) processor. Commands are not necessarily processed sequentially.

Initialize
To set counters, switches, or addresses to starting values at prescribed points in the execution of a program, particularly in preparation for reexecution of a sequence of code. To format a volume in a file-structured format in preparation for use by an operating system.

Input
The data to be processed; the process of transferring data from external storage to internal storage.

Instruction
A coded command that tells the computer what to do and where to find the values it is to work with. A symbolic instruction looks like ordinary language. Symbolic instructions must be changed into machine instructions before they can be executed by the computer.

Interactive processing
A technique of user/system communication in which the operating system immediately acknowledges and acts upon requests entered by the user at a terminal. Compare with batch processing.

Interface
A shared boundary. An interface might be a hardware component to link two devices, or it might be a portion of storage or registers accessed by two or more computer programs.

Internal storage
The storage facilities that form an integral physical part of the computer and that are directly controlled by the computer; for example, the registers of the machine and main memory.

Interpreter
A computer program that translates and executes a source language statement before translating and executing the next statement.

Interrupt
A signal that, when activated, causes a transfer of control to a specific location in memory and breaks the normal flow of control of the routine being executed.

Interrupt-driven
Software that uses the interrupt facility of a computer to handle I/O and responds to user requests. RT–11 is such a system.

Interrupt vector
Two words containing the address of an interrupt service routine and the processor state at which that routine is to execute.

I/O (Input/Output) device
A device attached to a computer that can bring information into the computer or get information out.

Iteration
Repetition of a group of instructions.

Job
A group of data and control statements that does a unit of work. A program and all of its related subroutines, data, and control statements; also, a batch control file.

KMON (Keyboard MONitor)
The part of the monitor that interprets the command language you use to communicate with the computer. See also DCL.

Label
One or more characters used to identify a source language statement or line.

Latency
The time from the initiation of a transfer operation to the beginning of actual transfer; that is, verification plus search time. The delay while waiting for a rotating memory to reach a given location.

Library
A file containing one or more macro definitions or one or more relocatable object modules, which are routines that can be incorporated into other programs.

LIFO (Last In/First Out)
A data manipulation method in which the last item stored is the first item processed; a push-down stack.

Linkage
The code that connects two separately coded routines and passes values and/or control between them.

Linked file
A file whose blocks are joined by references rather than by consecutive locations.

Linker
A program that combines many relocatable object modules into an executable module. It satisfies global references and combines program sections.

Listing
The printed copy generated by a printer or hard-copy terminal.

Load
To store a program or data in memory. To place a volume on a device unit and put the unit on line.

Load map

A table, produced by a linker, that provides information about a load module's characteristics; for example, the transfer address, the global symbol values, and the low and high limits of the relocatable code.

Load module
A program in a format that is ready for loading and executing.

Location
An address in storage or memory where a unit of data or an instruction can be stored.

Locked
Pertaining to routines in memory that presently cannot be swapped or transferred.

Logical device name
An alphanumeric name assigned by the user to represent a physical device. The name can then be used synonymously with the physical device name in all references to the device. Logical device names are used in device-independent systems to enable a program to refer to a logical device name assigned to a physical device at run-time. By default, RT–11 uses two logical device names: SY and DK. SY represents the device from which the operating system is booted, and DK represents the device on which RT–11 stores files by default or looks for files on which to operate. You create a logical device name when issuing the ASSIGN or MOUNT command.

Loop
A sequence of instructions that is executed repeatedly until a terminal condition prevails.

Low memory
Physical memory between 0 and 28K words.

Low-order byte
The least significant byte in a word. The low-order byte occupies bit positions 0 through 7 in a PDP–11 word and is always an even address.

Machine language
The language used by the computer when performing operations.

Macro
An instruction in a source language that is equivalent to a specified sequence of assembler instructions, or a command in a command language that is equivalent to a specified sequence of commands.

Main program

The module of a program that contains the instructions at which program execution begins. The main program usually exercises primary control over the operations performed; it also calls subroutines or subprograms to perform specific functions.

Mask
A combination of bits that is used to manipulate selected portions of any word, character, byte, or register while retaining other parts for use.

Mass storage
Pertaining to a device that can store large amounts of data that are readily accessible to the computer.

Main Memory
A series of storage locations within the computer where the computer processes your data.

Matrix
A rectangular array of elements. Any matrix can be considered an array.

Memory
Any form of data storage, including main memory and mass storage, in which data can be read and written. Memory usually refers to main memory.

Memory image
A replication of the contents of a portion of memory, usually in a file.

Mnemonic
An alphabetic easy-to-remember representation of a function or machine instruction.

Monitor
The master control program that observes, supervises, controls, or verifies the operation of a computer system. The collection of routines that controls the operation of user and system programs, schedules operations, allocates resources, performs I/O, and so forth.

Monitor command
An instruction or command issued directly to a monitor.

Monitor command mode
The state of the operating system — indicated by a period at the left margin — that allows monitor commands to be entered from the terminal.

Mount a volume

To logically associate a physical mass storage medium with a physical device unit. To place a volume on a physical device unit; for example, to place a magtape on a magtape drive and put the drive on line.

MSCP (Mass Storage Control Protocal)
A Digital control protocol for disk devices. Devices controlled by this protocol are called MSCP devices.

Multiprocessing
Simultaneous execution of two or more computer programs by a computer which contains more than one central processor.

Multiprogramming
A processing method in which more than one task is in an executable state at any one time, even with one CPU.

Nondirectory-structured
Refers to a storage volume that is sequential in structure and therefore has no volume directory at its beginning. File information (file name, file type, length, and date-of-creation) is provided with each file on the volume. Such volumes include magtape and cassette.

Non-file-structured device
A device, such as a printer or terminal, in which data cannot be organized as multiple files.

Object code
Relocatable machine language code.

Object module
The primary output of an assembler or compiler, which can be linked with other object modules and loaded into memory as a runnable program. The object module is composed of the relocatable machine language code, relocation information, and the corresponding global symbol table defining the use of symbols within the module.

Object Time System (OTS)
The collection of modules that is called by compiled code to perform utility or supervisory operations; for example, FORTRAN IV Object Time System.

Octal
Pertaining to the number system with a radix of eight; for example, octal 100 is decimal 64.

ODT
The On-line Debugging Technique: an interactive program for finding and correcting errors in programs.

Off-line
Pertaining to equipment or devices not currently under direct control of the computer.

Offset
The difference between a base location and the location of an element related to the base location. The number of locations relative to the base of an array, string, or block.

One's complement
A number formed by interchanging the bit polarities in a binary number; for example, 1s become 0s; 0s become 1s.

On-line
Pertaining to equipment or devices directly connected to and under control of the computer.

Op-code (operation code)
The part of a machine language instruction that identifies the operation the CPU is to perform.

Operand
The data upon which an instruction operates. An operand is usually identified by an address part of an instruction.

Operating system
The collection of programs, including a monitor and system programs, that organizes a central processor and peripheral devices into a working unit for the development and execution of application programs.

Operation
The act specified by a single computer instruction. A program step undertaken or executed by a computer; for example, addition, multiplication, and comparison. The operation is usually specified by the operator part of an instruction.

Operation code
See Op-code.

Operator's console
The set of switches and display lights used by an operator or a programmer to determine the status of the computer system and to start the computer.

Option
An element of a command or command string that enables the user to select alternatives associated with the command. In the RT–11 operating system, an option

consists of a slash character (/) followed by the option name and, optionally, a colon and an option value.

Output
The result of a process; the transferring of data from internal storage to external storage.

Overflow
A condition that occurs when a mathematical operation yields a result whose magnitude is larger than the hardware is capable of handling.

Overlay segment
A section of code treated as a unit that can overlay code already in memory and be overlaid by other overlay segments when called from the root segment or another resident overlay segment.

Overlay structure
A program overlay system consisting of a root segment and optionally one or more overlay segments.

Page
That portion of a text file delimited by form feed characters and generally 50 to 60 lines long. Corresponds approximately to a physical page of a program listing.

Parameter
A variable that is given a constant value for a specific purpose or process.

Parity
A binary digit appended to an array of binary digits to make the sum of all bits always odd or always even. It is used to check the validity of data.

Patch
To modify a routine in a rough or expedient way, usually by modifying the binary code rather than by assembling it again.

PC
See Program counter.

PDP
A Programmable Data Processor.

Peripheral device
Any device distinct from the computer that can provide input and/or accept output from the computer.

Physical device
An I/O or peripheral storage device connected to or associated with a computer.

Physical device name
A 2- or 1-letter mnemnonic, a unit number, and a colon (for example DU1:) by which the operating system identifies a device handler that enables you to interact with a device.

Priority
A number, associated with a task, that determines the order in which the monitor will process the request for service by that task, relative to other tasks requesting service.

Process
A set of related procedures and data that are executed and manipulated by a computer.

Processor status word (PSW)
A register in the PDP–11 that indicates the current priority of the CPU, the condition of the previous operation, and other basic control items.

Program
A set of machine instructions or symbolic statements that performs a task.

Program counter (PC)
A register used by the central processor unit to record the addresses of the instructions to be executed. The PC (register 7 of the eight general registers) always contains the address of the next instruction to be executed, or the second or third word of the current instruction.

Program development
The process of writing, entering, translating, and debugging source programs.

Programmed request
A set of instructions (available only to programs) that is used to invoke a monitor service.

Program section
A named, contiguous unit of code (instructions or data) that is considered as an entity and that can be relocated separately without destroying the logic of the program.

Prompt
Text displayed that requests information from the user. Also, a symbol displayed by the operating system or a program indicating the computer is ready to receive commands. The RT–11 monitor prompt is a period (.).

Protocol
A formal set of conventions governing the format and relative timing of information exchange between two communicating processes.

Pseudo device handler
A piece of code that looks like a device handler to the monitor but is not one; for example, the Symbolic Debugger. A pseudo device handler is similar to a device handler in that it can accept such monitor commands as INSTALL, LOAD, and SHOW. It is dissimilar to a device handler in that it does not handle devices and does not do any I/O.

PSW
See Processor status word.

Queue
Any dynamic list of items; for example, items waiting to be scheduled or processed according to system- or user-assigned priorities.

Radix
The base of a number system; the number of digit symbols required by a number system.

Radix–50
An octal numbering system that represents 50 octal (40 decimal) characters. Radix–50 uses a character-packing algorithm that permits the storage of 3 characters in a 16-bit word (rather than 2, which the standard octal code allows). Radix–50 is used in various ways, such as to store file names in RT–11 directories.

RAM (Random-Access Memory)
Memory that is accessed so that the next location from which data is to be obtained is not dependent on the location of the previously obtained data.

Random access
Access to data in which the next location from which data is to be obtained is not dependent on the location of the previously obtained data. Contrast Sequential access.

Read-only memory (ROM)
Memory whose contents are not alterable by computer instructions.

Real-time processing
The computation performed while a related or controlled physical activity is occurring. The results of the computation can be used for guiding the process.

Record
A collection of related items of data treated as a unit; for example, a line of source code or a person's name, rank, and serial number.

Reentrant
Pertaining to a program composed of a shareable segment of pure code and a nonshareable segment that is the data area.

Register
A temporary storage location in the hardware logic section of the CPU (rather than in main memory). Registers are divided into three types: general, processor, and device registers. See also General register.

Relative address
The number that specifies the difference between the actual address and a base address.

Relocate
In programming, to move a routine from one portion of storage to another and to adjust the necessary address references so that the routine, in its new location, can be executed.

Resident
Pertaining to data or instructions that are permanently located in main memory.

Resource
Any means available to users, such as computational power, programs, data files, storage capacity, or a combination of these.

Restart
To resume execution of a program.

RMON (Resident MONitor)
The part of the monitor that provides the console terminal service and central program code necessary for both system and user programs. It always remains in computer memory, regardless of system operations, and is the basic source of control in the computer.

ROM
See Read-only memory.

Root segment
The segment of an overlay structure that, when loaded, remains resident in memory during the execution of a program.

Routine
A set of instructions arranged in proper sequence to cause a computer to perform a desired operation.

Run
A single, continuous execution of a program.

Run-group commands
Those commands that can run a background job: RUN, R, VRUN, and V.

Scroll
A feature of moving upward or downward the lines of information displayed on a terminal screen.

Sector
A physical portion of a disk mass storage device.

Segment
See Overlay segment.

Sequential access
A method of data access in which the next location from which data is to be obtained immediately follows the location of the previously obtained data. Contrast Random access.

Software
The programs that control a computer. Software can be divided into five main categories: monitors, device handlers, utilities, language processors, and application programs.

Software bootstrap
A bootstrap that is activated by loading the instructions of the bootstrap and specifying the appropriate load and start address.

Source code
Text, usually in the form of an ASCII format file, that represents a program. Such a file can be processed by an appropriate system program.

Source language
The system of symbols and syntax used to describe a procedure that a computer can execute.

Spooling
The technique by which I/O with slow devices is placed on mass storage devices to await processing.

Startup Command File

An indirect command file that is automatically processed when you start RT–11. The startup files STRTSB.COM, STRTFB.COM, STRTXM.COM, and STRTZM.COM are provided to run under the SB, FB, XM, and ZM monitors respectively.

Storage

Pertaining to a device into which data can be entered, in which it can be held, and from which it can be retrieved at a later time.

String

A connected sequence of entities, such as a line of characters.

Subprogram

A program or a sequence of instructions that can be called to perform the same task (though perhaps on different data) at different points in a program, or in different programs.

Subroutine

See Subprogram.

Subscript

A numeric valued expression or expression element that is appended to a variable name to uniquely identify specific elements of an array. Subscripts are enclosed in parentheses. There is a subscript for each dimension of an array. Multiple subscripts must be separated by commas. For example, a 2-dimensional subscript might be (2,5).

Supervisory programs

Computer programs that have the primary function of scheduling, allocating, and controlling system resources.

Swapping

The process of moving data from memory to a mass storage device, temporarily using the empty memory area for another purpose, and then restoring the original data to memory.

Synchronous

Pertaining to related events where all changes occur simultaneously or in definite timed intervals.

Syntax

The structure of expressions in a language and the rules governing the structure of a language.

System program
A program that performs system-level functions. A program that is part of the basic operating system (for example, a system utility program) is a system program.

System support handler
A form of pseudohandler used by RT–11 to make certain necessary connections with a particular processor's hardware. Examples include PI for the CTI Bus-based processors and UB for UNIBUS processors that support UNIBUS Mapping Registers. System support handlers are loaded into memory before the system device handler.

System volume
The volume on which the operating system is stored.

Table
A collection of data in a well-defined list.

Terminal
An I/O device that includes a keyboard and video screen (or other display mechanism). In PDP–11 systems, a terminal is used as the primary communication device between a computer system and a user.

Time sharing
A method of allocating resources to multiple users so that the computer processes a number of programs concurrently.

TMSCP (Tape Mass Storage Control Protocol)
A Digital control protocol for tapes that is similar to MSCP, the Digital control protocol for devices.

Toggle
To use switches on the computer operator's console to enter data into the computer memory.

Translate
To convert from one language to another.

Trap
A conditional jump to a known memory location performed automatically by hardware as a side effect of executing a processor instruction. The address location from which the jump occurs is recorded. It is distinguished from an interrupt, which is caused by an external event.

Truncation
The reduction of precision by ignoring one or more of the least significant digits; for example, 3.141597 truncated to four decimal digits is 3.141.

Turnkey
Pertaining to a computer system sold in a ready-to-use state.

Two’s complement
A number used to represent the negative of a given value in many computers. This number is formed from the given binary value by changing all 1s to 0s and all 0s to 1s and then adding 1.

UCF (User Command First)
An RT–11 feature that lets you write your own keyboard monitor processing utility to intercept a command line before KMON attempts DCL, CCL, or UCL parsing.

UCL (User Command Linkage)
An RT–11 feature that lets you define your own commands.

Underflow
A condition that occurs when a mathematical operation yields a result whose magnitude is smaller than the smallest amount the hardware can handle.

UNIBUS mapping register (UMR)
A set of hardware registers in certain PDP–11 processors that direct memory access between an 18-bit bus and 22-bit memory.

User program
An application program.

USR (User Service Routines)
The part of the monitor that enables you to access information stored on disks and tapes.

Utility program
Any general-purpose program included in an operating system to perform common functions.

Variable
The symbolic representation of a logical storage location that can contain a value that changes during a processing operation.

Vector
A consecutive list of associated data.

Volume
A mass storage medium that can be used for file-structured data storage.

Wildcard
A valid substitute for characters in a file specification. Used to perform operations on multiple files. Can be asterisks to represent entire file names or file types, or percent signs to represent single characters in file names or file types.

Wildcard operation
A shorthand method of referring to all files with a specific characteristic in their name.

Word
Sixteen binary digits treated as a unit in PDP–11 computer memory.

Working system volume
The volume containing the operating system files adapted to your needs.

Write-enabled
The condition of a volume that allows information to be written on it.

Write-protected
The condition of a volume that protects the volume against information being written on it.