Tailoring RT-11

System Management and Programming Facilities

Simon Clinch
Stephen Peters
Kevin Small
Anne Summerfield

The RT-11 Technical User's Series

Tailoring RT–11

System Management and Programming Facilities

Tailoring RT–11

System Management and Programming Facilities

Simon Clinch
Stephen Peters
Kevin Small
Anne Summerfield

digital
DECbooks

Designed by Virginia J. Mason
Printed in the United States of America

10 9 8 7 6 5 4 3 2

Documentation number EY-00024-DP
ISBN 0-932376-34-7

Library of Congress Cataloging in Publication Data
Main entry under title:

Tailoring RT–11.

Includes bibliographical references and index.
1. RT–11 (Computer operating system) 2. Electronic digital computers—Programming. I. Clinch, Simon, 1959– II. Title: Tailoring R.T.–11.
QA76.6.T34 1984 001.64'2 84-9405
ISBN 0-932376-34-7

Contents

Acknowledgment

We would like to thank all those who contributed to this publication. In particular, we are grateful to Martin Gentry, Dennis Jensen, and Bernard Volz for reviewing the material in this book. The staff at Digital Press deserve special commendation for their invaluable assistance.

Introduction

Tailoring RT–11: System Management and Programming Facilities describes the RT–11 tools you will need to manage your system and use it efficiently. The first portion of the book examines system management functions: installing the system, performing system generation, maintaining the system volume, controlling the internal allocation of system resources. The latter part of the book focuses on system programming functions: using the scheduler, writing a device driver, using memory management directives.

Chapter 1, "Volume Maintenance," explains how to initialize and maintain system volumes by using DCL commands. Chapter 2, "Installation and System Generation," discusses how to install RT–11 by using the automatic installation procedure and how to customize the system with SYSGEN. Chapter 3, "System Maintenance," describes how to update system software and how to select and implement customizations. Chapter 4, "The Queue Package and the Error Logger," examines how to run the system jobs QUEUE and ERRLOG.

Chapter 5, "PDP–11 Architecture," describes how to use the device registers, the memory management unit, and the interrupt system. Chapter 6, "Extended Memory Management," examines how to access memory above the 28-Kword boundary. Chapter 7, "Device Communication," describes how to communicate with devices that are not sup-

ported by the standard RT–11 operating system. Chapter 8, "Writing an Interrupt Service Routine," explains how to write an interrupt service routine and how to use interrupt vectors. Chapter 9, "Writing a Simple Device Handler," examines how to write, assemble, link, install, and debug a device handler. Chapter 10, "Additional Features for Device Handlers," describes how to add optional features to a device handler.

Resources

In order to do the practice exercises, you will need access to an RT–11 system that has been bootstrapped.

Although every effort has been made to make *Tailoring RT–11* a self-contained volume, you may need to refer to the following manuals from the RT–11 documentation set for additional information:

- *RT–11 Automatic Installation Booklet*
- *RT–11 Installation Guide*
- *RT–11 Programmer's Reference Manual*
- *RT–11 Software Product Description*
- *RT–11 Software Support Manual*
- *RT–11 System Generation Guide*
- *RT–11 System Message Manual*
- *RT–11 System User's Guide*
- *RT–11 System Utilities Guide*

The documentation to which we refer throughout the text is written for RT–11 version 5.0. We also used a computer system equipped with RT–11 version 5.0 to generate the programs in our examples and practices. If you own a newer version of RT–11, you may also need a copy of the latest *System Release Notes* to determine the difference between

your system and the one described here. For a directory of documentation products, write:

Digital Equipment Corporation
Circulation Department, MK01/W83
Continental Boulevard
Merrimack, NH 03054

For additional information on RT–11 conventions, you may refer to the publications listed below:

- *Working with RT–11* (Digital Press, 1983)
- *Programming with RT–11, Volume 1 Program Development Facilities* (Digital Press, 1984)
- *Programming with RT–11, Volume 2 Callable System Facilities* (Digital Press, 1984)
- *PDP–11 Processor Handbook*

Notations

The following symbols are used in this book to represent specific elements:

⟨KEY⟩	indicates keyboard and keypad keys, their functions, or key combinations
COMMANDS	(uppercase) indicates input
Prompts	(upper and lowercase) indicates computer output
[ ]	indicates parts of a command that are optional (the brackets are not part of the command string)

An example box acts as a window that shows either the interaction between the user and the computer or a portion of the codes in a program. If the code in an example does not have a label, blank spaces have not been included for the label field.

Tailoring RT–11

System Management and Programming Facilities

1

1

Volume Maintenance

To make efficient use of physical storage media such as disks and magnetic tapes, you will need to know how file storage is organized in RT–11. This chapter discusses how to initialize volumes for storage and make backup copies of important volumes. We will also discuss bad blocks and how to use the SQUEEZE command to improve disk organization. The commands discussed in this chapter are: INITIALIZE, BACKUP, FORMAT, COPY, and SQUEEZE.

When you have completed this chapter, you will be able to describe the RT–11 disk and file structure; create a bootable system volume; recover data from a volume containing bad blocks; prepare new volumes for use as system or data volumes; and make backup copies of important volumes, including the system disk.

Volume Structure

RT–11 supports two types of storage devices: random access devices such as disks, diskettes, or DECtape, and sequential access devices such as magnetic tapes or cassettes. Files are stored according to the type of storage device used. On random access devices, the storage area is divided into 256-word blocks. A disk is known as a block-replaceable device because each block can be independently addressed, read from, or written to. Blocks are numbered starting at block 0. Figure 1 shows the structure of an RT–11 disk.

RT–11 reserves blocks 0 to 5 for system use. Block 0 contains the boot block. The PDP–11 bootstrap loader always reads in the contents of the boot block and then executes this code. If the storage device is a system volume, the boot block will read in and execute the code in blocks 2 through 5. System volumes contain a bootstrap program in blocks 2 through 5, but data volumes do not. If the storage volume is a data volume, then an error message will be produced if you try to boot from that volume.

Block 1 is known as the home block and contains system data such as volume identification and owner name.

The device directory always starts at block 6, and user data starts at the first block following the last directory block. On sequential access devices there is no directory. Tape files are preceded by file headers. Files are stored contiguously from the start of the tape, with the rest of the tape left blank. See figure 2 for the structure of an RT–11 magnetic tape.

Bad Blocks

If a hardware error is returned when you try to read from or write to a block, the block is known as a bad block. Bad blocks are caused by media failure, drive failure, or electromagnetic noise. They have different effects in different areas of the volume. For example, bad blocks in the directory area may mean the loss of all files covered by that segment of the directory, or they may mean a very long recov-

Figure 1.
Structure of an RT–11 Disk

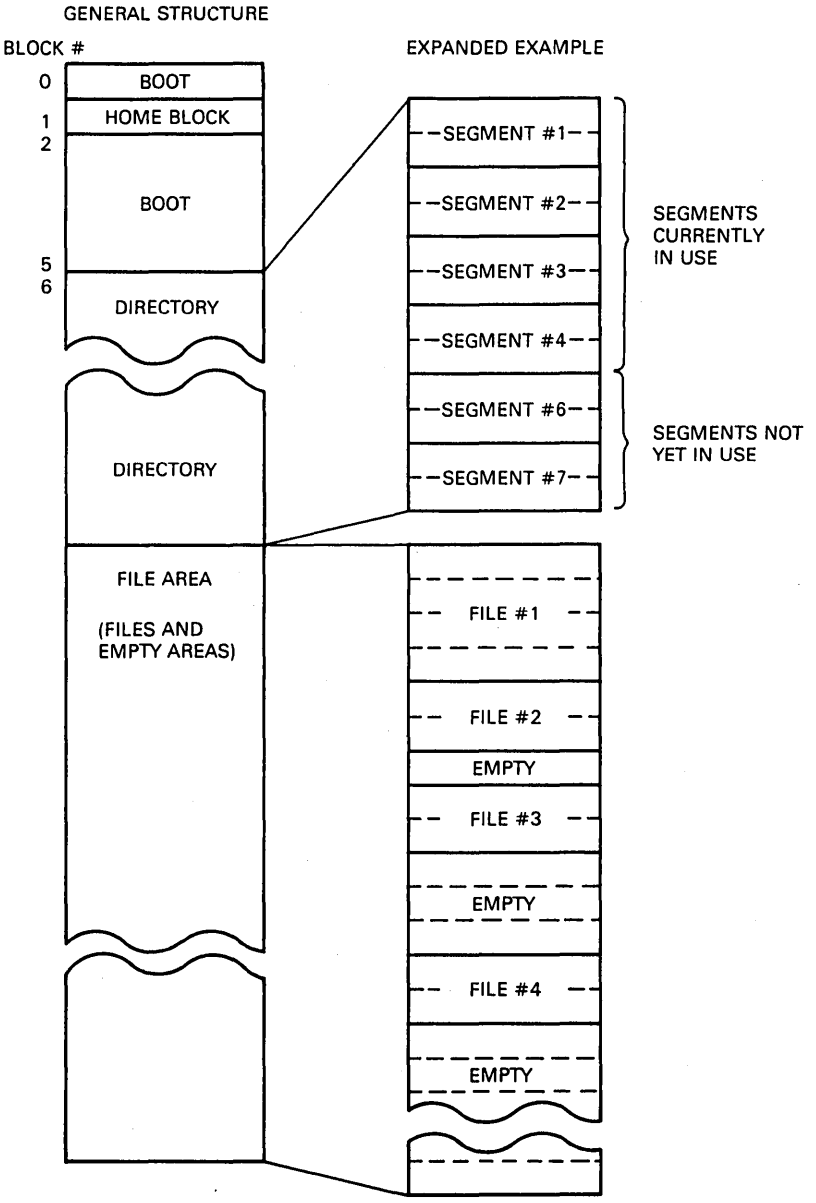
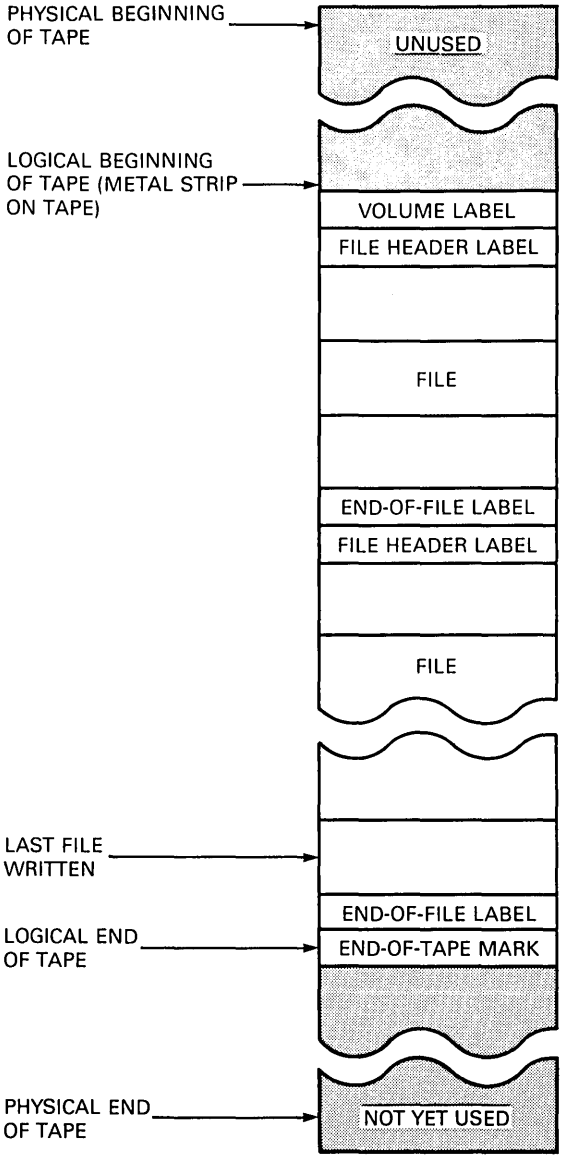


Figure 2.
Structure of an RT–11 Magnetic Tape



ery procedure. Bad blocks in the home or boot block area mean that a volume can no longer be used as a system volume. In the file storage area, they may mean that one file is lost.

The way you handle bad blocks depends on where they are located. You should use the block numbers to determine whether the bad blocks are in the home or boot block, the first segment of the directory, a file, or the empty area of the volume.

To determine whether there are bad blocks and what their locations are, use the DIR command with the /BAD and /FILES options. A sample directory listing showing positions of bad blocks is shown in figure 3.

Initializing Volumes

The first thing you must do when you receive a formatted physical storage volume from the manufacturer is to initialize the volume. To do this, write on the volume the information needed to identify it and the files the volume will contain. If you receive an unformatted volume—for example, when you get a new RK05 disk—you will need to format the device before initializing it. Look up the FORMAT command in the *RT–11 System User's Guide*.

The INITIALIZE command clears the directory of a directory-structured device. Initialization writes the home

Figure 3.
Directory Listing Showing Positions of Bad Blocks

```
.DIR/BAD/FILES
    Block          Type          File          Block
044632  18842.  Replaced     BASKIT.DSK  006335   3293.
077020  32272.  Replaced     < UNUSED >  000522    338.
077122  32338.  Replaced     < UNUSED >  000624    404.
077224  32404.  Replaced     < UNUSED >  000726    470.
107443  36643.  Replaced     < UNUSED >  011145   4709.
111047  37415.  Replaced     < UNUSED >  012551   5481.
?DUP-W-Bad blocks detected 6.
```

block, clears the boot blocks (except the block 0 message telling you there is no bootstrap on the volume), and sets up the directory. Before any files are created, the device directory contains a single active segment with a single entry that describes the available area of the disk, which extends from the end of the directory to the end of the storage volume. The number of segments in a directory varies and is controlled with the /SEGMENTS option. Larger directories allow for more file entries per volume.

Initializing a tape writes the volume label and places an end-of-tape indicator directly after it. This is the way an empty volume is represented on a non-directory-structured device. The boot for a tape is written as the first file, using the /FILES option.

The /VOLUMEID option allows you to specify the volume identification and owner name of the volume. To change the identification of a block-replaceable volume without initializing it, use the /VOLUME:ONLY option.

You may also specify the number of blocks used as directory segments on block-replaceable devices, using the /SEGMENTS:n option of the INITIALIZE command. Table 1 shows the default number of segments in the directory for standard RT–11 devices. Remember that one directory segment is made up of two blocks of storage and can contain a maximum of 72 entries.

During the initialization phase, you may examine the volume and protect it from bad blocks by using the /BADBLOCKS option of the INITIALIZE command. Using this option causes the system to write .BAD files over the bad blocks. This ensures that the system will not try to access the bad blocks during routine operations. Any file with the .BAD file type is generally treated as immovable. Available space is segmented into noncontiguous areas as a result of .BAD files. If the system finds a bad block in either the boot block or the volume directory, it prints an error message saying that the volume cannot be used.

If you use the /BADBLOCKS:RET option when you initialize a used volume, then initialization will keep all files with a .BAD file type that are found on the volume. This option does not scan for new bad blocks and thus, saves time during initialization. However, it does not give up-to-date information about the bad blocks on a volume, so you

Table 1.
Default Directory Sizes

Device	Number (decimal) of Segments in Directory
DD	1
DL (RL01)	16
DL (RL02)	31
DM	31
DU (disk)	31
DU (diskette)	1
DX	1
DY (single-density)	1
DY (double-density)	4
RK	16

must make a choice between time and accuracy. The bad block scan requires accessing every block; so the larger the disk, the longer it will take.

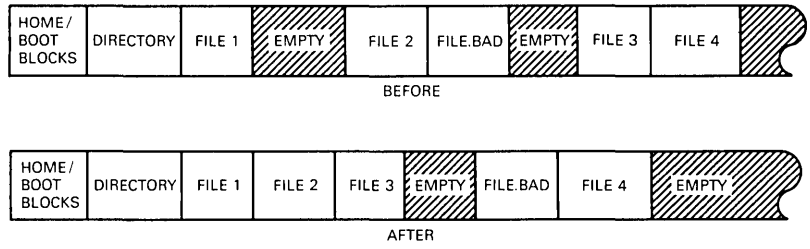
If you are using an RK06, RK07, RL01, or RL02 disk, or block-replaceable devices, you may use the /REPLACE option to scan for bad blocks. If the system finds any bad blocks, it creates a replacement table that maps a spare good block for each bad one so that the disk appears to have only good blocks. Use of a replacement table, however, makes response time slower.

The software always considers devices such as the RD51, RC25, and RA80 to be error-free. The devices cover bad blocks through automatic revectoring to spare tracks. Should bad blocks occur, .BAD files can be used to avoid an area until the device can be updated.

Squeezing Volumes

RT–11 files are stored on contiguous blocks so that deleting files leaves empty areas on the volume. These areas may be reused, but large numbers of creations and deletions will

Figure 4.
Using SQUEEZE on a Volume with .BAD Files



leave many small areas that are not reusable because of their size. SQUEEZE consolidates empty areas by moving all the files to form one large block of contiguous data. This operation is modified in the case of bad blocks covered by .BAD files. In order to prevent you from exposing bad blocks on a disk, .BAD files are not moved in the SQUEEZE operation. Files are inserted before and after the .BAD files until the empty space between the last file moved and the bad block is smaller than the next file to be moved. Figure 4 shows the effects of performing a SQUEEZE operation on a volume with .BAD files. In the next section, we will discuss using SQUEEZE to create backup copies of volumes.

Backup Operations

Backup versions of files and volumes are used to make sure that valuable information is not lost. Two types of backup operations will be described: archiving and creating working copies.

Archiving

You may use COPY/DEVICE to make an exact copy of a volume on an equivalent volume. The original file organization is retained and the boot and the volume ID are copied.

In general, this command is quick and appropriate as long as bad blocks are not a factor on either volume.

Another way of archiving is to use the BACKUP command. The BACKUP command uses the BUP utility program to copy the contents of a large file or volume to a set of smaller volumes for storage. The volumes generated by BACKUP cannot be used in this fragmented form. BACKUP is only used as a means of storing information for system and data archives. BACKUP/RESTORE copies files or the contents of volumes back to their original form for active use.

Creating Working Copies of the System Disk

You will need to copy the system disk so that you have more than one copy of the working system. By default, the COPY command does not transfer system files and the contents of blocks 0 through 5. If you want to create a bootable system volume, you must copy the .SYS files as well as the data files and then use the COPY/BOOT command to set up the boot blocks.

You may use SQUEEZE to compress a volume onto a new backup volume. Using SQUEEZE in this way is similar in effect to a COPY/SYSTEM operation, except that SQUEEZE compresses the directory and COPY does not. COPY/SYSTEM/QUERY should be used if the new volume is to be tailored differently from the input volume. The following example describes how to copy a bootable volume from DL0: to DL1:.

1. Initialize the new volume using the INITIALIZE command. The format of the command is:

```
.INI/REP/VOL DL1:
```

2. Copy the files to the new volume using the COPY/SYSTEM or SQUEEZE/OUTPUT commands. The formats for these commands are:

```
.COPY/SYS DL0:*.* DL1:*.*
.SQU/OUT:DL1: DL0:
```

3. Write the bootstrap program to the volume using COPY/BOOT:. This assumes the Foreground/Background monitor is to be booted:

```
.COPY/BOOT DL1:RT11FB.SYS DL1:
```

4. Boot the new system:

```
.BOOT DL1:
```

It is better to run the new copy of a system volume and store the old one because this tests whether the copy was successful (you know that the old version of the volume ran correctly) and because the new system was squeezed, so the files are contiguous and access is, therefore, more efficient.

Data Recovery from Storage Volumes

Before using any of the data recovery procedures described here, try to find out as much as you can about the cause of the error. For example, check that the drives are aligned compatibly before you go through a possibly long recovery procedure. Remember that data is lost most often because of bad blocks or user errors resulting in corrupted files or a corrupted directory.

Recovering a File Deleted in Error

The simplest type of recovery procedure is restoring a file deleted in error. When you delete a file, its status in the directory is changed, but the contents of the file stay on the storage volume until they are overwritten. To recover the file, you must restore its status to that of a permanent file. To do this, find the starting block and length of the file, using the command DIR/DELETED/BLOCKS/FULL. Restore the directory entry with the CREATE/ALLOC:size/START:x command. Here "size" represents the file length and "x" represents the starting block position. Check the extent of the newly created file to be sure it contains no unwanted material.

Recovering a Storage Volume Initialized in Error

To recover a storage volume that was initialized in error, use the INITIALIZE/RESTORE command. This command will only be effective if the volume was not written to between initialization and the recovery procedure. For example, any bad-block scan during initialization will prevent a recovery.

Recovering from Bad Blocks

If you find that blocks on a volume you have used have become bad, you should try selective copying to save any valuable information stored on that volume. If the volume was being used as a scratch volume or did not contain any important files, then reformat the volume and initialize it with a bad-block scan.

Bad Blocks in the File Area

Try to copy the file containing a bad block or blocks to another location on a volume by using the COPY/IGNORE command. This copies one block at a time, so there is less chance of an error than with multiple-block transfers. I/O errors are reported on the terminal, but copying continues. When you have copied the file, delete the original so that the bad block is in an empty area of the volume. To avoid the bad block prior to an initialization, use CREATE to create a .BAD file over the block.

Bad Blocks in Directories

Data recovery becomes more difficult when the directory has been corrupted because the system accesses files through the directory. If you access a bad block in a directory, an error message:

```
?MON-F-Directory I/O error
```

will be displayed. You also get this message if you access a volume that has not been initialized in RT–11 format.

The procedure for recovering files when the directory is corrupted involves patching around the bad blocks in the directory segments. You will then be able to copy the files to another storage medium. Refer to chapter 9 of the *RT–11 Software Support Manual*. All files are block contiguous, so if you have a current listing of DIR/BLOCKS, then you know where the files reside physically on the volume.

Bad Blocks Caused by Drive Failure or Electromagnetic Noise

If there is no damage to the storage medium itself, you can correct bad blocks in the file area by rewriting the header information. Check that the disk drive is correctly aligned before you try to perform this operation.

First, take a directory listing of bad files to check whether they are present. If so, note their position and copy all files from the bad volume to a good one. Use COPY/DEVICE if possible or COPY/IGNORE if errors are reported during the copying operation. Reinitialize the bad volume and copy the files back. If bad blocks occur again, compare their positions to those first recorded. If errors occur in the same positions, you should assume that the volume has physical damage. Cover the bad blocks or replace the volume.

Bad Blocks Caused by a Damaged Medium

If bad blocks cannot be removed by rewriting or reformatting, they must be isolated in some way so that they will not be accessed by normal file operations. There are two ways to protect the volume from this class of bad blocks. However, these methods should be used only when necessary because they result in a decrease in access speed and/or available disk space.

The first method is replacement (available only on RK06, RK07, RL01, or RL02 devices). With RK and RL volumes, a bad block is logically replaced by a good one, using a replacement table stored in block 1 and spare blocks reserved at the end of the disk. This removes bad blocks from all operations. Use the command line INIT/REPLACE dev: on the volume to scan for bad blocks and set up the replacement table. Use the /REPLACE:RETAIN option in future initializations to retain the replacements made previously. The number of blocks that may be replaced is limited physically to the replacement table space and available spare blocks.

The second method is covering, which is available on all devices, but file storage is segmented. To cover a bad block, create a file with the file type .BAD at the location of the bad block. The INIT/BADBLOCKS command covers all bad blocks when the volume is initialized. To cover without initializing, use the CREATE command to create a file with the type .BAD at the position of a bad block. You should not use COPY/DEVICE on any volume that has covered bad blocks because this command ignores the file structure and thus, will not pass over .BAD files.

References

RT–11 Software Support Manual. Chapter 9 describes how to restore files when the device directory becomes corrupted. It also offers examples of file recovery. Table 9-1 and figure 9-2 describe the contents of the home block.

RT–11 System User's Guide.

2

2

Installation and System Generation

The RT–11 software distribution kit, provided on magnetic tape, diskette, or cartridge disk, contains all of the software for the RT–11 system: the monitors, device handlers, utility programs. It also contains a number of other useful files such as help text and notices.

You use the distribution kit as a base from which to build the system volumes for your working system. The process of copying the kit and selecting files to transfer to a working volume is called installation. Certain system configurations use an automatic procedure to install the system for you.

The distributed monitors may be changed according to the specific needs of your application and computer configuration. Some of the changes may be made by using monitor commands or patching specific files, but many of the changes are implemented by using the system generation procedure.

This chapter discusses installation and how to use the automatic installation procedures. It also describes the procedures for system generation and the options available.

Distribution Kit

The RT–11 software, distributed on one or more volumes, includes the monitors, device handlers, utility programs, source files for SYSGEN, and command text files:

- Monitors include ready-to-run versions of the SJ, FB, and XM monitors, the Base Line (BL) monitor (a simplified version of the SJ monitor), and an Automatic Installation monitor (based on the FB monitor).
- Device handlers for the SJ and FB monitors are named dd.SYS; "dd" is the two-character permanent device name. The XM versions of these device handlers are named ddX.SYS.
- Utility programs include editors, libraries, and subroutines for program development. File volume and system maintenance utilities as well as other programs not supported by Digital are distributed with RT–11.
- Source files for SYSGEN include all files needed to generate customized monitors and device drivers.
- Command text files, the remaining files on the distribution kit, include indirect control files for installing and generating a working system, demonstration programs to verify installation, and text files with release notes.

Refer to table 2–1 of the *RT–11 Installation Guide* and appendix A of the *RT–11 System Release Notes* for a complete list of RT–11 files.

The distribution kit is your master copy of RT–11; therefore, do not write to it or use it as a system volume for development and production. Instead, copy the distribution software and work from the copy during installation.

Installing RT–11 requires at least one of the mass storage devices supported by RT–11. Although RT–11 supports a large number of storage devices, RT–11 is not distributed on all of the supported media. The software distribution media include:

- RK05, RL01, or RL02 cartridge disk
- RX01 or RX02 diskettes
- RX50 diskettes
- RC25 (25-Mbyte removable disk cartridge)
- Magnetic tape (9-track, 800 bits-per-inch)

The *RT–11 Software Product Description* (SPD) has a complete list of RT–11 distribution media. The system device (SY:) does not have to be the same type as the distribution device. For example, if your system has an RX02 drive unit and an RK07 drive, you may create an RK07 system volume by installing RT–11 from the RX02 distribution kit.

When you have installed RT–11 on a system volume, you can run any of the distributed monitors and use the device handlers and utility programs. The distributed RT–11 software has the necessary features for most applications.

Installation Procedures

When you install your system, you must perform the following steps:

1. Boot the distribution volume.
2. Create a backup copy. Store the backup copy in a safe place, separate from working volumes.
3. Create one or more bootable working system volumes by copying selected software components from the distribution kit. Make backup copies of these new volumes.
4. Test the working system by editing, assembling, linking, and running the demonstration programs.

Recent versions of RT–11 are distributed with automatic installation and verification procedures, which simplify installation.

Automatic Installation

To use the automatic installation procedure, you must possess the following minimum hardware configuration:

Processor	PDP–11 or LSI–11 with 24 Kbytes of memory
System Console	LA34, LA100 series, or VT100 series terminal
Mass Storage	dual RX02 diskettes, RD51/RX50 disks, dual RL02 disks, or two RC25 disks (one-drive unit)

If your configuration contains the hardware necessary for an automatic installation, your manual set will include an *RT–11 Automatic Installation Booklet* for your distribution kit. This booklet describes how to perform a hardware bootstrap from the distribution kit.

The automatic installation and verification procedure installs RT–11 by running an interactive control file. It gives you instructions to mount a volume in the destination device and asks you to respond when you have done so. It makes a backup copy of the distribution kit and creates a working RT–11 system volume. When the installation is complete, you must run a verification procedure to test the working system.

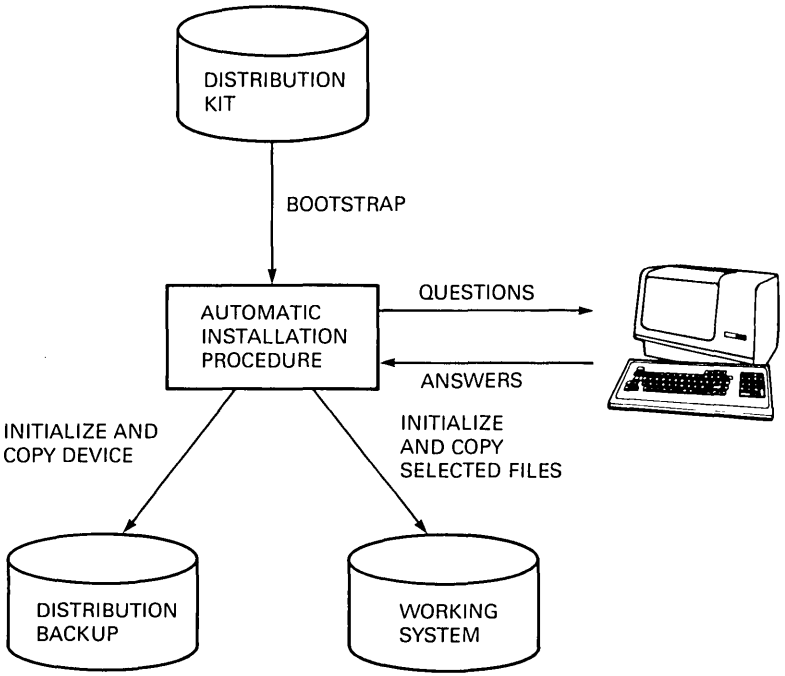
To run the automatic installation procedure, you boot the system from your distribution kit. The system automatically uses the automatic installation procedure when it starts up. If you do not have the configuration necessary for an automatic installation, the procedure displays an error message and exits.

Figure 5 shows the operation of the automatic installation procedure. Distribution kits on diskettes, such as RX02, have multiple volumes so the process is repeated for each volume.

When the working volume has been created, the procedure bootstraps the system from the working volume. You must then verify the installation by issuing the command:

```
.IND VERIFY
```

Figure 5.
Automatic Installation Procedure



The verification procedure carries out the following functions:

- Loads the LP or LS handler and FRUNs QUEUE
- Prints the *RT–11 System Release Notes*
- Assembles, links, and runs the terminal identification program IVP.MAC
- Prints the program listing and link map for IVP.MAC
- Terminates QUEUE and unloads it and the LP or LS handler

At the end of the procedure you will have a distribution kit, a backup of the distribution kit, and an installed working RT–11 system. If the verification procedure fails, it will issue a warning message.

Manual Installation

Although Digital recommends installing RT–11 automatically, it is possible for you to install the system manually. If your system configuration does not meet the requirements for automatic installation, you may have Digital install your system or follow the procedures described in the *RT–11 Installation Guide*, which apply to computer systems of various configurations.

To install a system, you will use a series of four commands: INITIALIZE, COPY/SYSTEM/QUERY, COPY/BOOT, and SQUEEZE.

Changing RT–11 Features

When RT–11 has been installed, the working volume will contain the software components you have selected, with the standard set of features. Table 1-1 of the *RT–11 Installation Guide* lists the features generated with each of the distributed monitors.

Although you do not need to be familiar with RT–11 to install a distributed system, you should modify the system only if you are familiar with RT–11. There are a number of ways to modify the RT–11 monitors and other software components, including:

- Issuing monitor commands to change device characteristics permanently (see SET dd: in the *RT–11 System User's Guide)*
- Placing monitor commands in the startup command file to initialize the system when it is bootstrapped
- Using system generation to implement major changes to the RT–11 components
- Patching files to make permanent changes to the monitor, device handlers, and utility programs

The features that you can select by patching components are listed in table 1-3 of the *RT–11 Installation Guide.*

Perform only the patches that are published by Digital. Chapter 3, "System Maintenance," describes in greater detail the use of SIPP and the procedures for patching software components.

System Generation

Many permanent customizations cannot be implemented by monitor commands or short patches. These customizations are implemented by a process called system generation, or SYSGEN, used to create RT–11 monitors and device handlers. Customization allows you to select system features and devices to be supported and to specify details of the configuration on which the software will be used. When you have selected the features you want, the requested components are assembled (using conditional assembly code to include the options you have selected) and linked.

The minimum hardware configuration recommended to perform a system generation is:

- A dual RL01 disk (or larger)
- 32 Kbytes of memory (56 Kbytes is preferable)

System generation can be done on systems that have only diskettes, but this is not recommended because of the amount of media swapping needed. Chapter 3 of the *RT–11 System Generation Guide* describes system generation on small systems.

Table 1-4 in the *RT–11 Installation Guide* lists the features that are available only through a SYSGEN. The features most often needed include multiterminal support, system job support, and error logging.

To perform a system generation, you should:

1. Plan the system generation
2. Run the SYSGEN command file
3. Assemble and link the components
4. Back up the generated system

Planning a System Generation

Planning is the most important part of any SYSGEN. The other steps use routine procedures.

First, you should collect information about the hardware configuration—what devices you have and their vector and register addresses. Then, determine which monitors you want. Refer to the *RT–11 Software Product Description* and the *RT–11 Installation Guide*, chapter 1, to find out the differences between the RT–11 monitors. Finally, study the available options, analyze the needs of your application, and decide which options you want. The features available through a SYSGEN are listed in table 1–1 of the *RT–11 System Generation Guide*.

You will not want to perform a system generation often, so you should plan for the needs of all the applications you expect to run in the near future. If this creates too large a system, you should consider generating a number of system volumes, one for each application.

Running SYSGEN

SYSGEN is an indirect control file and uses .ASK and .ASKS to issue questions. To run the SYSGEN command file, you issue the command:

```
.IND SYSGEN
```

The system prompts you with the questions. Your responses determine the programs that are built and the options selected. The default answer to each question is shown in brackets at the end of each question. If you want the default, press ⟨RETURN⟩. If you do not understand a question, press ⟨ESCAPE⟩⟨RETURN⟩; SYSGEN will type an explanation of the question and ask you again. You may abort SYSGEN at any time by pressing ⟨CTRL/C⟩ twice.

SYSGEN might not ask every question shown in the manual. For example, question 20 asks:

```
"Do you want all the keyboard monitor commands [Y]?"
```

Questions 21, 22, and 23 determine which subset of commands you want. If you answer Y to question 20, SYSGEN will not ask questions 21, 22, and 23. The SYSGEN questions are arranged in the following groups:

Initialization	determines whether you want to use the answers from a previous SYSGEN and whether you want to save the answers from this SYSGEN. The default name for the answer file is SYSGEN.ANS. If you do use a previous answer file, you may change the answers to any question by giving the number of the question you want to change.
Monitor Type	determines which RT–11 monitors you want built. The default is to build SJ and FB but not XM. The monitors will be generated with the name RT11mm.SYG; "mm" is SJ, FB, or XM.
Monitor Options	determine monitor options, such as error messages on system I/O errors, multiterminal support, and user command linkage.
Device Options	determine which devices the generated system will support. SYSGEN generates device handlers called xx.SYG, where "xx" is the permanent device name. Type a question mark (?) to get a list of the devices available. You may also use this section to add support for user-written device handlers. Unless you select additional device slots, the

	system will only contain enough slots for the devices you specified.
Graphics Options	determine the register and vector addresses for VT11 or VS60 graphics display terminals. Graphics support is not available if you select multiterminal support.
Terminal Interface Options	determine the vector and register addresses for local and remote DL11 and DZ11 lines if you selected multi-terminal support. At this stage SYSGEN asks if you want to change any of your responses. If you want to make only minor changes, you may go back to answer some questions again; although in general, if you have made a major mistake it is best to abort the SYSGEN and start again.
Physical Device Selection and SYSGEN Cleanup	determines the devices for the source input files, binary output files, and link maps and asks whether you want to keep the SYSGEN work files, object files, and map listing files.

A number of files are used in the process of system generation. These files are:

SYSGEN.COM	is the indirect control file that performs the system generation. It creates the output files using the answers to the SYSGEN questions.
SYSGEN.ANS	saves answers from a system generation. These answers can be used later in another system generation.

SYSGEN.CND	contains conditional assembly values, determined by the answers you give to the questions. This file is assembled with all of the software components generated by the SYSGEN.
SYSGEN.TBL	sets up device table entries for each device you specify. SYSGEN uses some of the conditionals to generate this file.
SYSGEN.MON	assembles and links the RT–11 monitors. SYSGEN creates this indirect command file.
SYSGEN.DEV	assembles and links the device handlers. SYSGEN creates this indirect command file.
SYSGEN.BLD	is the command file that runs SYSGEN.MON and SYSGEN.DEV.

SYSGEN.ANS is the default file name for the answer file. If you use a different file name with the .ANS file type, that name will be used for all the other output files. For example, if you call the .ANS file WORKFB.ANS, the other files will be called WORKFB.CND, WORKFB.TBL, WORKFB.MON, WORKFB.DEV, and WORKFB.BLD.

Assembling and Linking the Components

After completing the interaction with SYSGEN.COM, you must assemble and link the components.

If necessary (for example, if you are doing a SYSGEN on an RX01- or RX02-based system), enter the assembly and link commands yourself, moving and deleting files as necessary. Refer to the .MON and .DEV files for the necessary commands.

The rest of this section assumes that you are using the command files generated by SYSGEN.COM to build the components. You can do this by using one of the following commands:

.$@SYSGEN.MON	builds the RT–11 monitors
.$@SYSGEN.DEV	builds the device handlers
.@SYSGEN.BLD	builds all the generated components by executing both of the above indirect commands

SYSGEN.MON and SYSGEN.DEV refer to four logical device names. You specify to which physical devices they are assigned during the SYSGEN dialogue.

SRC: is the source input device. All the source files needed during assembly must be on this volume.

OBJ: is the object output device. All the object (.OBJ) files created during assembly will be written to this volume.

BIN: is the binary output device. The generated components will be written to this volume.

MAP: is the load map output device. The load maps (.MAP) from the linker will be written to this device. It may be a listing device such as TT: or LP:, but it is often useful to store the maps in files. You will need to examine the load maps if you want to patch the components later.

Use the largest, fastest storage devices available. If you only have one block-replaceable device unit, mount a copy of your installed system in it, and use that volume for all logical names.

If you have two block-replaceable units, you may mount the installed system in drive 0 and use it for SRC:, then mount an initialized disk in drive 1 and use it for BIN:, MAP:, and OBJ:. If you do this, the volume in drive 1 will contain only the generated components; you will have to transfer other components later.

After the assembly and link operations have been completed, the following files are left:

MAP: contains .MAP files from linking the monitors and device handlers, unless you asked for them to be deleted.

BIN: contains the generated monitors and device handlers.

OBJ: contains the .OBJ files from the assemblies, unless you asked for them to be deleted.

When the system has been generated, you should apply any of the patches selected from the *RT–11 Installation Guide, RT–11 System Release Notes,* or appendix D of the *RT–11 System Generation Guide.* Remember that .SYG is the file type given to all the device handlers and monitors created by SYSGEN. In general, patching should not be necessary.

To make the volume BIN: a system volume, you must change the .SYG files to .SYS. You should perform a software boot using the BOOT command to verify that the system you generated works:

EXAMPLE

```
.BOOT RT11SJ.SYS
```

You must then set up the bootstrap block with a COPY/BOOT command and reboot the system. If the BIN: volume is also the SY: volume during SYSGEN, you must be careful about the transition to the new system. Do not simply rename .SYG files as .SYS, for this may delete currently active system components. Instead, build a new system volume on some device other than SY:, using the .SYG components renamed .SYS., and then boot the new system volume.

Backing Up the System

When you have installed your system and added all the customizations you need (either by system generation or by patching), you should make a backup copy of the system volumes. You should keep backup copies of the following:

- Generated monitors, handlers, and utility programs
- Conditional files SYSGEN.CND and SYSGEN.TBL
- Command files SYSGEN.MON and SYSGEN.DEV
- Map files

If possible, you should also keep hard copy listings of the SYSGEN session to show which options have been selected. If you perform more than one SYSGEN, be sure to keep your files in order. For example, you may store the backup of each SYSGEN on a separate small volume, or give the files unique names.

Using Generated Monitors and Handlers

A monitor and the handlers it uses must be compatible. They must match the following system generation options: memory management, error logging, and device timeout support.

If the options of a handler do not match those of the running monitor, the monitor will not install the handler. For safety, you should use the distributed handlers only with the distributed monitors, and use monitors only with handlers that have been assembled using the same SYSGEN.CND file. It is possible to maintain the SJ, FB, and XM monitors on one system disk. You may then switch from one system to another using the BOOT command.

If you want to maintain the results of more than one system generation on one volume (or the distributed system and a generated system on one volume), you will have to change the names of one set of handlers and monitors. To identify different monitors, rename them any name you like, but remember that a monitor must have .SYS as its file type.

The default name for handlers is dd.SYS for SJ or FB systems or ddX.SYS for XM systems, where "dd" is the permanent device name. You may not have two handlers for one device on a volume with the name dd.SYS, so you must rename one set. A patch is given in appendix D, sec-

tion D.3 of the *RT–11 System Generation Guide* that enables you to specify a single-character suffix for the device handlers used by a specific monitor. For example, if you have two FB monitors on a volume—the distributed RT11FB and a multiterminal version named RT11MT—you can patch RT11MT to access handlers named ddM.SYS. You must name the generated handlers ddM.SYS to match, and then restore the original dd.SYS handlers from the distribution kit if necessary.

Preparing a Working System

To create an RT–11 system volume to work with your application you must perform the following: install the distributed system on a system volume, perform a system generation if necessary, and apply any selected patches.

Automatic installation transfers all the RT–11 software to one or more volumes. If you have large system disks, such as RK07s or an RA80, you may be able to fit all the RT–11 software and all your own files on one volume. If you have small volumes or do not want all of the RT–11 components on the production system disk, select which files to put on the system volume. Refer to table 2-1 in the *RT–11 Installation Guide* for a list and description of all files in the distribution kit.

The system volume must contain the handler for the system device, at least one RT–11 monitor to be used by the hardware boot, and the swap file SWAP.SYS. If the system was built to support startup command files, the system volume should also contain STARTS.COM for an SJ monitor, STARTF.COM for an FB monitor, and STARTX.COM for an XM monitor.

To hardware boot the RT–11 monitor, the bootstrap code must first be loaded on the volume using the COPY/BOOT command. A volume that contains only the files described above can boot RT–11, but you will not be able to do anything useful on such a system unless it contains other software.

The programs you put on the system volume depend on the size of the volume and the needs of your applica-

tion. On a large system volume you may keep programs that you do not use often. On small volumes you must select the components carefully. Here are some suggestions for which components to include:

- Device Handlers. Include handlers for all devices on your configuration. You should include TT.SYS if you are using the SJ monitor. Since TT.SYS is included with the FB and XM monitors, you do not need to specify it.
- Utilities. PIP, DUP, and DIR are almost always needed, as are RESORC and HELP. You do not need to keep HELP.TXT unless you want to modify the HELP text.

  If you have a printer and will be using FB or XM, the QUEUE package is useful for printing files. You will need the files QUEUE.REL and QUEMAN.SAV. The transparent spooling package (SPOOL) is especially useful for sending files to serial output devices. SPOOL consists of SPOOL.REL, SP[X].SYS, and SPOOL.SYS.

  You will probably need to create source files, so you will need an editor. You can select EDIT or KED, (KEX under the XM monitor). The terminal on which you edit will, in part, determine which editor to use.

  If you are going to develop programs on your system, you will need the program development utilities. If you are programming in MACRO–11, you need the assembler MACRO.SAV and the system macro library SYSMAC.SML. For cross-references, you need CREF.SAV.

  If you are programming in FORTRAN IV, you need the compiler FORTRA.SAV and the FORTRAN IV library. The library may be a separate file FORLIB.OBJ, or it may be included in the system library SYSLIB.OBJ.

  If you are using BASIC–11, you need a BASIC–11 interpreter. The default interpreter is BASIC.SAV.

  If you want to communicate with a host system, the virtual terminal communication package (VTCOM)

changes your stand-alone system to a local terminal. VTCOM consists of VTCOM.REL, VTCOM.SAV, XC[X].SYS or XL[X].SYS, and TRANSF.SAV.

Special Considerations for Small Volumes

If you are using a small device such as RX01, RX02, or RX50 as the system device, you may not be able to fit all the software components on a single volume. Even if you can, it may not have room for your own files.

You can avoid this problem if you:

- Keep the system volume (SY:) in unit 0, and assign DK: to unit 1. If you create all user files on a data volume in unit 1, then it does not matter if there is very little room left on the system volume.
- Create a separate volume for utilities that are not used very often. To use these utilities you must run them directly, specifying the device. This means that you may not use DCL commands that call those utilities unless you temporarily copy the utility to SY: when you want to use it. Note that you should not run PIP or DUP from any device other than SY:.
- Build special-purpose system volumes—for example, one for MACRO–11 program development, one for BASIC–11, and one for dedicated applications work.
- Keep only one monitor on a system volume and only the device handlers that will be used.

References

RT–11 Installation Guide. Chapter 2 describes the software components and discusses the arrangement of these components on a working system. Table 2-2 lists the capacities of various system disks.

RT–11 Automatic Installation Booklet.

RT–11 System Generation Guide. Chapter 1 explains the procedure for system generation. Appendix A contains an example of system generation.

RT–11 Software Product Description.

RT–11 System Release Notes. Chapter 11 of the release notes for RT–11 version 5.1 discusses SPOOL in detail. Chapter 12 describes VTCOM.

Introduction to RT–11.

3

3

System Maintenance

You will want to maintain up-to-date versions of the software products you use and be able to select from a range of system modifications. This chapter discusses the procedure for updating system software and selecting and implementing customizations. Some customizations must be implemented by patching system software components; others may be implemented by giving commands to the monitor or placing them in the startup command file. This chapter discusses these commands—which have a permanent effect and which must be placed in a startup file.

You will learn to use a software update kit to replace system software modules, use software patching utilities to implement published customizations, and customize a system by using monitor commands manually or by placing them in the startup file.

System software may be modified at three levels: source code level, object code level, or memory image level.

System updates are modifications issued by Digital to replace old software modules with new ones. Software customizations are optional modifications to the RT–11 operating system. You may select from a range of published modifications. For software customizations, old software modules are modified using patching utilities (as opposed to replacement). Other system customizations may be implemented by placing commands in the startup file. These may include commands to run programs other than Digital system programs.

Updating the System

Digital supplies updates to RT–11 in the form of update kits, which enable the system to be updated automatically. Each update kit includes:

- A letter from Software Product Services containing special instructions about the update
- Release notes describing the update changes and related user information
- User documentation on how to perform the update
- Distribution medium containing the new software modules and automatic update command file(s)

The license and support terms that apply to your installation will determine what updates you receive for your RT–11 system.

Customizing Software with Monitor Commands

Many customizations are best implemented using monitor commands:

- You can mount a logical disk using the monitor command MOUNT. The format of the command is:

  MOUNT LDn: FILESPEC

  The logical disk remains mounted even if you reboot the system.

- You can use the SET command to modify the characteristics of a device handler, and the new characteristics remain even after a reboot. The format of the command is:

  SET LP: CSR=aaaaaa

In this example, the command modifies the line printer device handler so that it uses the control status register (CSR) address "aaaaaa" (octal) for the line printer controller. Note that only the device handler on the system volume is modified by most SET commands. The device handler must be unloaded, installed, and loaded (or the system rebooted) before the change becomes effective.

By using these monitor commands you can perform many modifications to system software without having to perform a SYSGEN.

There will be other customization commands whose effects do not remain after a reboot. If your system always needs the effects of these commands, you should place the commands in the startup file—for example, SY:STARTF.COM for the FB monitor. Here are some other examples:

- If your system needs QUEUE, SPOOL, or the error logger, then the startup file should contain commands to run them.
- If your system console is a video terminal, you may type SET TT SCOPE in the startup file. (SCOPE is the default setting for RT–11 version 5.1.)

Practice 3–1

1. Perform a customization that will make the DIR command produce directory listings with one column instead of two by default. Refer to the *RT–11 Installation Guide* for details of the procedure.
2. Assume that all programs run on the system will need to have the User Service Routine permanently resident in memory. Make a backup copy of the startup file and edit the startup file so that the User Service Routine will not swap out during program execution.

Customizing Software with Patching Utilities

Software customizations are usually small changes to system software in the form of input to patching utilities. You make larger changes by performing a system generation.

The published customizations are found in appendix D of the *RT–11 System Generation Guide* and chapter 2 of the *RT–11 Installation Guide*. Other customizations are found in the *RT–11 System Release Notes*. These sources include procedures for implementing the customizations.

When you have generated a system, you should then look at these customizations to see whether any of them are needed for the specific applications you intend to use. For example, if your terminal requires fill characters to follow each carriage return, you need to implement a customization on the monitor to send the fill characters to the terminal.

The published customizations are implemented using the following patching utilities: SLP.SAV, PAT.SAV, and SIPP.SAV. All customizations published in chapter 2 of the *RT–11 Installation Guide* are implemented using SIPP.

Source Code Patching

Source code patching is a source code modification to a software component. The usual procedure is to run the source language patch program (SLP) with the command file to patch the program.

Object Code Patching

Object code patching is a technique for modifying the object code of a software component. Generally, you will perform the following:

- Create a source file containing the patch by using an editor
- Assemble the patch source file
- Run the object patch utility (PAT.SAV) to install the patch to the object module

Memory-image Patching

Memory-image patching is a technique for modifying the memory image of software components using the save image patch program (SIPP.SAV). The process for modifying the memory image is described below:

1. Run SIPP using the command:

   .R SIPP

2. When you have entered the command and pressed ⟨RETURN⟩, SIPP replies with an asterisk (*) prompt. You then reply with the file specification of the memory image you want to modify and press ⟨RETURN⟩:

   *FILESPEC

3. If the memory image is overlaid, SIPP then asks you for the number of the segment that contains the data you want to modify ("nnnnnn" represents the number in octal):

   Segment? nnnnnn

4. SIPP now prompts you for the base address and offset of the location you want to modify, regardless of whether the program is overlaid. You give these in octal.

   Base? bbbbbb

   Offset? oooooo

5. SIPP displays the contents of that location and prompts you for new values starting with the loca-

tion you specified. The segment column is only used for overlaid programs as shown:

Segment	Base	Offset	Old	New?
nnnnnn	bbbbbb	oooooo	xxxxxx	–

6. You may give the new value in octal, ASCII, or Radix–50 format as follows:

Octal	;Oyyyyyy	"yyyyyy" is the new value in octal. Since octal is the default format, you need to use ;O only after ASCII or Radix–50 input.
ASCII	;Ay	"y" is the character to be inserted in the byte of the current location.
Radix–50	;Ryyy	"yyy" is up to three characters to be inserted into the current word location in Radix–50 format.

You must press ⟨RETURN⟩ after entering each value. You can change the default radix by entering:

;O ⟨RETURN⟩	for octal
;A ⟨RETURN⟩	for ASCII
;R ⟨RETURN⟩	for Radix–50

7. To leave a value unchanged, press ⟨RETURN⟩.

8. To back up to a previous location, type a circumflex and press ⟨RETURN⟩:

```
^⟨RETURN⟩
```

9. When you have finished modifying values, you press:

```
⟨CTRL/Y⟩⟨RETURN⟩
```

which returns you to the asterisk (*) prompt.

10. To exit from your session with SIPP, press ⟨CTRL/C⟩ after the * prompt.

Let us assume that you want to implement a customization on LINK. Instead of looking for SYSLIB on the system device SY:, LINK is to search for SYSLIB on DK:. To accomplish this, run SIPP. When the system sends the offset prompt, type in the offset of SYSLIB.

EXAMPLE

```
.R SIPP(RETURN)
*SY:LINK.SAV(RETURN)
Segment?      0(RETURN)
Base?         0(RETURN)
Offset?       000000(RETURN)

Segment Base     Offset     Old      New?
000000  000000   000000     ??????   ;RDK(RETURN)
000000  000000   000000+2   ??????   (CTRL/Y)(RETURN)
*(CTRL/C)
```

References

RT–11 Update User's Guide.

RT–11 System Generation Guide. Appendix D contains information about published customizations and the procedure for implementing them.

RT–11 System Utilities Manual. Chapter 19 explains PAT in detail; chapter 20 covers SIPP in detail; and chapter 21 discusses the Source Language Patch Program (SLP).

RT–11 System User's Guide. Chapter 4 discusses the arguments of the SET command—those that have a permanent effect and those that reset to a default condition when the system is rebooted.

RT–11 System Release Notes.

RT–11 Installation Guide. Chapter 2 discusses customizations and the procedure for implementing them.

Solutions to Practices

3–1 (1) See Chapter 2 of the *RT–11 Installation Guide* for the customization procedure.

(2) The startup command file should now contain the command:

.SET USR NOSWAP

No other SET USR commands should be present in the command file.

4

4

The Queue Package and the Error Logger

When you issue the PRINT command to print a file, the keyboard monitor (KMON) performs the print operations by default. KMON runs in the background job environment, which means that you must wait until the listing is complete before you issue another monitor command. When you run a foreground or system job, you may continue using the background environment to issue monitor commands or run background programs independently while the other job is running.

Digital supplies a queue package as part of RT–11 for queuing output to any RT–11 supported device such as the line printer. QUEUE runs as a system job (or foreground job if the monitor does not have system job support). When you issue the PRINT command, the utility program QUEMAN sends information to QUEUE about the file to be printed. QUEUE performs the listing operation from the system job environment, so you may continue issuing monitor commands while the file is being printed.

You may run one system job in the foreground if you are using a Foreground/Background monitor. If the monitor has system job support, then you may run up to six system jobs, in addition to the foreground and background jobs.

Another package supplied with RT–11 that uses a system job slot is the error-logging package. This records I/O activity, device errors, and memory errors. Under the FB and XM monitors, the system job ERRLOG receives information about all errors and records them in a file. Under SJ, a pseudo device handler EL receives the information and holds it in a buffer. You then use the SHOW ERRORS command to get a report of the recorded information.

In this chapter you will learn how to use the queue package to print or transfer files while the system is running other operations, and you will learn to start error logging and produce error-logging reports.

How QUEUE and ERRLOG Can Be Run

If you are using a Foreground/Background or Extended Memory monitor that does not have system job support, you may run either queuing or error logging, but not both, as a foreground job. If you want to run both queuing and error logging at the same time or need to run a foreground job of your own, you must generate a monitor that includes support for system jobs, and run QUEUE or ERRLOG using the SRUN command.

In a system job environment it is recommended that you always use SRUN to run system jobs, so that they run using their correct logical job names. There are only two functional differences between running jobs as foreground jobs or as system jobs:

1. To start a job, use:

   FRUN for a foreground job

   SRUN for a system job

2. To abort a foreground job, use:

   ⟨CTRL/F⟩⟨CTRL/C⟩⟨CTRL/C⟩

   or

   ABORT F

To abort a system job, use:

⟨CTRL/X⟩JOBNAME⟨RETURN⟩

or

⟨CTRL/C⟩⟨CTRL/C⟩

or

ABORT JOBNAME

Queuing

The RT–11 queue package is designed to allow files to be printed on a line printer while normal operations, such as program development, proceed in the background. In addition to the printer, the queue package may also be used with other peripheral devices. The queue package stores any number of requests for transfers from one or more files to either another file or to non-file-structured devices.

If a transfer is interrupted by some event, for instance the printer runs out of paper, the queue package can restart the transfer from the beginning so that a complete copy of the requested file will be produced.

Components of the Queue Package

The queue package is made up of two programs and a data file:

QUEUE.REL	This is the main component of the package. It maintains a list of requested transfers and adds new requests to the end of the list as they are made. It uses this list to control the I/O transfers it performs. After each transfer has been completed, it deletes the appropriate entry from the top of the list.
QUEMAN.SAV	This utility communicates with the queue. You may issue commands using DCL (for example, PRINT

	FILNAM.TYP) or CCL (for example, QUEMAN FILNAM.TYP/P) to QUEMAN in order to: • Make a request for a transfer • Check the status of waiting requests • Delete a request from the queue • Set control parameters of QUEUE
QUFILE.WRK	This work file on SY: is used by QUEUE to maintain the list of waiting requests. By keeping this list in a file instead of in memory, the queue package can continue from where it left off (for example, if the system crashes and is rebooted).

Queuing Operations

In using QUEUE, you should make sure that QUEUE is running by issuing the appropriate SRUN or FRUN command. If the queue package is to be used very often on a system, this command should be placed in the startup file.

You may send files to the QUEUE device using the QUEMAN utility, discussed in chapter 17 of the *RT–11 System Utilities Manual*. If that device is LP:, then the PRINT command will direct output to QUEUE if it is running.

Because QUEUE runs as a foreground or system job, you must load any device handler (other than SY:) that is to be used by QUEUE. You may load the handler either before or after running QUEUE, but you must load it before QUEUE tries to execute a transfer that needs the handler. Therefore, load the handler before issuing the queuing command that needs it. This may be done by placing the appropriate command in the startup file.

If you will be queuing files to a sequential device and want to be sure that no program other than QUEUE uses that device, assign ownership of that unit to QUEUE within your LOAD command. If QUEUE is running in the foreground, use the command:

```
LOAD dev:=F
```

But if QUEUE is running as a system job, use the command:

```
LOAD dev:=QUEUE
```

Queuing a file

If QUEUE is running, the PRINT command causes QUEMAN to run, which passes the request on to QUEUE. When you issue a PRINT command that is passed on to QUEUE, all the files in that command are printed as a single operation. (You may use the /PROMPT option of PRINT to include more files than would fit on one line.) All the files of the command will be copied to the output device or file in the order in which they appear in your command. This constitutes one queue job.

By default, a queued file is sent to device LP:. Chapter 2 of the *RT–11 Installation Guide* documents a customization patch that you can use to change this default. By default, the job name is the same as the name of the first file of the job. The /NAME option can be used to override this default. The format is:

```
PRINT/NAME:[DEV:]JOBNAME
```

To check the status of a QUEUE job—whether it is completed, in progress, or waiting—use the SHOW QUEUE command to see a list of the current queue jobs and each job's status.

Deleting a job

If, before a job has completed, you determine that you no longer want the job to be run, you may delete that job from the queue. For example, you might have requested that a program listing file be queued to the printer; but before the file is printed you find an error, correct it, and want to print the later version instead. So, you would need to delete the job from the queue. To do this, use the DCL command DELETE/ENTRY.

Run-time QUEUE options

You may set the following options of the queue package:

- The default number of banner pages
- The status of the work file QUFILE.WRK (whether or not to delete it if you halt QUEUE)

In the output from a queue job, a banner page identifying the output file may precede any file in a job. The banner page contains the file name in large block letters, the job name, and the date and time the file was printed. Banner pages are useful when you are printing many files using the QUEUE system because they help you separate one listing from another.

When you first run QUEUE, it is set to a default of no banner pages. To change this default you must run QUEMAN directly and use the /P option. You may override the current default within a specific PRINT command by using the options /FLAGPAGE:n or /NOFLAGPAGE. These options set the number of banner pages to be used for each file of the job that you create with that PRINT command. They do not change the default for any other job.

If you want to set the number of banner pages for individual files within a job, you must run QUEMAN directly and use the /H:n and/or the /N option on the files of the job. These options are not implemented in the DCL PRINT command.

The /P option to the utility QUEMAN is also used to determine whether QUFILE.WRK is to be deleted if you halt QUEUE. If the work file is deleted when you halt QUEUE, then you cannot restart QUEUE where it left off; you must issue the commands again to queue any jobs that did not complete. On the other hand, if you decide to keep QUFILE.WRK when QUEUE is halted, the restart capability is enabled. By default, QUEUE is set to keep the work file if QUEUE is halted.

Stopping and restarting queuing

QUEUE keeps its job list and current status in the work file QUFILE.WRK. Keeping this information in a file allows

queuing to be restarted without reentering the commands for the jobs that were waiting to be run. Thus, the queue package allows you to recover from interruptions. You may also stop and restart queuing by aborting QUEUE, by suspending QUEUE using QUEMAN, or by using the monitor SUSPEND command.

Interruptions

QUEUE's operation will be interrupted by a system crash or fatal error. If this occurs, when you run QUEUE again, it will automatically restart the printing at the beginning of the file that was in progress at the time of the interruption.

Other interruptions can affect queuing but not abort QUEUE. For example, the printer may be taken off-line, run out of paper, or experience a paper jam. When you correct the condition, queuing will continue; however, the listing in progress at the time the condition took effect may not be usable. To receive a clean copy of the file, use the /R option of QUEMAN before bringing the printer back on-line. Printing will restart at the beginning of the file that was in progress at the time queuing was interrupted.

Aborting QUEUE

You may want to stop QUEUE during a file transfer. If so, you should abort QUEUE, either by using ⟨CTRL/C⟩ (in conjunction with either ⟨CTRL/F⟩ or ⟨CTRL/X⟩) or by using the /A option of QUEMAN. As soon as QUEUE receives the abort request, which may take a few seconds, it will stop its I/O, close any output files, and exit.

When you abort QUEUE, QUFILE.WRK will be kept or deleted according to the option you have selected with the /P option of QUEMAN. If you are unsure of the current settings, run QUEMAN and reset the options. If you have not set QUFILE.WRK to delete automatically, you may delete it manually using the monitor command DELETE. If QUFILE.WRK is deleted, either automatically or manually, when you next run QUEUE, the queue will at first be empty. If QUFILE.WRK is kept, then when you next run QUEUE, queuing will restart at the beginning of the file that was being transferred at the time that QUEUE was aborted.

Suspending queuing with QUEMAN

Although aborting QUEUE stops output immediately, suspending it allows QUEUE to complete the transfer of the current input file. After the current file completes, QUEUE will close the output file for the job and will not start any new I/O until it receives a command to resume. You suspend queuing by using the /S option of QUEMAN and resume it by using the /R option of QUEMAN. When you use the /R option to resume queuing, QUEUE starts with the file immediately following the one completed after you issued the suspend request.

Use the QUEMAN/S option to suspend queuing if you want to perform routine operations without damaging the output results—for example, to change paper in the printer or separate the files that have been printed so far. Use this method also if you are terminating your session on the computer and want to resume the printing job next time you reboot the system.

Using the DCL command SUSPEND

You can use the command SUSPEND to suspend QUEUE if it is running as a foreground job or use SUSPEND QUEUE if it is running as a system job. The DCL command RESUME (or RESUME QUEUE) will restart QUEUE. The I/O operations stop immediately when you issue the SUSPEND command; they start exactly where they left off when you use the RESUME command.

Under most conditions, it is beneficial to restart QUEUE (because QUEUE starts printing the current file again from the beginning) or suspend QUEUE from QUEMAN and resume it (allowing QUEUE to reach the end of the current file). Using the SUSPEND command from KMON accomplishes neither of these, nor does it provide for updating QUFILE.WRK as the other methods do.

Summary of Queuing

The queue package uses two utilities, QUEUE and QUEMAN. Queuing is used primarily to print files while

the operator performs other operations. Once you start QUEUE (using the FRUN or SRUN command), every PRINT command creates a job for QUEUE. Each job causes the copying of one or more input files to an output device or output file.

When queuing is active, the following special commands and options are available:

SHOW QUEUE	shows you the current status of jobs waiting to be processed by QUEUE
DELETE/ENTRY	removes a job from the queue

In the PRINT command, these special options are available:

/FLAGPAGE:n or /NOFLAGPAGE	controls whether files will be preceded by a banner page showing the file specification in large block letters
/NAME:DEV:JOBNAME	controls the name of the job and the device to which the job will be sent
/PROMPT	allows you to specify more than one line of files in the same job

QUEUE keeps its list of jobs in the file QUFILE.WRK. This allows queuing to be restarted in the event of termination of QUEUE, planned or otherwise. You may restart QUEUE using the /R option of the queue manager utility QUEMAN. Using the /S and /R options of QUEMAN, you may also suspend queuing between files and resume queuing at a later time.

Error Logging

During the normal running of system and user programs, error messages will appear from time to time at the terminal. This information will not always be as complete as you

need it to be. The error-logging system is designed to provide reports indicating the dependability of your system's peripheral devices and, if present on your system, parity and cache memory. These reports are useful as an indicator of developing hardware problems that could be prevented by maintenance. They also can help in the diagnosis of serious hardware problems.

Components of the Error-logging System

The error-logging system is made up of four programs:

EL.SYS	a pseudo device handler used with the SJ monitor to collect information about errors that occur during I/O transfers and store that information in an internal buffer
ERRLOG.REL	a foreground or system job that collects information about the same types of errors and stores them in a file for FB and XM systems
ELINIT.SAV	a background job used to create and maintain a file containing statistics used by ERRLOG.REL
ERROUT.SAV	a background program that creates a report of the error information collected by EL.SYS under the SJ monitor or from the storage file used by ERRLOG.REL under the FB or XM monitor

Error-logging Support under the SJ Monitor

If you are using the SJ monitor, you must first load the error-logger pseudo device handler EL.SYS, using the command:

```
LOAD EL
```

Then to start error logging, use:

```
SET EL LOG
```

The error logger then starts to collect I/O transfer and error information in an internal buffer. When multiple errors are detected within a short time, it is possible for this internal buffer to become full. Under these conditions, a warning message is printed at the terminal and error logging is suspended.

You may clear the contents of the internal buffer when it becomes full, or at any time, by issuing the command:

```
SET EL PURGE
```

Although logging is suspended when the internal buffer becomes full, you may generate a report from the error logger before you purge the buffer. The generation of error-logging reports is discussed later in this chapter.

To suspend error logging you use the command:

```
SET EL NOLOG
```

and resume by starting error logging as described previously.

To disable error logging, use:

```
UNLOAD EL
```

If you want to save the contents of the internal buffer, before unloading the pseudo device handler, use the command:

```
COPY EL:  DEV:FILNAM.TYP
```

where "DEV:FILNAM.TYP" is the specification of the file in which this buffer is to be stored.

Error-logging Support under the FB or XM Monitor

You run the program ERRLOG.REL as a foreground or system job by using either the SRUN or FRUN command. Running error logging as a foreground job means that you may only use the background environment. You should, therefore, run the error logger as a system job if you have

system job support. When the error logger runs, it gives you the message:

```
?ERRLOG-I-To initiate error logging, RUN ELINIT
```

To initiate error logging, issue the command:

```
R ELINIT
```

This program asks you for some statistics about the ERRLOG.DAT file:

- The name of the device for the ERRLOG.DAT file (default:SY:)
- Whether you want to initialize the ERRLOG.DAT file (default:NO)
- The number of blocks for the ERRLOG.DAT file (default:100)

To indicate the default answer to any prompt, simply press ⟨RETURN⟩.

When the error logger has been initiated, ELINIT prints the following message at your terminal:

```
RT-11 V5.0 ERROR LOGGING INITIATED
```

Getting Reports

The information used to make up an error report is stored in EL's internal buffer if you are using the SJ monitor, or in ERRLOG.DAT if you are using the FB or XM monitor.

You may instruct ERROUT to print an error report at the terminal, using the DCL command SHOW ERRORS. For example, the command:

```
SHOW ERRORS/SUMMARY
```

prints at the terminal a summary of the I/O transfer, memory parity, and cache memory errors that have occurred from the time that the current error-logging file or EL internal buffer was initialized.

You may run ERROUT directly by using the command:

```
R ERROUT
```

The Command String Interpreter then gives you an asterisk (*) prompt. At this point you may type one or more of a number of commands that enable you to control:

- The file specification of the report
- The file containing the error information for the report
- Whether the report is to include full information about each error that is logged or only a summary of the errors
- The dates of the earliest and latest errors that are to be included in the report

To specify that all information is to be included in a report, run ERROUT, using the /A option. For example, the command:

```
ERROUT/A REPORT.TXT=SY:ERRLOG.DAT
```

produces a full report REPORT.TXT from SY:ERRLOG.DAT. Used in the same way, the /S option limits the report to a summary of the errors that were detected.

Analyzing a Report from the Error Logger

The information for each error will appear in one of two formats, according to the type of error. These two types of errors are storage device errors and memory errors. A summary of these two types of errors appears after any report that you request. Thus, there are three basic formats that appear in an error-logging report:

- Storage device error report format

- Memory error report format
- Error summary report format

Storage device error report format

When a device handler detects an error during an I/O transfer, it attempts to repeat the transfer a number of times before it gives up. When the transfer remains unsuccessful, data about the error is sent to the error logger. The full information available for this type of error is as follows:

- The type of error
- The date and time it was logged
- The physical unit number
- The device type
- The number of repeated attempts
- A dump of the appropriate device registers
- Whether the error occurred during a read or a write operation
- The location and size of the data that was not transferred because of the error

A sample storage device error report is shown in figure 6. Table 2 is a key to the layout of this report.

Memory error report format

Memory parity errors and cache memory errors cause data to be sent to the error logger. For memory parity errors, the full information available is as follows:

- The type of error
- The date and time the error was detected
- A dump of the appropriate memory registers

A sample of the format of this information is shown in figure 7. Table 3 is a key to the layout of this format.

Figure 6.
Sample Storage Device Error Report

```
**********************************************************************
DISK DEVICE ERROR
LOGGED 27-APR-84 00:06:20
**********************************************************************

UNIT IDENTIFICATION
        PHYSICAL UNIT NUMBER                                    000001
        TYPE                                                RX211/RX02
SOFTWARE STATUS INFORMATION
        RETRY 6. OF 8. POSSIBLE TOTAL

DEVICE INFORMATION
        REGISTERS:
        RX2CS                                   114560
        RX2DB                                   010400
        RX2ES                                   000400

        ACTIVE FUNCTION                           READ
        BLOCK                                   000001
        PHYSICAL BUFFER ADDRESS START           003514
        TRANSFER SIZE IN BYTES                     512.
```

Table 2.
Line-by-line Analysis of the Sample Storage Device Error Report

Line	Explanation
1–4	Report header. Includes the date and time the error was logged.
6–8	Unit identification. Identifies the drive number, the device controller, and the storage device type.
10	Retry count.
12	Labels the section on device information. The lines that follow provide statistics on the device registers and address information.
13–16	Register contents. Each device has a number of hardware registers, the contents of which are listed in these lines.
18	I/O transfer type. Tells whether the I/O transfer was a read or write operation.
19	Device block number. Tells which device block the error occurred in.
20	Physical buffer start address. Tells the physical address in memory of the user data buffer for this I/O transfer.
21	Transfer size in bytes. Tells the size in bytes of the unit of data the device handler has attempted to transfer.

Figure 7.
Sample Memory Parity Error Report

```
*********************************************************
MEMORY PARITY ERROR
LOGGED 27-APR-84 00:06:20
*********************************************************

DEVICE INFORMATION
        MEMORY REGISTERS:
        ADDRESS      CONTENTS
        172100       100001

        ERROR TYPE IS MEMORY
```

Table 3.
Line-by-line Analysis of the Sample Memory Parity Error Report

Line	Explanation
1–4	Report header. Includes the date and time the error was logged.
6	Labels the section on device information.
7–9	Memory parity register contents. Identifies your system's memory parity control and status register(s) and gives their contents.
11	Error type. Tells whether the error was a memory error or a cache memory error.

For cache memory errors, the full information available is:

- The type of error
- The date and time the error was detected
- A dump of the appropriate memory registers
- A dump of the memory system error register, the cache control register, or the hit/miss register

A sample of the format of this information is shown in figure 8. Table 4 is a key to this format.

Figure 8.
Sample Cache Memory Error Report

```
**********************************************************
CACHE MEMORY ERROR
LOGGED 27-APR-84 00:06:20
**********************************************************
DEVICE INFORMATION
        MEMORY REGISTERS:
        ADDRESS      CONTENTS
        172100       100001

        MEMORY SYSTEM ERROR REGISTER:          000200
        CACHE CONTROL REGISTER:                000000
        HIT/MISS REGISTER:                     000032

        ERROR TYPE IS CACHE
```

Table 4.
Line-by-line Analysis of the Sample Cache Memory Error Report

Line	Explanation
1–4	Report header. Includes the date and time the error was logged.
6	Labels the section on device information.
7–9	Memory parity register contents. Identifies your system's memory parity control and status register(s) and gives their contents.
11–13	Cache memory register contents. This information is displayed for both a memory parity error and a cache memory error if your system includes cache memory.
15	Error type. Tells whether the error was a memory error or a cache memory error.

Error summary report format

The information included in the error summary report comprises three sections:

- Device statistics
- Memory statistics
- Report file environment

A sample printout of a summary report, including all three of these sections, is shown in figure 9. Additional information on the three formats can be found in chapter 16 of the *RT–11 System Utilities Manual*.

Practice 4–1

If you have a system that supports the Foreground/Background monitor and has a line printer, use it to perform the tasks specified in questions 2 through 7. If your system also includes a monitor that supports both error-logging and system jobs, perform all the tasks below.

For any tasks that you are not able to carry out because of hardware or software limitations, write down the commands that you would use to perform the specified tasks.

1. Bootstrap a monitor with system job and error-logging support. (If you have no such monitor, ignore this first step and give the commands to carry out the remainder of this question.) Start error logging. If a previous data file is on the disk, initialize it. Accept all other defaults.
2. Start the queuing system.
3. From a directory of your volume(s), identify some text files. (Files with file types .MAC, .FOR, or .COM are useful text files.) Queue two of these files to the line printer, giving each a single banner page.
4. Set the banner page default to one banner page. (Do not set QUFILE.WRK to be deleted.)
5. Queue a number of files to the printer. Make some single-file jobs and some multiple-file jobs. Get a listing of the queue while the files are being printed. Abort QUEUE while a file is printing.
6. Restart QUEUE. Notice which file is printed first.
7. Check the queue and make sure that there are at least two files waiting in the queue. If not, queue up a few more. Suspend QUEUE. Check the queue. Resume QUEUE.
8. Get an error-logging report from the session.

Figure 9.
All Three Sections of a Sample Error Summary Report

```
************************************************************************
DEVICE STATISTICS
LOGGED SINCE 27-APR-84 00:06:20
************************************************************************

UNIT IDENTIFICATION
        PHYSICAL UNIT NUMBER                                    000000
        TYPE                                                 RK11/RK05

DEVICE STATISTICS FOR THIS UNIT:
        NUMBER OF ERRORS LOGGED                  0.
        NUMBER OF ERRORS RECEIVED                0.
        NUMBER OF READ SUCCESSES                49.
        NUMBER OF WRITE SUCCESSES                0.

UNIT IDENTIFICATION
        PHYSICAL UNIT NUMBER                                    000000
        TYPE                                                 RX211/RX02

DEVICE STATISTICS FOR THIS UNIT:
        NUMBER OF ERRORS LOGGED                  1.
        NUMBER OF ERRORS RECEIVED                1.
        NUMBER OF READ SUCCESSES                 2.
        NUMBER OF WRITE SUCCESSES                0.

UNIT IDENTIFICATION
        PHYSICAL UNIT NUMBER                                    000001
        TYPE                                                 RX211/RX02

DEVICE STATISTICS FOR THIS UNIT:
        NUMBER OF ERRORS LOGGED                 16.
        NUMBER OF ERRORS RECEIVED               16.
        NUMBER OF READ SUCCESSES                 0.
        NUMBER OF WRITE SUCCESSES                0.


************************************************************************
MEMORY STATISTICS
LOGGED SINCE 27-APR-84 00:06:20
************************************************************************

STATISTICS:
        NUMBER OF MEMORY PARITY ERRORS           5.
        NUMBER OF CACHE ERRORS                   0.

REPORT FILE ENVIRONMENT:
        INPUT FILE              SY0:ERRLOG.DAT
        OUTPUT FILE             RK0:ERRORS.TXT
        OPTIONS                           /A
        DATE INITIALIZED             27-APR-84
        DATE OF LAST ENTRY           27-APR-84

TOTAL ERRORS LOGGED                      22.
MISSED ENTRIES                            0.
MISSED ERROR ENTRIES                      0.
UNKNOWN DEVICE STATISTICS ENTRIES         0.
UNKNOWN ERROR RECORD ENTRIES              0.
```

References

RT–11 System Utilities Manual. Chapter 16 details the components of the error-logging system and explains ELINIT along with the statistics it uses. Chapter 17 discusses the options of QUEMAN in detail.

RT–11 System User's Guide. Chapter 4 describes in detail the /ENTRY option of the DELETE command, the /NAME and /PROMPT options of the PRINT command, PRINT/FLAGPAGE and /NOFLAGPAGE, SHOW QUEUE, as well as the /TO and /FROM options of the SHOW ERRORS command.

RT–11 Installation Guide.

Solutions to Practices

4-1 (1) If you have a Foreground/Background monitor on the system volume:

```
.BOOT/NOQUERY RT11FB⟨RETURN⟩
```

(2) If you have system job support:

```
.SRUN QUEUE⟨RETURN⟩
```

Otherwise:

```
.FRUN QUEUE⟨RETURN⟩
```

(3) `.PRINT/FLAGPAGE:1 file1, file2⟨RETURN⟩`

(4)
```
.R QUEMAN⟨RETURN⟩
*/P⟨RETURN⟩
No. of banner pages? 1⟨RETURN⟩
Delete workfile? NO⟨RETURN⟩
*⟨CTRL/C⟩
```

(5) For each job, type:

```
.PRINT file⟨RETURN⟩
```

Then:

```
.SHOW QUEUE⟨RETURN⟩
```

To abort QUEUE if QUEUE is a system job, type:

```
⟨CTRL/X⟩
Job? QUEUE⟨RETURN⟩
⟨CTRL/C⟩⟨CTRL/C⟩
```

If QUEUE is a foreground job, type:

```
⟨CTRL/F⟩
F⟩ ⟨CTRL/C⟩⟨CTRL/C⟩
```

Otherwise, type:

```
R QUEMAN⟨RETURN⟩
*/A⟨RETURN⟩
*⟨CTRL/C⟩
```

(6) `.FRUN QUEUE⟨RETURN⟩`

or

`.SRUN QUEUE⟨RETURN⟩`

The file that starts printing should be the same file that was printing when you aborted QUEUE.

(7) QUEMAN continuing to run:

```
*/L⟨RETURN⟩
*file1, file2⟨RETURN⟩
*/S⟨RETURN⟩
*/R⟨RETURN⟩
*⟨CTRL/C⟩
```

(8) `.SHOW ERRORS/PRINTER⟨RETURN⟩`

This command produces a full error report at the printer.

5

5

PDP–11 Architecture

In order to understand how the RT–11 operating system supports device handlers and extended memory, you must understand how the PDP–11 interrupt system and the memory management unit function. This chapter discusses the purposes of the fields within the processor status word, the operation of the processor when an interrupt occurs, and the arrangement and operation of the memory management registers.

The PDP–11 Computer

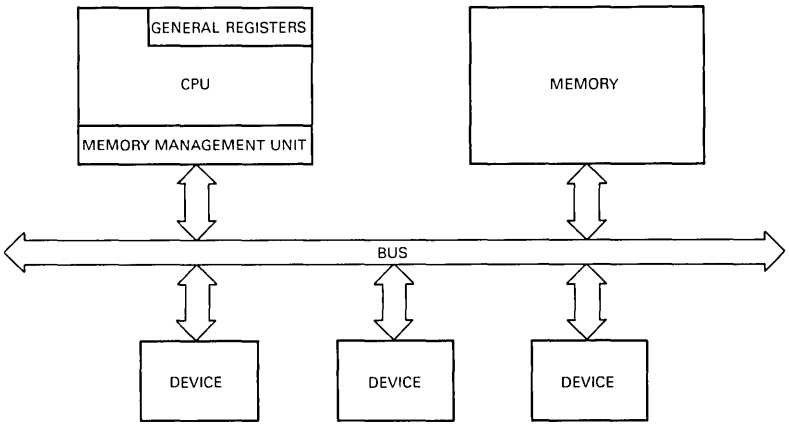
The PDP–11 is a 16-bit computer. The central processing unit (CPU) is the heart of the system. It is connected to the main memory and a number of peripherals by a bus (figure 10). The bus carries addresses and data for devices on the bus. There are also a number of control signals on the bus that enable the CPU and the other devices connected to the bus to communicate with one another.

The memory is read by placing the address of the data on the bus and then signaling a READ request. Similarly, to write data into memory, both the address and the data are placed on the bus and the WRITE control signal is raised. These signals can be issued by the CPU or any bus master device.

Usually, a device transfers data, at the request of the CPU, between the device and the CPU. Then, the CPU transfers that data to memory. Fast devices that transfer large amounts of data often transfer the data directly into memory on their own. This is called direct memory access (DMA).

Every device connected to a PDP–11 computer is controlled by a set of registers. Each of these registers has an address in the range 17760000 to 17777776. This 4-Kword

Figure 10.
System Bus Architecture



address range is not actual memory; it is reserved for device registers and is called the I/O page. The hardware automatically changes references to addresses in the range 160000 to 177776 to the equivalent 18- or 22-bit address.

The CPU contains a number of registers. Seven of these registers, called the general registers and numbered R0 to R7, are available to the software. R6 and R7 are used in a special way by the CPU and are referred to in a program by special names. R6 is the stack pointer register (SP), and R7 is the program counter register (PC).

The CPU executes instructions using the PC as the address of the next instruction to be fetched from memory. Depending on the type of instruction, more information may be fetched from memory and may be modified or written into memory. During the execution of an instruction, the PC is continually adjusted to contain the address of the next instruction to be executed.

The stack pointer keeps track of the latest entry on the stack. The stack is used for storing the information needed to return from a trap or interrupt (or subroutine).

The other register in the CPU that can be accessed by software is the processor status register (PS). This register contains information about the current state of the CPU, as shown in figure 11 and table 5.

Once started, the CPU fetches and executes instructions, following the path of the program through branches and subroutine calls until it executes a halt instruction. You might think that you could predict which instruction in the

Figure 11.
Contents of the Processor Status Register

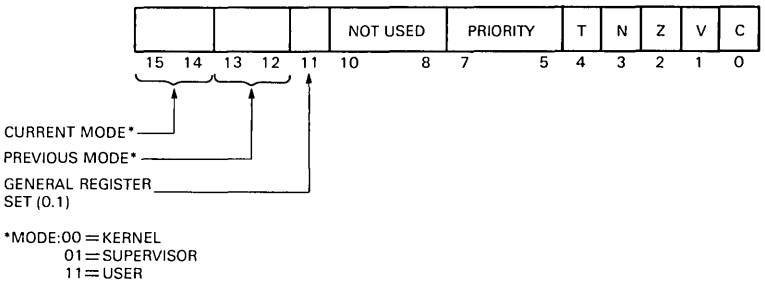


Table 5.
Contents of the Processor Status Register

Bits	Information
0–3	Condition codes are set and cleared depending on the result of previous instructions.
4	Trace trap forces a trap after the current instruction.
5–7	Current processor priority. Only devices with a priority higher than the value in this field may interrupt the CPU. When the CPU is running at priority 7, no interrupts will be accepted. The priority can be changed by executing an RTI instruction or by using other special instructions on some PDP-11s, for example, the MTPS instruction on an LSI-11/23.
8–10	Reserved.
11	Which of two sets of general registers in the CPU is active.
12–15	Previous mode (12,13) and current mode (14,15) bits show which mode (kernel, supervisor, or user) the CPU is running in and was running in before the last interrupt or trap. Some instructions (HALT and RESET, for example) are illegal in user mode. Only in kernel mode can all instructions be executed.

program the CPU would execute next, but there are two events that can modify the path of control—device interrupts and traps.

The Interrupt System

A device usually interrupts the CPU to indicate that it is ready to perform a transfer on the bus or has completed a requested task. The interrupt will only be accepted by the CPU if the priority of the device is higher than the current CPU priority, as given by the priority bits (5–7) in the processor status register. If the priority of the device is not higher, the interrupt request stays active until the CPU priority is lower than the priority of the interrupt request; then the CPU will accept the interrupt. When more than one interrupt request is pending, the highest priority request that is closest to the CPU on the bus will be acknowledged.

When the CPU accepts the interrupt, it must save enough information about the current state of the program it is running to be able to go back to it when the interrupt has been serviced. The program counter register provides one such item of information; the processor status register provides the rest. These two registers are pushed onto the system stack.

When a device's interrupt is acknowledged, the device passes the address of an interrupt vector to the CPU. The first word of the interrupt vector is the value to load into the PC in order to find the software that processes the interrupt, the interrupt service routine (ISR). The second word of the interrupt vector is the new value to load into the PS. It is as though the CPU had executed the following sequence of instructions:

```
MOV     PS,-(SP)
MOV     PC,-(SP)
MOV     INTVEC+2,PS
MOV     INTVEC,PC
```

As the new value is placed in the PS, the CPU priority is adjusted to the level given by the priority bits of the new value, usually equal to or greater than that of the interrupting device. The ISR continues execution until it has satisfied the device's request. The ISR then executes an RTI instruction to restore the PC and PS from the stack. The interrupted process continues from the point of the interruption.

The Trap System

A trap is similar to an interrupt but is not generated by a device. Traps are generated by the CPU, either because it has detected an error (illegal instruction, bus error, and so on) or because it has executed a special instruction (BPT, EMT, IOT, or TRAP). These events cause the CPU to follow a sequence of operations similar to that of an interrupt. The old PS and PC are stacked, and the new PC and PS are loaded from the appropriate trap vector in memory.

(RT–11 programs execute EMT instructions to gain access to functions that the RT–11 monitor will perform on their behalf.)

The trap vectors (at addresses 0–36 in memory) and fixed interrupt vectors (at addresses 60–276 in memory) are set up by the operating system to point at routines within the system that can handle each specific event. Vectors for a variable number of additional devices are allocated in the so-called "floating vector" space (starting at address 300 and usually kept below address 400).

The Memory Management Unit

Programs on a PDP–11 address memory in individual bytes using a 16-bit address. This means that the program can directly address 65536 bytes of memory (32 Kwords) at any given time. The bus and the memory, however, can use up to 22 bits to specify an address.

Although a PDP–11 system may be expanded to contain as much memory as is possible (124 Kwords for a Unibus; 2044 Kwords for a Q-Bus), the program can only directly address 32 Kwords. Of these addresses, the highest 4 Kwords (160000 to 177776) are normally reserved for communicating with devices. This leaves only the lowest 28 Kwords of memory available to the program. Accessing the additional memory requires additional hardware, called the memory management unit (MMU). Only the RT–11 XM monitor provides support for the MMU. In fact, the XM monitor cannot be used on systems without the MMU hardware.

When the MMU is disabled, addresses referred to by the program correspond directly with physical locations in memory. When the MMU is enabled, 16-bit virtual addresses referenced by the program are converted to 18- or 22-bit physical addresses by a process called mapping. A program's addressable region remains 16 bits wide but is thought of as virtual in that the actual addresses used by the program do not necessarily correspond directly to the same physical address.

The MMU relocates a program's virtual addresses in

pages. A page is a range of contiguous virtual addresses that starts on a 4-Kword boundary; it can be from 32 words to 4 Kwords in length, measured in units of 32 words. The virtual address space is divided into eight pages of 4 Kwords each.

The MMU maps each page to physical memory independently of the other pages. This means that a contiguous virtual address space can be mapped into regions of physical memory that are not contiguous. If a virtual page is less than 4 Kwords in length, the virtual addresses between the end of one page and the start of the next are not mapped into physical memory. As a result, these virtual addresses cannot be used by the program unless the mapping is changed; thus, the program has less than 32 Kwords of addressable memory.

The MMU maps a program's virtual address space into physical memory using a set of eight relocation registers called active page registers (APRs). Each APR is made up of a pair of 16-bit registers. The page address register (PAR) and page descriptor register (PDR) work together to relocate a page.

The PAR, shown in figure 12, contains the base address of the page in physical memory. Because the MMU considers memory in units of 32-word blocks, a page must begin on a 32-word boundary in physical memory. Therefore, the base address of a page may be specified using the 12 most significant bits of the address (16 bits for 22-bit addressing). The physical address of the page may be found by multiplying the contents of the PAR by 64.

The PDR (figure 13) specifies the page length (in 32-word blocks), the direction of expansion from the base address, and access control information. The MMU uses the page length to determine whether the specified virtual ad-

Figure 12.
Page Address Register (PAR)

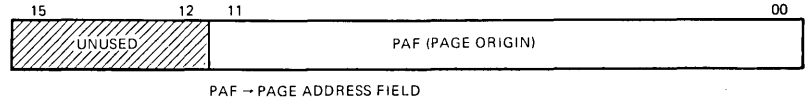
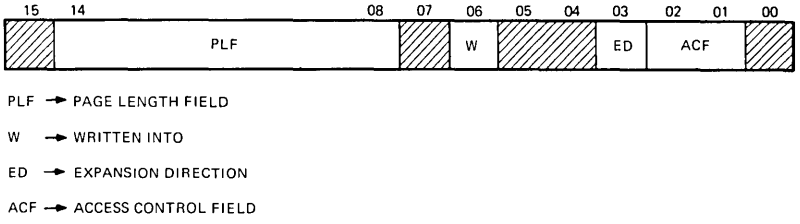


Figure 13.
Page Descriptor Register (PDR)



dress is within the mapping limits of the page. If not, the MMU generates a memory management fault. RT–11 ignores the remaining bits provided for use by the system software.

When the MMU is active, it converts every virtual address into a physical address. The MMU interprets a virtual address as being made up of three fields, as follows: page address field (PAF), block number (BN), and displacement in the block (DIB). These are shown in figure 14.

The last two fields, BN and DIB, are collectively called the displacement field (DF). The mapping is shown in figure 14. The PAF selects the APR that contains the origin of the appropriate page in physical memory. The MMU then adds the BN to the contents of the PAR and appends the DIB to the sum to form the physical address.

The MMU has two modes of operation to separate system functions from user functions: kernel and user. The MMU uses a separate set of APRs and a stack pointer for

Figure 14.
Construction of a Physical Address

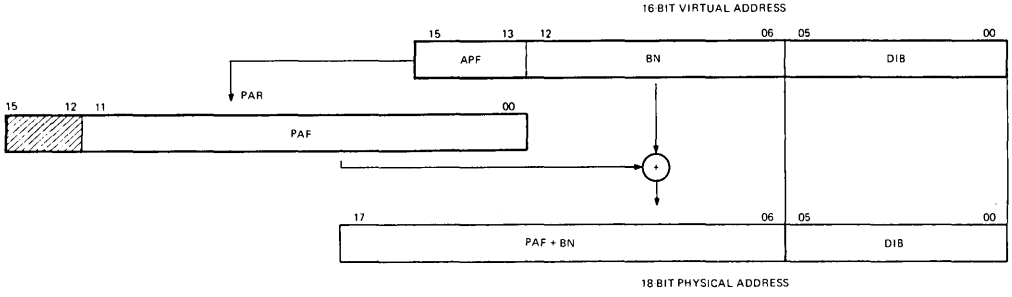
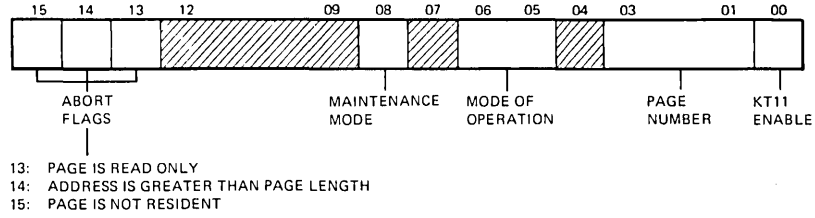


Figure 15.
MMU Status Register 0



each mode. The two current mode bits of the PS determine which set of APRs will be used for the current operation.

Kernel mode is the hardware term for system state. Because the two modes use different sets of APRs, it is not possible for a program executing in one mode to access the physical memory allocated to a program in the other mode unless the APRs map the same physical memory. This feature prevents user programs from modifying the monitor or each other.

User mode prevents execution of HALT and RESET instructions. Attempts to employ them in user mode will generate a trap. Because all traps and interrupts are serviced in kernel mode, a return to the system results.

The MMU has four status registers: SR0, SR1, SR2, and SR3. Status register 0, shown in figure 15, contains the memory management enable bit, abort error flags, and other information used by RT–11 to recover from an abort or to service a memory management trap. Status register 1 is available on some PDP–11s and is not used by RT–11. Status register 2 is a read only register that contains the virtual address currently being converted by the MMU. This may be useful when analyzing faults. Status register 3 is available on some PDP–11s and is used to control 18-bit/22-bit mode. (See chapter 4 of the *RT–11 Software Support Manual*.)

References

PDP–11 Processor Handbook.

RT–11 Software Support Manual. Chapters 4 and 7 contain additional information on PDP–11 memory management.

6

6

Extended Memory Management

This chapter describes how RT–11 allows you to access extended memory through a set of memory management programmed requests. It also outlines some applications for extended memory. Given the description of an operation that requires memory management programmed requests, you will be able to write and implement a simple program that performs the specified operation.

The XM Monitor

RT–11 supports the use of extended memory (up to 22 bits or 2044 Kwords on Q-bus and 18 bits or 128 Kwords on Unibus) through the extended memory (XM) monitor. The XM monitor is equivalent to the FB monitor plus the routines to support memory management. The memory management routines enable the monitor to control the mapping of virtual addresses to physical memory by changing the contents of the active page registers (APRs).

Because a working XM system is present on the distribution kit, a system generation is not necessary to create an XM monitor and the XM device handlers. If you select the XM monitor during the SYSGEN procedure, the monitor and device handlers will be assembled with the prefix file XM.MAC. This file contains the conditional assembly symbol definition MMG$T=1, which will cause all of the memory management support to be assembled in the monitor and the device handlers.

Your system must have the following hardware to run an XM monitor:

- At least 32 Kwords of memory
- A memory management unit (KT11)
- An extended instruction set (EIS)

The XM monitor boots into memory immediately below the 28-K limit and runs with the User Service Routine permanently resident (figure 4–22 in the *RT–11 Software Support Manual*.) All monitor routines (RMON, USR) and device handlers execute in kernel mode, which maps to low memory (0–28K) and the physical I/O page. KMON executes in user mode, but with the same mapping (APR contents) as kernel mode. User programs always execute in user mode. They expand their mapping using memory management programmed requests.

Accessing Extended Memory

In order to access extended memory in a user program you must:

- Specify the amount of physical memory needed
- Specify the virtual addresses you want to use
- Link the virtual addresses to the area of physical memory

You perform these operations using a set of memory management programmed requests that control the contents of the APRs. These programmed requests and the macros that create the data structures are listed in table 4–6 in the *RT–11 Software Support Manual*. The syntax of each programmed request is described in chapter 2 of the *RT–11 Programmer's Reference Manual*. Chapter 4 in the *RT–11 Software Support Manual* describes the sequence of operations the monitor performs for each request.

Creating Regions

The XM monitor allocates physical memory to a program in segments that are called regions. A region is a contiguous segment of physical memory that starts on a 32-word boundary and can be any size from 32 words to 96 Kwords in units of 32-word blocks. A program may have up to four regions at a time.

When the XM monitor runs a program, it automatically reserves one region for the program in low memory (0–28 Kwords). For virtual jobs this is called the static region, a fixed region the program cannot change. In addition, the program can also create up to three dynamic regions in extended memory (above 28 Kwords) at one time.

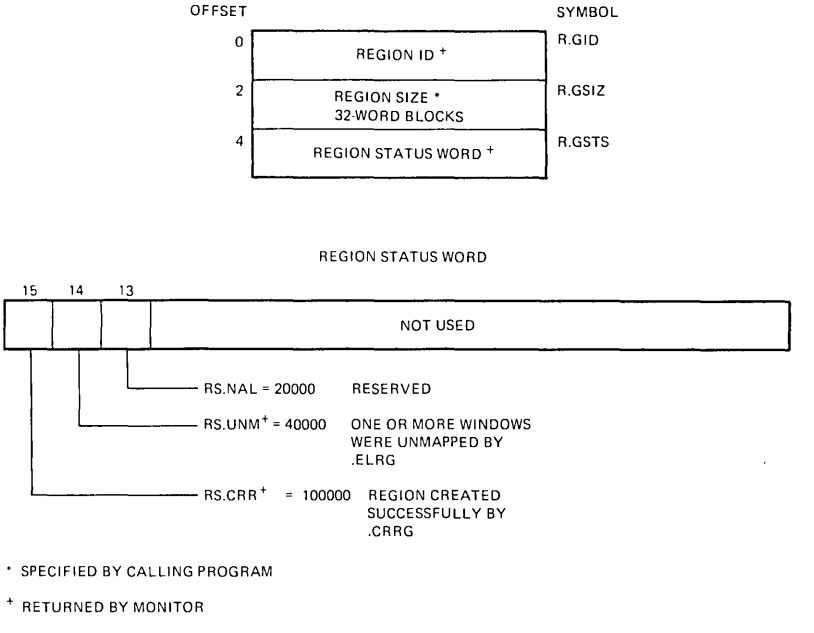
A program describes a dynamic region that it wants to create using a three-word block of memory called a region definition block (figure 16). The program creates a region definition block using the macro:

```
RGBLK:  .RDBBK  rgsiz
```

Here "rgsiz" is the region size in 32-word blocks.

The macro also defines the symbols shown in figure 16. The monitor uses the region definition block to return information such as the region identification and status to

Figure 16.
Region Definition Block



the program. The program creates a dynamic region using the macro:

```
.CRRG   area,rgblk
```

Here "rgblk" is the address of a region definition block specifying the size of the requested region in 32-word blocks.

If the region is created successfully, the monitor returns the region number (ID) in the first word of the region block and bit 15 is set in the status word (figure 16).

Each time the program creates a region in memory, the monitor fills in a region control block in the job's impure area. The region ID returned in the region definition block is the number of the corresponding control block.

Creating Windows

In order to access a dynamic region in extended memory, a program must specify to the monitor one or more contig-

uous ranges of virtual addresses that it wants to use to access that region. Each contiguous range of addresses is called a virtual address window. More precisely, a virtual address window is a contiguous segment of virtual address space that starts on a 4-Kword boundary and may be any size from 32 words to 32 Kwords in units of 32-word blocks.

When the XM monitor runs a program, it automatically creates a virtual address window corresponding to the region it created in low memory. This window includes all of the instructions and data in the loaded program and the program stack. For virtual jobs, this is the static window. The program can also create up to seven additional dynamic windows at one time.

A program describes a virtual address window to the monitor using a seven-word memory block called a window definition block (figure 17). The program creates this block using the macro:

```
WNBLK:  .WDBBK   wnapr,wnsiz
```

Here "wnapr" is the number of the APR that maps the base address of the page and "wnsiz" is the window size in 32-word blocks.

This macro also defines the symbols shown in figure 17. The monitor uses the window definition block to return information such as the window ID and status to the program. The program creates a virtual address window using the macro:

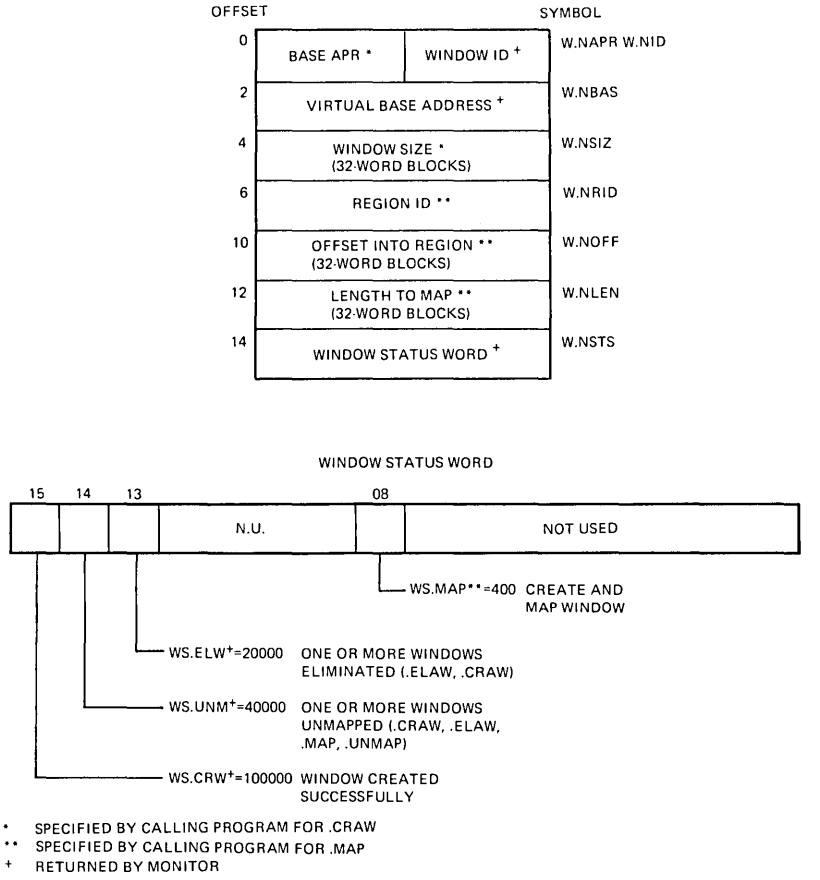
```
.CRAW   area,wnblk
```

Here "wnblk" is the address of a window definition block specifying the base APR to use for the window (0–7) and the size of the window (32-word blocks).

If the window is created successfully, the monitor returns the following information in the window definition block:

- The window number (ID)
- The virtual base address of the window
- Bit 15 set in the status word

Figure 17.
Window Definition Block



Each time the program creates a window, the monitor fills in a window control block in the job's impure area, which describes the window. The window ID returned in the window definition block is the number of the corresponding control block.

Mapping Windows to Regions

Although the program has created a region in physical memory and a virtual address window, the virtual ad-

dresses within the window will not be converted into physical addresses until the window is mapped to the region. Mapping is the operation that connects the virtual addresses within a window with a region in physical memory. In hardware terms, the mapping operation inserts values into the APRs so that the virtual addresses within the window will be converted into physical addresses within the appropriate region.

The program must specify the following information in the window definition block when mapping a window to a region:

- The region ID
- The offset into the region at which to start the map (in 32-word blocks)
- The length of the window to map (in 32-word blocks)

The map may start anywhere in the region and extend as far as needed. More than one window may be mapped to different parts of the same region, and different windows may be mapped to the same area in a region.

There are two methods by which a program may map a window to a region. First, the program may create and map a window in one operation. This is done by setting the bit WS.MAP in the window status word of the window definition block and then using the programmed request:

```
.CRAW   area,wnblk
```

Here "wnblk" is the address of the window definition block. The program must supply information in the window definition block necessary to map the window as well as its size and base APR. The program can do this by creating the window definition block with the macro:

```
.WDBBK  wnapr,wnsiz,wnrid,wnoff,wnlen,wnsts
```

Here "wnrid" is the region ID to which the window is being mapped; "wnoff" is the offset into the region at which the mapping starts; "wnlen" is the length of the window to map; and "wnsts" sets the WS.MAP bit in the window status word

to specify that the .CRAW request map the window after creating it.

In the second method the program can map a previously created window to a region using the macro:

```
.MAP    area,wnblk
```

Here, the program must load the necessary mapping information into the window definition block. Either way, once the window has been successfully mapped, the program can access physical memory locations using the virtual addresses in that window.

Job Mappings

When you run a job in an XM system, the monitor automatically creates a default mapping for the job in the user mode APRs. The monitor can create privileged or virtual mapping for a job. The job may specify the type of mapping it wants the monitor to create by using bit 10 in the job status word (JSW).

Privileged Mapping

When bit 10 in the JSW is zero (the default value), the monitor creates a privileged mapping. This mapping is identical to kernel mode mapping; the job's virtual address space is mapped to low memory (0–28K) and the I/O page. The monitor creates a privileged mapping by copying the contents of the kernel APRs into the user APRs. As a result, a privileged job has access to the physical vector area and the monitor.

A privileged job may change its mapping by creating up to seven virtual address windows and mapping them to dynamic regions in extended memory. When a privileged job maps one or more virtual address windows, the monitor removes the privileged mapping and maps the virtual addresses using the appropriate window control blocks.

When the job unmaps its virtual address windows, the monitor restores the privileged mapping.

Privileged mapping is the default in the XM systems in order to provide upward compatibility with SJ and FB systems. SJ and FB jobs will execute as privileged jobs in the XM system with no change. All RT–11 utility programs and the keyboard monitor (KMON) execute as privileged jobs in user mode in an XM system.

Virtual Mapping

If bit 10 in the JSW is set when the monitor loads the job into memory, the monitor will create a virtual mapping. With a virtual mapping, a job may use all 32 Kwords of virtual address space. On the other hand, a virtual job cannot access the monitor, the physical vector area, or the I/O page. To request a virtual mapping you must set bit 10 in the JSW, either with an .ASECT directive or with a patch to location 44 in the job's memory-image file.

The monitor always loads a virtual job into a region in low memory called the static region. The region ID for the static region is 0 because the region is defined by the first region control block. The job cannot modify the static region.

The monitor then creates a virtual address window called the static window (window ID 0), which extends from virtual address 0 to the job's high limit, and maps this window to the static region. A virtual job can access only those virtual addresses within the limits of the job; the job cannot modify the static window.

The static region for a virtual background job extends from physical location 500 up to the bottom of the USR. The job is loaded starting at location 500. The static window extends from virtual address 0 up to the job's high limit, and it is mapped to the static region starting at physical location 500. Virtual location 0 to 500 make up the virtual vector and system communication area.

The static region for a foreground job extends from virtual location 0 up to the job's high limit. The foreground job is loaded between the top of the USR and the last loaded

device handler or system job. The static window extends from virtual address 0 up to the job's high limit, and it is mapped to the static region starting at the first location above the impure area. As with virtual background jobs, virtual locations 0–500 are the virtual vector and system communication area.

A virtual job may use the virtual address space left between the job's high limit and the 32-Kword limit by creating virtual address windows (in that part of the virtual address space and dynamic regions in extended memory) and by mapping the windows to the regions. If the job tries to use a virtual address that has not been mapped, a memory management fault (trap) will occur.

Context Switching

When the monitor switches from one job to another, it saves the context of the job being switched out and restores the context of the new job. The context of a job in an XM system includes the following locations not saved in an FB system:

- Kernel PAR1
- BPT vector
- IOT vector
- MMU fault vector

The monitor does not save the contents of the APRs as a part of the job's context. It restores a job's mapping, using the information in the job's region and window control blocks. For this reason, a user job must always map using the memory management programmed requests. It must never change the APR's directly.

When the monitor switches to a new job, it starts by copying the contents of the kernel APRs into the user APRs. If the new job is KMON, it may start execution. If the job is privileged but not KMON, the monitor scans the region and window control blocks and restores any mapping the job may have created. If the job is virtual, the monitor clears

the registers and restores the mapping defined by the region and window control blocks. For more details about context switching in XM systems, refer to chapter 4 in the *RT–11 Software Support Manual.*

Synchronous System Traps and Virtual Vectors

A synchronous system trap (SST) is a system-generated interrupt that occurs synchronously with program execution. An SST can result from an error condition (bus time-out or illegal instruction) or a special instruction that generates a trap (TRAP, BPT, or IOT). Table 6 lists all of the SSTs with their trap vectors.

A user application in an SJ or FB system can service these traps by storing the address of the trap service routine in the first word of the appropriate vector, either directly or with a programmed request. When a trap occurs in an XM system, the CPU enters the trap service routine using the vector in kernel address space. This creates a

Table 6.
Synchronous System Traps and Their Vectors

Vector	Synchronous System Trap
4	Trap to 4, caused by a reference to an odd address or by a bus time-out.
10	Trap to 10, caused by an attempt to execute a reserved instruction.
14	Breakpoint trap, usually issued by a debugging utility program such as ODT.
20	I/O trap.
34	TRAP instruction, issued by a program to change the flow of execution.
114	Memory parity trap, caused by a memory parity error.
244	FPU trap, caused by a floating point unit exception or error.
250	Memory management trap, caused by a program's attempt to reference a virtual address that is not mapped to a physical address.

problem for virtual jobs that want to service an SST because the virtual vectors are mapped to physical memory starting at or above location 500. A virtual job cannot access the kernel vector area to modify vector 34, for example. The XM monitor solves this problem by passing control to the address specified in the virtual vector when an SST occurs. Therefore, a virtual job can service SSTs by modifying the appropriate virtual vector. Chapter 4 in the *RT–11 Software Support Manual* describes how the monitor processes each SST listed in table 6.

XM Bootstrap Action

Booting an XM system performs the following for the memory management system:

- Tests for the presence of a KT11 and prints an error message if it is absent
- Sets up both kernel and user APRs with the kernel mapping
- Enables the KT11
- Computes the size of extended memory in 32-word blocks

XM Program Applications

This section outlines some of the most common applications that use extended memory. It is planned as a guide only. For full details of a specific application, read the manual references given for that application.

XM Data Buffers

The simplest type of XM application is probably the program that stores one or more large data buffers or arrays in

extended memory. Chapter 4 in the *RT–11 Software Support Manual* indicates how to design this type of application.

XM Overlays

RT–11 supports program overlays in extended memory. These XM overlay segments may be either memory resident or disk resident. You may, therefore, implement a program that contains the following types of overlays in any combination:

- Disk-resident overlays in low memory
- Disk-resident overlays in XM
- Memory-resident overlays in XM

When you link your program you specify the overlay structure with the /O and /V options. /O:n specifies a disk-resident segment for an overlay region in low memory. /V:n specifies a memory-resident segment in XM for virtual overlay region n. /V:n:m specifies a disk-resident segment for virtual overlay region n that shares XM partition m.

The linker adds all of the code necessary to control your overlay structure in a transparent way.

Multiuser Applications

If you have generated an XM system that includes multiterminal support, you may implement a multiuser application using XM data buffers or XM overlays. Chapter 4 in the *RT–11 Software Support Manual* indicates a possible design using each method.

The XM .SETTOP Feature

The XM .SETTOP feature in RT–11 permits a virtual job to allocate additional virtual address space up to 32 Kwords

and map the additional addresses to an XM region in one simple operation. The virtual overlay handler uses this feature to create virtual overlays in extended memory. It is of little use to privileged jobs, as it will only allocate additional address space in low memory for these jobs.

To understand how the XM .SETTOP feature works, it is useful to review how .SETTOP works in an SJ or FB environment. Background or single-job tasks use the .SETTOP request to allocate additional memory to the job partition. The .SETTOP request will allocate memory up to the base of RMON if available. A .SETTOP to the base of the USR prevents swapping. The .SETTOP request is of no use to a foreground or system job unless memory has been previously allocated with the /BUFFER option of the FRUN or SRUN commands.

To understand how the .SETTOP request functions in an XM environment, you must understand the following concepts:

- The program high limit (PHL)
- The virtual high limit (VHL)
- The next free address (NFA)

The program high limit is the highest virtual address used by the root segment of the program or its low memory overlays, if there are any. The linker stores this address in location 50 when the program is linked. In memory, this location will always be lower than the base of the USR.

The virtual high limit is the highest virtual address used by any XM virtual overlays, rounded up to a 32-word boundary, minus 2. The octal address will always end in 76.

The next free address for programs with virtual overlays is the virtual high limit rounded up to the next 4-Kword boundary. The NFA will always be a multiple of 20000 (octal). For programs without virtual overlays, the NFA is PHL+2.

The result of a .SETTOP request in an XM environment depends on whether the XM .SETTOP feature is active. You enable the XM .SETTOP feature for a program

when you link the program. The following options enable this feature:

- LINK/V creates virtual overlays
- LINK/XM is for programs without virtual overlays

If the XM .SETTOP feature is not active, the .SETTOP request has limited value in an XM environment because it only allocates physical memory below 28K. It is of no use to virtual jobs. For a privileged job using the default mapping, .SETTOP functions as it does in an SJ or FB system. Even if the privileged job has mapped to extended memory, .SETTOP returns the highest available address below 28K.

When the XM .SETTOP feature is active, the .SETTOP request becomes a valuable tool for virtual jobs. When a virtual job issues a .SETTOP request, the .SETTOP allocates virtual address space to the job beginning at the NFA and automatically maps those virtual addresses to a dynamic region. With privileged jobs, the XM .SETTOP is useful only to background jobs because it only allocates memory up to the base of the USR in low memory. Thus, the XM .SETTOP feature provides a powerful and simple means for virtual jobs to create and map virtual address windows to dynamic regions without needing to go through each individual step. For more details on the XM. SETTOP feature, refer to chapter 4 in the *RT–11 Software Support Manual*.

Debugging an XM Application

You may debug both privileged and virtual application jobs in an XM environment using the virtual debugging tool (VDT), distributed as an object file VDT.OBJ. You use VDT in exactly the same way as ODT. Because VDT limits you to the memory mapped by the job you are debugging, you cannot access the monitor, the vector area, or the I/O page when debugging a virtual job. Refer to chapter 4 in the *RT–11 Software Support Manual* for more details on VDT.

Restrictions on XM Applications

Because of the way in which XM support has been implemented in RT–11, XM applications must conform to the following restrictions to assure correct operation:

1. Queue elements must be 10 words in length.
2. Channels allocated with .CDFN, queue elements allocated with .QSET, and interrupt service routines (ISRs) must reside in low memory.
3. User-allocated channels, queue elements, and ISRs must not be located in the virtual addresses mapped by APR1 (20000–37777) because the monitor may temporarily remap these addresses.
4. When the message handler (MQ) is active, user-allocated channels, queue elements, and ISRs must not be located in the virtual addresses mapped by APR2 (40000–57777) because MQ uses APR2. This restriction is in addition to restriction 3.
5. Virtual jobs may not use the .CNTXSW programmed request.
6. KT11 must not be changed by the program except through programmed requests.

Interrupt Service Routines in XM Systems

Chapter 8, "Writing an Interrupt Service Routine," describes how an SJ or FB user application program could service device interrupts in-line, as a part of the application. An XM application may also service interrupts in-line; however, there are additional restrictions when running under an XM system.

First, the ISR must execute within a privileged job with a mapping identical to the default (kernel) mapping. When an interrupt occurs in an XM system, the CPU transfers control to the ISR, using the interrupt vector in kernel ad-

dress space and kernel relocation for the ISR entry address. Therefore, the ISR must reside in low memory and must stay mapped while interrupts are enabled. In addition, it must be able to access the monitor and the I/O page and, therefore, must have a mapping identical to kernel mapping.

Second, the ISR must not use the virtual addresses mapped by APR1 (20000–37777) because the monitor changes APR1. In addition, if the MQ handler is active, the ISR must not use the virtual addresses mapped by APR2 (40000–57777).

Third, the code following a .SYNCH request must not use the virtual addresses mapped by APR1 (and APR2 if MQ is active) because the code executes using the user register contents, but with kernel mapping. For a more detailed discussion of these restrictions, refer to chapter 6 in the *RT–11 Software Support Manual.*

References

RT–11 Programmer's Reference Manual. Chapter 2 describes .SETTOP in detail.

RT–11 Software Support Manual. Figures 4–25 and 4–27 show the mapping of virtual background and foreground jobs. Figures 4–28 and 4–29 show privileged mapping for background and foreground jobs. Figure 4–33 illustrates the functions of .SETTOP under the XM monitor. Figure 4–41 shows the monitor filling in a region control block in a job's impure area. Figure 4–44 shows the monitor filling in a window control block in a job's impure area.

RT–11 System User's Guide.

RT-11 System Utilities Manual. Chapter 11 discusses the design and implementation of overlays.

7

7

Device Communication

This chapter describes the methods for transferring data to I/O devices that are not supported by the standard operating system. To establish communication between the system and these devices, you will learn to use programmed I/O, interrupt handling, interrupt service routines, and device handlers. You will also learn the benefits and disadvantages of each method and be able to determine the most efficient way to carry out I/O transfers.

The I/O Device

It is possible to connect a wide range of I/O devices to a PDP–11 computer system. For instance, you might want to connect a peripheral device that is not supported by the standard operating system or use a standard peripheral in a way not supported by the operating system. Under such conditions, you will have to write your own software to handle the I/O transfers between your programs and the device. The same means is used to communicate with a device, regardless of how simple or complex the interface is between the computer and the device.

To the PDP–11 hardware, the device appears to be no more than a set of control and data registers and an interrupt vector. It is these items that a program uses to communicate with the device. To the program, both the data registers and the interrupt vector are normal memory addresses.

The device's interrupt vector is made up of two words in the interrupt vector area that are set up by the program. The first word contains the address of the code that will receive control when an interrupt occurs, and the second carries the initial value of the PSW. If the program is certain that the device will not raise an interrupt, the interrupt vector may be ignored.

To the program, the device registers appear to be a few memory locations (160000 through 177777) found in the I/O page. The program reads and writes these registers to check the status of the device and to cause some action to be performed by the device.

Since each device has its own arrangement of registers and functional characteristics, the program has to issue commands to the device in a way acceptable to it. Certain operations may have to be performed in a specific order because each device has its own characteristic protocol. The program should also be capable of dealing with error conditions. When a device detects an error, it sets certain flags in its registers. These flags may cause the program to perform some special action, for example, trying the failing operation again or resetting the device.

The primary purpose in having a device connected to

the computer is to transfer data in one direction or the other. The main consideration is that the transfer take place as quickly as possible. The following sections examine the ways in which the transfer may be performed.

Communicating with the Device

The two methods of controlling a device are: programmed I/O and interrupt processing.

With interrupt processing, RT–11 provides a number of procedures to help the user's software perform interrupt-driven I/O. This support is provided for two forms of programming: interrupt service routines (ISR) and device handlers.

Programmed I/O

The simplest method of transferring data between memory and a device is to use noninterrupt programmed I/O, often called polling. Every device has a bit in its registers that indicates a ready state when set. Using this method, the program runs with the device's interrupts disabled and checks this bit in the device's registers. When the ready bit is set, the program may either transfer data or issue another command to the device. The program either waits in a loop for this bit to be set by the device or does other processing while the device is busy, checking the status of the device from time to time.

This method is very device specific and performs its task with no help from the operating system. This means that it cannot use any of the procedures provided by RT–11 for performing I/O, and it also ties up the CPU when it might possibly be better used. Nonetheless, there are times when programmed I/O is the best method to use. In conditions where the device's data is very volatile and time critical, the time needed by the system to respond to an interrupt may be unacceptable. A program loop with very few instructions could possibly service the device faster than

the interrupt system, but only if the program controlling the device were running as the highest priority job on the system and no other work could be done. It is usually better, therefore, to make use of the PDP–11 interrupt system.

Interrupt Processing

Interrupt processing is the second method of controlling a device. The program sets up the interrupt vector assigned to the device with the PC and PSW to be used by the CPU when the device causes an interrupt. The program next starts a data transfer by writing into the device registers. Then the program may either do some other useful work or let RT–11 pass control to another process.

When the transfer is complete, the device signals an interrupt. The CPU loads the contents of the interrupt vector into the PC and PSW, which starts up the interrupt service routine. The interrupt service routine, started in this way, checks the device status registers to see if any errors occurred. This routine may try the operation again, start another transfer, or set completion status flags within the main program. When the interrupt service routine has completed its processing of the interrupt, control returns to the activity that was interrupted via an RTI instruction.

The main benefits of interrupt-driven I/O are that it enables two or more processes to run at the same time and it allows the best use of a valuable system resource—the CPU. There is, however, something that you should bear in mind. RT–11 also uses the interrupt system. The operating system does not automatically know when an interrupt occurs. In order to keep the operating system running correctly, software that performs interrupt processing should keep RT–11 informed of each event. If this rule is not observed, RT–11 may not be able to respond to an interrupt in time. The total system interaction must be understood in order to synchronize device activities and avoid conflicts. For this reason, there are a number of macros in the system library that generate calls to RT–11 routines to help the interrupt service routines and the operating system cooperate. These macros provide support for two forms of in-

terrupt handler—the interrupt service routine and the device handler.

Interrupt Service Routines

An interrupt service routine (ISR), which is part of a program, does the interrupt processing for a device. The main program sets up the device's interrupt vectors to point at the routine and starts an I/O transfer by writing to the device registers. When the device generates an interrupt, the ISR takes control and performs the appropriate action.

Using an interrupt routine gives the program complete control of the device. The program has full access to all of the device's control and status registers as well as its data buffer registers. Because there is no operating system overhead when making an actual data transfer, an ISR responds to interrupts and transfers data between the device and the program at very high speed. Thus, the ISR provides the programmer with a great deal of flexibility. The programmer selects the way the data transfers are invoked, the amount of processing performed on the data, and the amount of information passed back to the main program.

The application program with the ISR is very device dependent. The program and the routine are linked into one job, which has exclusive access to the device. If there were a need to perform the same task using a different device, the ISR would have to be modified to take into account the differences in the device registers and vector addresses.

The use of an ISR enables only one application to communicate with the device. Thus, the device is not available to all of the programs on the system. The way a device is made more widely available is by use of a device handler, which we'll look at next.

Device Handlers

A device handler is the standard software interface between the RT–11 operating system and a peripheral de-

vice. It is stored on a mass storage device, usually the system device. When the operating system wants to communicate with a peripheral device, it loads the appropriate device handler into memory.

A program that wants to transfer data between itself and a device controlled by a device handler does so by asking the operating system to perform the transfer. The operating system passes the program's standard I/O transfer request on to the appropriate device handler. The device handler, which performs the transfer without any more action on the part of the program, starts the transfer by writing into the device's registers. The device handler has an interrupt service routine that processes interrupts generated by the device. Any tasks that must be performed at the end of the transfer, such as starting user completion routines, are performed by the device handler and the operating system.

Using a device handler to communicate with a device makes the device generally available to those programs on the system that use RT–11 programmed requests to perform I/O. Using these programmed requests also makes a program device independent because the form of the requests (.READ, .WRITE, and so on) is the same for all devices.

Because device handlers are the standard method of communicating with a device under RT–11, they are very simple to use. The operating system provides all of the program interfaces for the device handler. The procedure for writing a device handler is also very clear. RT–11 provides macros that help you write a device handler, as well as keyboard monitor commands that help you install the device handler and load it into memory.

Using a device handler to communicate with a random access device such as a disk has another important benefit. The RT–11 file structure is immediately available for use, with no additional effort.

8

8

Writing an Interrupt Service Routine

This chapter provides the system programmer with all of the information needed to write an interrupt service routine. First, we'll examine the general structure of an interrupt service routine and the actions such a routine may need to perform. Then we'll discuss the interfaces to the operating system that enable an interrupt service routine to perform its task without affecting other processes on the system. The programmed requests discussed in this chapter are: .PROTECT, .DEVICE, .INTEN, .FORK, and .SYNCH. Given the description of a device and an application, you will be able to implement a simple interrupt service routine that performs this task.

A program that uses the interrupt system may be divided into two parts. The first part, the main-line program code, is the program that performs all of the usual operations—modifying the general registers, issuing programmed requests, and accessing all of the available memory space. This ability to access memory at will also gives the main program access to the interrupt vectors and registers of the device. The second part of such a program is the routine started when an interrupt occurs—the interrupt service routine (ISR).

It is possible to write an ISR that performs all of its tasks without communicating with the operating system. This type of routine does its work quickly and then exits with an RTI instruction. The problem with using such a routine is that the time required to service the interrupt may be too long, locking out other processes, especially other interrupts. This could have a serious effect on the latency of the system, the time taken for the system to respond to an interrupt. This may not apply to the system for which the program is being written. For example, a dedicated data acquisition system running as the only job could be designed in this way. If, however, it became necessary to run the same program as a job under the FB monitor, the program might have to be modified to allow the other jobs to function correctly.

RT–11 provides a protocol for interrupt service routines that enables the operating system to optimize the performance of the system. You should follow the rules of this protocol when writing an ISR so as to have the least negative impact on the overall performance of the system.

RT–11 Interrupt Service Protocol

The RT–11 interrupt service protocol maximizes the response of the system to real-time events. It enables any ISR to move from one priority level down to the next, as the interrupt processing becomes less time critical. The lower the level of priority, the more freedom the ISR has. In return, as this ISR moves down to a less limited level, the rest of the system is more responsive to other events. The four levels in which an ISR can be running (from highest to lowest) are:

- Priority level 7
- Device priority
- Fork level
- Synch level

Moving from one level to another involves calling routines within the RT–11 operating system. Because the code of RT–11 is not reentrant, the initial acceptance of the interrupt must raise the CPU priority to the highest level—priority level 7. This means that the second word in the interrupt vector, the new PSW, must contain the octal value 340.

Some of the operations that we will describe are used by the FB and XM monitors but not by the SJ monitor. (Requests that are not required or used by the SJ monitor execute as no operation codes.) Nevertheless, you should always include these macros and programmed requests in your program so that the programs will run under any monitor.

Running at Priority Level Seven

When a device generates an interrupt, the CPU saves the current PC and PSW on the stack. It then loads the PC and PSW registers from the interrupt vector specified by the device. Because the second word of the interrupt vector contains octal 340, the CPU starts running at priority 7, blocking all other device interrupts. Keeping the CPU at this level for too long a time could result in loss of data from other sources, including the clock. So it is important that the priority level be lowered as soon as possible.

The main reason for taking the interrupt at priority 7 is to allow the ISR to inform RT–11 of the interrupt. All interrupts must be disabled when the ISR calls the .INTEN macro. This macro causes RT–11 to switch the CPU into system state and lower the CPU priority to the device priority. If, however, there is a sequence of instructions you need to perform without interruption, then the code should be placed before the .INTEN macro. The sequence should not be longer than about 50 microseconds, time for only a few instructions. Check the *PDP–11 Processor Handbook* for instruction timings. If the execution time at priority 7 is longer than this, the interrupt latency of the system will be degraded.

Running at Device Priority

Following the call to the .INTEN macro, the ISR continues at the priority you selected in the call. The ISR is now running in system state. The current stack is the system stack. Context switching is disabled. Interrupts at a lower or equivalent level are still blocked.

The instructions executed should now reset or prepare the device for the next transfer. This is where most of the actual device communication should be done. The device registers may be read and written as needed.

Because the ISR is running in system state, the ISR cannot issue any programmed requests. If the interrupt service needs a long time to perform additional processing, it should drop its priority to execute some of the operations at fork or synch level.

Running at Fork Level

Fork level may be entered when the ISR is running at device priority. This state is used by an ISR to perform additional operations in system state at priority 0. The state is changed to fork level by calling the .FORK macro, which causes an entry to be made on a queue called the fork queue. Execution of the ISR is suspended until all interrupts have been dismissed. Before RT–11 returns control to the user process, the entries in the fork queue are reactivated in the order in which they were queued. When the ISR proceeds from the .FORK macro as a fork process, it is again in system state but at priority 0 (interruptible). Context switching is disabled, but all interrupt levels are active.

Because fork processes are executed in the order in which they appear on the queue, they may be used to serialize access to a shared data structure or system resource. If the ISR needs to issue a programmed request, this must be done at synch level.

Running at Synch Level

An ISR enters synch level from either device priority level or fork level. Interrupt service routines execute asynchro-

nously in system state. Programmed requests are only made in user state with the correct job context. If you need to issue a programmed request from an ISR, you must first call the .SYNCH macro. The .SYNCH macro causes the ISR to be suspended until all interrupts have been dismissed and all fork processes have completed. When RT–11 is ready to return to the user process, the ISR is allowed to proceed.

When the ISR continues from the .SYNCH call, conditions are the same as with a completion routine for a standard RT–11 I/O request. The ISR is running in user state at priority 0. As such, the ISR is permitted to issue programmed requests within the context of the current job.

It could take a long time for the .SYNCH call to return to the ISR. Any fork processes must run to completion before the switch is made to user state. The actual switch needs a scheduling pass and possibly a context switch. As the ISR is now executing as a completion routine for the job, the ISR may also have to wait for any higher priority compute-bound jobs to become blocked.

Preparing for the Interrupt

There are two things that the main-line program must do:

- Set up the interrupt vectors with the PC and the PSW of the interrupt routine
- Prepare for transfers and set the interrupt-enable bit in the device registers

This may not be the only application program that uses this device. Therefore, the first thing that you should do is allocate the interrupt vector for the device with a .PROTECT request. This request asks RT–11 to make your job the owner of the interrupt vector for the device, so that no other job will be allowed to make use of it. If the .PROTECT request returns an error status, the interrupt vector has been given to another job or is in use by the operating system. Your program must not try to modify the vector. If the .PROTECT request is successful, your pro-

gram may set up the interrupt vector to point to the first instruction of the ISR.

Your program should also issue the .DEVICE request to make sure that the device stops if the program is aborted. This request asks RT–11 to write a given value into a given device register. The most common use for this feature is to disable interrupts when the program stops.

Your program should reset the device to put it into a known state and then set the interrupt-enable bit in the control status register (CSR) for the device to enable interrupts. When set, this may result in an immediate interrupt that may need to be ignored. For you to know what is valid at any given time, communication between the routine that sets up transfers and the ISR must be established.

There are two basic methods for handling interrupt-driven transfers. The first method is to initialize the vector and device prior to any transfers, leaving interrupts enabled for the duration of all transfers. The second method is to enable interrupts after a transfer has been set up, with the interrupts being disabled after the ISR services the current transfer. The best method is application and device dependent.

Once the device control has been initialized, the mainline program may now make transfer requests. Once completed with transfers, the device should be cleared. From the main program's point of view, the overall process is:

1. Initialize
2. Transfer
3. Clear

Lowering the Processor Priority

The ISR is entered with the processor running at priority 7, with all interrupts blocked. In order to allow a higher priority device to interrupt, you must lower the processor priority to a value that reflects the relative priority of your device. This is done by the .INTEN macro, which expands

into a call to the operating system routine $INTEN. This routine takes as its argument the priority level at which you want your code to continue running. Calling $INTEN informs RT–11 that an interrupt has occurred. The routine $INTEN forces a switch to system state and lowers the processor priority to the device priority given in the call.

The format of the .INTEN macro call is shown below. There are two expansions, depending on the presence or absence of the argument PIC. Either way, the important argument is the device priority you have selected. Location 54 is a pointer to the RT–11 routine $INTEN. If your code is not position independent, the following may be used:

```
.INTEN   PRIO
```

This macro expands to:

```
JSR      R5,@54
.WORD    ^C(PRIO*40)&340
```

If, however, you need to use position-independent code, you should use the other call:

```
.INTEN   PRIO,PIC
```

This macro expands to:

```
MOV      @#54,-(SP)
JSR      R5,@(SP)+
.WORD    ^C(PRIO*40)&340
```

When the $INTEN call returns, the ISR is no longer running as an interrupt routine but as a subroutine of the operating system.

At the point at which the interrupt service routine is entered, the registers and stack are those of the interrupted process. Both the stack and the registers must be preserved until the .INTEN macro is called. After the call, registers R4 and R5 are available for use. If you need to use the other registers, they must be saved and restored before you exit.

When you are running at device priority, you are using the system stack, so you should not place too much data on it. When you leave device priority level, the stack should be returned to the state it was in when the .INTEN call returned to the ISR.

Creating a Fork Process

The fork process is used in an interrupt service routine when the operations to be performed would hold up interrupts too long if performed at device priority but are too important to wait until synch level could be entered.

A fork process is created when the ISR is running at device priority and issues a .FORK request. This request calls an RT–11 routine that places a packet called a fork block on the fork queue. The ISR is then suspended until all interrupts have been dismissed. The .FORK request is accepted only if the stack's setup is the same as it was after the return from .INTEN. Also, your program must provide the fork block and set up a pointer called $FKPTR.

The fork block is a block of four words. The first word will be used as the link in the queue. This word must be zero when you make the .FORK request. The second word holds the address at which the ISR will continue as a fork process. The third and fourth words are used to save R5 and R4.

The fork block cannot be used again until the fork process starts to execute. If the device handler is entered again in response to another interrupt from the device before the fork process has started, the link word in the fork block will not be zero. If this happens, the ISR will not be able to use this block to queue another .FORK request.

The pointer $FKPTR must be set up in the main program to point at the routine within RT–11 that queues the block. The address of this routine is found in monitor fixed offset 402.

The .FORK request takes the fork block as its argument.

```
.FORK    FKBLK
```

The macro expands to:

```
JSR      R5,@$FKPTR
.WORD    FKBLK-.
```

The instructions that set up $FKPTR will be in the main program. The best way to set up this pointer is to use the following sequence of instructions:

```
        .GVAL   #AREA,#402
        ADD     @#54,R0
        MOV     R0,$FKPTR
        .
        .
        .
AREA:   .BLKW   2
$FKPTR: .WORD   0
FKBLK:  .WORD   0,0,0,0
```

When the .FORK request returns, the ISR is running in system state at priority 0. Registers R4 and R5 have been preserved across the call. Registers R0 through R3 are available for use after the call.

Issuing Programmed Requests

RT–11 programmed requests are issued only from user state. An ISR enters user state from either device priority level or fork level by calling the .SYNCH macro. The .SYNCH macro has two arguments. The first is the address of a seven-word synch block. The second, if present, causes the macro to generate position-independent code. This macro is unusual in that it returns to the instruction following the call only if the call failed. If the call was successful, control is returned to the second word following the call. The general form of the request is:

```
        SYNCH   #SYNBLK[,PIC]
        BR      SYNERR
SUCCESS:.
        .
        .
```

The macro generates a call to an RT–11 routine that stores the successful return address of the ISR in the last word of the synch block. The routine then enters the block in the completion queue of the specified job.

There are two locations in the synch block that must be set up by the program. The job number must be placed in the second word. The value placed in the fifth word will be loaded into R0 when a successful return is made from .SYNCH.

When the .SYNCH call returns successfully, the ISR is running as a completion routine in user state at priority 0. Registers R0 and R1 are available for use. You should note that R4 and R5 should be saved before issuing the .SYNCH since they are not saved across the .SYNCH call. The ISR is now allowed to issue programmed requests.

Leaving an Interrupt Service Routine

When the ISR has completed all of the processing needed to service an interrupt, it must return to RT–11. Once the ISR has executed the .INTEN macro, it is running not as an independent interrupt routine but as a co-routine of RT–11. The ISR must not issue the RTI instruction after executing .INTEN, .FORK, or .SYNCH because RT–11 will have to restore some registers first.

The way to leave an ISR running at any level—device priority level, fork level, or synch level—is to issue the instruction:

```
RTS     PC
```

This returns control to the operating system, which restores any registers it had saved. It is important that you make sure that all registers and the stack are restored to their initial state when you leave, that is, pop all data from the stack that you pushed.

Planning the Interrupt Service Routine

Careful planning is the most important part of writing an interrupt service routine. While every device and application have their own unique features, it is very important that you:

- Study and understand all aspects of the operation of the device
- Study and understand the structure of an interrupt service routine
- Consider how including interrupt service routines in the application program will affect the structure and operation of the overall application
- Align the logic flow and methods of communication between the main program, the transfer initiator, and the ISR

Once you have completed these planning phases in an organized manner, you may write, test, and debug the code.

References

PDP–11 Processor Handbook.

RT–11 Software Support Manual. Chapter 6 describes interrupt service routines in detail.

9

9

Writing a Simple Device Handler

This chapter describes the principles involved in writing a simple RT–11 device handler. The shape of a device handler is discussed in detail along with the use of the essential macro definitions. The macros discussed in this chapter are: .DRDEF, .DRBEG, .DRAST, .FORK, .DRFIN, and .DREND. Given a description of the operation of a device, you will be able to write, install, test, and debug a simple RT–11 device handler.

RT–11 Device Handlers

An RT–11 device handler is the standard software interface between the device and the RT–11 operating system. A device made available to RT–11 through a device handler is also available to any programs on the system that use RT–11 programmed requests to perform I/O.

The device handler is stored as a memory-image file on a mass storage device. When the operating system needs to communicate with a device, the appropriate device handler must be loaded from the file into memory. You can load the device handler by the programmed request .FETCH or by the console command LOAD. Once the correct device handler has been loaded, utility programs or user programs can access the device by using RT–11 queued I/O programmed requests.

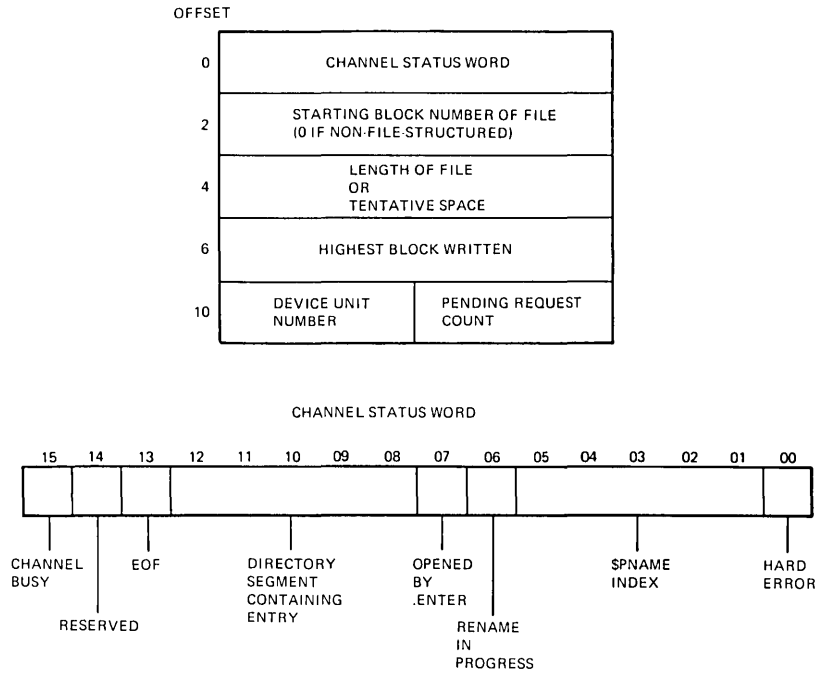
Queued I/O

Before a program issues I/O requests, it must open a channel to a file or a device with the .LOOKUP or .ENTER programmed requests. Logically, a channel is a unique path to a device or a file on a device. A channel is identified by a number specified in the request to open the channel. After the channel is open, all I/O requests refer to the channel by this number. Physically, a channel is a data structure in memory called a channel status block (CSB). When a channel is opened, the operating system fills in the CSB with information taken from both the programmed request and the device.

A channel status block, shown in figure 18, is a block of five words that contains all of the file and device information needed for the system to identify a file uniquely. The first word of the channel status block records the current status of the channel. The channel status word is also shown in figure 18.

An application program requests an I/O transfer by issuing an RT–11 programmed request (.WRITE, .READC, etc.). The request passes an EMT argument block to the res-

Figure 18.
Channel Status Block

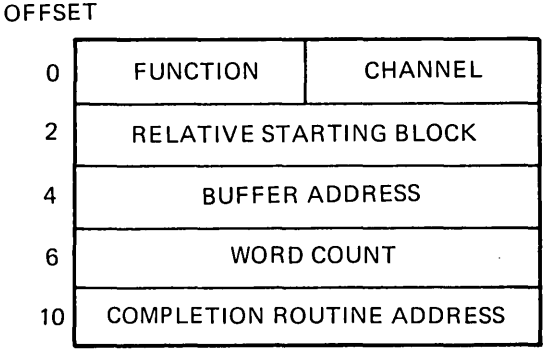


ident monitor RMON. The format of an EMT argument block is shown in figure 19.

RMON builds an I/O queue element using the information contained in the argument block and the CSB. The format of an I/O queue element is shown in figure 20. RMON then adds this queue element to the appropriate device handler queue.

Each device handler has its own queue of I/O requests. Two one-word locations in the device handler are pointers to the first and last elements in the queue. The monitor uses these pointers to add queue elements to the handler queue. Queuing I/O requests for each device handler permits I/O transfers to execute either synchronously or asynchronously with the main program. If the program had requested an asynchronous transfer, RMON returns to the program as soon as the element has been queued. If, however, the request had been for a synchronous transfer, the

Figure 19.
EMT Argument Block



monitor blocks the job and requests a scheduling pass to run another job.

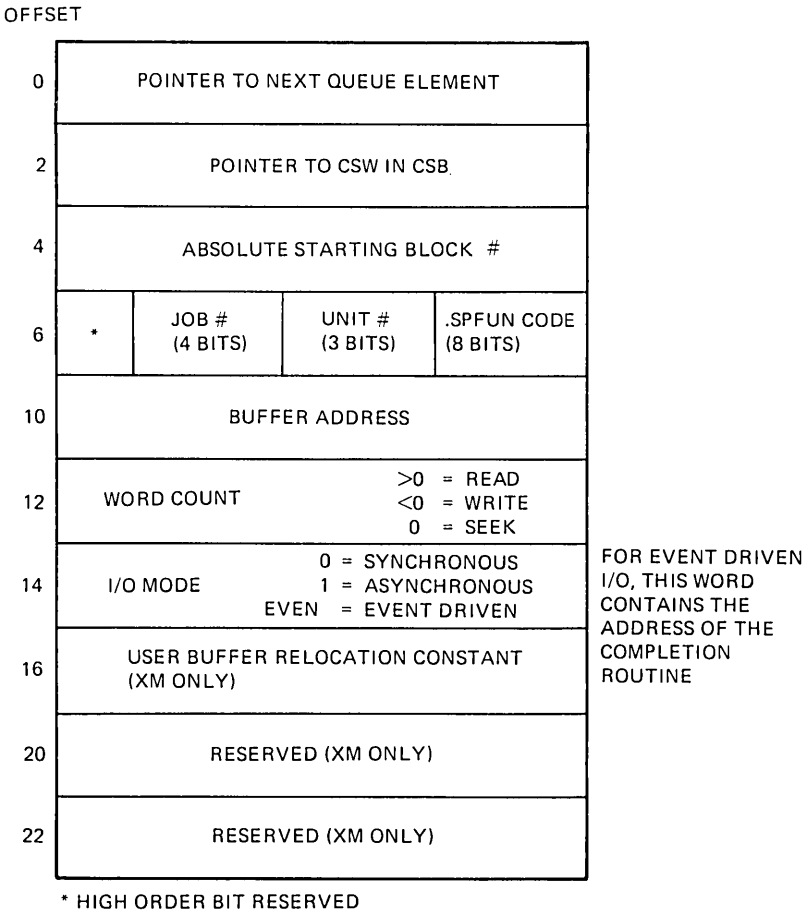
When the device is ready to perform another operation, the device handler takes the element at the front of its queue and issues the appropriate commands to the device. After the operation has been performed, the device handler checks to see if any errors occurred.

On completion of an I/O transfer, the device handler calls the I/O completion routine in the monitor. The monitor removes the old queue element from the queue and makes the device handler process the next request in the queue (if one is present). The monitor then uses the old queue element to start the user completion routine if there is one. The queue element is returned to the list of available queue elements.

Structure of a Device Handler

As stated before, an RT–11 device handler is stored in a file that must be structured in a certain way. Any device handler that conforms to this standard method for supporting a peripheral device can be loaded by RMON. There are a number of macros in the system macro library that make device handlers easy to write by providing the proper format for each element.

Figure 20.
I/O Queue Element

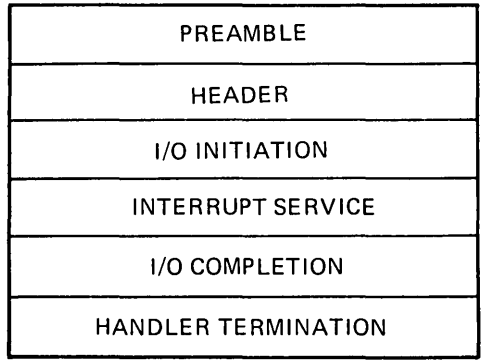


An RT–11 device handler consists of six parts, as shown in figure 21. Each part contains macro calls and code to perform a particular function:

- Preamble — symbol definitions and .MCALL directives
- Header — interrupt vectors and queue pointers

• I/O initiation	entry point to start a new I/O transfer
• Interrupt service	entry point for device interrupts
• I/O completion	handler exit path back to the monitor
• Handler termination	table of monitor entry points

Figure 21.
The Six Parts of a Device Handler



This chapter examines these parts in detail as well as the macro calls that you will use when you write a device handler for RT–11. Figure 22 shows the macros used in each of the six sections of a device handler.

Figure 23 shows a skeleton outline of a device handler. A user-chosen, two-character name for the device is used often in the code as a prefix to a symbol. This prefix is represented by the letters "dd" (for example, ddCQE). In the skeleton outline, the device name is SK, so ddCQE becomes SKCQE.

It is important to write the device handler in position-independent code because it may be placed anywhere in memory when loaded by the operating system. The *PDP–11 Processor Handbook* contains a section that discusses position-independent code.

Figure 22.
Device Handler Macros

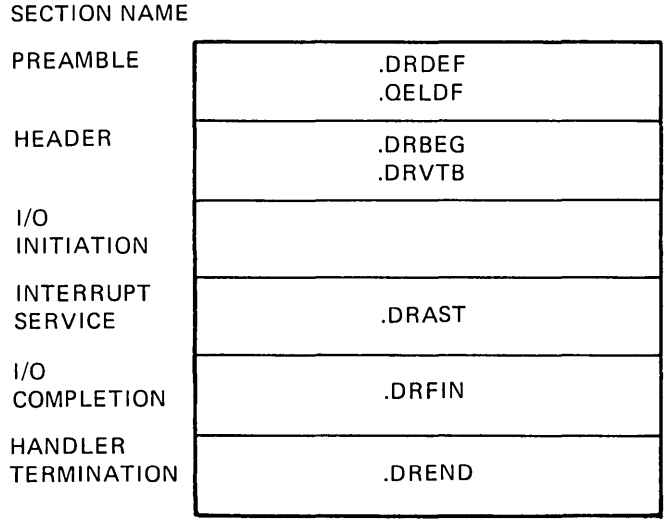


Figure 23.
Skeleton Device Handler

```
.TITLE  SK      V05.00

;  SK DEVICE HANDLER

.IDENT  /V05.00/
.SBTTL  PREAMBLE SECTION

.MCALL  .DRDEF
.DRDEF  SK,377,WONLY$,0,177514,200

SKBR    = SK$CSR+2              ;SK BUFFER REGISTER
SKIE    = 100                   ;INTERRUPT ENABLE BIT

.SBTTL  HEADER SECTION

        .DRBEG  SK

.SBTTL  I/O INITIATION SECTION

        MOV     SKCQE,R4        ;R4 POINTS TO CQE
        ASL     Q$WCNT(R4)      ;MAKE WORD COUNT BYTE COUNT
        BEQ     SKDONE          ;A SEEK COMPLETES IMMEDIATELY
        BCC     SKERR           ;THIS IS A WRITE-ONLY DEVICE
RET:    BIS     #SKIE,@#SK$CSR# ;ENABLE INTERRUPTS
        RTS     PC              ;WAIT FOR ONE
```

Figure 23. (continued)

```
.SBTTL  INTERRUPT SERVICE SECTION

        .DRAST   SK,4,SKDONE
        MOV      SKCQE,R4          ;R4 POINTS TO CQE
        BIT      #100200,@#SK$CSR  ;ERROR OR READY?
        BMI      RET               ;ERROR-HANG UNTIL CORRECT
        BMQ      RET               ;NOT READY - EXIT AND WAIT
        BIC      #SKIE,@#SK$CSR    ;DISABLE INTERRUPTS
        .FORK    SKFBLK            ;PROCESS REMAINING CODE AT
                                   ;FORK LEVEL
        ADD      #Q$WCNT,R4        ;OFFSET QUEUE ELEM POINTER

SKNEXT: TSTB     @#SK$CSR          ;READY FOR NEXT CHAR?
        BPL      RET               ;NO - BRANCH BACK
        TST      @R4               ;ANY LEFT TO PRINT?
        BEQ      SKDONE            ;NO - TRANSFER IS DONE
        MOVB     @-(R4),R5         ;GET A CHARACTER
        INC      (R4)+             ;BUMP BUFFER POINTER
        INC      @R4               ;BUMP CHARACTER COUNT
        BIC      #^C<177>,R5       ;7-BIT ASCII
        MOVB     R5,@#SKBR         ;SEND CHAR TO DEVICE
        BR       SKNEXT            ;TRY FOR ANOTHER

.SBTTL  I/O COMPLETION SECTION

SKERR:  BIS      #HDERR$,@-(R4)    ;SET ERROR BIT IN CSW
SKDONE: BIC      #SKIE,@#SK$CSR    ;DISABLE INTERRUPTS
        .DRFIN   SK                ;JUMP TO MONITOR

SKFBLK: .WORD    0,0,0,0           ;FORK QUEUE ELEMENT

.SBTTL  HANDLER  TERMINATION SECTION

        .DREND   SK
.END
```

Preamble

The first section in the device handler, the preamble, contains definitions that will be used by the other five sections. It starts with a .MCALL directive for the .DRDEF macro and a call to this macro. You should follow this call with any other symbol definitions that you need.

The .DRDEF macro is called in this section to issue .MCALL directives for the other device handler macros. The

.DRDEF macro also defines symbols for device characteristics and offsets into device handler data structures. The use of the .DRDEF macro is shown in chapter 2 of the *RT–11 Programmer's Reference Manual* and is discussed in detail in chapter 7 of the *RT–11 Software Support Manual*.

Header

The second part of an RT–11 device handler, the header section, is the first place that any code is produced. The header section contains a call to the .DRBEG macro, which takes one argument, the two-character name of the device. The .DRDEF macro must have been called before this macro because the .DRBEG macro uses some of the symbols that the .DRDEF macro defines.

The .DRBEG macro sets up the first five words of the handler. The data in these five words is shown in figure 24. This macro also stores some data in block 0 of the device handler file. This data, shown in figure 25, is used by the operating system to load the handler into memory.

Figure 24.
Device Handler Header Words

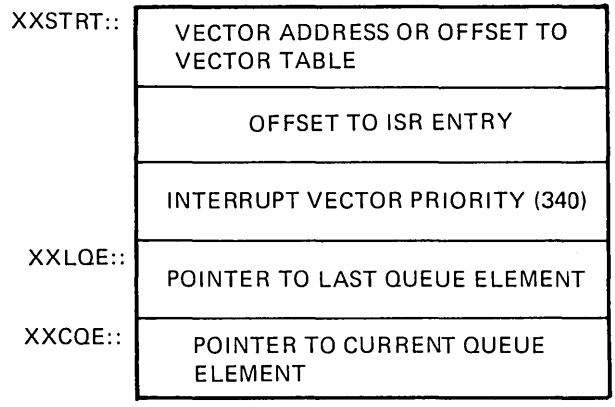
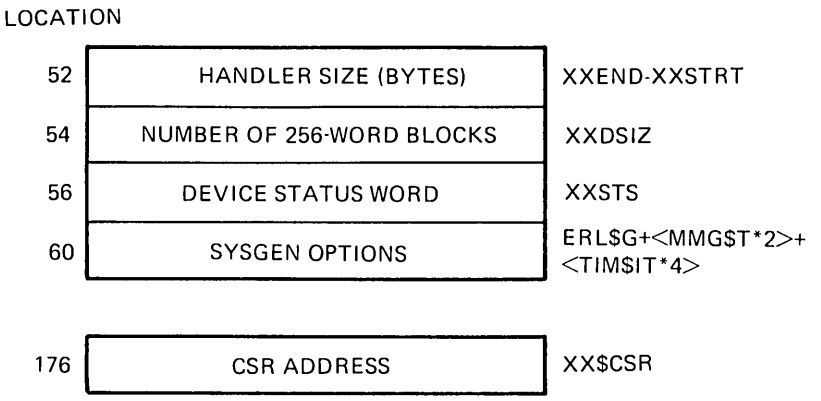


Figure 25.
Information in Block 0 of the Device Handler



I/O Initiation

The third part of the device handler, the I/O initiation section, contains the first executable instructions of the device handler. This section is called by the operating system to start a data transfer using the information contained in the queue element at the head of the queue. The handler is entered by a JSR PC instruction at the first location after the header. The CPU will be running in system state at priority 0. All registers are available for use by the device handler. A normal device handler does not need to be reentrant because the queued I/O system ensures that the next request will not be passed to the device handler until the previous one has completed.

The queue element your device handler will be expected to work on will be pointed at by the fifth word of the handler. The address in this location, ddCQE::, will be the address of the third word of the queue element Q.BLKN. The device handler may reference the fields of the queue element using the queue element offset definitions of the form Q$xxxx provided by the .DRDEF macro.

The I/O initiation section should check the request to ensure that it is valid. If there is something wrong with it,

the device handler should branch to the I/O completion section and signal a fatal error (by setting the hard error bit in the channel status word).

If the request seems to be correct, the device handler should issue commands to the device to start the transfer and then return to the operating system to wait for the interrupt. Some requests may not use interrupts; these should simply perform their operation and branch to the I/O completion routine.

Interrupt Service

There are two entry points in the interrupt service section of the device handler. The interrupt entry point is entered when the device generates an interrupt. The abort entry point is called by the operating system when the job that issued the current request asks to abort or is forced to abort. The .DRAST macro sets up both of these entry points. When the device generates an interrupt, the processor enters the device handler at priority level 7. As with interrupt service routines, the first step that must be taken is to lower the priority. This is another thing that the .DRAST macro does for you. The format of the .DRAST call is:

```
.DRAST   dd,pri(,abort)
```

In this call:

dd	is the two-character device name
pri	is the priority at which the interrupt service routine is to execute (the device priority)
abort	is the optional symbolic address of the abort entry in the handler

Interrupt control starts in the handler at the instruction following the call to the .DRAST macro. The processor is running in system state at the priority given in the call. Only R4 and R5 are available for use. Any other registers

must be saved before use and restored before returning to RT–11 through either an RTS instruction or a request. The interrupt routine has full access to the queue element (pointed to by ddCQE) and can use it to complete the transfer.

If an error has occurred, the handler should retry the operation. If the error cannot be found, the device handler should move on to the I/O completion section, signaling a fatal error.

If there are more characters or blocks of data to transfer, the device handler may restart the device and return to RT–11 (by means of an RTS PC instruction) to wait for the next interrupt. If the interrupt signals the completion of the requested transfer, the device handler may transfer control to the I/O completion section.

At any point in the interrupt service routine, the device handler may switch to a fork process by issuing a .FORK request. This macro takes as an argument the address of a four-word block called the fork block. The fork process suspends execution of the device handler until all other interrupts have been dismissed. An example of the .FORK request and its expansion follows.

EXAMPLE

```
        .FORK   RKFBLK
        JSR     R5,@$FKPTR
        .WORD   RKFBLK-.
                .
                .
                .
                .
RKFBLK: .WORD   0,0,0,0
```

The pointer $FKPTR is defined by the .DREND macro described below. The actual address of the routine is filled in when the device handler is loaded into memory by the monitor.

Following the .FORK call, the processor is running in

system state at priority 0. Registers R4 and R5 are preserved across that call. All registers are available for use when the .FORK macro returns.

Fork level is useful for performing retries when an error occurs during a transfer. It is important to remember, however, that a fork block is queued when the fork process is requested, and it cannot be used again before the fork process has completed. Thus, the fork request must not be used if the routine might be reentered in response to another interrupt that would use the same fork block. The fork request also should not be used with a device having continuous interrupts that cannot be disabled.

I/O Completion

The I/O completion section of the device handler is a common exit path from the device handler to the monitor I/O completion code. The device handler should transfer control to this section in the event that:

- A fatal error has occurred
- A recoverable error has exhausted its retry count
- A data transfer is complete

The code in this section must inform the monitor of the conditions under which the device handler has finished processing the current queue element. Chapter 7 of the *RT–11 Software Support Manual* gives details of the methods for flagging errors and other completion conditions.

The I/O completion section transfers control back to the operating system by calling the .DRFIN macro. This macro generates position-independent code that performs a jump into the operating system. When the operating system is given control by this macro, it releases the current queue element and takes care of any completion routines that the user may have requested.

Handler Termination

The last section in the device handler, handler termination, is a call to the .DREND macro. This macro marks the end of the device handler, allowing its size to be known. The .DREND macro also creates a table of pointers to routines in the RT–11 operating system. This table is filled in when the device handler is loaded.

Installation, Testing, and Debugging

Before you install and test a new device handler, you must assemble and link the source file to create an executable memory-image file. A device handler intended for the XM operating system has to be treated differently.

Assuming that the device handler for the device dd is in a file called dd.MAC, for the SJ and FB monitors use commands in the form:

```
.MACRO/CROSS/LIST SYSGEN.CND+dd/OBJECT
.LINK/MAP/EXEC:dd.SYS dd/NOBITMAP
```

But for the XM monitor, use:

```
.MACRO/CROSS/LIST XM+SYSGEN.CND+dd/OBJ:ddX
.LINK/MAP/EXEC:ddX.SYS ddX/NOBITMAP
```

Note that when assembling the source file for the device handler, you should include the system conditional file SYSGEN.CND (or the name entered during SYSGEN) if the monitor was created by a system generation. This ensures that the handler includes the same system generation options as the monitor. You will need the file XM.MAC when assembling a device handler for the XM monitor. Chapter 10, "Additional Features for Device Handlers," discusses how device handlers are affected by system generation features.

Installing a Device Handler

Once you have a memory-image device handler file (dd.SYS or ddX.SYS) on your system device, you should install the handler, either manually using the console INSTALL command or automatically by booting the system device. It is simplest to install the handler manually, using the command INSTALL dd.

If the system device tables do not have a free slot, you must use the REMOVE command to remove an installed handler in order to make room for your new one. You may also install the new device handler by booting your system device. The bootstrap will install the new handler provided that all of the following conditions are satisfied:

- The hardware driven by the new handler is present on your system.
- The file xx.SYS (or xxX.SYS) is on the system device.
- There is a slot in the system device tables to hold dd.
- The SYSGEN options in word 60 of the handler file match the system being booted.

See chapter 7 in the *RT–11 Software Support Manual* for a more detailed description of the various methods of installing a new device handler.

Testing and Debugging a Device Handler

Once the device handler is installed, you are ready to start testing it. Chapter 7 of the *RT–11 Software Support Manual* lists the steps you should follow when testing and debugging your device handler. This list includes these steps:

1. Use ODT to trace the processing of an I/O transfer through the handler.

2. Test the handler with as many system utility programs and KMON commands as possible.
3. Give the handler as heavy a workout as possible with an application program using all three I/O modes.

References

PDP–11 Processor Handbook.

RT–11 Programmer's Reference Manual.

RT–11 Software Support Manual. Chapter 7 describes RT–11 device handlers. Appendix A analyzes several distributed device handlers.

System Generation Conditionals
Multiple Vector Support
Internal Queuing
Device I/O Timeout
Set Options
Error Logging
Extended Memory Support
Special Functions
System Device Handlers
References

10

10

Additional Features for Device Handlers

This chapter describes optional features that may be added to an RT-11 device handler, as well as the methods by which these features are included. Among the features discussed in this chapter are: multiple vector support, internal queuing, device I/O timeout, SET options, error logging, extended memory support, special functions, and system device handlers. You will learn to write device handlers that do the following: support devices with multiple interrupt vectors, use internal queues, make use of the RT–11 device timeout routines, have SET options, include error logging support, can transfer data to extended memory, can perform special functions, and can be used to bootstrap the system.

In chapter 9, "Writing a Simple Device Handler," we looked at a simple device handler and the macro calls needed to make it work. In this chapter we will look at more macros and monitor routines that are used in an RT–11 device handler for special purposes.

System Generation Conditionals

Some optional features depend on system generation conditionals and will work only if the appropriate system generation conditional is enabled. These features include:

Device I/O timeout	depends on TIM$IT
Error logging	depends on ERL$G
Extended memory support	depends on MMG$T

If you want the device handler to work in all conditions, you should use conditional assembler directives to generate different code depending on the system generation conditionals.

EXAMPLE

```
.IF EQ   MMG$T          ;If no memory management
MOV      (R5)+,-(R4)
.IFF                    ;If memory management
JSR      PC,@$MPPTR
MOV      (SP)+,-(R4)
MOV      (SP)+,R0
BIT      #1700,R0
.ENDC                   ;EQ MMG$T
```

When you assemble the device handler, you must include the files defining system generation conditionals, so that the device handler can support those features that are also supported by the system.

Multiple Vector Support

The device handler discussed in chapter 9, "Writing a Simple Device Handler," was designed to work with a de-

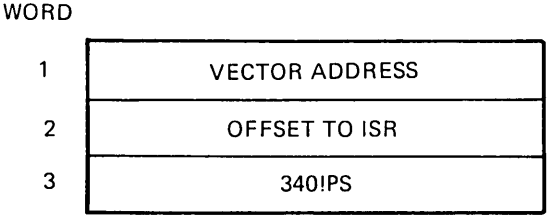
vice that had only one interrupt vector. Some devices have more than one vector. For example, the paper tape reader/punch has two vectors—one for the reader (vector 70) and another for the paper tape punch (vector 74). When the device has more than one interrupt vector, the device handler must specify the following information for each interrupt vector:

- The address of the interrupt vector
- The interrupt entry point in the device handler
- The processor status (PS) value

These values are given by using the .DRVTB macro to set up a table of three-word entries. The format of an entry is shown in figure 26. Each occurrence of .DRVTB defines one interrupt vector. You supply one call to .DRVTB for each interrupt vector used by the device. Chapter 7 in the *RT–11 Software Support Manual* describes the format and use of the .DRVTB macro in detail.

The .DRVTB macro calls may be placed anywhere between the .DRBEG and .DREND macros but must not be placed in the path of the instructions. When an interrupt is received from the device, the correct interrupt vector is used to transfer control to the appropriate entry point within the device handler.

Figure 26.
Multiple-vector Table Entry



Internal Queuing

Some device handlers perform more than one operation at the same time. The RT-11 queued I/O system, however, will not pass the next request to the device handler until it has completed the last request. To allow multiple requests to be active at the same time, the device handler is written in such a way that it queues requests internally.

The method used by the device handler is as follows. When RT–11 adds a queue element to an empty device handler queue, it calls the I/O initiation code of the device handler to start the request. If the request is invalid, control is passed to the .DRFIN macro to return a hard error. If the request is one that can be performed by the initiation code, the request is completed and the device handler exits via .DRFIN. In both cases, the queue element is returned to the operating system immediately.

Should the request be one that would require a long time to complete and could best be done asynchronously, the device handler adds the queue element to an internal queue. The next step is to clear ddCQE and ddLQE, making the device handler queue appear empty. The queue element is not returned to the available list. This ensures that the next request RT–11 adds to the device handler queue will once again be the first request and the I/O initiation section will be called. If the request that has been added to the internal queue is the first, the device handler starts the operation. The I/O initiation code exists with an RTS PC instruction.

On completion of an operation, the interrupt service code looks at the internal queues to determine which request caused the interrupt. The device handler must return the queue element to the operating system when the request is completed. This is done by setting the pointers ddLQE and ddCQE to point at the element. The .DRFIN macro passes the element back to RT–11 but the device handler is not able to continue execution. This is a problem because there may be other requests on the internal queue waiting to be started.

There is another way of accomplishing the same thing a .DRFIN macro does. Chapter 7 of the *RT–11 Software*

Support Manual shows the code that may be used in place of the .DRFIN macro to allow the device handler to continue running.

The last consideration with internal queuing is the abort procedure. When a job aborts, all the requests that are queued for I/O are also aborted. This is normally done by RT–11 by calling the device handlers at their abort entry points; however, the device handler is only called if there is an active queue element that came from the aborting job.

Handlers that queue internally do not keep the queue elements where RT–11 can locate them. To preclude this problem, there is a special bit in the device status word that has the symbolic name HNDLR$. If this bit (bit 11) is set in the device status word, the device handler is always called at its abort entry point whenever a job aborts. The handler must return all of the queue elements of the aborting job to the operating system. Chapter 7 in the *RT–11 Software Support Manual* describes the abort procedure to be followed by a handler that queues internally.

Device I/O Timeout

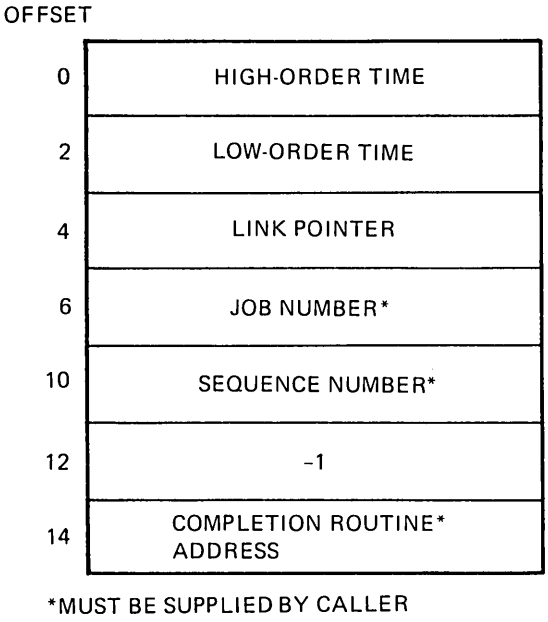
Device I/O timeout is a method of taking action if an interrupt does not occur within a certain time. This is useful for checking whether a device is ready to be used. For example, a line printer will not interrupt if it is not switched on.

The device handler may request that a completion routine be run if an interrupt does not occur within a specified time. This is the equivalent of a mark-time request.

Device I/O timeout needs some support from the operating system. This is an optional feature selected by system generation. If you enable device I/O timeout support when performing a system generation, the system generation conditional TIM$IT is set to 1. Two macros are called by a device handler that uses timeout support. These macros may be used within a device handler at any time except while the processor is running above priority level 0.

The .TIMIO macro is used after the device handler has started an operation that is expected to generate an interrupt when it completes. The main argument to the .TIMIO

Figure 27.
.TIMIO Timer Queue Element Block



request is the address of a seven-word timer queue element. The format of this element is shown in figure 27. This block contains:

- The time, specified as the number of clock ticks
- The job number of the request
- The sequence number of the request
- The address of the completion routine

If the time passes without an interrupt, the completion routine is executed in the context of the specified job. The sequence number of the .TIMIO request is passed to the completion routine in R0. If the device handler does receive the interrupt before the completion routine runs, the .CTIMIO macro is called to cancel the timer request. Chap-

ter 7 of the *RT–11 Software Support Manual* describes the use of device I/O timeout in detail.

Set Options

Giving a device handler SET options allows an operator to change some characteristics of a device handler after it has been assembled. The console command that the operator uses is in the form:

SET dd [NO]keyword[=value]

EXAMPLE

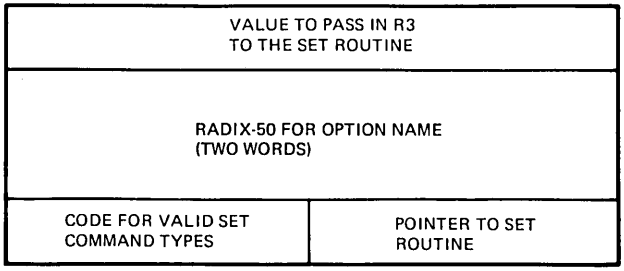
```
.SET LP CR
.SET LP NOCR
.SET LP WIDTH=80
```

When a SET command is issued, the monitor looks for the device handler file dd.SYS (ddX.SYS under XM) and reads blocks 0 and 1 into memory. Block 0 contains a table of valid option keywords and the forms the options may take. Also in block 0 are routines that implement the options by changing the code and data in blocks 0 and 1 of the handler file. When the entry for the keyword is found, the routine to support that option is called. After the routine has executed, the monitor writes the blocks back into the file.

The table of options starts at location 400 in block 0 and terminates with a zero word. Figure 28 shows the arrangement of each four-word entry in the table. The table is set up by calls to the .DRSET macro. The form of the .DRSET macro is:

.DRSET option,val,rtn[,mode]

Figure 28.
SET Option Table



In this call:

option	is a keyword of up to six alphanumeric characters
val	is the value to be passed to the routine in R3
rtn	is the name of the routine to process the option
mode	describes the forms the option may take

The mode argument is optional and takes the form of a keyword or list of keywords. The word NO indicates that the NO prefix is allowed; NUM specifies that a decimal value must be provided; and OCT states a need for an octal value. The words may be combined, as in ⟨NO,NUM⟩ or ⟨NO,OCT⟩ where the NO prefix may be used, but a value must also be provided.

EXAMPLE

If the macro call:

```
.DRSET  FORM,66.,SETERM,<NO,NUM>
```

appeared in the LP handler, the following commands would be acceptable from the console:

```
.SET LP FORM=20
.SET LP NOFORM=0
```

The routines to process the options should follow the calls to the .DRSET macro. Both the option table and all of the routines must be contained completely in block 0. Information is passed to a routine in the registers as follows:

R0 the numeric value from the SET command (if any)

R1 the device unit number from the SET command (If no unit was given, the sign bit is set.)

R3 the val word from the option table entry

If the carry bit is set when the routine returns to the monitor, an error message is printed and the blocks are not written back into the device handler file. Chapter 7 of the *RT–11 Software Support Manual* discusses SET options in more detail and gives some examples.

Error Logging

Error logging is a method used by a device handler to monitor the performance of the device. The device handler passes information to the error logger after each transfer. The error logger keeps a record of device activity to check the performance of the device. The error logger runs under the FB or XM monitor as either a system job or normal foreground job. For SJ, the error logger is a pseudo device handler. Error-logging support is selected by performing a system generation. Two system generation conditionals are involved in error logging:

- ERL$G is set to 1 if error logging is enabled.
- ERL$U is set to the maximum number of device units for which the error logger can collect information.

A device handler calls the error logger after every transfer and every retry. Calls to the error logger must be made serially, so that the device handler only calls the error logger during I/O initiation or at fork level.

The device handler makes a call to the error logger by loading the registers as follows:

R2 is a pointer to a buffer in the handler that contains the device registers to be logged.

R3 contains two bytes of information: the high byte holds the initial retry count; the low byte contains the number of device registers that should appear in the error report.

R4 has the device identified code dd$COD in the high byte. The low byte is −1 for a successful transfer, or the number of retries that remain to be performed for an error.

R5 points at the third word of the current queue element.

If the transfer is successful, only registers R4 and R5 are used by the error logger. When the registers have been set up, the device handler issues the instruction:

```
JSR     PC,@$ELPTR
```

If the error logger is not running, this call returns to the device handler immediately.

When you have written a device handler that uses the error-logging support, you must change the reporting system so that the program that prints the reports, will print the entries from your device in the correct format. Chapter 7 in the *RT–11 Software Support Manual* discusses error logging in more detail as well as the procedure used to make your device known to the error-reporting program (ERROUT).

Extended Memory Support

Extended memory support is the main feature of the XM monitor. Under this monitor, programs may access memory above the lower 32 Kwords on a PDP–11. Although programs running on the PDP–11 are limited to using a 16-bit address, the actual physical location to which a program refers is defined by the values in the memory management hardware registers. The virtual memory address space of the program is said to be mapped to physical

memory. This is explained in more detail in chapter 6, "Extended Memory Management."

There are two sets of mapping registers. The mode of the processor defines which set of registers is active. Kernel mapping is the set of register values that allows access to the lower 28 Kwords of memory plus the I/O page. The RT–11 monitor runs with kernel mapping. The other set of registers defines the current user mapping. This is the set of values that defines the user program's virtual address space.

A user program may run with any mapping. Its code and data areas may be anywhere in physical memory, not necessarily in the lower 28 Kwords of memory. The relationship between the program's virtual addresses and the physical addresses is given by the values in the current mapping registers. The mapping registers that are used to determine the physical address are the page address registers (PAR0–PAR7).

Device handlers run with kernel mapping, whereas the I/O queue element passed to the device handler contains the user virtual address of the buffer. This buffer address, as mapped for the user, may not be the same address that the device handler would use to reference the same location.

There are two features of the XM monitor that help the device handler perform transfers to and from a buffer in another address space. First, the I/O queue element contains, at offset Q.PAR, a value for a PAR1 register. This value is used to map the user buffer. Second, the XM monitor contains a number of routines that may be used to transfer data between the device handler's address space and the user's buffer. The routines and general considerations for using extended memory support in a device handler are discussed in detail in chapter 7 of the *RT–11 Software Support Manual*.

Special Functions

Some devices perform operations that cannot be fully supported by the standard queued I/O system programmed re-

quests—for example, rewinding a magnetic tape. When an application program wants to request one of these special operations, the device handler may make use of another facility that RT–11 supports, the special function (SPFUN). The application program issues the special function programmed request:

```
.SPFUN  area,chan,func,buf,wcnt,blk[,crtn]
```

The .SPFUN request is similar to the normal I/O requests .READ and .WRITE, except for the additional parameter "func." As with a normal I/O request, when the .SPFUN request is issued, a queue element is assembled. The offset Q.FUNC in the queue element (zero for a normal I/O request) contains the value of the func parameter.

A device handler that performs special functions must have the SPFUN$ bit set in the device status word (one of the arguments to the .DRDEF macro). If this bit is not set, RT–11 will not accept any special function requests for the device. When the device handler is entered, it should check offset Q.FUNC (or Q$FUNC) in the queue element. If this byte is zero, the request is a normal READ or WRITE. If, however, this byte is not zero, a special function is being requested.

The special function codes should be negative byte values. The meaning of each of the function codes is determined by the person who writes the device handler. Also, for each of the legal function codes, you may interpret the values passed as "buf," "wcnt," and "blk" in any way you wish.

Chapter 7 of the *RT–11 Software Support Manual* discusses special functions in more detail and gives some examples of their uses.

System Device Handlers

A system device handler is one that can be used as the system device for RT–11. This means that the device can be bootstrapped to load the RT–11 operating system into memory from the device. A system device handler is cre-

ated by adding a primary driver to a standard device handler. When a PDP–11 bootstrap sequence is started, the primary driver is loaded from the specified device into physical memory at location 0. The primary driver must be less than 1000 octal bytes long because the secondary bootstrap is loaded at location 1000.

To add a primary driver to a standard device handler, issue the .DRBOT macro immediately before the .DREND macro. Then insert the primary driver between the .DRBOT and the .DREND. The primary driver that you add to a standard device handler is in four parts:

- The bootstrap entry routine
- The software bootstrap routine
- The bootstrap READ routine
- The bootstrap error routine

The bootstrap entry routine is two instructions loaded at location 0. These instructions—a NOP followed by a branch to the software bootstrap—are generated by the .DRBOT macro.

The software bootstrap is entered from the bootstrap entry routine. All registers are available for use by the software bootstrap, which performs the following operations:

- Sets up the stack at location 10000
- Saves the number of the device unit being booted (usually found in the control status register of the device)
- Calls the bootstrap READ routine to read in the secondary bootstrap (blocks 2 to 5 loaded, starting at 1000)
- Sets B$READ to point to the bootstrap READ routine
- Sets B$DEVN to contain the Radix–50 device name
- Places the device unit number in B$DEVU
- Transfers control to the secondary bootstrap (JMP @#B$BOOT)

The bootstrap READ routine is called by both the software bootstrap of the primary driver and the RT–11 bootstrap, the secondary bootstrap. This routine reads the device by non-interrupt-driven programmed I/O using the following information passed in the registers:

R0	the block number to read
R1	the number of words to read
R2	the address at which to store the data

If the bootstrap READ routine fails to make a transfer, it should jump to the bootstrap error routine at location BIOERR. If the transfer was completed successfully, the routine should return via an RTS PC instruction with the carry bit clear. The bootstrap error routine starts at location BIOERR. The error routine is activated if the bootstrap fails for any reason. This routine is generated by the .DREND macro.

References

RT–11 Programmer's Reference Manual.

RT–11 Software Support Manual. Chapter 7 discusses the functions of the bootstrap and the process by which a system device handler is written and installed.

Index

Tailoring RT-11: System Management and Programming Facilities examines the RT-11 tools that enable system managers and programmers to maintain and modify their RT-11 systems. The first portion of the book covers the system management functions: installing the system, performing system generation, maintaining the system volume, and controlling the internal allocation of system resources. The last half of the book describes the system programming functions: using the scheduler, writing a device driver, and using memory directives.

The RT-11 Technical User's Series presents up-to-date information on RT-11. Other books in the series are:

Working with RT-11

Programming with RT-11, Volume 1
Program Development Facilities

Programming with RT-11, Volume 2
Callable System Facilities

The authors develop courses for Educational Services Division of Digital Equipment Corporation in Reading, England.

digital

For a complete list of DECbooks, write to:

Digital Press
Digital Equipment Corporation
30 North Avenue, Burlington, MA 01803

ISBN 0-932376-34-7