TOPS-10
Crash Analysis Guide

AA-H206D-TB

January 1989

The *TOPS-10 Crash Analysis Guide* presents methods for analyzing TOPS-10 system crashes. It describes the tools that can be useful in the process of diagnosing the cause of system failure, and suggests methods of solving the problem that caused the failure. This book is intended to be used by experienced TOPS-10 system programmers and assumes that the reader has adequate system privileges to complete the procedures presented.

Operating System	TOPS-10 Version 7.04
Software	GALAXY Version 5.1

digital equipment corporation
maynard, massachusetts

First Printing, November 1978
Revised, August 1980
Revised, April 1986
Revised, January 1989

The Reader's Comments form on the last page of this document requests the user's critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

CI	DECtape	LA50	SITGO-10
DDCMP	DECUS	LN01	TOPS-10
DEC	DECwriter	LN03	TOPS-20
DECmail	DELNI	MASSBUS	TOPS-20AN
DECnet	DELUA	PDP	UNIBUS
DECnet-VAX	HSC	PDP-11/24	UETP
DECserver	HSC-50	PrintServer	VAX
DECserver 100	KA10	PrintServer 40	VAX/VMS
DECserver 200	KI	Q-bus	VT50
DECsystem-10	KL10	ReGIS	
DECSYSTEM-20	KS10	RSX	

digital™

CONTENTS

PREFACE

CHAPTER 1 INTRODUCTION

1.1 SYSTEM ERROR RECOVERY 1-1
1.2 TYPES OF ERRORS 1-2
1.3 CRASH ANALYSIS TOOLS 1-3
1.4 CRASH ANALYSIS PROCEDURE 1-4

CHAPTER 2 EXAMINING A CRASH FILE

2.1 CREATING A CRASH FILE 2-1
2.2 USING FILDDT 2-3
2.3 ESTABLISHING PROPER MAPPING 2-4
2.3.1 FILDDT Mapping Instructions 2-5
2.3.2 Mapping the Crash 2-5
2.4 VERIFYING THE DUMP 2-7
2.5 FILDDT COMMAND FILES 2-8
2.6 STOPCODE INFORMATION 2-10

CHAPTER 3 LOCATING THE FAILURE

3.1 HARDWARE MAPPING 3-2
3.2 PAGING POINTERS 3-2
3.3 EXTENDED ADDRESSING 3-3
3.4 MONITOR-RESIDENT USER DATA 3-3
3.5 PROGRAM COUNTER WORD 3-4
3.6 PROCESSOR MODES 3-4
3.6.1 User Mode 3-5
3.6.2 Exec Mode 3-5
3.7 THE PRIORITY INTERRUPT SYSTEM 3-6
3.8 THE DEVICE INTERRUPT SERVICE 3-8
3.8.1 Standard Interrupts 3-8
3.8.2 Vectored Interrupts 3-10
3.9 TRAPS 3-10
3.9.1 Page Fail Traps 3-11
3.10 CLOCK LEVEL 3-12
3.11 ACCUMULATORS AND PUSH-DOWN LISTS 3-13
3.12 MONITOR ORGANIZATION 3-13
3.12.1 Monitor Startup Modules 3-14
3.12.2 Symbol Definition Modules 3-15
3.13 EXAMPLES OF LOCATING FAILURES 3-15

CHAPTER 4 EXAMINING THE DATA STRUCTURES

4.1 SYMBOLS 4-1
4.1.1 Naming Conventions 4-2
4.1.2 Symbol Files and Monitor Generation 4-4
4.2 READING THE CODE 4-5
4.2.1 How to Use a CREF Listing 4-5
4.2.2 Macros 4-5
4.2.3 Conditional Assembly 4-6
4.2.4 Finding Symbols 4-6
4.3 JOB-RELATED DATA STRUCTURES 4-7
4.4 CPU DATA STRUCTURES 4-9
4.5 MEMORY DATA STRUCTURES 4-10

4.6 COMMAND PROCESSING TABLES 4-11
4.7 UUO PROCESSING TABLES 4-11
4.8 I/O DATA STRUCTURES 4-11
4.9 THE JOB DEVICE ASSIGNMENT TABLE 4-11
4.10 THE DEVICE DATA BLOCK 4-12
4.11 FINDING DDB INFORMATION 4-13
4.12 LINE DATA BLOCKS (LDBS) 4-15
4.13 THE SCNSER DATA BASE 4-16
4.14 TERMINAL CHUNKS 4-16
4.15 TERMINAL DEVICE DATA BLOCKS 4-17
4.16 FINDING TERMINAL I/O INFORMATION 4-18
4.17 TAPE DRIVES 4-19
4.18 DISKS 4-21
4.18.1 Finding Information on Disk 4-23
4.18.2 In-Core File Information 4-26
4.18.3 The Software Disk Cache 4-28
4.18.4 Finding In-Core File Information 4-29

CHAPTER 5 ERROR HANDLING ROUTINES

5.1 HARDWARE ERRORS 5-1
5.1.1 APR Interrupt Routine 5-3
5.1.2 Page Fail Trap Routine 5-4
5.1.3 Saved Hardware Error Information 5-5
5.1.4 Hardware Error Checking 5-6
5.2 STOPCODES 5-9
5.2.1 Stopcode Processing 5-10
5.2.2 Continuing from Stopcodes 5-11
5.2.3 Special Stopcodes 5-11
5.3 ERRORS DETECTED BY RSX-20F 5-13

CHAPTER 6 DEBUGGING THE MONITOR

6.1 PATCHING WITH FILDDT 6-1
6.2 USING EDDT 6-2
6.2.1 Starting the Monitor 6-2
6.2.2 Breakpoints 6-2
6.3 DEBUGF FLAGS 6-3
6.4 MULTI-CPU ENVIRONMENT 6-4
6.5 CAUTIONS 6-4

APPENDIX A GLOSSARY

APPENDIX B ADDRESS SPACE LAYOUT

INDEX

FIGURES

B-1 Monitor Code Section Layout B-2
B-2 DECnet Code Section Layout B-3
B-3 Monitor Data Section 3 Layout B-4
B-4 Monitor Data Sections 4,5 Layout B-5
B-5 Monitor Data Sections 6,7 Layout B-6
B-6 Monitor Data Sections 35,36,37 Layout B-7

TABLES

3-1	Interrupt Level Indicators	3-7
4-1	Monitor Accumulators	4-2
5-1	Hardware Errors	5-8
A-1	Glossary of Acronyms	A-1

PREFACE

The TOPS-10 Crash Analysis Guide is a procedural and reference manual that you can use to diagnose the causes of TOPS-10 system failures and to correct these problems.

The TOPS-10 Software Notebook Set contains several documents that you should use while analyzing system crashes. In particular, you will find the TOPS-10 Monitor Tables Descriptions and the Stopcodes Specification are most important for symbol definitions, and the TOPS-10 DDT Manual is a useful reference for the debugging tools used in the procedures.

Before you can reliably diagnose and repair system problems, you must be able to use DDT commands to examine and patch the TOPS-10 monitor modules. You must also be familiar with any local modifications that have been made to the monitor.

There are a few symbols shown in this manual that indicate special characters. They are:

Character	Meaning
^\	<Control-backslash> is the character to type on the CTY to get the attention of the parser.
$	The ESCape character, or altmode, is used in commands to DDT and TECO.
<CTRL/Z>	This control character is used to terminate a TOPS-10 process, such as DDT. It is displayed as ^Z.

CHAPTER 1

INTRODUCTION

Crash analysis is used in the process of solving system problems. You can analyze a crash by examining a copy of memory that is stored in a crash file when the operating system stops running. There are different methods of analyzing different types of system problems. It may be helpful, for example, to isolate the cause of a problem as either the hardware or the software on a preliminary investigation, but it is important to understand and recognize all symptoms of system problems, including those involving the interaction of both hardware and software.

This manual describes methods that you can apply to various system problems. As you become more familiar with the monitor and the tools you use to debug the system, you will be able to customize these methods to your own needs.

1.1 SYSTEM ERROR RECOVERY

To successfully analyze different types of system problems, you should try to view the system as a whole, investigating hardware status and software conditions, as well as the interaction of the two. You can use many informational tools to detect and correct system problems: hardware diagnostics verify the hardware state of the machine, and software test packages verify the performance and validity of software components. The monitor itself is an excellent test program for both hardware and software. It prints and saves information about the problems it encounters on the console terminal (CTY). Each CPU in a multiple-CPU configuration has a CTY, where it prints information about the stopcodes it encounters, messages for the operator, and a log of system events.

The TOPS-10 monitor and hardware systems are designed to prevent the system from crashing when a minor error is encountered. Timesharing is only interrupted by an unrecoverable, or fatal error. Most system problems are not fatal, and in most cases system operation continues normally.

A hardware or software error that prevents normal timesharing operation causes a crash; that is, the system performs certain error recovery operations, terminates all user and system jobs, and restarts operation with a fresh database. If a hardware or software error is serious enough to warrant this procedure, the system is halted and a copy of memory is written to disk (or dumped) before the system is reloaded. This copy of memory, called the crash file, is useful because the system uses this file to record the contents of many registers and data structures. This manual describes how to examine the crash file to find information that might indicate the reason for the crash.

Not all hardware and software errors cause the system to crash. The software is equipped with a number of special error recovery procedures to continue operation after a system or user error. The software generates a stopcode, which provide the system manager with information about the cause of the error, and lists system modules and data locations useful in analyzing the source of the stopcode. This information is printed on the system's CTY to inform the operator of the status of the system. A continuable stopcode does not cause a system reload or halt, but, in most cases, produces a crash file.

A system error that causes a crash, like a program error that causes a halt, is called a fatal error, because all the jobs on the system must be halted and restarted. The system records as much information as possible before the crash. However, in the act of reloading memory or processing a hardware error, the operating system may lose or overwrite applicable data locations, and a certain amount of information may be lost. In every crash, it is important to be aware that information recorded during the crash may be invalid or corrupted.

The way the monitor processes the error depends on the type of failure that occurred. The method you use to analyze the crash depends on the type of information that the monitor saved before the crash. This manual is organized to provide crash analysis information for different types of crashes. Remember that this manual can only explain ideal and general situations. As the system analyst, you should be familiar with the specific aspects of the system you are analyzing, because you may face unique problems at your site. If possible, review the system build procedure, especially the information about hardware and software configuration. This type of information is described in the TOPS-10 Software Installation Guide.

DIGITAL provides software error reporting and revision services for problems you cannot solve. If you cannot solve a problem that prevents system operation, submit a Software Performance Report (SPR) through your DIGITAL Service Representative. Be sure to include all the information required to analyze a system crash. This manual describes that information.

1.2 TYPES OF ERRORS

The hardware and software handle each type of system problem differently. Most problems do not result in a crash; many errors are handled locally for a specific program or device, without affecting the entire system. For example, TOPS-10 is designed so that unprivileged user jobs cannot directly crash the system. If a user program develops a fatal error, the monitor aborts the program without affecting the other users on the system. If the monitor data base entries for a particular user job are destroyed, the monitor tries to eliminate the job without affecting other jobs. However, changes to system-wide variables such as those affecting memory and CPU usage may cause the system to crash.

In almost all cases, the software detects and handles errors by gathering information and taking corrective action. In the case of a fatal error, the system reloads automatically. Fault continuation allows the system to correct certain types of errors and continue operation without affecting the execution of user programs. In most cases, corrective action affects only the process at fault. Such action might include repeating an I/O operation or stopping exection of a user job.

Fault continuation allows the system and user jobs to continue with little or no interruption, but continuable stopcodes are recorded on the CTY for later examination. It is important to be aware of all previous errors in the process of analyzing a crash, even those that did not directly cause the system to crash. Internal discrepancies that corrupt an important data structure may in turn affect other routines, and the error propagates, or the software goes into an infinite loop.

Crash files and CTY listings are the main sources of information about the system before the time of the crash. However, error recovery code can contain errors of its own. The history of a crash, including data from the time leading up to the crash, is an important source of information in these situations.

When the system crashes, you must be prepared to verify that the system actually crashed, and determine the extent to which the software was affected. You must isolate the problem that caused the error by defining the point in the code where the error was detected, then identify the problem that caused the error condition, record that information, and correct the problem if possible.

This procedure, and the tools you will need to analyze crashes, are described in the following chapters. Remember that your success in these areas depends on many factors, and that it may not be possible to correct the error immediately. It is more important to continue system operation as soon as possible. Later, you can address the crash using the tools described in this manual.

1.3 CRASH ANALYSIS TOOLS

To analyze a system crash, you need several sources of information, and you must use system programs to examine the information. You must use all your knowledge of the DECsystem-10 and the TOPS-10 monitor, as well as the GALAXY system, ANF-10 network communications, and all other software running on the system. The specific sources of information about a system crash are:

- The CTY output for the time before the crash
- The crash file
- Listings or microfiche of the monitor sources, describing the algorithms, data structures, symbols, and bit definitions
- The operator log book
- The Monitor Tables descriptions from the TOPS-10 Software Notebook Set

You will use the following tools in analyzing system crashes.

- FILDDT (File DDT) allows you to examine files or the running monitor. Sections 2.3 though 2.4 describe FILDDT.
- EDDT (Exec DDT) allows you to examine, breakpoint, and patch the running monitor. Section 6.2 describes EDDT.
- CRSCPY copies crash files and stores information about them in a database. The TOPS-10 Operator's Guide describes CRSCPY.

- SPEAR creates reports, based on the system error log file (ERROR.SYS), which are useful for tracing non-fatal errors that may have led to the system crash. Refer to the TOPS-10/20 SPEAR Reference Manual for more information about SPEAR.

- OPR, the operator interface to the DECsystem-10, provides commands that allow you to change the system configuration and to control software processes. Refer to the TOPS-10 Operator's Command Language Reference Manual for more information about this program.

You will also need to use a text editor such as TECO to patch the monitor sources or system startup files after you have solved a software problem.

1.4 CRASH ANALYSIS PROCEDURE

To isolate a system problem, you must use FILDDT to examine the crash file. The crash file records the state of the system at the time of the crash, including information you can use to determine the cause of the crash, such as:

- Processor mode (user, user I/O, or exec mode)
- Stack pointer and stack in use
- Contents of accumulators
- Stopcode information

First you must obtain the crash file. In Chapter 2, you will learn how the monitor creates and maintains crash files. Chapter 2 also contains procedures for loading the monitor symbols for FILDDT and using the symbolic FILDDT to examine a crash file and extract the information listed above.

Chapter 3 explains how to interpret the information you obtain from the crash file, to determine the state of the system at the time of the crash.

Chapter 3 contains a discussion of processor modes, job scheduling, and the priority levels that the monitor uses in timesharing, and how the information from the crash file can point to the faulty code that caused the crash.

After you have determined the monitor process that failed, you can begin to investigate the crash file for the actual routine that failed. Chapter 4 contains a description of the monitor's data structures and how to obtain information about them from the crash file and the source code.

The monitor may crash, or hang without crashing, because an error has occurred in the error handling and recovery procedures. Chapter 5 contains descriptions of the the system error recovery routines. Continuable stopcodes are described in more detail. You can use the information in this chapter to determine whether error handling routines are functioning properly.

It is sometimes necessary to analyze and correct a system error while the monitor is running, either because a system reload does not correct the error, or the error only becomes apparent while the system is running. If you encounter a problem that defies analysis using FILDDT to examine crash files, you can use EDDT to examine and correct locations in the running monitor. For example, if the system halts or hangs without dumping or without reloading, or if a problem exists that does not interfere with timesharing, you can use EDDT to examine the running monitor. This procedure is described in Chapter 6.

Appendix A contains a Glossary of the acronyms used in this manual.

Appendix B contains illustrations of the general layout of monitor code in virtual address space, for TOPS-10 Version 7.04.

CHAPTER 2

EXAMINING A CRASH FILE

When the system crashes, the monitor attempts to record information about the state of the system at the time of the crash. Normally, the system writes a copy of memory to disk before beginning system reload operations. This copy of memory is called a crash file, or just "a crash". You can examine this file using a special version of DDT called FILDDT. This chapter explains in more detail how the crash file is created and how to locate the crash file for a particular crash. The procedure for preparing FILDDT so that you can examine the crash file is also described, as well as some of the information that you can obtain immediately by examining the CTY output of stopcode information.

2.1 CREATING A CRASH FILE

When a stopcode occurs, BOOT automatically creates a crash file of the contents of memory, called CRASH.EXE, and copies it to the system crash list. If BOOT cannot dump memory automatically, you can force a dump by typing the following command on the CTY:

```
BOOT>str:/D
```

Use /D to force the crash file to be written. You may include the name of a file structure (str:).

If this action fails, the CRASH.EXE file on every file structure in the system crash list may be unprocessed by CRSCPY.

The allocation of CRASH.EXE space is accomplished when you define file structure information in the ONCE dialog. You can modify the amount of space reserved for crash files by running the monitor in user mode. Refer to the TOPS-10 Software Installation Guide for complete information about ONCE.

To stop the machine when a malfunction occurs, deposit a non-zero value into physical location 30. The monitor checks this location at every clock tick. If it finds a non-zero value, the monitor jumps into BOOT. You can initiate this procedure using one of the following commands.

The first example is a command to the PARSER on a KL system. Type <CTRL/backslash> where you see ^\. In the following examples, semicolons precede comments that should not be included in your input.

```
^\                              ;invoke the PARSER
PAR>SHUTDOWN                    ;shut down the system
[Dumping on DSKA:CRASH.EXE[1,4]]
```

For a KS system, you type the following commands:

```
^\                          ;invoke the console
ENABLED
KS10>SHUTDOWN               ;shut down the system
USR MOD
[Dumping on DSKA:CRASH.EXE[1,4]]
```

If the monitor can reach clock level, this command will start BOOT. BOOT stops the machine, writes a crash file, and begins automatic reload procedures. If the monitor has been up less than five minutes, BOOT starts, but does not initiate the dump and reload action. Instead, BOOT prints the BOOT> prompt and waits for you to type a command.

If the SHUTDOWN command is ineffective, you must instruct the monitor to begin system shutdown procedures. The following commands to the PARSER accomplish that on a KL system:

```
^\                          ;invoke the PARSER
PAR>SET CONSOLE MAINTENANCE
PAR>HALT
PAR>EXAMINE KL
PAR>JUMP 407
```

This instructs the monitor to execute the instruction at location 407, which signals the policy CPU to initiate a system shutdown procedure. In multiple-processor systems, it may be desirable to initiate system shutdown procedures on the current CPU instead of the policy CPU. To accomplish this, jump to location 406 instead, using the following command:

```
PAR>JUMP 406
```

For the KS, you might use the following procedure to force a system shutdown:

```
^\                          ;invoke the console
ENABLED
KS10>HALT                   ;halts the system
KS10>MR                     ;forces exec mode
KS10>SM                     ;halts at default location
KS10>ST 407                 ;loads BOOT
USR MOD
```

You should try to use the SHUTDOWN procedure first, because a forced reload does not save the PC, and there is danger of losing device and interrupt status information.

After a fatal stopcode or a manual dump operation, BOOT displays the following information on the CTY:

```
[Dumping on DSKA:CRASH.EXE[1,4]]
[Loading from DSKA:SYSTEM.EXE[1,4]]
```

As the second message indicates, BOOT automatically reloads the monitor. The automatic reload function can be disabled using the OPR program. This function is useful when debugging the monitor, as described in Chapter 6.

The CRSCPY program runs when the system is reloaded, to copy the CRASH.EXE file to a unique file name that will not be superseded by subsequent CRSCPY runs. If your system did not run CRSCPY when it reloaded, you must copy the CRASH.EXE file to a safe area manually.

As soon as you can log into the system, save the crash in the XPN: area of the disk structure by typing the following command:

```
.R CRSCPY
CRSCPY>COPY
```

The CRSCPY program copies the file using a unique file name and reports it when the operation is finished. For more information about CRSCPY, refer to the TOPS-10 Operator's Guide.

You can use SYSTAT to obtain an overview of the status of the system at the time of the crash. Use the /X switch to SYSTAT to indicate a crash file, and include the name of the crash file. For example, to examine the SYSTAT information for a crash file named SER003.EXE, type the following command:

```
.SYSTAT/X XPN:SER003.EXE
```

The /X switch specifies that the SYSTAT program should read the file XPN:SER003.EXE (the file name assigned by CRSCPY) instead of the running monitor.

2.2 USING FILDDT

FILDDT is a system debugging tool designed for debugging files that are stored on disk. Because FILDDT is a modified version of DDT, you must be familiar with DDT before you attempt the procedures described in the following sections. For more information about DDT, refer to the TOPS-10 DDT Manual.

FILDDT has all the commands of regular DDT, with one major difference: commands that control program execution do not work. Those commands are:

$G	Start the program.
$X	Execute a single instruction.
$P	Proceed with execution.
$B	Set breakpoints.

The monitor, because of its large size, runs with local and global symbols removed. You cannot examine the crash file without these symbols, so you must load the symbol table of the monitor into memory with FILDDT and save the modified version of FILDDT. To create this special monitor-specific FILDDT, follow the procedure explained below.

First, run the standard version of the FILDDT program:

```
.R FILDDT

File:
```

You must type the name of the file from which the symbols are to be loaded. This file must be the runnable monitor; that is, the monitor before loading (often SYS:SYSTEM.EXE). Include the /S switch to indicate that symbols are to be loaded.

```
File:SYS:SYSTEM.EXE/S
```

The /S switch tells FILDDT to load the symbols for this file. When FILDDT displays another File: prompt, type <CTRL/Z> to exit from FILDDT, then type the SAVE command to the monitor with the file name you choose for the the symbolic FILDDT, to save the runnable file. In the following example, the symbolic FILDDT is called MONDDT.

```
File:^Z

.SAVE MONDDT
MONDDT saved
```

After you save the symbolic FILDDT program, you can use the RUN command to start the new FILDDT at any time. For example, the following commands start the symbolic FILDDT and give it the name of a crash file (XPN:SER003.EXE) to examine:

```
.RUN MONDDT

File:XPN:SER003.EXE
```

When FILDDT reads the crash file, it reports the mapping of the ACs in the following message:

```
[Looking at file DSKA:SER003.EXE[10,1]]
[Paging and ACs set up from exec data vector]
```

The monitor locations saved in the crash file must now be mapped to the virtual monitor addresses. FILDDT provides special commands for mapping the monitor and the user address space. Before you issue a mapping command, FILDDT assumes all locations are physical references.

2.3 ESTABLISHING PROPER MAPPING

Virtual addressing machines require special consideration. Instructions in programs are loaded into memory by a mapping scheme based on page maps. The actual physical location of a word in the monitor will not necessarily be the same as the virtual location.

The symbolic FILDDT contains the virtual address of each location, but not its physical address. You must map FILDDT memory references through the Exec Process Table (EPT) to examine monitor locations, or through the User Process Table (UPT) to examine user locations. To establish mapping, you must perform the following steps:

1. Find the page numbers of the page maps.
2. Issue the FILDDT mapping instruction (a $nU command).
3. Verify that the mapping is correct.

The following sections describe two methods for mapping the dump and obtaining preliminary information concerning the state of the processor at the time of the crash. The instructions used in the following procedure may be included in a FILDDT command file (also called a patch file).

To map a crash, you must provide FILDDT with pointers to mapping tables and other locations in the monitor. The mapping tables and monitor locations are described in more detail in Chapters 3 and 4.

2.3.1 FILDDT Mapping Instructions

FILDDT allows you to specify the type of address mapping to use in locating information. You can specify virtual or physical addressing. The mapping instructions are:

$U enables virtual addressing. This instruction also sets the FAKEAC flag, indicating that physical locations 0-17 are to be interpreted as the user accumulators (ACs).

$$U enables physical addressing. The FAKEAC flag is cleared, indicating locations 0-17 are interpreted as hardware registers 0-17.

By default, physical addressing is enabled. FILDDT interprets all addresses as physical until you issue a virtual mapping instruction. The mapping is correct only for the data in portions of the monitor's low segment, because the low segment virtual addresses equal the physical addresses.

The TOPS-10 monitor uses KL-paging, also called "extended addressing" (described in Section 3.3). By default, FILDDT is enabled for KL-paging. If it is necessary to disable KL-paging (for an older version of the monitor, for example), you can issue the following command to FILDDT:

```
0$11U
```

To enable KL-paging, type the following command:

```
1$11U
```

The command n$11U establishes the mapping scheme so that FILDDT will read the page maps correctly.

Next, you must point FILDDT at the correct page maps that associate virtual addresses (loaded into the symbolic FILDDT) with the physical addresses (saved in the crash file), and establish virtual mapping.

2.3.2 Mapping the Crash

To map virtual addresses to physical ones, FILDDT needs the locations of the Exec Process Table (EPT) and the Special Pages Table (SPT). The EPT allows FILDDT to map exec virtual memory. The SPT is used to map the user job that was running at the time of the crash.

On a multiple-processor KL system, the dump contains an EPT for each CPU in the system. To analyze the dump, you must map FILDDT through the EPT for the CPU that crashed. A CPU Data Block (CDB) exists for each CPU in the system. On a single-processor system, there is one CDB. The CDB contains the address of the EPT. Therefore, you must first find the CDB for the CPU that crashed. The location DIECDB contains the pointer to the CDB of the CPU that crashed.

NOTE

The contents of DIECDB are written when the system crashes, but not when the system hangs. When you are analyzing a hung system, the contents of DIECDB (if nonzero) were written by a previous crash, and therefore may be invalid.

You can see the contents of DIECDB by typing the following command to FILDDT:

```
DIECDB[  12000
```

In this example, the physical starting address of the CDB is 12000. The location of the EPT is stored in the CDB at the offset symbolized by .CPEPT. Use the following command to open .CPEPT and read its contents:

```
$Q+.CPEPT-.CPCDB[  1000
```

The first part of the instruction ($Q) refers to the last value displayed (that is, the contents of the currently open location). This value is 12000. Starting from location 12000, the pointer moves to the offset indicated by the difference between the values of .CPEPT and .CPCDB. The new location is the offset into the CDB of the EPT address (.CPEPT). The instruction opens the location .CPEPT and displays its contents. The EPT address is displayed as physical location 1000.

FILDDT needs the page number for the EPT, not its physical address. Therefore, you must divide the contents of .CPEPT by 1000.

Submit the result of this division operation to FILDDT using the $0U command. For example, to calculate the page number and map the EPT, type the following FILDDT instruction:

```
$Q'1000$0U
```

This command divides the previous value (using the $Q command) by 1000 and submits the result to FILDDT as the EPT page number. In this example, the page number is 1.

Exec virtual memory is mapped after the $0U command. This is sufficient for examining monitor memory locations in the crash. However, to examine user data, you must map the current user job. The FILDDT command n$6U maps the user job and its associated per-process storage in exec virtual memory (funny space). The value of n is the page number of the UPT (User Process Table).

The SPT contains a word for the current job running on each CPU in the system, plus a word for each user job. The right half of each SPT slot contains the page number of the UPT for the current CPU. When extended addressing is enabled, the SPT points to the UPT.

The following FILDDT command sets the SPT base address:

```
JBTUPM+(job#)-(CPU#)$6U
```

To map a user job other than the current job on the current CPU, add the contents of the right half of JBTUPM to the job number, then submit the result to the $U command.

FILDDT provides temporary registers to contain either hardware registers or user accumulators. When hardware mapping is established, FILDDT assumes that locations 0-17 refer to hardware registers 0-17. However, when you issue a virtual mapping command ($U), the user ACs can be mapped through the temporary registers. This allows you to load the user ACs into the temporary registers and then refer to the user ACs as locations 0-17.

You can use the following FILDDT instruction to map the current AC block to the temporary registers provided by FILDDT. The instruction to open and map the current AC block is:

```
.CPACA[ $Q$5U
```

This instruction is useful only if the location .CPACA contains the address of the current AC block. If, however, a UUO at interrupt level occurs (UIL stopcode), this instruction cannot be used successfully. Instead, you must determine the location of the current AC block by defining the interrupt level in progress at the time of the crash. The AC blocks and interrupt levels are described in more detail in Chapter 3.

The user job in memory may not match the UPT currently in use at the time of the crash. You can check the user job that was running by comparing the contents of offset .CPJOB in the CDB with the contents of .USJOB in the UPT. If these values do not match, the interrupt routine was switching UPTs at the time of the crash; use the UPT for the job number that is in .USJOB.

Look at the code that you are familiar with, in the high segment, to make sure the dump is mapped correctly. Also check location 410 (ABSTAB), which should point to NUMTAB, which is one of the first locations in the low segment.

If you set up mapping through the wrong page map, FILDDT returns a question mark whenever you try to reference an unmapped location. For example, this could occur if you use the null job's UPT to set mapping. To reset mapping, use the "$$U" command to set physical mapping by FILDDT.

2.4 VERIFYING THE DUMP

Occasionally, your monitor will crash in the process of upgrading to a new version, or when you are making modifications to the code. In these cases, it is possible that your crash file will be based on a different version of the monitor than the monitor-specific FILDDT you created. You should make sure that the symbols in the monitor-specific FILDDT match the crash that you are examining. If values of the symbols do not match, the information in the crash file may be useless, misleading, or corrupted.

There are several ways to check the symbols. One is to make sure the version number of the crashed monitor matches that of your current monitor. Another is to examine addresses in the monitor with known contents and verify that they contain the right information.

Monitor location CNFDVN contains the monitor version number and edit number. This version number should match the version number displayed by the DIRECTORY monitor command.

```
.DIRECTORY IEZ093.EXE
IEZ093  EXE  8196 <155> dd-mmm-yy  704(33432) DSKB:[10,1]

.RUN MONDDT

File:DSKB:IEZ093.EXE[10,1]
[Looking at file DSKA:SER003.EXE[10,1]]
[Paging and ACs set up from exec data vector]

$$C
CNFDVN/     70400,,33432
```

Note that the DIRECTORY command reports version and edit numbers 704(33432), matching the contents of CNFDVN: 704 in the left half, and 33432 in the right half.

You can obtain the name of the monitor by reading ASCII text starting at location CONFIG, as shown in the following example:

```
CONFIG$0T/      RL371A  DEC10 Development
```

In this case, the full system name is "RL371A DEC10 Development".

If these values match, you can be relatively sure that the monitor-specific FILDDT and crash file match.

2.5 FILDDT COMMAND FILES

FILDDT command files are used to map a dump and obtain preliminary information that might be relevant to analyzing the crash. A command file is a set of FILDDT commands that are executed automatically when you issue the $Y command to FILDDT. Command files are also used to edit the runnable monitor (as opposed to making edits to source modules and rebuilding the monitor).

The FILDDT command $Y invokes a series of FILDDT commands stored in a file on disk. This allows you to easily execute a set of commands that you use frequently instead of typing them in. You could use a command file to map and verify a dump and to extract information you are likely to need while diagnosing a crash, as described below.

NOTE

> The $ (dollar sign) is displayed when you press the ESCape key in FILDDT. It is used here to show where you must insert an ESCape character into the file. Most text editors require a special procedure for inserting ESCape and other non-printing characters into a file. You must use the text editor documentation to find the method for quoting characters if you do not know how to insert an ESCape character into a file.

The following command file maps a crash file for a multiple-processor KL system. The same command file is equally useful on a single-CPU KL or a KS system. The command file also verifies the correspondence of the dump with the monitor-specific FILDDT and displays pertinent system information about the crash.

Comments are included here to describe the functions of the commands, However, FILDDT will not accept a command file with comments. Your actual command file should NOT contain the comments in the following example:

```
.TYPE VERIFY.DDT          ;display contents of patch file
DIECDB[                   ;gets addr of CDB for CPU that crashed
$Q+.CPEPT-.CPCDB[         ;gets addr of the EPT
$Q'1000$U                 ;divides addr by 1000 to get page number
SPTTAB$6U                 ;sets the SPT base address
.CPACA[$Q$5U              ;maps AC references
.CPCPI[                   ;gets PI status
.CPPGD[                   ;gets DATAI PAG results
.CPSPT[[                  ;gets the address of the SPT
.CPDWD[                   ;gets CPU's DIE interlock word
```

```
.CPCPN[                 ;CPU number of crashed CPU
.CPJOB[                 ;gets job number of current job
.USJOB[                 ;job number in funny space
.CPTCX[                 ;process context word on page fails
```

You can include these and other FILDDT commands in a command file to obtain initial information about the crash. The locations referenced in this file are described in Chapter 3.

The following example shows the types of information that might be displayed and how to interpret the information. Again, the comments are included for descriptive reasons, but comments are not allowed in an actual command file.

```
.R MONDDT               ;run the symbolic FILDDT

File: SYS:CRASH
[Looking at file DSKA:SER003.EXE[10,1]]
[Paging and ACs set up from exec data vector]
$Y                      ;execute a command file
File: MON.DDT           ;command file is MON.DDT

DIECDB[   13000         ;the address of the CDB for the
                        ;CPU that crashed is 13000

$Q+.CPEPT-.CPCDB[   3000 ;compute the offset into the CDB
                        ;address of the EPT is stored

$Q'1000$U               ;compute the page number of the EPT
                        ;and point FILDDT to the EPT

SPTTAB$6U               ;set the SPT base address

.CPACA[   402077$Q$5U   ;map AC references

.CPCPI[   377           ;377 indicates PI levels are enabled

.CPPGD[   700100,,2600  ;DATAI PAG shows that:
                        ;current AC block is 0 (exec)
                        ;previous AC block is 1 (user)
                        ;previous context section is 0 (exec)
                        ;UPT page number is 2600

.CPSPT[SPTTAB+1[  2600  ;shows UPT page number of currently
                        ;mapped job on this CPU

.CPDWD[   0             ;Die interlock word

.CPCPN[   1             ;CPU1 failed

.CPJOB[   5             ;Job 5 was running

.USJOB[   5             ;Job 5 is mapped on this CPU

.CPTCX[   701100,,2364  ;Process context information:
                        ;current AC block is 1 (user)
                        ;previous AC block is 1 (user)
                        ;previous context section is 0
                        ;user base page number is 2364
```

It is important to compare the value of .CPTCX with the contents of .CPPGD. The process context word stored in .CPTCX and the DATAI PAG word stored in .CPPGD are different when the state of the processor at the time of the crash is indeterminate (for example, for IME or EUE stopcodes).

2.6 STOPCODE INFORMATION

The following information is useful when the system crashed with a stopcode. You can determine the stopcode information by looking at the CTY for the CPU that crashed. The stopcode name is printed on the CTY, and is stored in location .CnSNM, where n is the CPU number. Use the Stopcodes Specification in the TOPS-10 Software Notebook Set to look up the module that generated the stopcode.

The stopcode routines in the monitor also store and print the following types of information on the CTY:

- Date and time of crash

  This information is stored in a series of locations starting at LOCYER:

  LOCYER - Year of the crash
  LOCMON - Month of the crash
  LOCDAY - Day of the crash
  LOCHOR - Hour of the crash
  LOCMIN - Minute of the crash
  LOCSEC - Second of the crash

  Remember to display these locations in decimal, not octal.

- Current job

  The word at address .CnJOB holds the job number of the current job on CPUn.

- PPN of current job

  The PPN is stored in the JBTPPN table, indexed by the job number.

- Program name of current job

  The program name is stored in SIXBIT in the JBTNAM table, indexed by the job number.

- Terminal of current job

  The terminal name is stored in SIXBIT in the first word of the Terminal DDB, pointed to by TTYTAB (indexed by job number).

- CPU number

  The CPU number of the CPU that crashed is determined from the value of .CnDWD (where n is the CPU number). Test this symbol for a negative value (-1) for each CPU in a multiple-CPU system. A negative value indicates that the CPU did not crash. If the contents of .CnDWD are equal to zero, the current CPU is the CPU that crashed.

Refer to Section 5.2 for more information about the types of stopcodes and the information they provide.

CHAPTER 3

LOCATING THE FAILURE

The monitor is the portion of the software that is responsible for interfacing user programs to hardware. Specifically, the monitor is responsible for the following functions:

1. Performing tasks for a user before and after running a program, such as copying or deleting files, finding the status of the system, and running or stopping programs. TOPS-10 provides the user interface in the form of the command language.

2. Executing the program. The user must make requests for all services (including I/O). The user programming interface is standardized in the form of monitor calls, also called Unimplemented User Operators (UUOs).

3. Providing access to the data base. This is done by creating a logical file system for data stored on disk devices.

4. Controlling CPU usage. A timesharing system must know how to determine who should get control of the computer. This is called scheduling.

5. Controlling memory usage. For the system to run efficiently, jobs must be moved in and out of memory at the right time. This operation is known as swapping and paging.

6. Controlling access to sharable devices. The main sharable devices on timesharing systems are disks. Because many jobs will be using files on the same disk drive, adequate control must be maintained to prevent destructive interference.

7. Controlling access to single-user (non-sharable) devices. The monitor must implement a way to allocate these devices to the right users and control the I/O. TOPS-10 does this with the GALAXY batch and spooling system.

8. Providing error analysis when hardware or software errors occur (DAEMON and SPEAR).

9. Providing accounting information so the system can be fairly allocated and users charged for what they use (ACTDAE).

3.1 HARDWARE MAPPING

The hardware uses three types of tables to establish and maintain mapping of locations in memory for a job: process tables, section tables, and page tables:

- The process table describes characteristics for a specific job and includes a pointer to each section map required to map the job. There are two process tables: the Exec Process Table (EPT) and User Process Table (UPT).
- The section map contains pointers to the page map for each virtual section for the monitor or user job.
- The page maps contain locations for each physical and virtual page allocated to the monitor or user job.

The paging system uses two process tables: the UPT to map the user job and the EPT to map the monitor. The UPT (User Process Table) is the table used to describe user address space. Each user job has its own UPT, which must be loaded before the job can be run. The EPT (Exec Process Table) is used to describe the monitor address space.

The processor runs by switching between user mode and exec mode. To perform address translation quickly, the hardware must know the locations of the process tables. Two registers are used to find the process tables: the User Base Register (UBR) points to the UPT and will vary for each job that is loaded into memory. The Exec Base Register (EBR) points to the EPT. On multiple-CPU systems, each CPU has an EBR and a UBR at all times.

3.2 PAGING POINTERS

The page maps contain pointers to physical pages of data. The page maps are read by the microcode, which evaluates two kinds of pointers: section pointers that point to section maps, and page map pointers that point to physical pages. Section and page pointers have identical formats. There are four types of pointers, indicated by a code stored in Bits 0-2 of the word. The access code is applied to the address by ANDing Bits 3-6 of all pointers used to evaluate the address.

The pointer to non-accessible pages has code (0) in Bits 0 through 2.

The pointers to accessible pages also include accessibility codes in Bits 3 through 6. Bit 3 (P), if set, indicates that the page is public. Bit 4 (W) indicates whether the page is writable, and Bit 6 (C) indicates whether the page can be cached.

Bit 5 of the pointer to an accessible page is used by the MCA25 harware option as the "Keep Me" bit. That is, if Bit 5 is set in the page pointer, the address translation for that page is not cleared in the hardware pager, providing that the DATAO PAG (context switch) is issued with Bit 3 set.

3.3 EXTENDED ADDRESSING

The KL processor uses KL-paging to allow code and data to be grouped into virtual sections; each section is a maximum of 512 pages of virtual memory. The monitor layout for a KL with extended addressing enabled is illustrated in Appendix B.

The KS processor does not support extended addressing. However, because KL-paging is required in order to run TOPS-10 Version 7.04, the KS processor simulates KL-paging by choosing an alternate page map when necessary.

The primary page map for the KS monitor is the Section 0 page map. To perform a monitor call to an extended section, the KS monitor changes the page map pointer. For example, to execute the DNET. monitor call, a special macro reads the Section 2 page map pointer (from SECTAB+2 in the EPT) and writes the address into the Section 0 page map pointer (at SECTAB in the EPT). The KS accesses locations in the Section 2 page map until the monitor call has been serviced. A similar macro restores the Section 0 page map pointer to SECTAB.

3.4 MONITOR-RESIDENT USER DATA

Some information that pertains to the specific user is kept in the monitor's address space, in the exec page maps. Each word in a page map can point to a physical page in memory, but the Section 0 Page Map also contains indirect pointers to the UPT. The monitor uses these virtual addresses to reference job-specific locations, such as funny space.

The job-specific data in monitor address space is composed of the following areas, which are described separately below.

```
Funny Space (Per-Process Area)
UPT
.UPMAP (Section 0 page map)
.UPMP/.UUPMP (UPT origin)
JOBDAT
Vestigial JOBDAT
```

The information in these pages is specific to the current user, so the job's page maps in the crash file contain virtual and physical addresses. In a multiple-CPU system, the SPT (Special Pages Table) for that CPU contains the current user page map page. When a new job is selected to run, only the UBR and the SPT words need to be changed.

Certain pages of the executive virtual address space are designated as the per-process monitor free core, also known as funny space, for the job that is currently running on that CPU. This is monitor memory that is swapped with the job, and contains information pertaining to its disk DDBs, monitor buffers, SWITCH.INI, the extended channel table, and so forth.

The monitor references the user's funny space with the symbol .UPMP, which points to the first location in the UPT, and reads the physical location in memory from the page table for user page 0.

User page 0 contains JOBDAT locations, which are used by the monitor for handling the user job.

Vestigial JOBDAT is the job data area for the job's high segment.

3.5 PROGRAM COUNTER WORD

The PC (Program Counter) double-word contains the location of the next instruction that the system will execute, including flags to indicate whether the processor is in user mode or exec mode. The PC is stored in the job's UPT (at USRPC) and in the CDB (at .CnPC). When you analyze a crash, you must examine Bit 5 of the PC word to determine whether the processor was in user mode or exec mode at the time of the crash. If Bit 5 of the PC is set, the processor was in user mode. If Bit 5 is clear, the crash occurred in exec mode. The remaining PC flags indicate arithmetic overflow conditions and so forth.

The PC contains a thirty-bit address, which points to the next instruction to be executed. When control passes to a section other than the section where the instruction was issued, that instruction must refer to a 30-bit address. To store the 30-bit PC with flags, the flag-PC doubleword is used. The flag word contains the PC bits in Bits 0-12, in a format identical to the single-word PC. Bits 13-17 are unused. The right half of the first word is used by the hardware. The second word contains the page number and address. Bits 0 through 5 of the second word are zero. The format of the PC doubleword allows the flags (including the mode bit) to be read in the same manner as a single-word PC. You can also read the address in a double-word PC in the same way as a single-word PC, after you add 1 to the location of the PC word.

Most instructions that use 30-bit addresses cannot be issued in Section 0. Global section references are illegal in Section 0, except for the OWGBP instruction, the XJRST and XJRSTF instructions, and the XBLT function of the EXTEND instruction. Any other instructions with global section references must be made from a non-zero section.

3.6 PROCESSOR MODES

The processor reads the PC to determine whether the instruction is to be executed in user or exec mode. User mode allows user jobs to run programs and request the monitor for system resources. Exec mode allows the monitor to satisfy user requests for system resources and perform overhead functions.

You can determine the processor mode at the time of the crash by reading the PC word from the CDB. Bits 5 and 7 of the PC word are useful in determining the processor mode. If Bit 5 is clear, the processor was in exec mode. If Bit 5 is set, the processor was in user mode. In user mode, if Bit 7 is set, the job is in public mode; if Bit 7 is clear, the job is in concealed mode. In exec mode, if Bit 7 is clear, the process is in kernel mode. If Bit 7 were set in exec mode, this would establish supervisor mode, but this mode is not used by TOPS-10.

Processor modes, PCs, and paging pointers are described in the DECsystem-10/20 Processor Reference Manual.

3.6.1 User Mode

Normally a user program runs in user mode. When the program requests a monitor service, using a monitor call, the current processor flags and PC are saved. The program is stopped temporarily while waiting for the monitor service to be completed; this is called "blocking." Control of the processor is then passed to the monitor in exec (kernel) mode by clearing the processor flags and starting at a new PC.

When an I/O operation is requested or completed, a device interrupt causes the monitor to service the device. On a regular basis, the monitor receives a clock interrupt, which initiates job scheduling and system maintenance (overhead functions). When the clock service routine is finished, control passes to the appropriate user program, and the processor switches back to user mode by setting the flag bits (Bits 5 and 7) and restoring the user's PC.

A user program runs in either User Public or User Concealed mode. User mode begins with a monitor command and ends when the program exits or encounters an error. Normally the program runs in public mode: Bits 5 and 7 of the PC word are set. The user program runs in concealed mode if Bit 7 is clear and Bit 5 is set.

3.6.2 Exec Mode

When a user program requests a service by the monitor, using a monitor call or a command, the processor must switch from user mode to exec mode. Exec mode allows the monitor to perform privileged services and provides the user's interface to file management, device control, and hardware communication in general.

User programs run in user mode, and cannot perform direct I/O instructions. A range of I/O instructions, with device codes from 740 to 774, are reserved for customer definition, and are therefore designated as unrestricted codes.

When a UUO is executed, a hardware trap condition occurs, causing the the microcode to store the following information in the UPT:

- PC doubleword
- 30-bit effective address
- Opcode and AC (from the instruction)
- Process context word

The new PC word is taken from one of the MUUO dispatch locations in the UPT, depending on the processor mode and whether or not the UUO occurred during the processing of another trap condition (a PDL overfolw, for example). Control passes to the MUUO routine in the monitor, where UUO processing begins. The monitor uses AC Block 0. The user program uses AC Block 1. To switch to AC Block 0 from Block 1, the monitor issues the following instruction:

```
DATAO PAG, addr
```

Where: addr contains the value [400100,,0].

When the job is not running, the user accumulators are stored in JOBDAT in the user's address space. The monitor's accumulators are stored in the next higher locations in the user's address space.

Once in the MUUO routine, the monitor checks the UUO for legality by checking the instruction stored in .USMUO of the UPT. The return PC from USRPC in the UPT is placed on the monitor's stack for this job. Then control passes to the appropriate routine to perform the function for the user.

The execution of the user function may finish or it may block, waiting for something to happen (I/O, for example), before it can continue. If control can be returned to the user job, the user AC set is restored and control passes to the location pointed to by the PC in USRPC. If the job blocks, the monitor goes to clock level. After the blocking condition is serviced, the job can run again. At the time of the block, the monitor's PC is stored at USRPC in the UPT.

The MUUO routine uses a stack, also located in the UPT, which the monitor can address because it is mapped through a monitor virtual address (refer to Section 3.3).

Some values in the UPT can be cached without interfering with the system, such as the stack. These locations are referenced by the symbol .UUPMP. Other locations are not cached; they are referenced by the symbol .UPMP, which also points to the first location of the UPT. On a single-CPU system, the monitor caches the contents of all locations in the UPT from .UUPMP to .UPMP. On multiple-CPU systems, however, the system only caches the contents of .UUPMP.

3.7 THE PRIORITY INTERRUPT SYSTEM

In exec mode, the monitor can service the user program, a device request, or a clock-level interrupt. Interrupts can be caused by devices or by the clock. While in exec mode, the monitor services interrupts according to the Priority Interrupt (PI) level assigned to the interrupting process. A typical set of priority interrupt levels (also called PI channels) might be:

```
Level 0   DTE (Byte Xfer,Deposit,Examine only)
          CI/NI (limited set of functions only)
Level 1   none
Level 2   DTA (DECtape)
Level 3   Card reader, APR, clock
Level 4   Line printer, magtape, NI, DTE (doorbell)
Level 5   Disk, CI
Level 6   ANF-10 network
Level 7   Monitor
```

To distinguish the interrupt level of the system at any one time, four pieces of information are used:

- The set of accumulators currently in use, which reveals the stack in use.
- The processor mode (exec or user).
- The status of the PI system.

- The process context word. When the monitor is called to perform a service for a user job, as with a command or UUO, the microcode creates the job's process context word and writes it into the UPT. This process context word is displayed by a DATAI PAG instruction where Bit 2 is cleared, and contains the current AC block number, the previous AC block number, section bits, and the current UBR (User Base Register).

A summary of the interrupt levels and how to distinguish them is shown in the following table:

Table 3-1: Interrupt Level Indicators

	AC Block	PDL	Mode	PI Status
User Job	1	Variable	User	No PIs active
Null Job	1	N/A	User	No PIs active
UUO Level	0	JOBPDO	Exec	No PIs active
Clock Level	0	NUnPDL	Exec	PI 7 active
Device Interrupts:				
Terminal driver	2	CnxPD1	Exec	PI SCNCHN active
Disk service	3	CnxPD1	Exec	PI DSKCHN active
Network service	4	CnxPD1	Exec	PI NETCHN active
Other (level y)	0	CnyPD1	Exec	PI y active
Page Fail	0	NUnPDL ERnPDL	Exec	Variable

You can find the stack by finding the current set of ACs. The process context word, stored in the UPT, contains the current AC block.

You can determine the status of the priority interrupt system by looking at the PI status word, stored at location .CnCPI in the CDB. This word is read by the monitor with a CONI PI instruction and stored in the CDB when the monitor starts to process a stopcode. Using this information you can determine whether the PI system was enabled, what PI levels were enabled, and what kinds of interrupts were in progress.

The PI status word on a KL system has the following format:

Bits	Meaning
0-10	Not used.
11-17	Level on which a program requests an interrupt (Bit 11 = Level 1, Bit 12 = Level 2, and so forth).
18-20	Write even parity (KL diagnostics only).
21-27	Levels on which an interrupt is in progress.
28	PI system is on.
29-35	Enabled levels.

3.8 THE DEVICE INTERRUPT SERVICE

A device interrupt occurs when an I/O transfer is complete, a device has changed status, or an error has occurred. There are two types of device interrupts: vectored and nonvectored interrupts. A nonvectored, or standard interrupt, is handled by the software. The interrupt handling instruction is read from the EPT and control passes to the CONSO skip chain to determine the device that generated the interrupt. Section 3.81 describes standard, nonvectored interrupts.

The DTEs (doorbell function only), the interval timer (on the same level as APR interrupts), RH10, and RH20 MASSBUS controllers all perform vectored interrupts. Vectored interrupts are not dispatched by the software but are automatically dispatched by the microcode. Section 3.8.2 describes nonstandard, vectored interrupts.

3.8.1 Standard Interrupts

An interrupt can occur on Levels 1 through 7 only if the PI system is turned on, there are no higher-level interrupts in progress, and the PI system is enabled for interrupts on that level on which the interrupt is requested. If these conditions are met, the interrupt will stop the processor and turn on a bit in the PI status word. The bit indicates the level on which the interrupt is requested. The processor then executes the instruction for handling an interrupt on the requested PI level.

The location of the interrupt handling instruction is stored in the EPT. The exact location in the EPT is calculated from the following:

```
EPT+40+2*n      ;where n is the PI interrupt level
```

The next instruction to execute in the handling of the interrupt is stored in the EPT and depends on the PI level on which the interrupt was requested. The above calculation results in an offset into the EPT where the instruction is stored. Thus, if a BA10 (unit record) I/O bus controller is assigned to PI Level 2, the formula would result in EPT+40+(2*2). The system then executes the instruction stored at offset 44 into the EPT.

Interrupt level 0 is reserved for certain types of I/O transfers with DTE and CI/NI (KLIPA/KLNI) devices. Level 0 bypasses the software and is handled by the microcode, which handles interrupts on Level 0 automatically without requiring the software to store context information and so forth.

In general, the interrupt instructions in the EPT are formatted as:

```
XPCW CHnm
```

where n is the CPU number (omitted for CPU0), and m is the level number on which the interrupt is in progress. For example, CH7 means Level 7 on CPU0. CH27 indicates Level 7 on CPU2 (the third CPU in the configuration). The new PC flags at CHnm+2 usually include the previous context user flag. This allows the interrupt service routine to access the user's address space using the PXCT instruction.

The location following each XPCW in the EPT contains an instruction that will cause an I/O page fail condition (setting the APR flag), which will usually result in an IOP stopcode.

Using a data structure known as the CONSO skip chain, the interrupt routine polls the devices on that interrupt level and services the interrupt. With the XPCW instruction, control passes to the skip chain. Each channel has its own skip chain, starting at the address pointed to by CHnm+3, whose function is to find the specific device that created the interrupt and then service its needs.

The monitor performs CONSO instructions to decide which device generated the interrupt. If it finds the interrupting device, control passes to the interrupt handling routine. If the device is not requesting an interrupt, the monitor performs a JRST instruction to the next CONSO instruction. If it reaches the end of the CONSO skip chain, it dismisses the interrupt with the following instruction:

```
XJEN CHnm
```

When control passes to the interrupt handling routine, the monitor reads the status of the device, using a CONI or DATAI instruction. On that basis, it may stop the device, advance buffer pointers, or perform cleanup operations. A CONO or DATAO instruction clears the device interrupt status. Failure to do so would cause continual loops in the interrupt handling routine, and eventually the keep-alive count would expire.

The KL processor uses the following instructions to perform I/O:

```
DATAI       CONI
DATAO       CONO
BLKI        CONSZ
BLKO        CONSO
```

KS I/O processing uses the following set of instructions:

```
TIOxb
RDIOb
WRIOb
BCIOb
BSIOb
```

When the interrupt routine is completed, control returns to the routine that was running before the interrupt (which may be another device interrupt at a lower PI level). Each interrupt routine has its own push-down list. The push-down lists are named CnxPD1, where n is the CPU number (omitted for CPU0), and x is the interrupt level (from 1 to 6).

Device service routines preserve the state of the machine as it existed before it was interrupted. They can use AC Block 0, as UUO level does. Accumulators used by the interrupt routine are saved on the stack before processing, and restored when processing is complete.

The SAVnx routines (n = CPU number, omitted if 0, and x = interrupt level) are used to save/switch ACs during device interrupts. For example, SAV1 is the routine to save the ACs for PI Level 1 on CPU 0; SAV11 is the routine to save the ACs for PI Level 1 on CPU 1.

Certain device interrupt routines have dedicated AC blocks, listed below:

AC Block	Used for
0	Exec-mode
1	User-mode
2	Terminal Scanner Interrupt Service
3	File Interrupt Service
4	Network Interrupt Service
5	Reserved for Realtime Interrupt Service
6	KL-paging Microcode
7	Microcode

Interrupt service routines may also need to use the UPT of a job that is waiting for the completion of I/O, rather than the current job. In that case, the UBR and SPT must be modified to point to the correct UPT, and then switched back when the interrupt is through. The monitor routines that accomplish this are SVEUF, SVEUB, and SVPCS. When you are examining a dump, be sure to check the correspondence between the job and the UPT/SPT.

3.8.2 Vectored Interrupts

The KL hardware also uses vectored interrupts, which differ from the standard, nonvectored interrupts in that the vectored interrupt goes directly to the interrupt-handling routine, using a different interrupt location in the EPT. The interval timer, the DTE (doorbell function only), RH10s, and RH20s may do vectored interrupts.

The DTE interrupts to a location in the EPT, which is calculated as follows:

```
EPT+142+10n     ;where n is the DTE number (0-3)
```

For the RH10/RH20 devices, the system has an internal register called IVIR (Interrupt Vector). When an RH10/RH20 device requests an interrupt, the EBOX hardware/microcode dispatches to the location in the EPT calculated as follows:

```
EPT+contents(IVIR)
```

This interrupt method allows the disk interrupt to vector for the standard interrupt location for that channel, providing device independence in the device interrupt handling routine. Thus, the disk RH10 or RH20 can load the IVIR with 40+2n and the magtape RH10 or RH20 will dispatch directly into the middle of the skip chain to service a specific controller.

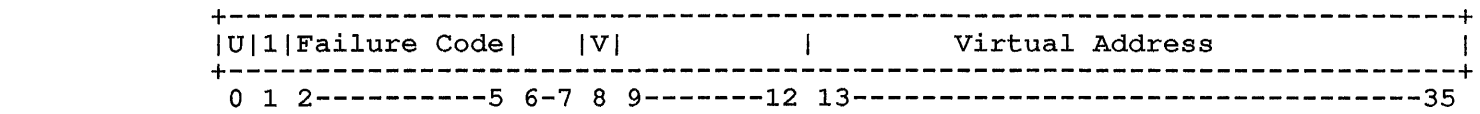
3.9 TRAPS

Traps differ from interrupts in that they are caused by the execution of a specific instruction rather than by some asynchronous event. When a trap occurs, the microcode stores the current PC and flags in the UPT. A new PC double-word, also in the UPT, specifies where control will pass and in what mode the processor will operate (exec or user mode).

3.9.1 Page Fail Traps

When a program attempts to access a page of data that is not available, the hardware generates a page fail trap. A page fail trap can occur for one of two reasons: the user tries to reference an address that cannot be accessed (page not in memory, page write-locked) or a hardware error (AR/ARX parity error, page table parity error) occurs. When a page fail trap occurs, the processor stores information about the trap in location 500 (.USPFW) of the current UPT. This location is known as the page fail word.

The page fail word is formatted differently for a page reference that is not available and for a hardware error. The page reference to an address that cannot be accessed has the following format:



In either type of page failure, the virtual address is stored in Bits 13 through 35. Bit 0 is on if the page failure occurred in user virtual address space. If Bit 0 is off, the failure occurred in executive virtual address space.

If Bit 1 is on, a hardware-detected error occurred, and the failure code is stored in Bits 1-5. The failure codes are:

Code	Meaning
20	No device response on UNIBUS (KS only)
21	Proprietary violation (KL only)
23	Address break (KL only)
24	Illegal indirect word in EA calc (KL only)
25	Page table parity error (KL only)
27	Section number in EA calc greater than 37 (KL only)
36	AR parity error (KL only)
37	ARX parity error (KL only)

If Bit 1 is off, Bits 2-7 have the following format:

Bit	Name	Meaning
2	A	Indicates whether the mapping is valid (0 means a page refill is required).
3	M	Indicates that the page has been modified.
4	S	Reserved for use by the monitor.
5	T	Indicates the type of page reference (0 for reading, 1 for writing).
6	P	Indicates the page is public, if set.
7	C	Indicates whether the page is cachable.

At the same time the page fail word is stored, the flag-PC doubleword is stored at .USPFP (location 501) in the UPT and control passes to the address stored at .USPFP+2 (location 503), which usually contains:

```
EXP     SEILM
```

Certain error handling routines modify .USPFP+2. If this location does not contain SEILM, the cause of the crash may have been a failure in an error recovery routine.

SEILM examines the page fail information stored in the UPT and breaks down the code to find the specific cause of the problem. The error-handling routines are described in Chapter 5.

Note that traps cannot be disabled and they can occur during the service of an interrupt. To return to the correct location, the Flag-PC doubleword is used.

The page fault trap routine uses AC Block 0 and a push-down list in the job's UPT.

3.10 CLOCK LEVEL

All functions that must be performed on a periodic basis are done at clock level, in exec mode. Clock level may be entered in one of the following ways:

- The clock ticked when the processor was in user mode.
- A UUO could not continue execution (was blocked).
- The null job was running and a new job became runnable.
- A UUO completed and a clock tick occurred previously, during the processing of the UUO.

A full cycle occurs when the processor enters clock level as the result of a clock tick; a partial cycle occurs when the processor enters clock level as the result of a job blocking or the null job detecting a newly runnable job. The full cycle starts at location CLKINT; a partial cycle starts at WSCHED or SCDCHK.

A clock tick interrupt occurs at APR interrupt level but is rescheduled to run at Level 7. The clock tick initiates accounting and scheduling functions, then generates a PI Level 7 interrupt.

Only the software will generate a Level 7 interrupt. Level 7 interrupts and ANF-10 network interrupts are controlled by the software. If the scheduler is running, a Level 7 interrupt will not be processed.

During the full cycle, the monitor performs the following tasks:

- User time accounting
- System time accounting
- Processing timing requests
- Checking for hung devices

- Command processing (policy CPU only)
- Choosing a job to run
- Choosing a job to swap

On a partial cycle, the system only performs user time accounting and then selects a job to run. A software interlock prevents a Level 7 interrupt from interrupting the partial cycle.

The scheduler uses the null job's push-down list, NUnPDL and AC Block 0. When a partial or full cycle has been done, the scheduler prepares and runs either a user job or the null job.

3.11 ACCUMULATORS AND PUSH-DOWN LISTS

The first step in finding the correct push-down list (or stack) is to get the right set of accumulators. When a crash occurs, the accumulators are saved in the following places:

AC	Block	Location
0		.CnCA0 = .CnCAC = CRSHAC (for CPU0)
1		.CnCA1
2		.CnCA2
3		.CnCA3
4		.CnCA4
6		Portions of .Cn6
7		Portions of .Cn7

The accumulators are stored when stopcode processing starts. The error processing routines in the monitor use a special stack, ERnPDL. If this is the current stack, be aware that an error may have occurred within the error routine. You must do the mapping, or certain stacks may be inaccessible. Once you have the correct accumulators, the stack currently in use will become readily apparent. You should check the stack to make sure the information in it appears to be current.

This information is fundamental to analyzing any crash, and it may lead directly to the cause of the crash. Often crashes occur because the ACs are misused, the stack is corrupted, or there is confusion in the Priority Interrupt handling system. Software crashes are not always the result of oversights in a complicated algorithm. However, if the crash is due to a more obscure problem, you can use the information you have gathered so far to begin your investigation of the state of the software at the time of the crash.

You can continue your investigation of the crash by comparing the state of the crash with the monitor sources. The following section lists the more prominent monitor modules and their functions.

3.12 MONITOR ORGANIZATION

Like the hardware, the software is composed of modules. Each module of the monitor is compiled separately, and then linked with the others to make up the monitor. A module is a monitor source file with related routines in it. For example, FILUUO deals with monitor calls for file access.

The CLOCK1 module controls the following activities:

- Perform system time accounting
- Perform user time accounting
- Initiate terminal command processing (COMCON)
- Initiate scheduling (SCHED1)
- Initiate swapping (SWPSER)
- Perform job context switching

The modules called from UUO level are organized hierarchically. At the highest level is the UUOCON module, which is responsible for UUO preprocessing, dispatching to the correct routine, and cleaning up after the function has been performed. It also contains the code for some of the UUOs.

For I/O-related UUOs, UUOCON performs device-independent functions before dispatching to a lower level for the device drivers. The drivers are responsible for calling the specific modules that issue the I/O instructions and start the transfers.

Most hardware interrupts enter the CONSO skip chain, which is in COMMON. From there, control passes to the appropriate low-level I/O module, or the skip chain may call a routine in the device driver. Certain types of hardware generate vectored interrupts, which do not access the skip chain.

3.12.1 Monitor Startup Modules

The monitor uses the following modules when it loads and starts the system, discarding some of them when normal timesharing begins:

- SYSINI initializes devices and the monitor's data base in preparation for timesharing. It performs system startup, running an operator dialog to obtain date and time, and performs device initialization. The monitor reclaims the memory space used by SYSINI and uses it for dynamic storage.
- ONCMOD holds the routines related to disk units and file structures. The monitor reclaims the memory for dynamic storage.
- REFSTR refreshes file structures at startup time. The monitor reclaims the memory for dynamic storage.
- PATCH contains extra space to patch the monitor during timesharing. Patch space is reclaimed starting at the location referenced by PATSIZ, and continues up. SYSINI and patch space are preserved when the monitor is run with EDDT loaded.
- AUTCON dynamically configures RH10, RH20, DX10, DX20, CI20, NIA20, and most I/O bus hardware. The monitor does not reclaim AUTCON memory space, because reconfiguration might be required during timesharing.

The following are optional modules that can be omitted from the monitor during monitor generation:

- CPNSER holds the routines that control the processors in a Symmetrical MultiProcessing (SMP) system.
- CTXSER performs job context service.
- IPCSER handles the InterProcess Communications Facility (IPCF).
- LOKCON locks jobs in core.
- PSISER handles the Programmable Software Interrupt (PSI) service.
- QUESER controls the ENQ/DEQ facility.
- RTTRP allows for real-time programming.

3.12.2 Symbol Definition Modules

Some modules contain only symbols that are used by other modules. They do not appear in the assembled monitor:

- F.MAC contains feature test switches.
- S.MAC contains system symbols.
- DEVPRM contains hardware device related symbols.
- DTEPRM contains DTE20 parameters.
- NETPRM contains network parameters.
- JOBDAT contains user job data area addresses.
- D36PAR contains DECnet parameters.
- SCPAR contains Session Control Parameters (DECnet).
- MACSYM contains DECnet macros.
- KLPPRM contains CI20 parameters.
- SCAPRM contains SCA parameters.
- MSCPAR contains MSCP driver parameters.
- ETHPRM contains Ethernet parameters.

3.13 EXAMPLES OF LOCATING FAILURES

The remainder of this chapter illustrates the crash analysis procedure for three types of crashes. The examples display the information gathered with the FILDDT patch file described in Section 2.5. Comments have been added here to describe the information gained from each command; in an actual command file, comments are illegal.

Example 1: IME Stopcode (Illegal Memory Reference in Exec Mode)

```
.RUN MONDDT                                ;Run the monitor-specific FILDDT

File: IME004                               ;Enter crash file name
[Looking at file DSKT:IME004.EXE[30,5653,CAG]]
[Paging and ACs set up from the Exec Data Vector]

diecdb/   CPU0                             ;Check that FILDDT found the
                                           ;right CDB
.cpslf/   CPU0                             ;DIE agrees with FILDDT
.cpdwd/   0                                ;This CPU was in DIE
.cppgd[   700100,,4325                     ;Mapping information saved by
                                           ;DIE
.cptcx[   700100,,4325                     ;It matches that saved by
                                           ;SEILM
.uspfw/   DFDV NTLFRE#(P)                  ;The page fault word
=113001,,552104                            ;A write attempt to 1,,NTLFRE
.USPFP/   CAIA 0   =304000,,0              ;The page fault PC flags
.USPFP+1/   P,,TIC+4                       ;an address
$q/   XCT 0(T4)   t4/   COMTIV+4           ;at which we find part of
                                           ;SCNSER's
1,,COMTIV+4/   MOVEM T1,CRSHWD+3(U)        ;typein processing
u[   1,,552051   _P,,NTLCKC#+4             ;However, U contains an
                                           ;apparent PC,
                                           ;rather than an LDB address
p/   .UUPMP+616,,NU0PDL+22                 ;We are on the clock-level
                                           ;stack
1,,NU0PDL+22/   P,,CTICOM#+5   ^           ;The call within SCNSER which
                                           ;failed
1,,NU0PDL+21/   ADD 0   ^                  ;some saved data
1,,NU0PDL+20/   P,,TTYCM7#+4               ;The return PC from the call
                                           ;to SCNSER
ttycm7?                                    ;Where is this label defined?
COMCON                                     ;In COMCON.
                                           ;This is part of the TTY
                                           ;command.
.cpcml/   P,,NTLCKC#+4                     ;COMCON's saved LDB address
                                           ;has the same incorrect value
                                           ;as AC U.
.cpisf/   .UUPMP+602,,NU0PDL+6   $c        ;However, COMCON's saved PDL
                                           ;pointer
1,,NU0PDL+6[   4,,15772                    ;points at a likely LDB
                                           ;address
$q+ldbclp/   CAIL U,43711                  ;And this LDB has a command
                                           ;line
                                           ;pointer established
.-ldbclp+ldbtit/   CCI 43705               ;So we trace its input chunk
                                           ;stream
=1400,,43705                               ;(POINT 12,addr,35)
ttchks=20                                  ;These chunks are 16 words
                                           ;long
4,,43705/   UNWNDC,,PLTS5A#+1   $12t       ;12-bit ASCII, starting next
                                           ;word
4,,43706/   tt
4,,43707/   21_
4,,43710/   115
4,,43711/    ec
4,,43712/   ho^@

;"tt 21_115 echo" was the command being executed.
```

```
1,,ttycmd/   PUSHJ P,SSEC1              ;We proceed to trace the
                                        ;execution
1,,TTYCMD+1/   PUSHJ P,SAVE2            ;of the command to see where
                                        ;U got
1,,TTYCMD+2/   PUSH P,U                 ;clobbered.
1,,TTYCMD+3/   MOVE P1,U
1,,TTYCMD+4/   PUSHJ P,CTEXT1
1,,TTYCMD+5/   CAIE T3,JOBVER
=302200,,137                            ;"_" is character code 137,
1,,TTYCMD+6/   JRST TTYC0#              ;so we skipped this
                                        ;instruction,
1,,TTYCMD+7/   PUSHJ P,NTLCKJ           ;and executed this code.

;NTLCKJ is called as a result of the NETDBJ macro

ntlckj/   PUSHJ P,NTCHCK                ;This routine checks for
                                        ;nesting of
1,,NTLCKJ+1/   JRST NTLCKJ+3            ;the NETSER interlock (false)
1,,NTLCKJ+2/   POPJ P,0
1,,NTLCKJ+3/   SKIPE .CPISF             ;It then checks for COMCON
                                        ;(true)
1,,NTLCKJ+4/   JRST NTLCKC#
1,,NTLCKC#/   PUSHJ P,NTLCKI            ;Get the interlock
1,,NTLCKC#+1/   JRST ANFMDL+5           ;(failure branch not taken)
1,,NTLCKC#+2/   POP P,0(P)              ;Proceed as a coroutine
1,,NTLCKC#+3/   PUSHJ P,@P(P)
u/   P,,NTLCKC#+4                       ;This is the return address
                                        ;in U!

;At TTYCMD+2, we pushed U on the stack.  We then called a coroutine.
;We should have called NTLCKJ before we pushed U onto the stack.
```

Example 2: UIL Stopcode (UUO at Interrupt Level)

```
.RUN MONDDT                             ;Run the monitor-specific FILDDT

File: uil002                            ;Enter crash file name
[Looking at file DSKT:UIL002.EXE[30,5653,CAG]]
[Paging and ACs set up from the Exec Data Vector]

diecdb/   CPU0                          ;Check that FILDDT found the
                                        ;right CDB
.cpslf/   CPU0                          ;DIE agrees with FILDDT
.cpdwd/   0                             ;This CPU was in DIE
.usmuo/   CAIA 0                        ;UUO PC flags
.USMUP/   BOOTPA   =20                  ;The UUO was in the ACs
.USMUE/   MAPBAX+1   =702432            ;UUO effective address
.USUPF/   TLNE T1,4   =603100,,4        ;AC block 3 was current,
$5u/   .CPCA0                           ;but FILDDT set up AC block 0,
.cpca3$5u                               ;so we set up AC block 3 by
                                        ;hand.
p/   .UUPMP+623,,C4PD1+23               ;We have an interrupt level
                                        ;stack
C4PD1+23/   CAIA FREIN5#+5              ;which points to this return PC
FREIN5#+5/   JRST FREIN3#   ^
FREIN5#+4/   PUSHJ P,CALMDA#            ;We had called this routine to
                                        ;notify
CALMDA#/   MOVE T1,0(U)                 ;the MDA of a new disk unit
CALMDA#+1/   MOVEI T2,0
CALMDA#+2/   PUSHJ P,SNDMDC
CALMDA#+3/   POPJ P,0
CALMDA#+4/   JRST F                     ;Aha!

;An editing error would seem to be responsible.
;The "JRST F" should be a "JRST CPOPJ1".
```

Example 3: KAF Stopcode (Keep-Alive Failure)

```
.RUN MONDDT                                ;Run the monitor-specific FILDDT

File: KAF003                               ;Enter crash file name
[Looking at file DSKT:KAF003.EXE[30,5653,CAG]]
[Paging and ACs set up from the Exec Data Vector]

diecdb/   CPU0                             ;Check that FILDDT found the
                                           ;right CDB
.cpslf/   CPU0                             ;DIE agrees with FILDDT
.cpdwd/   0                                ;This CPU was in DIE
.cppgd[   700100,,4325                     ;Mapping information saved by
                                           ;DIE
.cpcpi[   1,,777                           ;CONI PI, result saved by DIE

kafloc/   XPCW @.CPKAF                     ;Where a KAF STOPCD gets its
                                           ;start
                                           ;(RSX20F does an XCT of this
                                           ;location.)
AP0KAF#
AP0KAF#/   CAIA 0   =304000,,0             ;PC flags
AP0KAF#+1/   P,,LOKNPI   $c                ;and location
AP0KAF#+2[   4000,,0   $s                  ;new PC flags
AP0KAF#+3/   APRKAF                        ;and location
APRKAF/   MOVEM P,.CPSVP                   ;Where the real stack pointer
                                           ;was saved
/   .UUPMP+603,,NU0PDL+7                   ;So we examine it
NU0PDL+7/   WRSLOC,,0   ^
NU0PDL+6/   P,,XMTECH#+17   ^              ;We're inside XMTECH in
                                           ;SCNSER
NU0PDL+5/   P,,TTDSC1#+1                   ;from the call of XMTCHR in
                                           ;TTDINT.
NU0PDL+6/   P,,XMTECH#+17                  ;Let's look for a loop in
                                           ;XMTECH.
$q/   JRST XMTCH1#                         ;We're about to restart
                                           ;XMTCHR
1,,XMTCH1#/   PUSHJ P,LOKSCI
1,,XMTCH1#+1/   SKIPE T1,W(U)              ;Check for output state bits
$[   100,,0                                ;We have one,
1,,XMTCH1#+2/   JFFO T1,APCSET+11          ;so this jumps.
1,,APCSET+11/   JRST @XMTDSP#(T2)
1,,XMTDSP#/   SETZ XMTXFP#                 ;Bit 11 was set,
.+11./   SETZ XMTMIC#                      ;so we dispatch through this
                                           ;location,
1,,XMTMIC#/   MOVE T2,ARSLOC(U)            ;getting here.
$[   430400,,2                             ;These are our LDBMIC bits
1,,XMTMIC#+1/   TLNE T2,20                 ;(true)
1,,XMTMIC#+2/   SKIPE KAFLOC(U)            ;(skipped)
1,,XMTMIC#+3/   JRST MICLG3#
1,,MICLG3#/   PUSHJ P,HPOS                 ;Get horizontal position
1,,HPOS/   PUSHJ P,SSEC1
1,,HPOS+1/   LDB T2,LDPWID                 ;Get terminal width setting
$1t/   10 10 JOBBLT+4(U)                   ;(POINT 8,addr,35-8)
$[   2000,,50020                           ;from this value
$q'400=4,,120                              ;Dropping the low-order 8
                                           ;bits reveals
1,,HPOS+2/   ADD T2,JOBERR+1(U)            ;a width of ^O120
$/   -120                                  ;Adding this gives zero
1,,HPOS+3/   POPJ P,0   $
1,,MICLG3#+1/   JUMPN T2,XMTOK#            ;(Branch not taken)
1,,MICLG3#+2/   SKIPE T2,ARSLOC(U)         ;LDBMIC again
$[   430400,,2
```

```
1,,MICLG3#+3/    TLNN T2,140              ;(false)
1,,MICLG3#+4/    JRST XMTOK1#
1,,XMTOK1#/    TLNE T2,40                 ;(true)
1,,XMTOK1#+1/    JRST XMTECH#             ;(skipped)
1,,XMTOK1#+2/    SKIPN KAFLOC(U)
$[    0                                  ;(false)
1,,XMTOK1#+3/     JRST XMTCH2#
1,,XMTCH2#/    SOSGE T4,BOOTPA(U)
$[    0                                  ;(non-skip)
1,,XMTCH2#+1/    JRST ZAPBUF#
1,,ZAPBUF#/    MOVSI T1,DTEDRW#+31
=205100,,200
1,,ZAPBUF#+1/    TDNE T1,W(U)
$[    100,,0                             ;(true)
1,,ZAPBUF#+2/    JRST ZAPPI1#             ;(skipped)
1,,ZAPBUF#+3/    SETZM BOOTPA(U)
1,,ZAPBUF#+4/    MOVE T1,F(U)
$[    1400,,37654
1,,ZAPBUF#+5/    CAME T1,R(U)
$[    1400,,37654                        ;(true)
1,,ZAPBUF#+6/    PUSHJ P,RCDSTP#          ;(skipped)
1,,ZAPBUF#+7/    SKIPL SLJOBN#(U)
$[    0                                  ;(false)
1,,ZAPBUF#+10/    JRST XMTECH#
1,,XMTECH#/    MOVE T1,JOBBLT+2(U)
$[    200,,200115
1,,XMTECH#+1/    TLNE T1,100000           ;(true)
1,,XMTECH#+2/    JRST ECHCNR#             ;(skipped)
1,,XMTECH#+3/    MOVE T1,JOBBLT+3(U)
$[    10,,400
1,,XMTECH#+4/    TLNN T1,10               ;(true)
1,,XMTECH#+5/    TRZ T1,400               ;(skipped)
1,,XMTECH#+6/    SKIPL WRSINS+1(U)
$[    0                                  ;(false)
1,,XMTECH#+7/    TRNE T1,400              ;(false)
1,,XMTECH#+10/    TRNE T1,3000            ;(true)
1,,XMTECH#+11/    TLNE T1,400             ;(skipped)
1,,XMTECH#+12/    CAIA 0
1,,XMTECH#+13/    JRST ECHCNR#            ;(skipped)
1,,XMTECH#+14/    HLLZ T1,W(U)
$[    100,,0
1,,XMTECH#+15/    JUMPE T1,XMTIDL#        ;(branch not taken)
1,,XMTECH#+16/    PUSHJ P,UNLSCI
1,,XMTECH#+17/    JRST XMTCH1#            ;We're back where we started.

;We have uncovered a loop in XMTCHR processing.
;Comparison with the source shows that this occurs when
;TTY DEFER is set and the line is under MIC control.
;This can be solved by inserting a "TLZ T1,LOLMIC" just before the
;"JUMPE T1,XMTIDL" at XMTECH+15.
```

CHAPTER 4

EXAMINING THE DATA STRUCTURES

After you have isolated the failure in the monitor code, you will need to interpret the source code to make corrections. You must be able to read and understand the source code, and compare it to the instructions in the crash file.

For this purpose, the monitor uses symbols to represent almost all values: bits, words, offsets, instructions, and more. Symbols make the code easier to read and modify. This chapter describes the conventions used in choosing symbolic names, and the tools for finding the symbols in the source code.

4.1 SYMBOLS

This section describes the types of symbols, how they are named and where they are stored. There is more information about symbolic representation and usage in the MACRO Assembler Reference Manual.

The TOPS-10 software is made up of modules, each of which has its own symbolic definitions. By default, a symbol is defined and used only in a single module. The same symbolic name can be defined and used differently by different modules.

A global symbol is available to modules other than the one in which it is defined. The addresses of shared tables or commonly used subroutines are examples of symbols defined as global.

4.1.1 Naming Conventions

TOPS-10 uses a consistent scheme for naming and using symbols. This helps you read and understand the sources. For example, the monitor accumulator locations have names that are consistent throughout most of the monitor, and they have the following values:

Table 4-1: Monitor Accumulators

Number	Name	Description
0	S	Contains the I/O status word from a DDB (DEVIOS) while the monitor is processing I/O operations.
1	P	Contains the push-down list pointer currently in use.
2	T1	is an unpreserved, temporary AC.
3	T2	is an unpreserved, temporary AC.
4	T3	is an unpreserved, temporary AC.
5	T4	is an unpreserved, temporary AC.
6	W	usually contains the pointer to the process data block (PDB) or the tape controller data block (KDB).
7	M	contains the user virtual address for getting and putting data during UUO execution. During command processing, M contains the command dispatch bits.
10	U	contains the Unit Data Block (UDB) address (for FILSER or TAPSER), or the Line Data Block (LDB) address in SCNSER.
11	P1	is a preserved AC.
12	P2	is a preserved AC.
13	P3	is a preserved AC.
14	P4	is a preserved AC.
15	J	contains the job number, high segment number, or disk controller data block (KON) address at interrupt level.
16	F	contains the DDB address during I/O. It is used as a temporary register in non-I/O situations.
17	R	is a general-purpose, scratch AC.

The uses for each accumulator may change from one release of the software to the next. You should always check the source code to see how the program uses a specific accumulator in a specific situation.

To restore accumulators correctly, several standard subroutine return sequences have been set up. The main subroutine does a JRST to one of the following locations:

Subroutine Name	Function
CPOPJ	Regular POPJ return
CPOPJ1	Increment return address and then POPJ (skip return)
CPOPJ2	Double skip return
TPOPJ	Restore T1 and return
TPOPJ1	Restore T1 and skip return
T2POPJ	Restore T2 and return
T2POJ1	Restore T2 and skip return
MPOPJ	Restore M and return
FPOPJ	Restore F and return
FPOPJ1	Restore F and skip return
WPOPJ	Restore W and return
JPOPJ	Restore J and return

Symbolic names for locations in the monitor are one to six characters in length. Usually, all six characters are used. The first three characters identify the data structure and type of symbol; the last three describe the unique word or field.

Symbols for data structures usually take one of two forms:

```
dddxxx

.ddxxx
```

where ddd or dd represents the data structure and xxx represents the field or word. Some data structures are:

Symbol	Data Structure
.C0xxx	CPU data block for CPU0 (in low segment)
.C1xxx	CPU data block for CPU1 (low segment)
.Cnxxx	CPU data block (n = CPU number)
.CPxxx	CPU data block for current CPU (high segment)
.PDxxx	Process data block
.USxxx	User Process Table
.CTxxx	Context block offsets
.CXxxx	Context saved parameters block offsets
ACCxxx	Access table
BAFxxx	Bad allocation file block
CHNxxx	Channel data block
DEVxxx	Device data block
HOMxxx	Home blocks
JBTxxx	Job tables
JOBxxx	Job data area
KDBxxx	Common controller data block

KONxxx	Disk controller data block
LDBxxx	Line data block
NMBxxx	File name block
PPBxxx	Project programmer number data block
RIBxxx	Retrieval information block
SABxxx	Storage allocation block
STRxxx	File structure data block
TKBxxx	Tape controller data block
TTFxxx	Forced command table
TUBxxx	Magnetic unit data block
UDBxxx	Common unit data block
UFBxxx	UFD data block
UNIxxx	Disk unit data block

Byte pointers referencing fields within these data structures are named in the following way:

aacbbb

where:

aa represents the first two letters of the three letter name
c represents one of Y, M, B, P, S, or N
bbb represents the name of the pointer

For example, a pointer in the BAF block is named BAYbbb.

Bits within words are usually defined as one of the following:

xx.yyy

xxPyyy

where:

xx is the data structure
yyy is the bit name

Here are some examples:

TO.yyy	Bits in CONO TIM,
TI.yyy	Bits in CONI TIM,
LI.yyy	Bits in CONI/CONO PI,
LP.yyy	Bits in CONO/CONI APR,
JS.yyy	Bits in JBTSTS (job status word)

4.1.2 Symbol Files and Monitor Generation

Several of the monitor modules contain only symbol definitions. They are used to define the software features and hardware configuration in the process of building the monitor.

The first step in generating the monitor is to run the MONGEN program (MONitor GENerator). It asks a series of questions about the hardware configuration and the software options to be selected. For more information about the MONGEN program, refer to the TOPS-10 Software Installation Guide.

MONGEN creates symbol-definition files that describe the aspects of the system. After running MONGEN, the system installer can build the monitor with standard source code libraries, or, if changes have been made to the sources, the monitor must be built from separate modules.

If the systems programmer does not want to make any changes to the standard release of TOPS-10, the programmer compiles the common modules and loads them with a distributed library file of the remaining monitor modules.

It is common practice, however, to make modifications to the TOPS-10 source code. If changes have been made to one or more TOPS-10 source modules, the modules of the monitor must be assembled separately to build a library file.

Next, the MONGEN files must be assembled with the monitor's common modules, which are:

- COMMOD defines the disk data base.
- COMDEV defines all other devices.
- COMMON describes the CPU, memory, scheduler, job tables, and so forth.

4.2 READING THE CODE

There are two important sources of information in analyzing system crashes: the crash file and the monitor source code. The key to successful crash analysis is to be able to compare the crash file and the source code. Refer to the TOPS-10 MACRO Assembler Reference Manual for information about the source code and assembler language conventions.

4.2.1 How to Use a CREF Listing

The listings of the monitor source code should be cross-referenced (CREF) listings. You will find a CREF listing more useful than unassembled source code because CREF produces a sequence-numbered assembly listing, followed by tables showing where symbols are defined and referenced. To find a symbol in a module, you need only look in one of these tables, which points to a line number in the assembly listing. The CREF program is described in the TOPS-10 User Utilities Manual.

4.2.2 Macros

A macro is a set of frequently used instructions in a sequence that can be called with a single pseudo-instruction. A macro allows the system programmer to supply arguments to a single instruction, which the assembler expands to the desired instruction(s). Macros make it difficult to read the code, however, unless you understand the purpose of some commonly-used macros.

Several macros are used to define symbols. These macros are defined in S.MAC:

- XP (A,B) defines the global symbol A as being equal to B, but DDT will not display A (A==:B).
- ND (A,B) defines A as a global symbol equal to B using the XP macro, if A has not already been defined.

There are many other commonly-used macros in the monitor, including:

- $XHGH, $HIGH, $LOW, $CSUBS, and $ABS, which place code in the extended high segment, high segment, low segment, common subroutines, and an absolute physical location, respectively. Code usually goes in the monitor's high segment, which is write-protected; data goes in the low segment, which is writable. $ABS is usually used to place data in physical Page 0 of memory (Words 0-777).

- Ordinarily, an instruction in a user program is executed entirely in user address space, and an instruction in the monitor is executed in the executive address space. But to facilitate communication between the monitor and users, the monitor can execute instructions to refer to locations in the other address space. This feature is implemented by the previous context execute (PXCT) instruction. The following macros allow you to execute PXCT:

  1. EXCTUX moves information from the user's address space to the monitor.

  2. EXCTXU moves information from the monitor's address space to the user's.

  3. EXCTUU moves information from one location in the user's address space to another.

- The USERAC and EXECAC macros generate code to switch between accumulator blocks. USERAC switches to AC Block 1. EXECAC switches to the monitor's AC block. If no argument is given, the switch is made to AC Block 0. If an argument is given, the AC block specified by the argument is used.

4.2.3 Conditional Assembly

Parts of the monitor are assembled on an optional basis, depending on conditions defined by an assembler IF statement.

F.MAC has most of the symbol definitions that are used for conditional assembly. Most symbols are of the form FTxxxx, where FT stands for Feature Test and xxxx is the specific option. Some of the feature test symbols and the functions they enable are:

FTKL10	KL10 processor
FTKS10	KS10 processor
FTMP	SMP (multiple-processor) system
FTDUAL	Dual-ported disks are supported

4.2.4 Finding Symbols

When trying to find a symbol in the monitor, you should follow these steps:

1. Check the symbol table at the back of the CREF listing you are currently looking at. If one of the numbers after the symbol name has a pound sign (#) next to it (as in number#), the symbol is defined on that line of the code. If the symbol appears in the CREF listing with no line numbers that have pound signs, the symbol is global, or it is defined in a universal file.

2. If a symbol is defined in a universal file, check your CREF listings of S.MAC, DEVPRM.MAC, DTEPRM.MAC, NETPRM.MAC, MACSYM.MAC, and JOBDAT.MAC. If the symbol is not defined in any of these modules, the symbol is probably global.

3. If the symbol is not defined in the source module or the universal files, you must obtain a GLOB listing of the monitor. The GLOB listing points to the modules where global symbols are defined and used. Search the symbol tables at the back of those modules. (GLOB creates listings of global symbols from binary files. It is described in the TOPS-10 User Utilities Manual.)

4. If you are not successful in searching the listings, run the monitor-specific FILDDT and use the "symbol?" instruction to find the module where it is defined. If you type a symbol name followed by a question mark, FILDDT displays the module where it is defined.

   Monitor parameters used by certain modules are often associated with global symbols that are defined in those modules. LINK can detect the parameters that are assigned different values by different modules. FILDDT lists only one module where each global symbol is defined, and displays a "G" next to global symbols. If a symbol is not global, several modules may be listed as containing the symbol. You can unlock the local symbols for a certain module by issuing the following FILDDT command:

```
module$:
```

The monitor uses many fixed and dynamic data structures for job control, for memory management, and for device control. Some of the data structures that are important for crash analysis are described briefly in the following sections. For more specific information about the contents of these data structures, refer to the TOPS-10 Monitor Tables descriptions.

4.3 JOB-RELATED DATA STRUCTURES

Information about a job is kept in the monitor's low segment or in per-process address space (such as the UPT and JOBDAT). Most of the following data structures are job tables, and have JBT as the first three letters of the symbolic name (an exception is TTYTAB). Most job tables have one entry in the table per job. Some of these tables also have entries for high segments, because the monitor sometimes treats high segments like jobs.

The following job tables hold information about the status and condition of the job:

- JBTSTS, JBTST2, and JBTST3 contain the current state of the job, including the processor queue, execution status, swapping status, event wait condition, and whether the job is logged in.

- JBTCQ and JBTCSQ hold the processor queue number, subqueues, and scheduler class for each job. These tables are organized as a series of linked lists.

- JBTSWP holds the disk address of the swapped-out job.

The following tables hold the features and options for the job:

- JBTPRV holds the job's privileges.
- JBTSPL holds the spooling bits for the job. These control how and when requests to spooled devices (LPT, PLT, and so forth) are handled.
- JBTSCD holds the job's scheduler class.
- JBTWCH controls the WATCH information displayed by the monitor for the job.
- JBTLIM holds the CPU run-time limit for the job. The monitor checks this value before processing batch jobs.

The following tables describe the user and the program being run:

- JBTNAM holds the program name.
- JBTPPN holds the project-programmer number.
- JBTLOC holds the ANF-10 node number for remote spooling.
- JBTUPM, a component of the SPT, points to the physical page of this job's UPT when the job is swapped in.

The following tables are used to point to the location of another job-related table:

- JBTSGN contains the address of the job's high segment descriptor blocks.
- JBTPDB holds the address of the job's Process Data Block (the PDB).

The Process Data Block (PDB) stores more job-related information, including:

- User name (in SIXBIT)
- Accumulated run-time, core and disk usage
- Virtual memory limits
- IPCF information
- Current program name and directory
- The job's search list
- Context flags, quotas, and chain pointers

The words in the PDB are named .PDxxx, where xxx is the specific word.

The remainder of the job-related information is stored with the job itself in JOBDAT or the UPT. JOBDAT holds the user accumulators when the job is not running, the starting address of the program, the addresses of DDT and the symbol table, and other locations required to run the program.

4.4 CPU DATA STRUCTURES

The CPU Data Block (CDB) contains most of the CPU-specific information. On a multi-processing system of two or more KL processors, the monitor maintains a different CDB for each processor.

The CDB is is divided into two sections: one for constant definitions and the other for variable definitions. The constants area holds such information as the following:

- CPU number
- Instructions to execute in certain situations, such as device interrupts
- Bit masks
- Hardware constants

The variables area stores such information as:

- Stopcode information
- Hardware error information
- Performance information
- Frequency of certain events
- Per-CPU patch space

The CDB words are named .CPxxx or .Cnxxx, where n is the CPU number and xxx is the unique symbol for the word. On a single-CPU system, the .CPxxx format is always valid. In a multi-CPU system, .CPxxx refers to the current CPU (or, in FILDDT, the CPU that is currently mapped). To refer to the data on a CPU other than the one you are currently accessing, use the .Cnxxx formation, replacing n with the CPU number (0 through 2).

The COMMON module contains the CWRD macro to define constants and variables in the CPU Data Block (CDB). CWRD is called in the following way:

```
CWRD (nam, val, len, lbl)
```

where:

nam	is the word name
val	is the optional value to store in this address (default=0)
len	is the optional length of storage area (default=1)
lbl	is the optional alternate lable for old-style CPU0 references

For example, the following instruction defines .CnOK as a global symbol with a value of -1:

```
CWRD (OK,-1)
```

For example, the following instruction defines .CnACN as a word in the CDB variables area, with the alternate name APRSTS:

```
CWRD (ACN,,1,APRSTS)
```

The scheduler uses a series of tables to control the use of the CPU. Some of the scheduler tables are:

- QBITS determines how the scheduler should move a job from one wait state to another.
- SSCAN and SQSCAN tell the scheduler the order and direction the run queues should be scanned to find a runnable job.
- Transfer tables control the destination queue for requeued jobs.

The AVALTB table contains flags to indicate whether a sharable resource has become available. A sharable resource is a portion of the monitor that can only be used by one process at a time.

Some of the sharable resources are:

Name	Resource
AU	Alter UFD (one per UFD, per structure)
CX	PDB/context block interlock word (one per job)
DA	Allocate disk space (one per disk unit)
EV	Use executive virtual memory
MM	Memory management (for modifying the data base)

REQTAB contains the number of jobs waiting for each resource. A value of -1 in REQTAB indicates that the resource is available; a value of zero means that a job has the resource and no other job is waiting.

INTTAB describes each hardware interrupt routine. Each two-word entry contains the PI level, the address of the DDB (or prototype DDB), and the CPU to which the device is connected.

4.5 MEMORY DATA STRUCTURES

The monitor uses PAGTAB and PT2TAB to allocate user and monitor memory space (usually referred to as "core"). The tables contain one word for each page of physical memory. A job's allocation of pages is maintained as a forward linked list using PAGTAB, and as a backward linked list with PT2TAB. All the pages for a job are linked using the right half of a PAGTAB and PT2TAB entry. PAGPTR contains the starting address for the linked list of free pages. The left half of the PAGTAB and PT2TAB entries contain bits describing how the page is used: whether it is locked, locked in executive virtual memory, and so forth. The monitor uses PT2TAB to obtain information about swapped-out pages.

MEMTAB also has one entry for each page in memory. The monitor uses MEMTAB during swapping and paging requests, to keep track of where pages are stored in the swapping area and which page to transmit next.

The monitor also maintains areas of dynamic storage called free core, allocated in four-word chunks, using a bit table to determine which chunks are in use and which are not.

4.6 COMMAND PROCESSING TABLES

The command processor uses several tables to verify and control monitor commands, including COMTAB, DISP, and UNQTAB. COMTB2, DISP2, and UNQTB2 are used to describe SET commands. COMTBC, DISPC, and UNQTBC are for customer use.

TTFCOM is the forced commands table. This table is used if the monitor determines that a job must execute a command immediately, regardless of the job's current state. The monitor does not place the commands in the TTFCOM table into a terminal input buffer before processing the command.

4.7 UUO PROCESSING TABLES

UUOTAB contains the addresses of the operator-dependent UUO routines. The addresses are arranged in order of UUO opcode, with one halfword devoted to each address. The UUO handler verifies whether the UUO is valid and dispatches to the address stored in UUOTAB. If the UUO is illegal, control passes to an error routine called UUOERR.

The tables UCLJMP and UCLTAB are used for the CALL and CALLI UUOs. UCLTAB contains the names for the CALL UUOs; UCLJMP contains the addresses of the CALL/CALLI routines.

4.8 I/O DATA STRUCTURES

The most dynamic and interrelated data structures in the monitor are those related to I/O. The data structures that are common to almost all I/O operations are the Job Device Assignment table (JDA), the device data block (DDB), and user I/O buffers. Other data structures exist to control specific types of hardware: disk or tape units, device controllers, or software I/O channels. For certain devices (such as disk), an extra level of organization is imposed: the logical file structure, requiring additional data structures.

4.9 THE JOB DEVICE ASSIGNMENT TABLE

The Job Device Assignment table (starting at USRJDA in the UPT) holds the addresses of the DDBs currently in use by the job. It is indexed by the software channel number. When the user issues a UUO to initiate I/O, a software channel number must be supplied, which is associated with the device or file to be accessed. More channels are available in the extended channel table, stored in funny space. Extended channel table entries are in the same format as the JDA table. The contents of .USCTA in the UPT point to the extended channel table.

The left half of the JDA entry for a channel contains status bits that indicate which UUOs have been successfully completed for this channel. Following are some of the status bits, which are defined in S.MAC:

Bit	Symbol	Meaning
0	INITB	An OPEN or INIT has been done on this channel.
1	IBUFB	INIT specifying input buffers was done.
2	OBUFB	INIT specifying output buffers was done.
3	LOOKB	LOOKUP was done.

4	ENTRB	ENTER was done.
5	INPB	INPUT was done.
6	OUTPB	OUTPUT was done.
7	ICLOSB	CLOSE (input side of channel) was done.
8	OCLOSB	CLOSE (output side of channel) was done.
9	INBFB	INBUF was done.
10	OUTBFB	OUTBUF was done.
11	SYSDEV	System device, or [1,4] for disk area.
12	RENMB	RENAME UUO in progress.
13	RESETB	RESET UUO in progress.

4.10 THE DEVICE DATA BLOCK

The monitor uses the Device Data Block (DDB) to control each device. The information in the DDB comes from a monitor call and is read by the interrupt handling routine to perform the I/O. The handler records the status of the operation in the DDB. The monitor and the user can read the status of the I/O operation from the DDB. For example, the monitor can detect a hung condition by checking a timer in the DDB.

User programs can include the same instructions to perform I/O with disk devices, magnetic tapes, and line printers, because the format of the DDB is similar for all devices. The monitor handles the devices differently by handling the DDBs differently and by ignoring any information in the DDB that is not relevant to the specific device. For example, the monitor creates DDBs for single-user devices when the system comes up; these DDBs are never deleted. The monitor simply updates the information in the data block. For sharable devices, such as disk devices, the monitor creates DDBs dynamically in the user's funny space, when a channel is opened. The DDB for the channel is deleted when the channel is closed. Spooled devices, such as line printers, are handled in a similar manner.

A device on an ANF-10 network front-end requires a special kind of DDB, because remote stations can have line printers or card readers. When a user first accesses the remote device, NETSER creates a DDB for the device. COMDEV contains the prototype network DDB.

NETDEV contains the I/O routines for specific network devices. For example, the RDXSER routine, in NETDEV, handles RDA devices, and the TSKSER routine handles intertask communication.

DTESER contains the DTE device handling routine for DECnet front-ends (DN20s running MCB software). The DTE DDB is dynamically created for the purpose of loading and dumping the front-end memory.

All DDBs include the following locations:

- DEVNAM contains the SIXBIT device name.
- DEVBUF contains the addresses of the user buffers.
- DEVMOD describes the type of device.
- DEVIOS is the I/O status word.
- DEVSER contains a pointer to the next DDB and the address of the dispatch table.

Most devices are configured dynamically by the monitor. A prototype DDB exists for each type of device. When a recognized hardware device is detected by the monitor, a DDB is created and the contents of the prototype DDB are copied into the new DDB. Then, specific information (device names, unit numbers, and so forth) are filled in. Prototype DDBs are linked into the DEVLST chain. They may also by found by indexing into DDBTAB using the .TYxxx value for the device in question. For example, .TYMTA has a value of 2. DDBTAB+2 contains the address of the prototype magtape DDB.

Device	Module	DDB	Hardware Interface
Card reader	CDRSER	CR1DDB	CR10 I/O BUS
	DCRSER	DCRDDB	CD20/RSX-20F
Card punch	CDPSER	CDPDDB	CP10/CP10D I/O BUS
Line Printer	DLPSER	DLPDDB	LP20/RSX-20F
	LP2SER	LP2DDB	LP20/UNIBUS (KS10 only)
	LPTSER	LPTDDB	BA10/LP100 I/O BUS
Magtape	TAPUUO	TDVDDB	All interfaces
Plotter	PLTSER	PLTDDB	XY10 I/O BUS
Paper tape reader	PTRSER	PTRDDB	CR04 I/O BUS
Paper tape punch	PTPSER	PTPDDB	CR04 I/O BUS

4.11 FINDING DDB INFORMATION

The following example shows how to look at a crash file to find the DDBs and other information about I/O. In this example, Job 7 was running LPTSPL. You must first issue the mapping command ($6U), to map the UPT through Job 7, rather than through the UPT for the job that was currently running. A typical command sequence might be:

```
JBTNAM 7$6T/   LPTSPL
JBTUPM 7[   42000,,152   .-n$6U
```

where n is the CPU number of the CPU that is currently mapped.

The commands to look at the user job device assignment table are:

```
 USRJDA[     506000,,65334            ;Channel 0
.UPMP+652[   506000,,65414            ;Channel 1
.UPMP+653    0                        ;Channel 2
.UPMP+654    0                            .
.UPMP+655    0                            .
.UPMP+656    0                            .
.UPMP+657    0
.UPMP+660    0
.UPMP+661    0
.UPMP+662    0
.UPMP+663    0
.UPMP+664    0
.UPMP+665    0
.UPMP+666    0
.UPMP+667    0
.UPMP+670    0                        ;Channel 17#(octal)
```

The commands to display the devices associated with the DDBs are:

```
$6T     65334/   LPT0
        65414/   LPT1
```

Both devices are printers, controlled by LPTSPL.

The left half of each JDA entry contains bits indicating the UUOs executed for that channel. The left half of the JDA entry shown above contains 506000, which indicates Bits 0, 2, 6, and 7 turned on. These bits are set for the following UUOs:

```
Bit 0     OPEN/INIT
Bit 2     OUTBUF
Bit 6     OUTPUT
Bit 7     CLOSE (input side, as input is not allowed in LPTs)
```

The user buffers are the next source of information. Find the output buffer for LPT261 by examining the left half of the DEVBUF word in the DDB, which holds the address of the output ring header:

```
65414+DEVBUF/   45150,,0          ;output-header,,input-header
```

The user buffers are always in user address space. To examine locations in user address space, switch mapping to the user job. JBTUPM shows that the UPT starts at 152; therefore, the command to switch mapping to user space is:

```
152$1U
```

Now you can examine the contents of the output ring header:

```
45150/   44351                    ;Current buffer addr+1
45151/   10700,,0                 ;Byte pointer
45152/   -1                       ;Byte count
```

Location 45150 contains the address of the second word of the current buffer, which contains the address of the next buffer in the buffer ring, and so forth. You can locate all the buffers in the ring using the same method:

```
44351/   176,,44551               ;Buffer 1
44551/   176,,44751               ;Buffer 2
44751/   176,,44151               ;Buffer 3
44151/   176,,44351               ;Buffer 4
```

Therefore, there are four buffers set up. The right half of the header word points to the next buffer in the ring. The left half holds the use bit and the buffer size. Bit 0 is the use bit (BF.IOU), and its setting indicates the following state in the following types of buffers:

	Buffer Empty	Buffer Full
Input Buffer	0	1
Output Buffer	1	0

In the left half of the header words listed above, Bit 0 is off, indicating that the output buffers were full. The remainder of the left half holds the buffer size, in this case, 176 (octal) words.

To read the contents of the first buffer, use the following commands:

```
$$7T
44151/   @pHt@
44152/   }
44153$0T/   GLE  File format:ASCII  Print mode:ASCII /DELETE _^L
GGGGGGGGGGGG     RRRRRRRRRRRR    IIIIIIIII   PPPPPPPPPPPP    EEE
EEEEEEEEEEEEEEE ...
```

The rest of the buffer contains the banner page printed by LPTSPL immediately before printing a file. LPTSPL had just begun printing a file when the system crashed.

Job 7 is using two DDBs, but it is also important to check the extended channel table for the job. In this case, it reveals more DDBs. Note that the left half of the pointer to the extended channel table does NOT contain a section number, as might seem immediately apparent. Only the right half of this word is a valid pointer to data:

```
.UPMP+USCTA[    21,,341200

341200[    651500,,340000          ;Channel 20
341201[    651400,,340063          ;Channel 21
341202[    651400,,340146          ;Channel 22
```

These DDBs are in funny space, so they are disk DDBs. They contain the following file names: SYS:LPFORM.INI[1,4], DSKC:ERROR.FS[6,6], and DSKC:GRIPE.SRJ[1,2]. The DDBs are displayed as follows:

```
340000/   SYS
340000 DEVNAM/  LPFORM
340000 DEVEXT/  INI  (
340000 DEVPPN[  1,,4

340063/   DSKC
340063 DEVNAM/  ERROR
340063 DEVEXT/  FS    A
340063 DEVPPN[  6,,6

340146/   DSKC
340146 DEVNAM/  GRIPE
340146 DEVEXT   SRJ
340146 DEVPPN[  1,,2
```

Because the banner page that was being printed has the file name GRIPE, it is clear that the third disk DDB is associated with the file that was being printed at the time of the crash.

4.12 LINE DATA BLOCKS (LDBS)

The monitor uses terminals in two different ways: they are the means to enter commands directly to the monitor, and they are also subject to control by user programs. To serve both functions, there are two data structures: the terminal DDB and the Line Data Block (LDB).

LDBs contain information about a terminal line. There is one LDB for each terminal and it is built when the monitor is initialized. LDBs are not created dynamically; they continue to exist as long as the system is in operation. This allows users to type commands on terminals even though they are not logged in, and permanent LDBs speed response because the monitor does not have to spend the time allocating an LDB. The code to allocate and initialize the LDBs is in SCNSER, and it is discarded when system initialization is complete.

In general, an LDB contains:

- Pointers to input and output chunks (terminal I/O buffers)
- Counts of how many characters are currently in the chunks
- Pointer to its associated DDB
- Line status bits
- Line characteristic bits
- Position counter
- MIC information
- Break characters
- Count of characters to echo

You can use LINTAB to locate the LDB entry for a terminal line. LINTAB contains one entry for each terminal in the system (including CTYs and PTYs). Use the TTY number as the offset into LINTAB. The LINTAB entry (a fullword global address) points to the LDB, and the first word of the LDB points to the terminal DDB (if the terminal DDB exists).

4.13 THE SCNSER DATA BASE

SCNSER processes user input and calls the appropriate module to handle the I/O. The SCNSER data base is composed of the following virtual memory sections:

Data	Memory Section	Used for
LINTAB	Section 0	Translates line no. to LDB addr
DSCTAB	Section 0	Translates modem no. to line no.
DDB pool	Section 0	TTY device data blocks
LDBs	Section 4	Line data blocks
Chunk pool	Section 4	Buffers

4.14 TERMINAL CHUNKS

Terminal data is usually stored in eight-word buffers called TTY chunks. In 12-bit ASCII mode, the terminal chunk size varies. Examine the value of TTCHKS to see the current size of a terminal chunk. The terminal chunk starts with a pointer to the previous chunk, and a pointer to the next chunk, followed by the character data.

Chunks are maintained as doubly linked lists, using halfword links relative to Section 4. Each terminal line can potentially have four linked lists of chunks: one for input, one for output, a list for filler characters, and a list for out-of-band characters. When chunks are no longer needed by a terminal line, they are returned to a free list of chunks. The LDB contains pointers to the chunks.

Each character in a chunk is stored as a 12-bit byte, permitting a maximum of 21 characters to be stored in a chunk (3 to a word). In reading the characters in terminal chunks using FILDDT, use the $12T command to break up the 36-bit word into 12-bit bytes (4 bits for flags + 8 bits for data).

The monitor keeps all the chunks in a pool. The TTYINI routine, in SCNSER, initializes the chunks, allocating space for them and creating the links.

The location TTFTAK points to the first free chunk in the pool. When a terminal needs a chunk, it gets the chunk pointed to by this location. TTFPUT points to the last free chunk in the list and returned chunks are stored after this chunk. TTFREN contains the number of free chunks in the system. The following macros place characters in the chunks and remove characters from the chunks: LDCHK, LDCHKR, and STCHK. The following macros are useful in terminal handling. However, these macros should not be called when SCNSER interrupts are enabled.

- LDCHK takes a character out of a chunk, and does not give back used chunks (useful when echoing input).
- LDCHKR takes a character out of a chunk and returns used chunks to the pool, if necessary.
- STCHK puts a character in a chunk, allocating chunks from the pool, if necessary.

4.15 TERMINAL DEVICE DATA BLOCKS

Terminal device data blocks are allocated from the TTY DDB pool as jobs are created, or as the terminal is assigned by a job on another terminal. Some types of information that are stored in the terminal DDB are:

- Pointers to user buffers
- Device and logical names for the terminal
- I/O status information (DEVSTA)
- Device mode information (DEVMOD)
- CPU number of the CPU that owns this terminal
- Pointer to the LDB

Every job has a terminal DDB for its controlling terminal, whether the job is attached or not. Terminal DDBs are created when a job number is assigned (that is, when a program is run) and when a terminal is assigned or OPENed by another job. If the job is not logged in when the program finishes, the DDB is deleted. If the job is logged in, the DDB remains until the job logs out or detaches.

TTYTAB is a table in COMMON that has one entry per job and points to the DDB of the controlling (attached) terminal of the job. If a program opens a software channel for a terminal, an entry is made in the channel table for the terminal.

LDBs and DDBs are linked when a job is created or a terminal is attached to a job. These links are destroyed when:

- o You log out or detach your job.
- o A node goes down when the terminal is connected.
- o You hang up the modem of a terminal that is connected.
- o You release a terminal on a software channel.

TTYATI attaches the terminal to the job when the job is created; TTYATT attaches the terminal for the ATTACH command.

4.16 FINDING TERMINAL I/O INFORMATION

The following example shows how to extract information from the terminal chunks for a job. In this case, you are examining Job 17, which is running PIP. First, look at TTYTAB, which points to the terminal DDB for the job:

```
TTYTAB+21[   102206

102206$6T/   TTY124
```

As the first word of the block verifies, it is a terminal DDB. Next, find the LDB by looking at the DDBLDB word:

```
102206+DDBLDB[   4,,450430

4,,450430[   102206
```

The DDB pointer in the first word of the LDB is correct. Next, examine the LDB:

```
4,,450431[   0
4,,450432[   100000,,0
4,,450433[   10000,,0
4,,450434[   0
4,,450435[   0
4,,450436[   0
4,,450437[   1400,,426522
4,,450440[   1400,,426522
4,,450441[   0
4,,450442[   0
4,,450443[   301400,,422450     ;Ptr to put output characters
4,,450444[   301400,,430276     ;Ptr to take output characters
4,,450445[   2137               ;No. of characters in output
```

The pointers are PDP-10 byte pointers. The memory address in the right half points to the terminal chunk, which can be displayed by:

```
4,,430276$12T/   <space><space><
```

The pointer is in the middle of the chunk. Determine the chunk size, in order to know where the chunks begin and end:

```
TTCHKS=10
```

Now, start from a few locations back, and you can see:

```
4,,430275/    10    ^
4,,430274/          ^
4,,430273/   MEM    ^
4,,430272/   DT     ^
4,,430271/   ^@!^Q   =417221
```

The contents of location 4,,430271 are a backward pointer in the left half, and the location of the next chunk in the right half. The chunk itself holds the text "DT MEM 10 <***>."

By examining the next chunks, you can deduce the entire message:

```
DT      MEM    10  <***>   666405   9-Jul-80
BOOT11  DOC    10  <***>   157023  27-Jul-79       4A(46)
BOOT11  EXE    28  <***>   411354  26-Jul-82       4A(46)
BOOT11  HLP     2  <***>   500576   5-Jan-75
BOOT11  MAC   108  <***>   010501  27-Jul-79
BOOT11  MEM    29  <***>   544353  27-Jul-79
BOOTS   DOC    35  <***>   352703  17-Jul-79
BOOTS   EXB    10  <***>   556224  26-Jul-82
BOOTS   MAC    92  <***>   764007  31-Jul-79
BT128K  EXB    10  <***>   605464  26-Jul-82
BT256K  EXB    10  <***>   556224  26-Jul-82
WIBOOT  EXE    32  <***>   607553  30-Nov-79         7(12)
WLBOOT  EXE    32  <***>   325717  30-Nov-79         7(12)
WSBOOT  EXE    24  <***>   631454  30-Nov-79         7(12)
WTBOOT  DOC    18  <***>   451662  28-Jun-79
WTBOOT  MAC    29  <***>   007472  20-Jul-79
DML6A   DOC     3  <***>   331675   7-Mar-79
DMPFIL  EXE    16  <***>   071372  16-Jul-80         6A(7)
DMPFIL  MAC    34  <***>   661675   7-Mar-79
DMPFIL  MEM     5  <***>   077054   8-Mar-79
COPY    EXE     8  <***>   605250  17-Jul-80        7(101)
CPY007  DOC     4  <***>   507510   8-Mar-79
DTC007  DOC     3  <***>   204110   8-Mar-79
DTCOPY  EXE    20  <***>   456574  17-Jul-80        7(101)
DTCOPY  MAC    43  <***>   303311
```

The user was reading a BACKUP tape directory listing when the system crashed.

4.17 TAPE DRIVES

The data structures for tape drives parallel the actual hardware components. Depending upon the hardware interface, a magtape controller may be connected to as many as 15 drives. The software has up to 15 tape unit data blocks (TUBs) connected to a tape controller data block (KDB), which then points to a channel data block (CHN).

There is one TUB for each tape unit in the system. It contains the unit name, pointers to the DDB and controller, error counts, tape label information, and a pointer to the IORB (I/O request block, the request to the controller outlining the I/O transfer). The first word in each TUB is the SIXBIT name of the tape unit, in the form:

MTxy

where x = the controller name and y = the unit number. For example:

MTA0

The prototype TUBs are:

Symbol	Units
DX1UDB	DX10/TX01/TX02
T78UDB	TM78
TCXUDB	TC10C
TM2UDB	TM02/TM03
TMXUDB	TM10B
TS1UDB	SA10/TX01/TX02
TX2UDB	DX20/TX02

The KDB identifies a controller and there is one for each tape controller in the system. It holds the name of the controller, a pointer to the next KDB, the channel command list, a list of TUBs owned by the controller, and controller-dependent information. In the monitor, KDBs are pointed to by KDBTAB+.TYxxx. The name of the controller is stored in the first word as MTn, where n is the controller number. The KDB also points to the channel it is connected to.

The prototype KDBs are:

```
DX1KDB
T78KDB
TCXKDB
TM2KDB
TMXKDB
TS1KDB
TX2KDB
```

Channel data blocks exist for channels that are connected to any type of controller. They hold enough information to start and monitor the channel transfer, including:

- Error counts
- Retry information
- Channel status
- Channel queue

At system startup, AUTCON creates one magtape DDB for each unit on each controller. The start of a magtape DDB can be obtained from DDBTAB+.TYMTA. The magtape DDB is named:

```
MTxu
```

where:

- x is the alphabetic controller name (A for controller 0, B for controller 1, and so forth)
- u is the unit number

A special magtape DDB (called a Label DDB) is required for the tape label processor (PULSAR). This is needed so I/O can be performed by two different jobs (the user job and the job running PULSAR), while the device remains assigned to the user job. The label information is stored in the Tape Unit Data Block (TUB), which is common to both the magtape and the label DDB.

The name of a label DDB is in the form:

'Lxu

The values of x and u are the same as shown above for the magtape DDB. The label DDB has the same format as a magtape DDB.

4.18 DISKS

Disks are the most complex peripheral I/O devices in a timesharing system. They are shared among jobs, using a logically structured file system to store data and prevent destructive interference. The basic unit of disk storage is one block (equal to 128 words).

TOPS-10 organizes information into logical groups known as files. The contents of a file are referenced by the file specification, which uniquely identifies the file. A file specification has four components:

- A file structure name, which identifies the disk drive or group of disk drives where the file is stored
- An ordered list of directory names (MFD, UFD, and SFDs, if any)
- A file name of one to six alphanumeric characters
- A file extension of zero to three alphanumeric characters

A file structure is a logical device name that refers to one or more physical disk units. Using the file structure name, the user job need never know the exact physical unit where data is stored.

The directory where a file is stored helps to uniquely identify the file. TOPS-10 organizes files by using file structures, User File Directories (UFDs), and Sub-File Directories (SFDs). A UFD or SFD is itself a file, and contains a list of all files for a user, and a pointer for accessing those files.

The Master File Directory (MFD) points to all the UFDs on a particular disk file structure. There is one MFD for each file structure, containing the names and addresses of all the UFDs on that structure.

Each UFD can optionally contain Sub-File Directories (SFDs). An SFD is a logical group of files within the UFD. SFDs can contain their own sub-file directories, which can be nested to a level of five SFDs in a single UFD.

The UFD is named with the user's PPN, in brackets. For example, the user with PPN 10,507 has the following UFD:

[10,507]

You specify an SFD by typing the name of the UFD, followed by the name of the SFD (up to six alphanumeric characters). For example, the UFD [10,507] could contain a file called FIRST.SFD. To access the files in this SFD, the user specifies the following directory:

[10,507,FIRST]

In the SFD, the user keeps a file called SECOND.SFD, which points to a nested SFD. To access files in the nested SFD, the user types the following directory name:

```
[10,507,FIRST,SECOND]
```

The monitor does not write the data on disk in physically consecutive disk blocks. The monitor must allocate disk space effectively in a dynamic situation where users are constantly creating, deleting, modifying, and appending to variable-length files. Therefore, the monitor segments disk space into blocks and stores files in space that is available throughout the file structure.

To maintain this complex storage system, the monitor must maintain some amount of overhead data for retrieving files and allocating disk space. The RIB (Retrieval Information Block) contains the retrieval information for the file.

A RIB is a block on the disk that contains retrieval pointers to the blocks making up the entire file. The UFD points to the first RIB for each file. Each retrieval pointer in the RIB describes a contiguous block of data called a "group." The retrieval pointer contains the first physical disk address of the group and the number of blocks that are in the group. UFDs and MFDs also have RIBs to describe their locations on the disk unit.

A retrieval pointer contains the following information:

- The number of clusters in this group
- The cluster number where the group starts
- The checksum for the group

One of the following conditions is possible, if the left half of the retrieval pointer is zero:

- If Bit 18 = 1, Bits 19 through 35 contain the logical unit number of the next unit to get data from. This allows one RIB on one unit to hold pointers to data on another unit in the same structure.
- If the right half is zero, there is no more data in the file.

If a file needs more than one RIB to retrieve the data, it has extended RIBs at the start of subsequent groups. The monitor also writes an extra copy of each RIB as the last block pointed to by the RIB, for disk error recovery purposes. That copy is known as the spare RIB. The first RIB is known as the prime RIB.

Each disk unit contains a HOME block, which describes the file structure that contains the disk unit, and points to the MFD. Blocks 1 and 10 (decimal) on the disk contain the HOME block, which records the following information:

- The file structure to which this unit belongs, and the unit's position within the structure
- The characteristics of the unit and file structure
- A pointer to the MFD

The monitor uses the HOME block to find the MFD when the file structure is mounted for a user.

The monitor keeps information about used disk blocks in the Storage Allocation Tables (SAT blocks). The SAT block on each file structure is stored as SYS:SAT.SYS. Each bit in the SAT block represents a group of contiguous disk blocks called a cluster.

The smallest unit of data on disk that the monitor can allocate is the cluster, which is composed of a specific number of disk blocks. A small disk unit might use a cluster size of 3 blocks (600 words). If the monitor must allocate space to a file that is smaller than 200 (octal) data words, an entire cluster is allocated. When the cluster size increases, fewer SAT blocks are required for storage allocation information; with fewer reads/writes to the SAT, a smaller number of operations is required to assign and release disk space.

Large clusters save memory at the expense of disk space. Because disk space is allocated in clusters, short files result in wasted space if the cluster size is too large.

The MFD contains pointers to the UFDs on the disk unit. The UFD contains a two-word entry for each file in the UFD. The UFD entry specifies the file name in the first word, and file extension in the left half of the second word and a pointer to the file in the right half of the second word called the compressed file pointer (CFP). The CFP is the 18-bit address of the RIB of the file, pointing to the first supercluster of the file. A supercluster is a set of clusters stored contiguously on disk. A file always starts at the supercluster boundary, but one file may fill many superclusters of disk space.

The number of blocks per cluster is usually equivalent to the number of blocks per supercluster. However, if the total number of clusters on a file structure is greater than 262,143, the clusters are regrouped into superclusters such that the number of superclusters is less than or equivalent to 262,143 (the largest number that can be stored in the right half of the second word in the UFD entry). The number of clusters per supercluster is stored in the HOMe block, and in the STR block when the monitor is running.

4.18.1 Finding Information on Disk

The following example shows how to use FILDDT to retrieve information stored on a disk, using the /U switch to look at a disk unit. This example shows how to locate the contents of the file DSKA:H616.TXT[64,2]; DSKA is mounted on RPB1.

First, run the monitor-specific FILDDT (MONDDT in this manual), and specify the physical disk unit you want to examine, followed by the /U switch:

```
.R MONDDT

File:RPB1:/U
```

/U requires that you be logged in as [1,2], and instructs FILDDT to treat the disk as addressable.

The first data structure to use in examining the file is the HOME block. It holds pointers to other files, and can always be found at Blocks 1 and 10 (decimal) on a disk. To access the first word of the HOME block, specify location 200 to FILDDT. Each block is 128 (decimal) words, which equals 200 (octal).

Remember to convert disk block numbers to FILDDT addresses by multiplying by 200. If converting cluster addresses, multiply by 200*n, where n is the cluster size. For example, if the cluster size is 5, use the following calculation to specify the block number. (The numeric base of the following calculations are indicated by (8) for octal and (10) for decimal).

```
Block 15(10) = Block 17(8) * 200 = 3600(8) in FILDDT

Cluster 11(10) = Cluster 13(8) * 5 = Block 67(8) = 67 * 200 = 15600
```

To examine the HOME block, type the following:

```
200/  HOM                ;Name of HOME block
201/  DSKA01             ;Unit ID
202/  0
203/  0
204/  DSKA               ;Structure name
```

The pointer to the MFD's RIB is at offset HOMMFD:

```
200+HOMMFD/ 4204
```

This location contains the block number. All subsequent addresses are cluster numbers. The size of a cluster is stored in the HOME block at location HOMBSC:

```
200+HOMBSC/ 12           ;Blocks per supercluster
200+HOMBPC/ 12           ;Blocks per cluster
```

In this case, a cluster is 10 (decimal) blocks.

The MFD's RIB confirms that you have the correct RIB:

```
4204*200/  777653,,41
1,,41001/  1,,1          ;Owner of file
1,,41002/  1,,1          ;File name
1,,41003/  UFD)EC        ;File extension in left half
```

Examine the first retrieval pointer to find the MFD itself. The right half of the contents of the first word in the RIB contains the offset within the RIB to the first retrieval pointer. The left half of the first word is the negative of the maximum number of retrieval pointers that may be stored in the RIB.

```
1,,41001+41/  400000          ;Unit change pointer to Unit 0
1,,41002+41/  4010,,100332    ;1st real retrieval pointer
```

The first cluster of the MFD is number 332. This corresponds with Block 332*12=4204 (octal), the address of the RIB (stored in HOMMFD, shown above). The RIB is stored in the first block of the supercluster when the file is initially allocated. The monitor checks to see if the RIB address is the same as the first group of data. If so, the monitor retrieves the second block for data. Look at 1,,41200 (4204*200) for the MFD:

```
1,,41200/  1,,1                ;[1,1] UFD
1,,41201/  UFD. : = 654644,,332
1,,41202/  1,,4                ;[1,4] UFD
1,,41203/  UFD    = 654644,,3
1,,41204/  3,,3                ;[3,3] UFD
1,,41205/  UFD  > = 654644,,336
1,,41206/  10,,1               ;[10,1] UFD
1,,41207/  UFD  ? = 654655,,337
```

```
1,,41210/  1,,2                ;[1,2] UFD
1,,41211/  UFD  @ = 654644,,340
1,,41212/  1,,5                ;[1,5] UFD
1,,41213/  UFD  A = 654644,,341
1,,41214/  1,,3                ;[1,3] UFD
1,,41215/  UFD  B = 654644,,342
1,,41216/  64,,2               ;[64,2] UFD
1,,41217/  UFD  E = 654644,,345
```

The first word of each two-word MFD entry contains the UFD name. The second word contains the UFD extension in the left half and the supercluster address of the RIB in the right half. The pointer to the UFD RIB is located at supercluster 345 (assuming the supercluster size is equivalent to 1).

```
345*12*200/  777653,,41
1,,RNA2CB+71[  1,,1            ;Owner of file
1,,RNA2CB+72[  64,,2           ;File name
1,,RNA2CB+73/  UFD)EC          ;LH = file extension

345*12*200 41/  400000
1,,RNA2CB+133/  1000,,345      ;Location of UFD
```

Again, the RIB takes up the first block of the cluster. Add 200 (octal) to the address of the RIB to get the first data block of the UFD. If the cluster size is 1 block, you have to read the retrieval pointer for the first data block.

```
345*12*200+200/  F601
1,,RNA3CB+71/  EXE &S
1,,RNA3CB+72/  D602
1,,RNA3CB+73/  EXE GN
        .
        .
        .
1,,D3KDB+1/  H616
1,,DSKDB+2/  TXT!T4  =647064,,16424
```

The location of the RIB for the file is at Supercluster 16424:

```
16424*12*200/  777653,,41
44,,262001/  64,,2
44,262002/  H616
44,262003/  TXT)CT             ;LH = file extension

16424*12*200+41/  400000
44,,262042/  1655,,616424
```

Finally, you reach the file, which contains:

```
44,,262200/
 DATA
44,,262201/  A AT
44,,262202/  TIME
44,,262203/  OF SE
44,,262204/  R062.
44,,262205/  CRASH
44,,262206/
```

```
44,,262207/  VMA,
44,,262210/  PC=53
44,,262211/  7771
44,,262212/
         (FRO
44,,262213/  M KLD
44,,262214/  CP AL
44,,262215/  L COM
44,,262216/  MAND)
         .
         .
         .
```

Reformatting to make reading easier yields the following:

```
DATA AT TIME OF SER062.CRASH

VMA, PC=537771
(FROM KLDCP ALL COMMAND)...
```

4.18.2 In-Core File Information

To keep accurate information in a readily accessible place, the monitor maintains information about the following, in memory:

- Structure information
- Device information
- File information
- User information

To access a file structure, the monitor keeps a file structure data block called STR. It contains the name of the structure, allocation information, swapping information, and pointers to MFD and HOME blocks. The STRs are stored in a linked list, each entry pointed to by the system table TABSTR. A structure is identified by the offset into TABSTR where its entry is stored. The word SYSSTR points to the first structure. The STR also points to the physical units in the file structure.

The Unit Data Block (UDB) contains information about the physical disk unit, including:

- Physical unit name
- Pointers to related UDBs
- Pointers to HOME blocks and SAT blocks
- Unit parameters (cluster size, and so forth)

The UDBs for each structure are linked and each UDB points back to the STR. Because of these linkages, the STR points only to the first UDB. The UDB addresses are dynamically assigned by AUTCON.

The STR accesses the following data structures:

- SABs (Storage Allocation Blocks) are in-core copies of the SAT tables. Copies of the SATs are read into memory at system startup and updated on disk after every write operation.

- SPTs (Storage allocation Pointer Tables) contain pointers to all SAT blocks for a unit. Do not confuse the SPTs (Storage allocation Pointers Tables) used in disk I/O, with the SPT (Special Pages Table) used in mapping user jobs into physical memory.

- The PWQ (Position Wait Queue) is an ordered list of DDBs that have positioning requests for that unit.

The controller data block (KON) is connected to the UDB and contains information about the device controller for that unit. The channel data block (CHN) is linked to the KON and contains information about the hardwar channel associated with that disk controller. The CHN holds the transfer wait queue (TWQ) for the disk drives on that channel.

The PWQ and the TWQ contain information for performing I/O requests, and the order in which they are to be serviced. Both of these queues are required to drive a disk device. The format and naming scheme is the same as the channel data block for tape drives.

Only the static state of the file system can be described here. In a timesharing environment, jobs can modify files while the same files are being used by other jobs. The monitor requires special information for the contention-free management of the files. To keep track of currently open files, the monitor's data base shows the versions of all open files for all PPNs at any given time.

The file data base is organized using the following data structures:

- The PPB, the PPN data Block, contains information about all files for a specific PPN. There is one PPB for each PPN that has open files. All PPBs for all jobs are linked together; the first is pointed to by SYSPPB.

- The NMB, the Name Block, contains the file names of all open files on all file structures for a PPN. There is one NMB for each open file of each PPN, regardless of the number of versions of the file that are in existence. A word in the PPB points to the the first NMB in a list.

- The ACC, the access table, contains information needed to gain access to a specific version of a specific file. The location of the first RIB is stored here, with the file structure number. The ACC entries are linked in a ring through the NMB.

  At any time there are two possible versions of a file: the current version and the superseding version. Usually there is only one ACC; but while the file is being superseded, both the old and new versions of the file have ACCs linked to the NMB. There may be several ACCs if the file exists on more than one file structure, or older versions of a file are still open.

- The UFB is a UFD data block. The monitor keeps a UFB for each UFD for each file structure for your job. Each UFB contains the first retrieval pointer to the UFD. The PPB contains a pointer to the UFB for the first structure.

Every LOOKUP to a file is recorded in the PPB, the NMB, and the UFB. If the monitor cannot find a file, it marks the NMB to indicate that the file does not exist. Likewise, if the UFD does not exist, the monitor marks the UFB accordingly. There are two words in each of these data structures to contain this information. The first word is the KNO word, short for KNOW. This is set to tell whether the monitor checked to see if the file or UFD exists. If the bit is zero, a disk read will be required to find out if the file exists. If the bit is one, the second word, the YES word, is valid. If the YES word contains 0, the file does not exist; if the word is one, the file does exist and there is probably information about it in the PPB and NMB.

The goal of this information storage is to reduce the number of disk reads for discovering whether a file exists and where it is stored. This is especially useful during debugging, when the same group of files are used over and over again (source program, compiler, and linker, for example). Of course, not all the file information can fit into memory. The disk data structures are managed like a cache, where the oldest entries are discarded in favor of those accessed more recently.

The disk DDB is extremely important because it is the central source of information for all disk I/O operations. It contains pointers and links to many other data structures, including:

- The current retrieval pointers being used by the disk routines, and the block numbers to which the pointers refer.
- Pointers to the UDB and STR where the file resides.
- Pointers to the buffer ring header and user buffers.
- The PWQ and the TWQ, which make a linked list of DDBs waiting to use the disk and channel.
- Pointers to the ACC and UFD.

Disk DDBs are created when the device is OPENed and a software channel is created; they are deleted when the channel is closed. Disk DDBs are stored in the user's funny space.

4.18.3 The Software Disk Cache

The in-core file information that is being input or output can be cached in memory, allowing the monitor to access disk information more efficiently. The following data blocks are used in caching disk I/O information.

The data structures for the software disk cache are two doubly linked lists, a list header, and a hash table. Each entry in the list contains forward and backward pointers for each of the two lists, (.CBNHB, .CBPHB, .CBNAB, and .CBPAB), a UDB address (.CBUDB), a block number (.CBBLK), and a pointer to the address in free core where the block is (.CBDAT). For statistical purposes, the entry also contains a count of the number of times the block has been accessed since it was included in the list (.CBHIT).

The list header points to the two linked lists. The first linked list is the "access" list. The most recently accessed block is at the top of the list; the least recently accessed block is at the end. The access list is linked through the .CBNAB/.CBPAB words.

The second linked list is the "free" list. It contains a list of all blocks that are not currently in use and do not appear in the hash table. The free list is linked through the .CBNHB/.CBPHB words.

The hash table consists of pointers to the free list corresponding to the blocks that hash to the same position. Thus, the hash table consists of separate list heads for the lists of blocks that hash to that position in the hash table.

At initialization time (CSHINI), all the blocks are allocated and linked into the free list. They are also linked into the access list. The hash table entries are linked to themselves because the table is empty.

To find an entry, given its UDB and block number, use the block number as the offset into the hash table. Use the hash table entry as a list head, following the list until you either find a match, or return to the header. This is done with the CSHFND routine. In general, these lists are very small, most commonly only one or two blocks.

The main cache handling routine is CSHIO, which will simulate I/O from the cache, doing the necessary physical I/O to fill and write the cache. Note that this is a write-through cache, so no sweeps are required, and the data in the cache always reflects the blocks on disk.

4.18.4 Finding In-Core File Information

The following example finds the file information stored in memory for Job 3. First, you must set up paging for the job:

```
.C0EPT/     .E0EPT
$Q'1000$U
JBTNAM+3$6T/   ACTDAE          ;Program name
JBTUPM 3[   42000,,354         ;UPT at page 354
.$6U                           ;Mapping command
```

Then search for the assigned DDBs:

```
USRJDA[  0                     ;Channel 0
FOPBUF#+52[  0                 ;Channel 1
FOPBUF#+53[  0                 ;Channel 2
FOPBUF#+54[  0                    .
FOPBUF#+55[  0                    .
FOPBUF#+56[  0                    .
FOPBUF#+57[  0                    .
FOPBUF#+60[  0                    .
FOPBUF#+61[  0                    .
FOPBUF#+62[  0                    .
FOPBUF#+63[  0                    .
FOPBUF#+64[  0                    .
FOPBUF#+65[  0                    .
FOPBUF#+66[  0                 ;Channel 15
FOPBUF#+67[  0                 ;Channel 16
FOPBUF#+70[  0                 ;Channel 17

.USCTA[  20,,741200            ;Check for extended channels
```

```
741200[  564200,,740000        ;Channel 20
741201[  560200,,740066        ;Channel 21
741202[  474000,,740154        ;Channel 22
741203[  403000,,740242        ;Channel 23
741204[  441100,,740330        ;Channel 24
741205[  474100,,740416        ;Channel 25
741206[  0                     ;Channel 26
741207[  0                     ;Channel 27
```

In this case, there are six open DDBs, all in the extended channel table. They point to DDBs in funny space, so they must be for disk files. Looking closer, you can find the names of the files. The examples below show how this was done for the first three DDBs listed above.

```
740000$6T/  ACT
DDB20:                         ;Label this as the DDB
DDB20+DEVFIL$6T/   USAGE       ;for Channel 20.
DDB20+DEVEXT$6T/   OUT   !
DDB20+DEVPPN[   1,,7           ;ACT:USAGE.OUT[1,7]

740066$6T/  ACT
DDB21:                         ;Label this as the DDB
DDB21+DEVFIL$6T/  FAILUR       ;for Channel 21.
DDB21+DEVEXT$6T/  LOG   =
DDB21+DEVPPN[  1,,7            ;ACT:FAILUR.LOG[1,7]

740154$6T/  ACT
DDB22+DEVFIL$6T/  USEJOB
DDB22+DEVEXT$6T/  BIN   W
DDB22+DEVPPN$6T[  1,,7         ;ACT:USEJOB.BIN[1,7]
```

Now examine the USEJOB.BIN file. From the DDB, you can find which unit the file is on:

```
DDB22+DEVUNI/  142314,,142314 ;original UDB,,current UDB

142314$6T/RAJ3                 ;Physical device name
RAJ3:                          ;Label the UDB
RAJ3+UDBKDB[   136770          ;KDB
RAJ3+UNILOG$6T/   DSKA0        ;Logical name within structure
RAJ3+UNIHID$6T/   DSKA0        ;HOME block ID name
RAJ3+UNISYS[   142444,,46000   ;Next UDB in system,,bits
RAJ3+UNISTR[   145324          ;Next UDB for STR
RAJ3+UNICHN[   142444          ;Next UDB on channel
RAJ3+UNIKON[   142444          ;Next UDB on controller
     .
     .
     .
```

The unit is RAJ3, which is part of the structure DSKA.

Included in the UDB is a pointer to the structure data block (STR).

```
145324$6T/   DSKA              ;STR name
DSKA:                          ;Label the CHN
DSKA+1[   145274,,10           ;Next STR,,STR number
DSKA+2[   142314,,0            ;First UDB for STR,,K for CRASH.EXE
DSKA+3[   1                    ;Number of units in STR
DSKA+4[   3,,41577             ;Quota words
DSKA+5[   3,,41600             ;      .
DSKA+6[   0                    ;      .
DSKA+7[   0                    ;      .
DSKA+10[   0
DSKA+11[   266532
DSKA+12[   777777,,777014
DSKA+13[   7                   ;Mount count
DSKA+14[   410,,512304         ;First retrieval pointer to MFD
      .
      .
      .
```

There are two other methods for locating a disk structure. The first is to start with SYSSTR and follow the links to each structure:

```
SYSSTR/   247103,,1            ;Pointer in left half
247103$6T/   SIRS              ;1st STR in linked list
247104[   240137,,15
240137$6T/   BADP              ;2nd STR in list
240140[   110521,,14
110521$6T/   7A                ;3rd STR in list
110522[   145324,,1
145324$6T/   DSKA              ;4th STR in list
```

Or, with the file structure number, you can index into TABSTR:

```
TABSTR/   777733,,1
TABSTR+1/   110521
TABSTR+2/   145324
145324$6T/   DSKA
```

Notice that the links started by SYSSTR are not in the same order as TABSTR.

You can use the UDB to find several other structures:

```
RAJ3: UNIQUE/  0               ;Position wait queue
RAJ3: UNIPTR/  0               ;-Length,,addr of swap SAT
RAJ3: UNISAB/  7,,31271        ;First SAB in ring,,addr of SPT
```

From the UDB, you can find the KDB:

```
RAJ3: UDBKDB/    136770        ;Ptr in UDB to KDB

136770$6T/   RAJ               ;Controller name
RAJ:                           ;Label this
RAJ+1[   76237                 ;Next controller on system
RAJ+2[   7                     ;CPU accessibility mask
RAJ+3[   136704                ;KDBCHN -- CHN
RAJ+4[   777740,,137063        ;KDBIUN -- Initial pointer to units
      .
      .
      .
```

You can get the channel data block from the KDB:

```
RAJ KDBCHN/  136704          ;KDB pointer to CHN

136704/  0                   ;-1 if channel idle
CHN:                         ;Label it
CHN+1/  142750,,0            ;Next CHN,,last UDB with error
CHN+2/  0                    ;Error information
CHN+3/  0
CHN+4/  0
```

The other file information can be found by starting with SYSPPB and following pointers to the correct PPB, NMB, and ACC. (DEVACC in the DDB also points to the ACC.)

```
SYSPPB/  120140,,0           ;Pointer to first PPB

120140[  1,,4                ;Project,,programmer number
120141[  120440,,0           ;Next PPB in system,,0
120440[  1,,7                ;Project,,programmer number
PPB:                         ;Label it
PPB+1[   120560,,0           ;Next PPB in system,,0
PPB+2[   120450,,0           ;First UFB this PPN,,0
PPB+3[   120460,,0           ;First NMB this PPN,,bits
PPB+4[   6                   ;Use count
PPB+5[   410                 ;KNO bits
PPB+6[   410                 ;YES bits
PPB+7[   0                   ;Interlock bits
```

Now you can look for the file USEJOB.BIN in the NMB:

```
120460$6T/   USAGE           ;File name - USAGE
120461[   120510,,0          ;Next NMB,,0
120510$6T/   FAILUR          ;File name - FAILUR
120511[   120540,,0          ;Next NMB,,0
120540$6T/   USEJOB          ;File name - USEJOB
NMB:                         ;Label it
NMB+1[   122670,,0           ;Next NMB,,0
NMB+2[   26325               ;Compressed file pointer
NMB+3[   120550,,425156      ;ACC,,file extension in SIXBIT
NMB+4[   110000,,0           ;File structure number
NMB+5[   400                 ;KNO bits
NMB+6[   400                 ;YES bits
NMB+7[   2                   ;Use count
```

And finally, you can get to the ACC from the NMB:

```
120550[   156                ;Highest block allocated
ACC:                         ;Label the ACC
ACC+1[   120542,,200000      ;NMB,,bits
ACC+2[   1100,,26325         ;First retrieval pointer
ACC+3[   0                   ;Dormant ACCs
ACC+4[   110020,,120440      ;Bits,,PPB
ACC+5[   222136,,410
ACC+6[   145
ACC+7[   55744,,332136
```

The ACC points back to both the NMB and PPB. Note, however, that the ACC may point to another ACC, which may point to the NMB. This is ascertained by examining the last digit of the left half of the NMB. If the last digit is 2, as in this example, the left half of the NMB ACC word points to an NMB. If the digit is not 2, the NMB points to another ACC.

The PPB also points to the UFB.

```
DDB22 DEVUFB/  120450          ;DDB pointer to UFB

PPB PPBUFB/  120450,,0         ;PPB pointer to UFB

120450/  377777,,700521        ;Total blocks left this UFD
UFB:                           ;Label it
UFB+1[   122420,,775400        ;Next UFB,,bits
UFB+2[   100,,52166            ;First retrieval PTR to this UFD
UFB+3[   5                     ;Bits
UFB+4[   110000,,0             ;File structure number
UFB+5[   104,,0                ;N if job N owns AU for this UFB
UFB+6[   0                     ;Non-zero if waiting for AU
UFB+7[   0                     ;=1 if UFD has empty data blocks
```

In all cases, check the Monitor Tables Descriptions and the source listings to find the interconnections between the data structures and how to interpret what is stored in them.

CHAPTER 5

ERROR HANDLING ROUTINES

The monitor reports hardware and software problems by displaying error messages on the CTY, but these messages include only a small portion of the information that the monitor stores in its database.

This chapter will show you how to take a message from the CTY and use it to trace through the dump to obtain more information. This involves working with the APR interrupt routine, the page fail trap routine, and the stopcode routine. You can use this information to deduce the scope and nature of the problem more accurately.

The error routines of the monitor are designed to handle both software and hardware errors. When software errors are detected, control usually jumps to an error handling routine for processing. Hardware errors, however, can interrupt processing and sometimes halt the system.

5.1 HARDWARE ERRORS

You can use the CTY message to trace an error to the actual hardware that failed. The following types of hardware-related messages may appear on the CTY.

The most serious hardware error is indicated by one of the following messages:

```
?NON-RECOVERABLE MEMORY PARITY ERROR IN MONITOR

[CPU HALT]

   or

?NON-EXISTENT MEMORY DETECTED IN MONITOR

[CPU HALT]
```

In this case, the error is so serious that the processor is halted immediately and no further error processing can be done.

A second type of problem is an AR/ARX parity trap, indicated by the following message:

```
************
CPU0 AR/ARX PARITY TRAP AT USER PC 401123 ON dd-mmm-yy
JOB 1 [SYSTAT] WAS RUNNING
PAGE FAIL WORD = 000000,,00011
MAPPED PAGE FAIL ADDRESS = 547000,,560271
INCORRECT CONTENTS = 000000,,000000
CONI PI, = 000000,,000377
RETRIES UNSUCCESSFUL, OFFENDING LOCATION ZEROED
************
```

Another type of parity trap is a page table parity trap, indicated by the following:

```
************
CPU0 PAGE TABLE PARITY TRAP AT EXEC PC 414555 ON dd-mmm-yy hh:mm:ss
PAGE FAIL WORD = 000000,,00011
CONI PI, = 010000,,020377
************
```

A CPU interrupt due to a parity or NXM error is reported as:

```
************
CPU1 PARITY ERROR INTERRUPT AT USER PC 343413 ON dd-mmm-yy hh:mm:ss
JOB 2[WBKI] WAS RUNNING
CONI APR, = 003002,,312022
CONI PI, = 010000,,020377
ERROR INVOKED BY A message
************
```

This report can have several variations, depending on the CPU and the specific error. The monitor can include any of these error messages:

CACHE WRITE-BACK FORCED BY A SWEEP INSTRUCTION.

CHANNEL STATUS WORD WRITE.

CHANNEL DATA WORD WRITE.

CHANNEL READ FROM MEMORY.

CHANNEL READ FROM CACHE.

CPU WRITE TO MEMORY (NOT CACHE).

CACHE WRITE-BACK FORCED BY A CPU WRITE.

CPU READ OR PAGE REFILL FROM MEMORY.

PAGE REFILL FROM CACHE.

After this or other errors, the monitor may also attempt to check for problems by scanning memory for parity errors or nonexistent memory. A memory scan can produce one of the following reports:

```
************
MEMORY PARITY SCAN INITIATED BY CPU0 ON dd-mmm-yy hh:mm:ss
NOTHING WAS FOUND
***********

************
NON-EXISTENT MEMORY SCAN INITIATED BY CHANNEL 1 ON CPU1 ON dd-mmm-yy
hh:mm:ss
NON-EXISTENT MEMORY DETECTED:
AT 314243 (PHYS.)
***********
```

The channel number (CHANNEL 1) listed in this message refers to the sofware channel data block (CHN) number, not an RH20 channel.

Memory parity errors or nonexistent memory errors on a channel produce a special message:

```
************
CPU1 CHANNEL MEMORY PARITY ERROR ON dd-mmm-yy hh:mm:ss
DEVICE IN USE IS RPA2
CHANNEL TYPE IS type
TERMINATION CHANNEL PROGRAM ADDRESS = 000477
TERMINATION DATA TRANSFER ADDRESS = 251470
LAST THREE CHANNEL COMMANDS EXECUTED ARE:
   760000,,252777
   760000,,251777
   760000,,250777
************
```

The CHANNEL TYPE listed in this message may be DF10C, DX10, RH20, CI20, NIA20, or SA10. Hardware errors signal the software in either of two ways: by a processor (APR) interrupt or by a page fail trap. APR interrupts are usually generated on the highest PI level, because CPU errors are serious and must interrupt other devices. When notified of such errors, the monitor reads the hardware registers and takes the appropriate action.

To obtain more information about the error and the state of the monitor, you must examine the dump. It is important to understand how the monitor handles hardware errors. The following sections describe the routines in the monitor that handle errors.

5.1.1 APR Interrupt Routine

The routine to handle APR interrupts is APnINT, where n is the CPU number. It is defined by a macro in COMMON, and handles all the possible conditions that could cause a processor interrupt, which are:

- Cache-sweep-done
- Power fail
- Timer timeout (clock tick)
- I/O page fail error

- NXM error
- Cache directory parity error
- MB parity error
- Address parity error
- SBUS error

A clock tick or cache-sweep-done interrupt happens frequently and the monitor deals with them quickly. The other conditions require more extensive processing.

MB and NXM errors undergo even more analysis and eventually produce one or more of these error reports: CPU parity error or NXM interrupt, a memory scan, or the nonrecoverable error message.

5.1.2 Page Fail Trap Routine

Page fail traps are caused by one of the following conditions:

- Page fault
- Proprietary violation
- AR/ARX parity error (KL10 only)
- Page table parity error (KL10 only)
- Page refill failure (KL10 only)
- Address break (KL10 only)
- Illegal section number (KL10 only)
- Illegal indirection (KL10 only)
- Non-existent device or register (KS10 only)
- Hard memory error (KS10 only)
- NXM error (KS10 only)

Some of these conditions are the result of normal operations, such as an address break, proprietary violation, or page fault. Others are handled as error conditions. The page fail word describes the type of page fault that occurred. The trap handler is located at SEILM in APRSER.

The APR interrupt routine and the page fail trap routine use the same push-down list, ERnPDL, once an error has been detected. The power fail routine uses another push-down list, PWFPDL.

The channel error report is produced at the interrupt level of the device that was doing the transfer. This report usually occurs for disk and tape devices.

If a parity error is detected in fast memory, DRAM, or CRAM, the EBOX stops immediately by turning off its clocks. The front-end processor performs any diagnostic action that is necessary.

5.1.3 Saved Hardware Error Information

The error handling routines store information about hardware errors in the CPU Data Block (CDB). Some of those locations in the CDB are:

```
.CnACN (APRSTS)     CONI APR,
.CnAEF              APR error flag
```

Parity Error Information:

- .CnTPE contains the total number of parity error words in memory.
- .CnSPE contains the total number of nonreproducing parity errors in memory.
- .CnMPA contains the memory parity address for this CPU.
- .CnMPW contains the memory parity word for this CPU.
- .CnMPP contains the memory parity PC for this CPU.
- .CnSB0 contains the SBUS Diag 0 instruction.
- .CnS0A contains the answer from the SBUS Diag 0 instruction.
- .CnSB1 contains the SBUS Diag Function 1 instruction.
- .CnS1A contains the answer from the SBUS Diag Function 1 instruction.

NXM Information:

- .CnTNE contains the total number of NXMs for this CPU.
- .CnSNE contains the total number of nonreproducible NXMs for this CPU.
- .CnMNA contains the first address found with NXM.

AR/ARX Parity Information:

- .CnPBA contains the physical address that registered bad parity on last AR/ARX parity trap.
- .CnTBD contains the contents of the bad word on the last AR/ARX parity trap.
- .CnNPT contains the total number of AR/ARX parity traps.
- .CnAER contains the results of RDERA on a parity/NXM interrupt.
- .CnPEF contains the results of CONI APR on a parity/NXM interrupt.
- .CnPPC contains the PC on the last AR/ARX parity trap.
- .CnPFW contains the page fail word on the last parity trap.
- .CnHPT contains the number of hard AR/ARX parity traps.
- .CnSAR contains the number of soft AR/ARX parity traps.
- .CnPTP contains the total number of page table parity traps.

5.1.4 Hardware Error Checking

The KL10 processor is made up of the following hardware components, the EBOX, the MBOX, and various interfaces and buses. The EBOX, short for Execution BOX, is responsible for the execution of the instructions. The MBOX, short for Memory BOX, controls transfers to and from memory, cache, channels, and the EBOX.

The EBOX is composed of the following:

- Instruction Register (IR) receives the instruction code from the Arithmetic Logic Unit and passes it to the CRAM/DRAM for execution.

- Dispatch RAM (DRAM) and Control RAM (CRAM) hold the microcode that implements the PDP-10 instruction set.

- Arithmetic Logic Unit (ALU) is the major working area of the processor. It has three fullword registers:

  AR (Arithmetic Register)
  BR (Buffer Register)
  MQ (Multiplier/Quotient Register)

  The first two registers also have fullword extensions: ARX and BRX.

- Fast Memory (FM) contains the accumulators (ACs). The EBOX has eight AC sets.

- Virtual memory address (VMA) keeps the PC and sends the virtual address to the pager in the MBOX.

- Virtual memory address adder (VMA AD) helps the VMA in its computations.

- Program Counter (PC) holds the virtual address of the next instruction to be executed.

The MBOX is composed of:

- Pager (also known as the hardware page table), which holds 512 (MCA20) or 1024 (MCA25) mapping entries from the EPT or UPT.

- Physical Memory Address register (PMA), which holds the physical memory address of the next instruction.

- Cache (data and directory): high-speed semiconductor memory that stores copies of data from regular memory in order to speed up memory fetches. (MCA20 allows up to 2K of storage; MCA25 allows up to 4K of storage.)

- Memory Buffer (MB), to control the flow of data to and from cache, channels, memory, and the EBOX.

- Cache/MB interface, connecting cache to MB.

In addition, a number of buses and interfaces may be connected to the MBOX, EBOX, and other parts of the system, such as:

- E/M interface connects the MBOX and EBOX.
- S/X BUS/MB interface connnects the MBOX with the core/MOS controllers. The DMA20 is on the SBUS and interfaces to external memory.
- EBUS connects the EBOX to four DTE20s or eight RH20 slots (which may contain RH20 or KLIPA/KLNI controllers) and the DIA20/DIB20 interface to the traditional I/O bus devices.

Combinations of the following modules connect memory and MASSBUS devices:

- Channel/MB interface connects MB with the channel controller.
- Channel controller controls the flow of data through the CBUS.
- CBUS and CBUS interface handles data transfers that go directly to the MBOX, bypassing the EBOX.
- RH20 MASSBUS controller connects the CBUS to the MASSBUS.
- MASSBUS is a standard bus for interfacing tapes and disks to the KL.
- Device controller (BA10, TD10, RH10,...).
- I/O bus (PTP, PTR,...).
- Channel interfaces (DX10, DX20,...).
- CI20 port connecting the KL10 with the CI20 bus.
- NIA20 port connecting the KL10 with the Ethernet cable.

The KL10 dynamically generates parity in the following places:

- On the output side of the channel status RAMs
- On the output side of the AR
- Entering the pager from MB or AR
- Data stored in fast memory
- Data stored into the channel data buffers (18-bit parity is generated)

Parity is checked after the following operations:

- On all requests from the MBOX
- Data leaves MB to go to the DMA20, pager, channel, cache, AR or the arithmetic extender
- Data is paged out
- Data enters and leaves the RH20 or the MASSBUS

- Data enters the AR from the MBOX
- Data enters and leaves AR during DTE PI Level 0 interrupt handling
- Data enters the ARX from the MBOX
- Data leaves fast memory
- Control leaves CRAM/DRAM

Errors detected through parity checking in the last two conditions cause the KL (EBOX/MBOX) clock to halt immediately, provided that the correct conditions have been enabled. The relationships among the places where errors are detected and the condition they evoke is shown in the following table. Note that parity is generated by the transmitting device. This table does not include power-fail conditions.

Table 5-1: Hardware Errors

Component	Error	Error Indicator
MA20	Incomplete cycle	SBUS error bit
	Address parity error	Address parity bit
DMA20	Data parity error	SBUS error bit
	Address parity error	Address parity bit
	NXM error	SBUS error bit
MB	Data parity error	MB parity error bit
	Nonexistent memory	NXM error bit
Pager	Page table parity error	Page fail trap code=25
	Pager to cache directory	CD parity error bit
Arithmetic Logic:		
(AR, ARX)	AR parity error	Page fail trap code=36 (for Exec) code=76 (for User)
	ARX parity error	Page fail trap code=37 (for Exec) code=77 (for User)
	AR/ARX/EBUS parity error*	I/O page fail bit
RH20	Data parity error	Device interrupt
DX10	Data parity error	Device interrupt

* This type of error includes any type of paging failure while PI CYCLE is set. The PI CYCLE is a microcode condition that is enabled when the microcode honors a PI request and is disabled when the first XPCW instruction occurs for Levels 1-7 or a Level 0 request is completed.

5.2 STOPCODES

Stopcodes are symbolic names representing errors detected by the monitor. Stopcodes are generated by the STOPCD or BUG. macros. The DIE routine records error information and initiates a reload, if required. For a complete list of stopcodes, refer to the Stopcodes Specification.

The CTY for each CPU in a multi-CPU configuration records the stopcodes that occur on that CPU. You can use FILDDT to find the module where a stopcode is defined. You can find a stopcode in the crash file by looking for a symbol of the form S..name (for 3-character stopcode names) or just name (for 6-character stopcode names). The following example shows how to find the module where a KSW stopcode is defined:

```
S..KSW?

TAPSER G
```

Stopcodes are defined in many modules of the monitor, but they are generated by the same macro, the STOPCD macro. The STOPCD macro is called with:

```
STOPCD cont,type,name,disp
```

where:

cont — is the location to jump to after processing the error.

type — is the type of failure and determines the specific course of action. It can have one of the following values:

- HALT
- STOP
- JOB
- CPU
- DEBUG
- INFO
- EVENT

name — is the unique stopcode name.

disp — is the address of the routine containing additional information, if appropriate.

The severity of the error is indicated by the type of stopcode. The types of stopcodes are:

- HALT stopcodes occur after the most severe errors. The CPU cannot continue automatically after a HALT, no additional information is displayed on the CTY, and no information is saved (no crash file is automatically created). HALT stopcodes are also the least likely of the stopcodes to occur, and are usually caused by recursive calls to the DIE routine.

  HALT stopcodes indicate serious problems that endanger further system operation. The RSX-20F console front-end (using the HALT.CMD file) gathers pertinent status and error information.

- STOP stopcodes are the also serious, and cause the system (all CPUs) to put their status into memory and wait for the policy CPU to dump and reload the monitor.

- JOB stopcodes are those that affect only one job but may indicate problems in the system. If there is an interrupt in progress, the system will be reloaded. If not, only the faulty job will be terminated. Then a dump is taken and the system continues.

- A CPU stopcode is important only for multiple-CPU systems. This stopcode will stop only the current CPU, leaving the others running. It acts as a STOP stopcode in any of the following cases:

  - Single-CPU systems

  - Only one processor running in an multiple-CPU system

  - If DF.CP1 is set in the DEBUGF word.

- A DEBUG stopcode affects the system in different ways, depending on the contents of the DEBUGF word (short for DEBUG Flags). By setting certain bits in this word, a system programmer can control the effect of certain stopcodes, and manner in which the system is reloaded. The DEBUGF flags are listed in Section 6.3.

- An INFO stopcode displays a message on the CTY and rings the terminal bell, informing the operator of an event that may be of interest. Most INFO stopcodes are harmless and can be ignored. They do not halt the system or job, do not initiate a memory dump, and do not cause a system reload.

- An EVENT stopcode displays a message on the CTY, similar to an INFO stopcode, but does not ring the terminal bell.

5.2.1 Stopcode Processing

The DIE routine in ERRCON processes stopcodes in the following manner:

1. Increments .CnDWD to indicate that this CPU has died and to protect the code from being entered twice by that CPU.

2. Saves the PI status in .CnCPI and turns off the PI system.

3. Saves AC Blocks 0, 1, 2, 3, and 4 in memory.

4. Stores stopcode PC in %SYSPC and .CnSNM.

5. Sets up error stack from ERnPDL.

6. Creates CPU and device status block data using RCDSTB, and calls DAEMON to output those buffers.

7. Initiates a cache sweep and waits with control in the ACs until the sweep is finished.

8. Enters the secondary protocol.

9. Attempts to get the DIE interlock.

10. Prints stopcode information on CTY.

11. Dispatches to the routine that will take the dump and handle the specific type of stopcode.

INFO and EVENT stopcodes perform all the functions listed here, except that they do not turn off the PI system, do not halt the system, and do not perform a dump and reload. The EVENT output on the CTY is formatted differently from the other types of stopcodes.

5.2.2 Continuing from Stopcodes

JOB and DEBUG stopcodes do not ordinarily crash the system. They allow error collection to be done, and then the system can continue. Whenever a JOB or DEBUG stopcode occurs, the default action of the monitor is to dump memory to disk for later analysis. This is known as a continuable stopcode dump and is handled by BOOT. This allows the system to continue to do work even though the state of the machine is being saved.

The majority of stopcodes are caused by a corruption of some portion of the monitor's database. Often, a corrupted piece of data will cause several stopcodes, one right after the other. However, the first dump is the most important. When you are analyzing a series of crashes, look at the first crash in the series.

If two or more crashes have the same time stamp, you should look at the dump with Bit 8 clear in the DEBUGF word. You can probably ignore the other dump(s). Refer to Section 6.3 for more information about DEBUGF flags.

5.2.3 Special Stopcodes

Certain stopcodes occur more frequently because they represent a wide range of problems. Under these conditions, debugging becomes more difficult. The stopcodes of this type that you should be aware of are KAF, IME, UIL, and EUE. The causes for them mentioned in the following paragraphs are not complete, but they illustrate the way such a stopcode could occur.

Keep-Alive Fail (KAF) stopcodes occur when the system is hung or looping. In this situation, you cannot get response from the terminals, there are no jobs running, and no I/O is being done. Eventually, the front-end, RSX-20F, realizes the keep-alive count has expired, and forces the KL to execute the instruction in physical location 71 of memory, XPCW@.CnKAF, which stores the contents of P in KFnSVP, and issues the KAF stopcode. The address (a double-word PC) of the instruction that was being executed is stored at APnKAF and APnKAF+1.

A KAF occurs when something prevents the processor from reaching clock level, thus preventing the keep-alive count from being updated and scheduling from being done. This can occur if a process at a higher PI level never exits, which could be caused by one of the following:

- A higher level interrupt goes into an infinite loop.
- A higher level interrupt does not clear an interrupt signal when the interrupt routine exits. The signal, being constantly asserted, causes one interrupt after another.
- The clock does not tick because it has malfunctioned.
- The clock does not tick because the PI system has been disabled.
- A monitor routine does not release an interlock.
- A CPU in a multiple-CPU system does not release a CPU interlock.

IME stands for Illegal Memory Reference from Executive and is issued when an unexpected page fault occurs in exec mode. Some of the potential causes for an IME include:

- An attempt to write into the monitor's high segment.
- An attempt to reference data mapped through a UPT that is not addressable.
- Invalid indexing because accumulators were misused.

To solve IMEs, you can look at the following locations in the UPT:

- .USPFW (location 500) contains the page fail word.
- .USPFP (501) contains the flags in the left half.
- .USPFN (502) contains the PC of the page fail instruction.

The CDB also contains some relevant information, referenced by the following symbols:

- .CnAPC contains the APR error or trap PC on this CPU.
- .CnPFW contains the page fail word on traps to SEILM.
- .CnPPI contains the results of CONI PI, on a parity/NXM trap.
- .CnTCX contains the page fail word context word on traps to SEILM.

EUE stands for Executive UUO Error and occurs when the monitor attempts to execute an illegal UUO (usually with an opcode of 0). This stopcode is usually the result of the monitor branching to an address that contains data instead of an instruction. Its causes are very similar to that of an IME. The same problem may produce an EUE one time and an IME another time, depending on specific conditions.

To solve EUE and UIL stopcodes, you should look at the contents of the following locations in the UPT:

- .USMUO contains the flags and left half of the UUO.
- .USMUP contains the address of the UUO routine.
- .USMUE contains the effective address half of the UUO.
- .USUPF contains the process context word at the time of the UUO.

5.3 ERRORS DETECTED BY RSX-20F

When the RSX-20F console front-end detects certain KL error conditions, it collects data using command files (sometimes called TAKE files). The error conditions and the command file for each are listed below.

The command files are used to gather status and error data for special cases, and (on single-CPU systems) to assist in system continuation after a stopcode.

When the RSX-20F reload-enable flag is set, the following command files are automatically executed for the following conditions:

File	Error Condition
CLOCK.CMD	Field service probe clock error stop
CRAM.CMD	Control RAM (CRAM) clock error stop
DRAM.CMD	Dispatch RAM clock error stop
EBUS.CMD	EBUS parity error
FMPAR.CMD	Fast memory parity clock error stop
DEX.CMD	Deposit/Examine failure
HALT.CMD	KL executes HALT instruction
TIMEO.CMD	Protocol timeout condition
KPALV.CMD	Keep-alive failed condition (*)
DUMP.CMD	Optional system hung file

---

* When a Keep-Alive Fail occurs, the KPALV.CMD file is not used immediately. Instead, RSX-20F attempts to reload the monitor at location 71 (described in Section 5.2.3). If the front-end fails to reload the monitor, RSX-20F takes a Keep Alive Fail and executes the KPALV.CMD file. However, if the Retry-Enable Flag (which is set, by default) is cleared, the KPALV.CMD file is executed immediately without trying a reload.

  The KPALV.CMD is useful when the system hangs without doing any productive work. You can execute KPALV.CMD to gather status information and force a dump. To invoke KPALV.CMD, type the following commands on the CTY:

```
^/                          ;<CTRL-backslash>
PAR>TAKE KLPALV             ;initiates the .CMD file
```

CHAPTER 6

DEBUGGING THE MONITOR

There are two ways to make corrections to the monitor. The first method is to alter the running monitor using the monitor-specific FILDDT. You can use this method when the changes are small and it is unlikely that the system will crash due to patching errors. The second method involves taking the system standalone and loading the monitor with EDDT.

6.1 PATCHING WITH FILDDT

The monitor-specific FILDDT contains functions that allow you to change or patch the running monitor. To run FILDDT and patch the monitor, you must use the following commands:

```
.R MONDDT
File: /M/P
```

The /M switch indicates that all Examine and Deposit functions will refer to the running monitor. The /P switch allows you to patch the monitor. To use these switches, your job must have PEEK and POKE privileges.

Often the changes to be added in the monitor do not fit easily into the existing code. To add several lines of code, you must access the pre-allocated patching space that is resident in the running monitor. The patching space starts at the address pointed to by the symbol PATCH. The amount of words reserved for patching space is assembled into the monitor module PATCH.MAC (the symbol is PATSIZ), but the patch area is usually 50 (octal) words long. It is recommended that large changes be made directly to monitor sources, not to the running monitor.

CAUTION

When you install a change to the running monitor, remember that the monitor code should not dispatch to the patched location until you have installed the entire patch. Therefore, the instruction that dispatches to the changed code should be the last instruction you install. It is recommended that you use the $< command to FILDDT specifying PATCH as the patching area.

6.2 USING EDDT

EDDT is a version of DDT that runs in both user and exec modes. EDDT is part of the monitor, in the sense that it resides in the monitor's .EXE file and is loaded into core with the monitor. The command to BOOT to enable debugging with EDDT is:

```
BOOT>monitor-filespec/EDDT
```

The /EDDT switch instructs BOOT to start at the EDDT start address rather than the monitor's normal starting address. You can type /EDDT or /START:401.

When BOOT starts the monitor at location 401, the CPU is running unmapped. In this mode, EDDT could run, but the symbol table is inaccessible. Since this situation would provide only limited debugging capabilities, the monitor sets up minimal page mapping. When this is done, all monitor code and the symbol table will be accessible from EDDT. The monitor than jumps to EDDT.

When EDDT starts, it displays "EDDT" on the CTY and it is similar to user-mode DDT. There is no prompt, and the command syntax is nearly identical to DDT. For more information on the exec-mode debugging commands, refer to the TOPS-10 DDT Manual.

6.2.1 Starting the Monitor

When the monitor is loaded into core, data storage mapping and devices have not been configured. However, most of the useful information on the status of the monitor is contained in the monitor's high segment.

The monitor will be mapped after you start it, but normally the monitor's symbol table, EDDT, and the SYSINI locations are cleared after initialization. You can preserve the symbol table, EDDT, and SYSINI initialization code by starting the monitor at location DEBUG, using the following command to EDDT:

```
DEBUG$G
```

On a normal startup, the monitor discards its symbol table, EDDT, and SYSINI initialization code. The address space is reclaimed for the monitor's Section 0 free core pool. However, when you use EDDT to load the monitor (using the DEBUG$G command), this address space is preserved, and the symbol table is moved into Section 35 (KL10) or out of the monitor's address space into unmapped core (KS10). A pointer to the physical address of the symbol table is stored in the Exec Data Vector for use by EDDT.

6.2.2 Breakpoints

You can insert breakpoints anytime after the EDDT prompt. Unless you are debugging system initialization code, it is useful to set an initial breakpoint at the label "HIGHIN". When this point in the code has been reached, the monitor is ready to run. That is, all other CPUs have been started, channels can be autoconfigured, and so forth.

After the monitor starts running, you can type <CTRL/D> on any CTY to enter EDDT on the current CPU. SCNSER intercepts the <CTRL/D> character at interrupt level, saves the contents of the current AC block, and executes an unsolicited breakpoint entry into EDDT. Then you can type any valid EDDT command on the CTY. You can resume monitor execution by typing $P. SCNSER will ignore the <CTRL/D> character that caused control to pass to EDDT. The <CTRL/D> facility is controlled under timesharing by the use of the following monitor command on the CTY:

```
.SET EDDT BREAKPOINT [OFF/ON]
```

The default setting for this command is ON when Bit 0 is set in the DEBUGF word.

6.3 DEBUGF FLAGS

The DEBUGF word contains the following flags, which can be set and cleared using OPR commands. The most useful flag for the systems analyst is Bit 0, the sign bit. This flag indicates that EDDT is loaded for debugging the monitor and enables breakpointing monitor code.

Bit	Name	Description
0	DF.SBD	System being debugged (EDDT loaded).
1	DF.RDC	Reload on DEBUG stopcodes.
2	DF.RJE	Reload on JOB stopcodes.
3	DF.NAR	Do not automatically reload.
4	DF.CP1	Stop entire system on any CPU stopcode.
5	DF.DDC	Do not output a memory dump on a DEBUG stopcode.
6	DF.DJE	Do not output a memory dump on a JOB stopcode.
7	DF.DCP	Do not output a memory dump on a CPU stopcode.
8	DF.RQC	Start CRSCPY program to copy the previous crash file at the time of the next clock tick on the policy CPU.
9	DF.RQK	Call KDPLDR on the next clock tick.
10	DF.RQN	Call KNILDR on the next clock tick (obsolete).
11	DF.WFL	Copy output to FRCLIN at system CTY.
12	DF.DDC	Disable next CRSCPY request.
13	DF.RIP	Reload in progress (RECON. function .RCRLD)
14	DF.RAD	Reload after dump (don't dump twice in BOOT).
15	DF.RLD	Stopcode caused by a reload (used CRSCPY).
18	DF.BP0	Can enter EDDT on CPU0 using XCT .C0DDT.
19	DF.BP1	Can enter EDDT on CPU1 using XCT .C1DDT.
20	DF.BP2	Can enter EDDT on CPU2 using XCT .C2DDT.
21	DF.BP3	Can enter EDDT on CPU3 using XCT .C3DDT.
22	DF.BP4	Can enter EDDT on CPU4 using XCT .C4DDT.
23	DF.BP5	Can enter EDDT on CPU5 using XCT .C5DDT.

For example, suppose you want to stop the system before reloading to reconfigure the hardware. To do this, Bit 3 in the DEBUGF word should be set. To disable automatic reloads, run the OPR program and type the following commands to CONFIG:

```
.R OPR<RET>
OPR>ENTER CONFIG<RET>
CONFIG>SET NO AUTO-RELOAD<RET>
CONFIG>EXIT<RET>
```

6.4 MULTI-CPU ENVIRONMENT

Debugging a multiple-CPU system requires special considerations. EDDT performs all terminal I/O for the CTY that encountered the breakpoint. It is not unusual to use all CTYs on the system during a debugging session.

When a CPU stops at a breakpoint, normally the other CPU(s) will continue to run. If the breakpoint occurred on a non-policy CPU, the CTY on the policy CPU will report the following message:

```
problem on CPUn ...
```

However, if the breakpoint occurs on the policy CPU, a role switch occurs and another CPU assumes the role of the policy CPU. Although this behavior is desirable during timesharing, the role switch makes it very difficult to debug a multiple-CPU monitor when more than one CPU is running. Also, when the CPUs in the system detect the fact that one of the CPUs is not running, interlocks owned by the halted CPU are broken. If the CPU was actually paused at a breakpoint, and then continued, CIB stopcodes can occur.

To prevent role switching, a flag (DEBCPU) is set, and contains the CPU number on which you typed DEBUG$G. DEBCPU is checked in the BRKLOK and BECOM0 routines, to prevent possible role switches. This may be circumvented by patching a JFCL at DDTCPU prior to typing DEBUG$G.

Monitor messages are sent once per hour on the CTY. The following patch will circumvent this BIGBEN routine:

```
BIGBEN/POPJ P,
```

6.5 CAUTIONS

Remember, EDDT provides little protection against user errors. Keep the following points in mind when you are debugging a running monitor:

- EDDT cannot execute a UUO when you issue the $X and $$X commands. This is a restriction. Attempts to do this on a KL usually result in a PI Level 0 Interrupt Error from RSX-20F. The monitor performs some UUOs internally, in the SAVE/GET code, and the CLOSE and FINISH commands.

- You can change the AC block for EDDT when the monitor is at a breakpoint and you wish to deposit data into an AC block other than the current one. Use the following command to change to the AC block you specify (n):

  ```
  n$4U
  ```

  Do not attempt to use AC Blocks 6 or 7 on a KL10. This will crash the system because the microcode uses portions of AC Block 6 and all of AC Block 7.

- On a multiple-CPU system, there are locations in ONCMOD and SYSINI where the CPU must wait for another CPU to finish an operation. If that other CPU is halted at a breakpoint, the waiting CPU will time out. You must devise specific patches at CPUXCT to prevent this situation.

APPENDIX A

GLOSSARY

The table below provides an alphabetized list of the abbreviations and acronyms used in this manual, with expanded names to define them.

Table A-1: Glossary of Acronyms

Acronym	Meaning
AC	Accumulator
APR	Arithmatic Processor
BR	Buffer Register
CDB	Central Processing Unit Data Block
CFP	Compressed File Pointer
CHN	Channel Data Block
CI	Computer Interconnect
CPU	Central Processing Unit
CRAM	Control Random-Access Memory
CTY	Console Terminal
CX	A job context
DDB	Device Data Block
DDT	DEC Debugging Tool
DRAM	Dispatch Random-Access Memory
EBR	Exec Base Register
EPT	Exec Process Table
EVM	Exec Virtual Memory
FM	Fast Memory
I/O	Input/Output
IORB	Input/Output Request Block
IPCF	Interprocess Communication Facility
IR	Instruction Register
JDA	Job Device Assignment table
KDB	Controller Data Block
KON	Disk Controller Data Block
LDB	Line Data Block
MB	Memory Buffer
MFD	Master File Directory
MQ	Multiplier/Quotient Register
MUUO	Monitor UUO (see UUO)
NI	Network Interconnect
NZS	Non-Zero Section
PC	Program Counter
PDB	Process Data Block
PI	Priority Interrupt
PMA	Physical Memory Address
PPB	PPN Data Block
PPN	Project-Programmer Number

PTY	Pseudo-Terminal
PWQ	Position Wait Queue
RAM	Read-Access Memory
RIB	Retrieval Information Block
SAT	Storage Allocation Table
SCA	Systems Communications Architecture
SCS	Systems Communications Services
SFD	Sub-File Directory
SMP	Symmetric Multiprocessing
SPR	Software Performance Report
SPT	Special Pages Table (for mapping) Storage Allocation Pointer Table (for disk I/O)
STR	Structure Data Block
TKB	Tape Controller Data Block
TTY	Terminal
TUB	Tape Unit Data Block
TWQ	Transfer Wait Queue
UBR	User Base Register
UDB	Unit Data Block
UFD	User File Directory
UNI	Disk Unit Data Block
UPT	User Process Table
UUO	Unimplemented User Operation (monitor call)
VMA	Virtual Memory Address

APPENDIX B

ADDRESS SPACE LAYOUT

Monitor Code Section Layout

NOTE

The specifications shown in the following figures are subject to change without notice. Addresses are shown for comparison purposes only; actual addresses may be different depending on your specific monitor configuration.

Monitor Code Section Layout

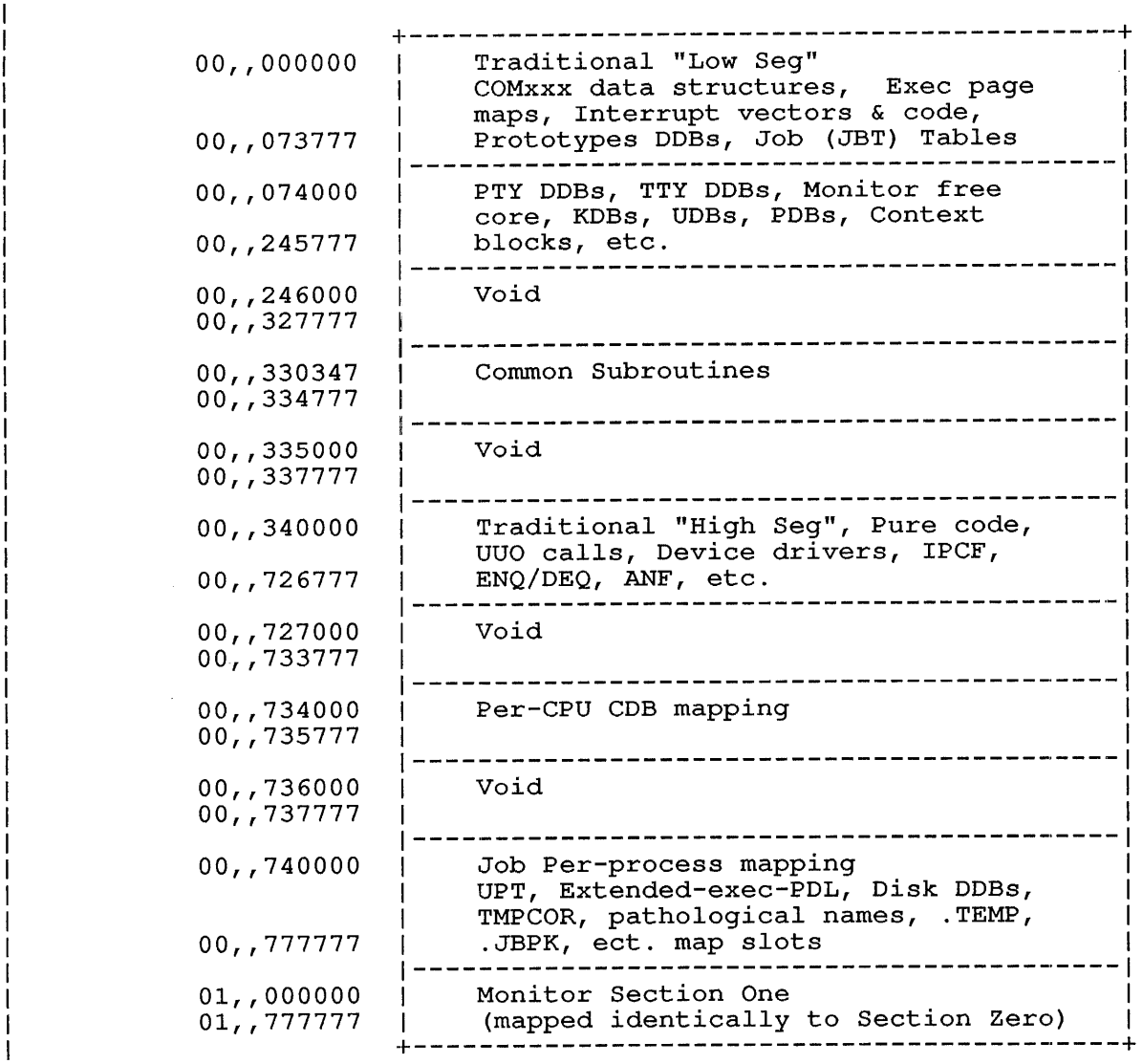


Figure B-1: Monitor Code Section Layout

DECnet Code Section Layout

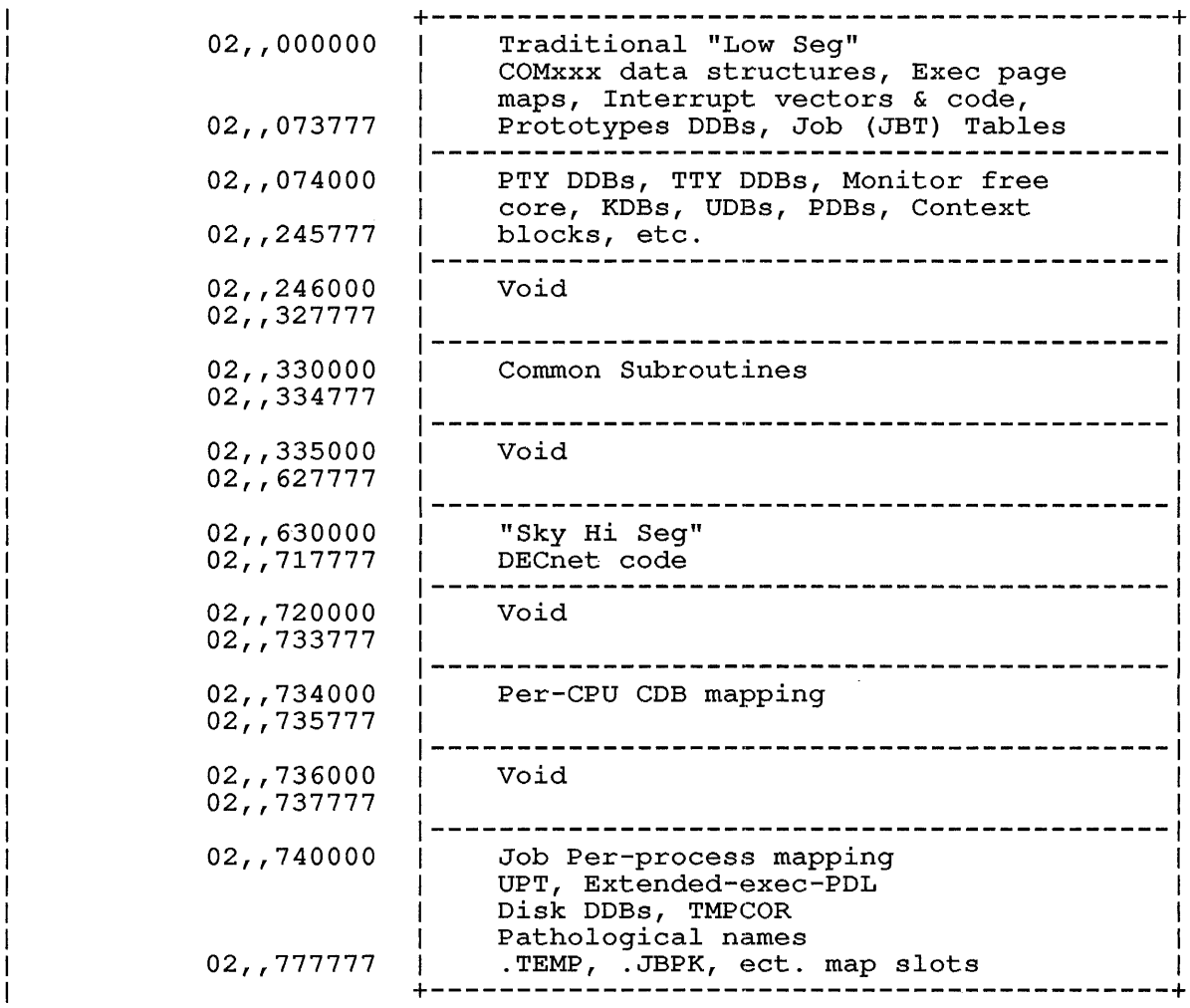


Figure B-2: DECnet Code Section Layout

Monitor Data Section 3 Layout

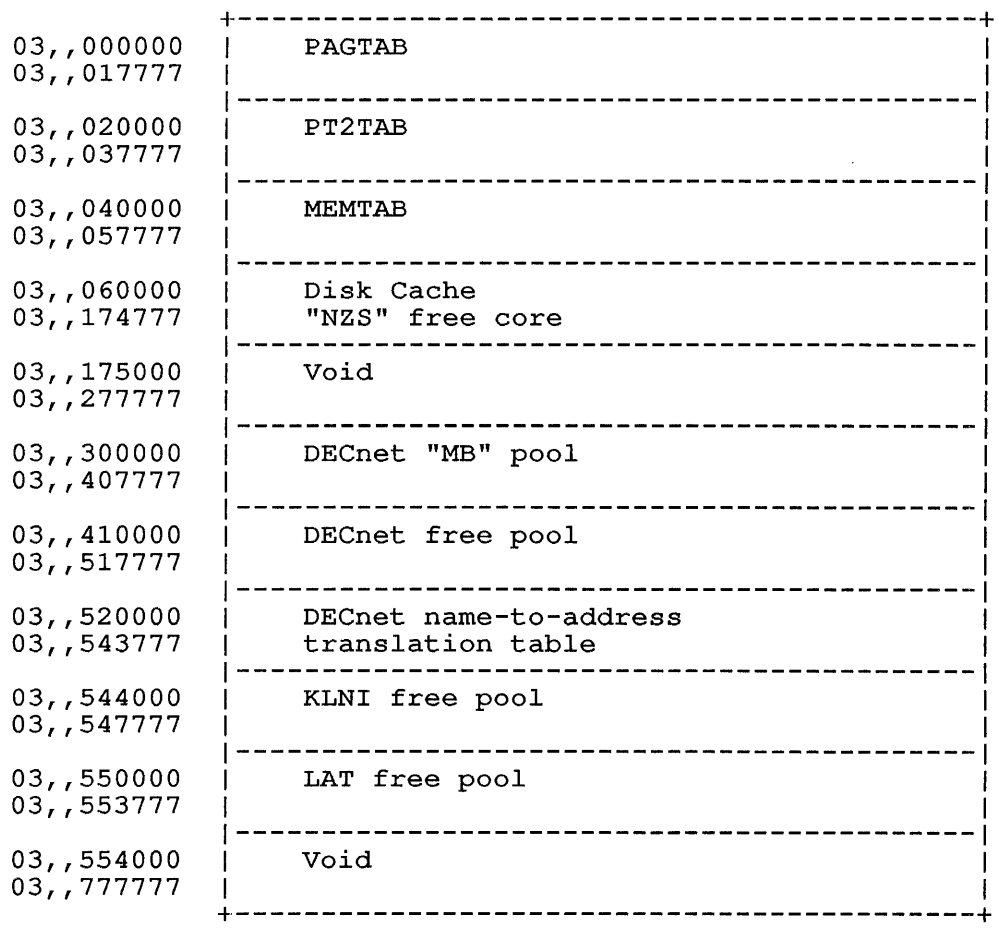


Figure B-3: Monitor Data Section 3 Layout

Monitor Data Sections 4,5 Layout

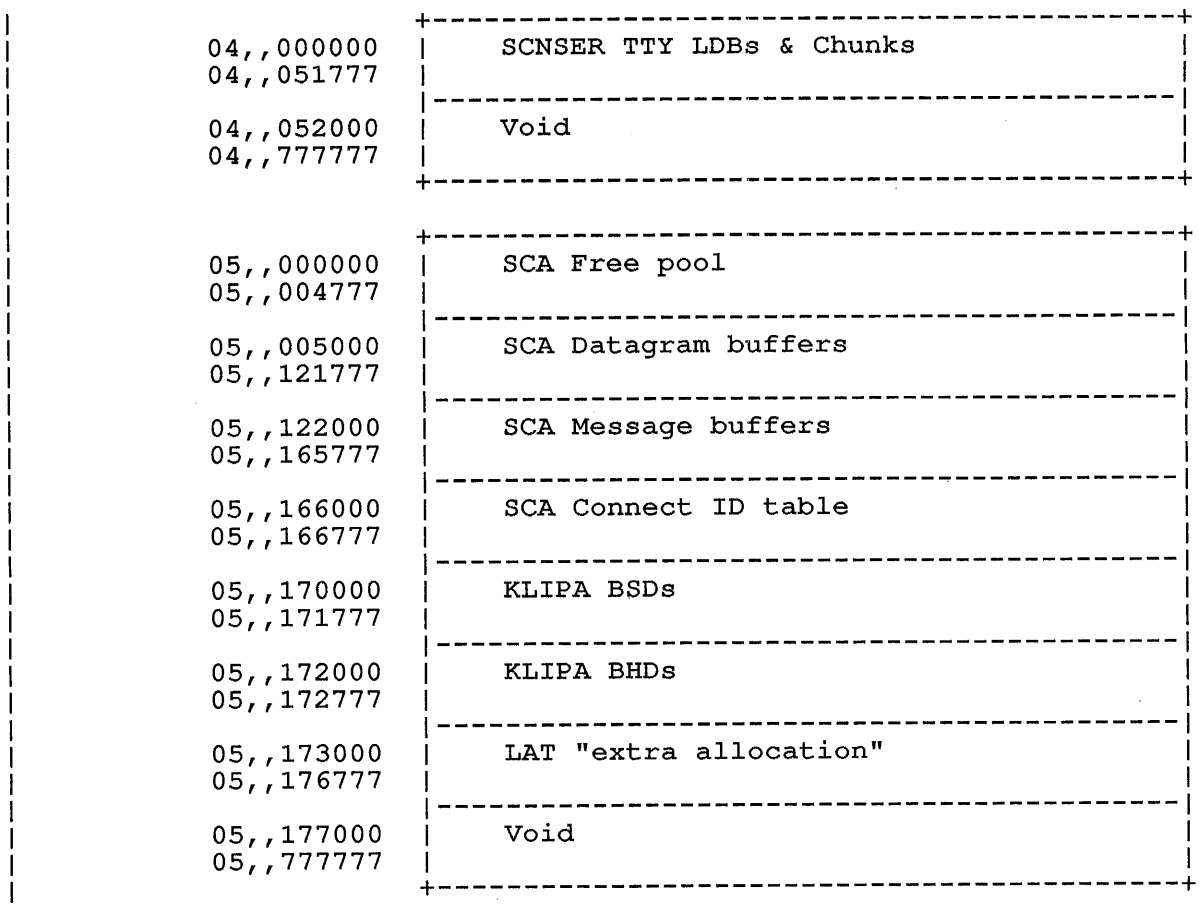


Figure B-4: Monitor Data Sections 4,5 Layout

Monitor Data Sections 6,7 Layout

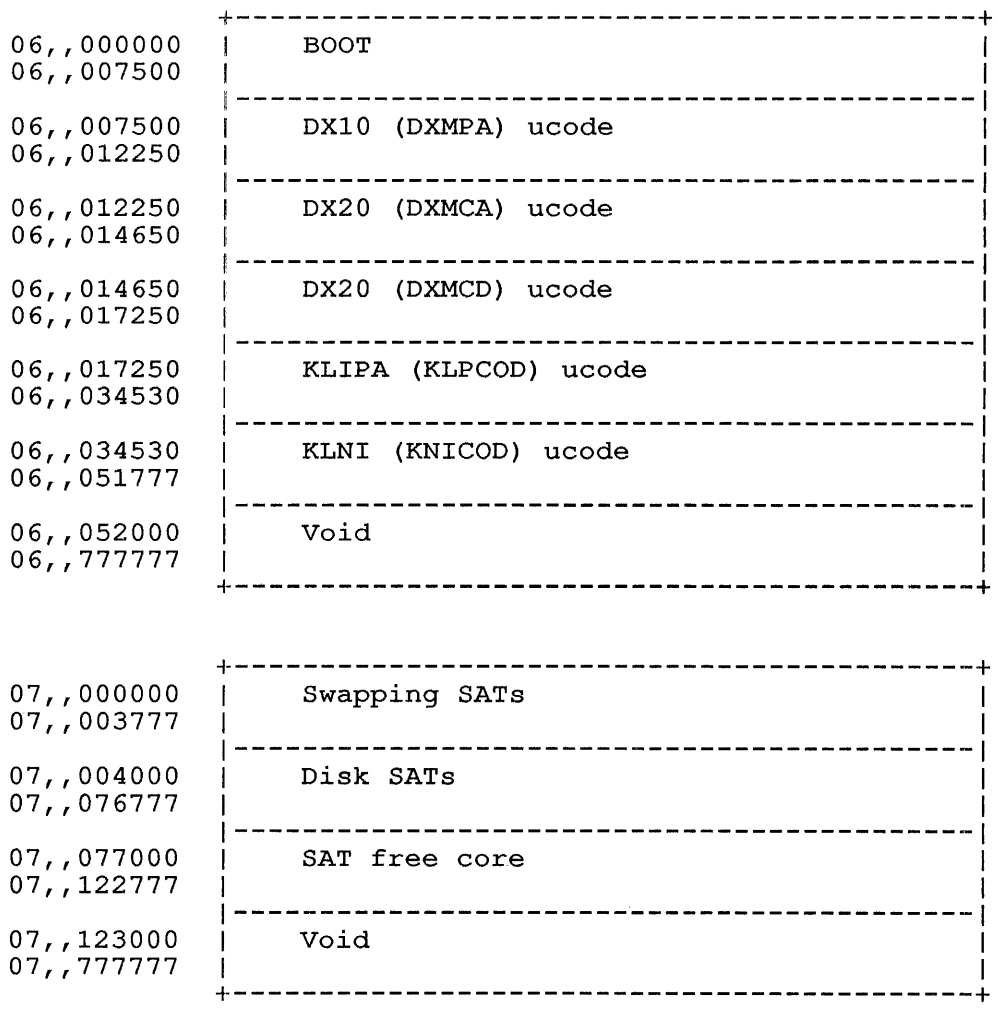


Figure B-5: Monitor Data Sections 6,7 Layout

Monitor Data Sections 35,36,37 Layout

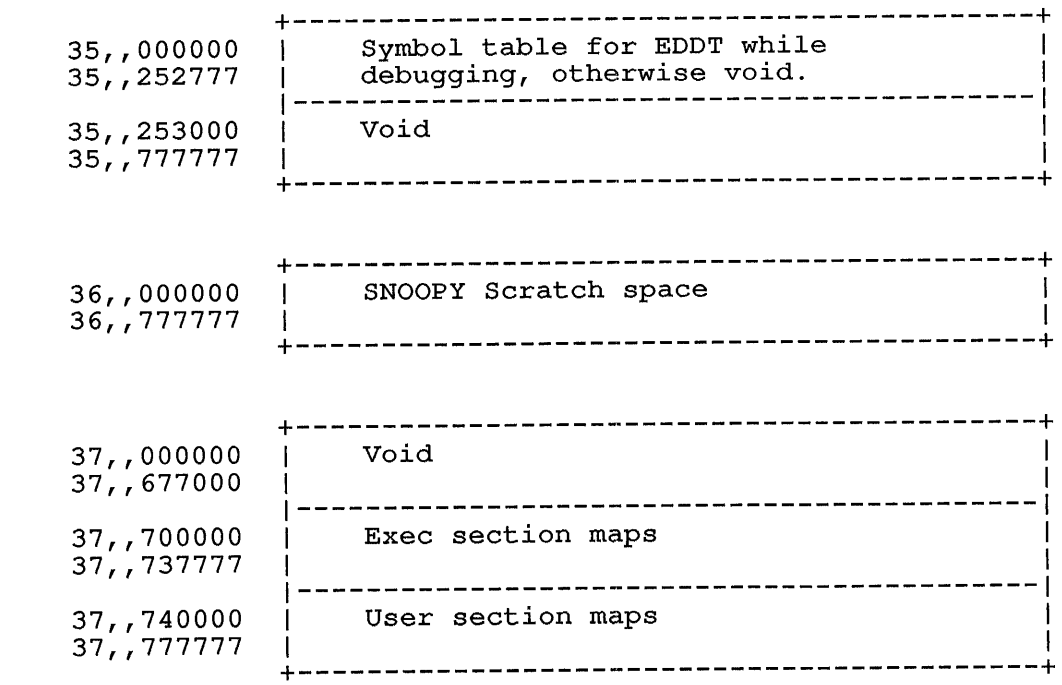


Figure B-6: Monitor Data Sections 35,36,37 Layout

-A-

AC blocks
  finding, 3-13
  switching, 3-5, 4-6
Access
  codes, 3-2
  table (ACC), 4-27
Accumulators, 2-5
  locations, 3-13
  monitor, 3-6, 4-2
  saving, 3-9
  scheduler, 3-13
  traps, 3-12
  user, 4-8
Addressing non-zero sections, 3-4
Allocating disk space, 4-23
Alternate page maps, 3-3
ANF-10 networks, 4-12
APR interrupts, 5-3
APRSER module, 5-4
AR/ARX parity errors, 5-5
Arithmetic Logic Unit (ALU), 5-6
Assigning channel numbers, 4-11
Attached terminals, 4-17
AU resource, 4-10
AUTCON module, 3-14, 4-20, 4-26
Automatic reloads, 2-2
AVALTB table, 4-10

-B-

30-bit addressing, 3-4
Blocking
  programs, 3-5
  user jobs, 3-6
BOOT, 2-1, 2-2
Booting systems, 2-2
Break characters, 4-16
Breakpointing monitors, 6-2
BUG. macro, 5-9
Building monitors, 4-4
Byte pointers, 4-4

-C-

Cacheable pages, 3-2
Caching
  disk information, 4-28
  UPT locations, 3-6
CALLI UUOs, 4-11
CDB
  constants area, 4-9
  defining locations, 4-9
  variables area, 4-9
Changing AC sets, 6-4
Channels, 4-11
  data blocks (CHN), 4-19, 4-20, 4-27
Channels (Cont.)
  error report, 5-4
  status bits, 4-11
Checking parity, 5-7
Chunks
  counts, 4-16
  terminal, 4-16
Clearing virtual addressing, 2-5
Clock, 3-12
CLOCK1 module, 3-14
Clusters, 4-23
CNFDVN location, 2-7
COMDEV module, 4-5
Command
  dispatch bits, 4-2
  files
    FILDDT, 2-8
    RSX-20F, 5-13
  tables, 4-11
COMMOD module, 4-5
COMMON module, 3-14, 4-5, 4-9, 4-17, 5-3
Common modules, 4-5
Compressed File Pointer (CFP), 4-23
COMTAB table, 4-11
Concealed mode, 3-4, 3-5
Conditionals, 4-6
Connecting devices, 5-7
CONSO skip chain, 3-9, 3-14
Console
  front-ends, 5-13
  terminal, 1-1
Continuable stopcodes, 1-2, 5-11
Control RAM (CRAM), 5-6
Controller data block (KON), 4-27
Controlling terminal, 4-17
Copying crash files, 2-2
CPNSER module, 3-15
CPU
  Data Blocks (CDBs), 2-5, 4-9, 5-5
  interlocks, 6-4
  stopcodes, 5-10
Crash
  analysis, 1-1
  files, 1-1, 2-1
  space, 2-1
Crash files, 1-1, 1-4, 2-1, 2-2
CRASH.EXE file, 2-1
Creating
  crash files, 2-1
  FILDDT command files, 2-8
  symbolic FILDDT, 2-3
CREF
  listings, 4-6
  program, 4-5
CRSCPY program, 2-1, 2-2
CTXSER module, 3-15

CTY, 1-1
Current ACs, 2-7
Cursor position counter, 4-16
CX resource, 4-10
CYCLE error, 5-8
Cycles, 3-12

-D-

D36PAR module, 3-15
DA resource, 4-10
DDBs, 4-11
DEBUG stopcodes, 5-10
DEBUGF word, 5-11, 6-3
Debugging the monitor, 6-1
DECnet
  front-ends, 4-12
  layout, 3-3
Defining
  CDB locations, 4-9
  symbols, 3-15
Device
  codes, 3-5
  Data Blocks (DDBs), 4-2, 4-12
  information, 4-12
  interrupts, 3-8
  status word, 4-2
Devices
  RDA, 4-12
DEVIOS word, 4-2
DIE routine, 5-9, 5-10
DIECDB location, 2-5
Directories, 4-21
Disabling
  extended addressing, 2-5
  time messages, 6-4
  user addressing, 2-6
Disk
  cache, 4-28
  controller data block (KON), 4-2
  device data blocks, 4-28
  dual-ported devices, 4-6
  file structure, 4-21
  I/O, 4-21
  on-line information, 4-26
  storage allocation, 4-23
Dismissing interrupts, 3-9
DISP table, 4-11
Dispatch RAM (DRAM), 5-6
DN20 front-ends, 4-12
Doubleword PC, 3-4
DTE
  DDBs, 4-12
  interrupts, 3-10
DTEPRM module, 3-15, 4-7
DTESER module, 4-12
Dual-ported disks, 4-6

-E-

Echo count, 4-16
EDDT, 6-2
Enabling addressing, 2-5
ENQ/DEQ
  module, 3-15
ERnPDL stack, 3-13, 5-4
ERRCON module, 5-10
Error
  handling, 5-1
  hardware codes, 3-11
  parity, 5-8
  processing routines, 3-13
ETHPRM module, 3-15
EUE stopcodes, 5-13
EV resource, 4-10
EVENT stopcodes, 5-10
Exec
  Base Register (EBR), 3-2
  kernel mode, 3-4
  mode, 3-2, 3-4, 3-5
  Process Table (EPT), 2-4, 2-5, 3-2
Exec-mode DDT, 6-2
EXECAC macro, 4-6
Execute-only programs, 3-5
Executing command files, 2-8
Execution Box (EBOX), 5-6
Executive UUO Error (EUE), 5-13
Exiting FILDDT, 2-4
Extended
  addressing, 2-5, 3-3
  channel table, 4-11
  software channels, 3-3

-F-

F module, 3-15, 4-6
FAKEAC flag, 2-5
Fast Memory (FM), 5-6
Fatal errors, 1-1, 1-2
Fault continuation, 1-2
Feature test options, 4-6
FILDDT
  command files, 2-8
  mapping commands, 2-5
  program, 2-3
Finding
  AC blocks, 3-13
  DDBs, 4-11
  stopcodes, 5-9
  symbolic definitions, 4-6
Flag-PC doubleword, 3-4
Flags for DEBUGF, 6-3
Forced commands, 4-11
Forced system dumps, 2-1
Forcing reloads, 2-1
Free core, 4-10
Front-ends, 4-12
Full clock cycle, 3-12
Funny space, 3-3

-G-

Generating parity, 5-7
GLOB program, 4-7

Global
  section references, 3-4
  symbols, 4-1, 4-7
Groups of disk data, 4-22

-H-

HALT stopcodes, 5-10
Halting systems, 2-2
Handling
  errors, 5-1
  interrupts, 3-9
Hardware
  addressing, 2-5
  error codes, 3-11
  errors, 5-1
  interrupts, 3-14
  mapping, 3-2
HOME blocks, 4-22

-I-

I/O
  channels, 4-11
  Request Block (IORB), 4-19
  status word, 4-2
  tables, 4-11
IF statement, 4-6
IME stopcodes, 5-12
INFO stopcodes, 5-10
Inserting breakpoints, 6-2
Instruction Register (IR), 5-6
Interlocks between CPUs, 6-4
Interrupt, 3-6
  accumulators, 3-9
  error-handling, 3-8
  handling routine, 3-8
  levels, 3-6
  PDLs, 3-9
  processor, 5-3
  stacks, 3-9
  Vector (IVIR), 3-10
Interrupting
  on Level 0, 3-8
  on Level 7, 3-12
Intertask communication, 4-12
INTTAB table, 4-10
Invalid mapping, 2-7
IPCSER module, 3-15
IVIR register, 3-10

-J-

JBT tables, 4-7
JBTPPB table, 4-27
Job
  context module, 3-15
  Device Assignment table (JDA), 4-11
  stopcodes, 5-10
  tables, 4-7
Job-specific monitor locations, 3-3
JOBDAT
  area, 3-6
  locations, 4-8
  module, 3-15, 4-7
  vestigial, 3-3

-K-

Keep Me bit, 3-2
Keep-Alive Fail (KAF), 5-12, 5-13
Kernel mode, 3-4
KL interrupt handling, 3-9
KL-paging, 2-5, 3-3
KLPPRM module, 3-15
KNO word, 4-28
KS
  alternate page maps, 3-3
  interrupt handling, 3-9
  reloading systems, 2-2

-L-

Label DDBs, 4-20
Line
  characteristics bits, 4-16
  Data Blocks (LDBs), 4-2, 4-15
LINTAB table, 4-16
Loading FILDDT symbols, 2-3
Local symbols, 4-1
  unlocking, 4-7
Locating EPTs, 2-5
Locations
  0-17, 2-5
  30, 2-1
  406, 2-2
  407, 2-2
  500, 3-11
  DIECDB, 2-5
LOKCON module, 3-15
Low segment addresses, 2-5

-M-

Macros, 4-5
MACSYM module, 3-15, 4-7
Magnetic tape devices, 4-19
Mapping
  ACs, 2-6
  dumps, 2-5
  exec virtual memory, 2-6
  extended sections, 2-5
  user jobs, 2-6
  verification, 2-7
  virtual addresses, 2-4, 2-5, 3-2
Master File Directory (MFD), 4-21
MCA25 bit, 3-2
MCB software, 4-12
Memory
  Box (MBOX), 5-6
  dump, 1-1
  tables, 4-10
MEMTAB table, 4-10

MIC information, 4-16
MM resource, 4-10
Mode flag, 3-4
Modules, 3-13
  common, 4-5
  monitor startup, 3-14
  optional, 3-15
  symbol definition, 3-15
MONGEN program, 4-4
Monitor
  ACs, 4-2
  breakpointing, 6-2
  building, 4-4
  command processing, 4-11
  functions, 3-1
  macros, 4-5
  modules, 3-13
  name, 2-8
  sources, 4-5
  startup modules, 3-14
  symbols, 4-1
  version numbers, 2-7
Monitor-resident user data, 3-3
Monitor-specific FILDDT, 2-3
MSCPAR module, 3-15
Multiple-KL systems, 4-6
MUUO, 3-6

-N-

Name Block (NMB), 4-27
Nested SFDs, 4-21
NETDEV module, 4-12
NETPRM module, 3-15, 4-7
NETSER module, 4-12
Network devices, 4-12
Non-Zero Sections (NZS), 2-5, 3-3, 3-4
Nonvectored interrupts, 3-8
NXM errors, 5-5

-O-

ONCE module, 3-14
Optional modules, 3-15

-P-

Page
  faults, 3-11
  map pointers, 3-2
  maps, 2-4, 3-2, 3-3
  tables, 4-10
Page fail
  codes, 3-11
  traps, 5-3, 5-4
  word, 3-11
PAGTAB table, 4-10
Parity
  errors, 5-4, 5-8
  generating, 5-7
Partial clock cycle, 3-12
Patch
  files, 2-8
  space, 6-1
PATCH module, 3-14
Patching monitors, 6-1
Per-process monitor free core, 3-3
Performing terminal I/O, 4-17
Physical addresses, 2-5
PI
  channels, 3-6
  CYCLE error, 5-8
  status word, 3-7
Pointers, 3-2
  compressed file, 4-23
  DDB, 4-2
  MFD, 4-22
  retrieval, 4-22
Policy CPU, 2-2, 6-4
Position Wait Queue (PWQ), 4-27
Power-fail stack, 5-4
PPN Data Block (PPB), 4-27
Prime RIB, 4-22
Priority Interrupts (PI), 3-6
Process
  context word, 3-7
  Data Block (PDB), 4-2, 4-8
  tables, 3-2
Processing
  errors, 3-13
  UUOs, 4-11
Processor
  interrupts, 5-3
  modes, 3-2, 3-4
Program Counter (PC), 3-4, 5-6
Prototype KDBs, 4-20
Pseudo-instructions, 4-5
PSISER module, 3-15
Public
  mode, 3-4, 3-5
  pages, 3-2
PULSAR module, 4-20
Push-down lists, 3-9
  scheduler, 3-13
  traps, 3-12
PWFPDL stack, 5-4
PXCT instruction, 4-6

-Q-

QBITS table, 4-10
QUESER module, 3-15

-R-

RDA devices, 4-12
Reading monitor sources, 4-5
Real-time module, 3-15
Recovering from errors, 3-12
REFSTR module, 3-14
Registers, 5-6
Reloading automatically, 2-2
Reloads, 2-1

REQTAB table, 4-10
Resetting mapping, 2-7
Resources, 4-10
Restoring accumulators, 4-3
Retrieval Information Block (RIB), 4-22
RH10 interrupts, 3-10
RH20 interrupts, 3-10
RH2PRM module, 4-7
Role switching, 3-5, 6-4
RSX-20F errors, 5-13
RTTRP module, 3-15
Run queues, 4-10
Running
  FILDDT, 2-3
  symbolic FILDDT, 2-4

-S-

S module, 3-15, 4-7, 4-11
Saving symbolic FILDDT, 2-4
SAVnx routines, 3-9
SCAPRM module, 3-15
Scheduler
  ACs, 3-13
  tables, 4-10
SCNSER module, 4-15, 4-16
SCPAR module, 3-15
Sections, 3-3
  DECnet, 3-3
  mapping, 2-5
  pointers, 3-2
  references, 3-4
  tables, 3-2
SEILM routine, 3-12, 5-4
Servicing interrupts, 3-9
SET commands, 4-11
Sharable resources, 4-10
Shutting down systems, 2-2
Skip chain, 3-9
Software
  channels, 4-11
  disk cache, 4-28
Source code, 4-5
Spare RIB, 4-22
Special Pages Table (SPT), 2-5, 3-3
SPT slot, 2-6
Stacks
  error processing, 3-13
  interrupt, 3-9
Starting BOOT, 2-2
Startup modules, 3-14
Status bits
  channels, 4-11
  I/O, 4-2
STOP stopcodes, 5-10
STOPCD macro, 5-9
Stopcodes, 1-2, 2-10, 5-9
Storage Allocation Blocks (SABs), 4-27
Storage allocation Pointer Tables (SPTs), 4-27
Storage Allocation Tables (SATs), 4-23
Structure
  data blocks (STRs), 4-26
  disk, 4-21
Sub-File Directories (SFDs), 4-21
Superclusters, 4-23
Swapped-out pages, 4-10
SWITCH.INI files, 3-3
Switching
  AC blocks, 3-5, 3-9, 4-6
  CPUs, 6-4
  modes, 3-5
  UPTs, 3-10
Symbol definition, 3-15
Symbolic FILDDT, 2-3
Symbols
  monitor, 4-1
  verifying, 2-7
Symmetric Multi-Processing (SMP), 4-6, 6-4
SYSINI module, 3-14, 4-17
SYSPPB table, 4-27
SYSSTR table, 4-26
SYSTAT program, 2-3

-T-

TABSTR table, 4-26
Tape
  controller data block (KDB), 4-2, 4-19
  I/O, 4-19
  label processing, 4-20
  unit data block (TUB), 4-19
Terminal
  chunk pointers, 4-16
  chunks, 4-16
  controlling, 4-17
  DDBs, 4-16, 4-17
  Device Data Blocks, 4-15
  I/O, 4-17
TMPCOR, 3-3
Transfer
  tables, 4-10
  Wait Queue (TWQ), 4-27
Trapping
  page faults, 3-11, 5-4
  UUOs, 3-5
Trapping page faults, 5-3
TSKSER module, 4-12
TTFCOM table, 4-11
TTYINI routine, 4-17
TTYTAB table, 4-17

-U-

UCLJMP table, 4-11
UCLTAB table, 4-11
UFD Data Block (UFB), 4-28
Unit
  Data Blocks (UDBs), 4-2, 4-26
Universal files, 4-7

Unlocking local symbols, 4-7
UNQTAB table, 4-11
Unrestricted device codes, 3-5
UPT locations, 3-6
User
  accumulators, 2-5, 2-6
  ACs, 4-8
  Base Register (UBR), 3-2
  buffers, 4-17
  concealed mode, 3-4
  DDBs, 3-3
  File Directories (UFDs), 4-21
  jobs
    blocking, 3-6
    mapping, 2-6
    switching, 3-10
    verifying, 2-7
  mode, 3-2, 3-4, 3-5
  Process Table (UPT), 2-4, 2-5, 3-2
  public mode, 3-4
USERAC macro, 4-6
Using
  command files, 2-8
  CREF listings, 4-5
  EDDT, 6-2
  FILDDT, 2-3
  SYSTAT, 2-3
USRJDA location, 4-11
UUOCON module, 3-14
UUOERR routine, 4-11
UUOs
  processing, 4-11
  trapping, 3-5
  verification, 3-6
UUOTAB table, 4-11

-V-

Vectored interrupts, 3-8, 3-10
Verifying
  FILDDT mapping, 2-7
  UUOs, 3-6
Virtual
  address mapping, 3-2
  addressing, 2-4, 2-5
  Memory Address (VMA), 5-6
  sections, 3-3

-W-

Writeable pages, 3-2

-X-

XPN: area, 2-3

-Y-

YES word, 4-28

READER'S COMMENTS

Your comments and suggestions help us to improve the quality of our publications.

For which tasks did you use this manual? (Circle your responses.)

(a) Installation (c) Maintenance (e) Training
(b) Operation/use (d) Programming (f) Other (Please specify.) _____

Did the manual meet your needs? Yes ☐ No ☐ Why? _____

_____

Please rate the manual in the following categories. (Circle your responses.)

	Excellent	Good	Fair	Poor	Unacceptable
Accuracy (product works as described)	5	4	3	2	1
Clarity (easy to understand)	5	4	3	2	1
Completeness (enough information)	5	4	3	2	1
Organization (structure of subject matter)	5	4	3	2	1
Table of Contents, Index (ability to find topic)	5	4	3	2	1
Illustrations, examples (useful)	5	4	3	2	1
Overall ease of use	5	4	3	2	1
Page Layout (easy to find information)	5	4	3	2	1
Print Quality (easy to read)	5	4	3	2	1

What things did you like *most* about this manual? _____

_____

_____

What things did you like *least* about this manual? _____

_____

_____

Please list and describe any errors you found in the manual.

Page Description/Location of Error

_____ _____
_____ _____
_____ _____
_____ _____

Additional comments or suggestions for improving this manual: _____

_____

_____

Name _____ Job Title _____
Street _____ Company _____
City _____ Department _____
State/Country _____ Telephone Number _____
Postal (ZIP) Code _____ Date _____

Fold Here and Tape

Affix
Stamp
Here

DIGITAL EQUIPMENT CORPORATION
CORPORATE USER PUBLICATIONS
200 FOREST STREET MRO1-3/L12
MARLBOROUGH, MA 01752-9101

Fold Here