Advanced Computer Design

PDQ-3 System User's Manual

PDQ-3 SYSTEM USER'S MANUAL

VERSION 3.1

April 1981

Advanced Computer Design

Author: Rich Gleaves

Document: 0.3-S

TABLE OF CONTENTS

II OPERATING SYSTEM (continued)

I. INTRODUCTION

1.0 Scope of this Manual

This is the reference manual for the UCSD Pascal system, version III.1. Users are assumed to be familiar with the UCSD Pascal system; if this is not the case, the following book is recommended:

Beginner's Guide for the UCSD Pascal System
Kenneth L. Bowles
Byte Books (McGraw-Hill), Peterborough, New Hampshire, 1979.

Other documents related to the PDQ-3 Computer System include:

PDQ-3 Hardware User's Manual
Describes the physical characteristics of the computer.

PDQ-3 Programmer's Manual
Describes the PDQ-3 Pascal language implementation.

PDQ-3 Architecture Guide
Provides details of the system software to experienced programmers.

NOTE - The following conventions are used throughout this manual to facilitate the description of various system concepts.

Angle brackets ("<" and ">") are used to indicate metasymbols; these are generic names of symbols. Optional items are delimited by square brackets ("[" and "]"). Some examples of metasymbols and optional items follow:

Typing <cr> completes the input prompt.

President <surname> should be [<expletive>] impeached!

The syntax for Pascal's IF statement is:

IF <Boolean expression> THEN <statement> [ELSE <statement>];

DISCLAIMER - much of the software provided with this system is maintained and controlled by Western Digital, Inc., the makers of the MicroEngine; because of this, Advanced Computer Design cannot guarantee its correctness. Bugs that we are aware of are documented in the appropriate sections of this manual.

1.1 System Organization

The PDQ-3 system software is a superset of the UCSD Pascal system, which was designed as an interactive, single-user system for program development and execution. The system has been extended with multi-processing capabilities and an asynchronous I/O system to allow the development of multi-user and real-time applications. The minimal hardware configuration required to use the system is a CRT terminal and a mass storage device (typically one or more floppy disk drives).

The system consists of the following parts:

Operating System - Provides an interactive command interpreter to control the rest of the system, and run-time support for the execution of Pascal programs.

Command File Interpreter - Automates repetitive tasks by feeding the system a predefined sequence of system commands to execute.

File Handler - Provides disk file management.

Editor - A screen-oriented editor used to create and maintain source files containing Pascal programs. It also provides text editing features for basic word processing tasks.

Pascal Compiler - A fast, one-pass compiler which can produce either executable Pascal programs or library routines.

Linker - Combines Pascal programs and separately compiled library routines into executable programs.

Monitor - Allows the user to examine and modify the contents of memory.

Printer Spooler - A utility program which allows text file printing to proceed concurrently with normal system operation.

Utility Programs - Various programs which aid program development.

1.2 Command and Data Overview

This section describes the various operations performed with the PDQ-3 system; these include action commands which invoke system parts, and data prompts which supply input to the system parts.

1.2.0 Promptlines

Promptlines are a commonly used method of displaying the commands available to the user in various parts of the system. Here are some examples of promptlines found in the system:

Command: E(dit, R(un, F(ile, C(omp, L(ink, S(ubmit, X(ecute

Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans

Responses consist of a single character; a carriage return is not required to complete the command. Command characters are capitalized and separated from the command abbreviation with a left parenthesis. Promptlines displaying alphabetic character commands accept both lower and upper case characters. With some promptlines, typing a "?" redisplays the promptline with a different set of commands. This is done to accommodate wide promptlines on narrow screens. Promptlines are usually referred to as "prompts"; thus, the promptline for the operating system is called the "system prompt", and for the file handler, the "filer prompt".

Many system parts display their current version number in their promptlines; it is usually delimited by square brackets.

1.2.1 File Names

Software development on the UCSD Pascal system largely consists of manipulating files; hence, file name prompts appear rather frequently. Because of this, users who understand the file system find the system easier to use, as many aspects of the file naming conventions involve simplifying the specification of a file name; it is therefore worthwhile to study chapter 2 (section 2.1 - the file system) and the sections describing file name prompts for the various system parts.

1.2.2 Data Prompts

Data prompts are used to obtain input data needed by the system parts. They usually appear in the form of questions; for instance:

```
Compile what file?

Are you sure you want to crunch DISK1: ?

Bad blocks scan for how many blocks?
```

Responses to data prompts usually come in one of two forms: the single character response to a "yes/no" question (such as the second example), and the input data response requiring a string of input characters followed by a carriage return.

An affirmative response to a "yes/no" question is indicated by typing "y" or "Y". Negative responses generally are indicated by typing "N" or "n"; however, some system parts (such as the filer) interpret any characters other than the affirmative ones as a negative response.

Input data responses are usually file names, but can be other items such as the current date or an integer value. These responses almost always require a carriage return to be typed after the input data. The backspace key erases mistakes in the typed input, and the rubout (or delete) character deletes all of the typed input.

Most system prompts requiring input data recognize "escape" inputs that cause the initial system command to abort. For instance, typing only a carriage return after the compiler prompt:

```
Compile what file?
```

... aborts the compiler and returns control to the system prompt. An immediate carriage return is generally accepted throughout the system as an escape; however, in some cases a carriage return has another meaning, so a different method of escape is required. These exceptions are described in the appropriate sections of this manual.

1.3 Key Commands

This section describes some Key commands used throughout the system. Key command definitions are described in section 9.3 (terminal configuration). Key command definitions for some common terminals are listed in Appendix F.

1.3.0 Accept and Escape

Two Key commands are used for terminating input data and commands: the accept Key and the escape Key. Accept is used in the editor; it is denoted in this manual by the metasymbols <accept> and <etx>. Escape is used throughout the system to abort commands; it is denoted by the metasymbols <escape> and <esc>. Key command usage is described in appropriate sections of the manual.

1.3.1 Console End of File

The "end of file" Key is used to terminate character sequences read from the Keyboard by a program or system part which uses the console as an input file; it is denoted by the metasymbol <eof>. See section 2.1 and the Programmer's Manual for more details.

1.3.2 Cursor Movement

Some system parts depend on the user's ability to move the cursor across the screen. Cursor movement is performed with the terminal's space bar (denoted as <space>), backspace Key (denoted as <backspace> or <bs>), and the vector Keys (i.e., <left>, <right>, <up>, and <down> Keys).

1.3.3 User Interrupt Commands

Most Key commands are synchronous with respect to system operation; i.e., they are not executed until the system reads them after issuing an input prompt. User interrupt commands, on the other hand, are executed immediately after being typed. This section describes the user interrupt commands.

1.3.3.0 Monitor Trap

The monitor Key interrupts the currently executing user or system program and passes control to the system monitor (described in chapter 8); program execution may be resumed from the monitor. The monitor Key is defined to be <control-P>.

1.3.3.1 Stop and Start

The stop and start keys suspend and resume console output. Once console output is suspended with the stop key, typing any key other than the start key "single-steps" the output; specifically, it allows one character to be written to the screen before resuspending output. The stop key is defined to be <control-S>. The start key is defined to be both <control-S> and <control-Q>.

1.3.3.2 Console Output Flush

The flush key causes the system to discard all console output until a subsequent console read operation is completed. Flushing is disabled by retyping the flush key. A practical example of the flush command is the interruption of the filer command T(ransfer when it is transferring text files to the console. Typing the flush key causes the I/O system to discard all characters written to the console, thus speeding up the transfer. When the transfer is complete, the filer attempts to restore its promptline; however, screen output is still being flushed, so it doesn't appear. Typing the space bar causes the prompt to reappear; normal system operation is then resumed. The flush key is defined to be <control-F>.

1.3.3.3 Keyboard Type-ahead Flush

The keyboard type-ahead flush key removes all characters queued in the type-ahead buffer; it is defined to be <control-X>.

The type-ahead buffer is used to hold keyboard input that is entered ahead of an input prompt. Input prompts always read characters queued in the type-ahead buffer before reading input from the keyboard. The type-ahead buffer is filled in one of two ways:

1) By typing keys when the system is not waiting for an input response. The input is queued in the type-ahead buffer.

2) By the command file interpreter, as it queues commands and data for future execution.

The type-ahead buffer holds a maximum of 64 characters. When it is full, subsequent keyboard input is not queued; instead, the system rings the terminal bell.

1.3.3.4 Disk Type

The disk type key allows on-the-fly alteration of the software controlling the floppy disk drives. Users can specify whether a drive reads single-sided, double-sided, DEC format, or Western Digital format disks. Users can also control the generation of floppy disk error messages (see section 2.0.2).

NOTE - Double-sided floppy disks require double-sided disk drives. The drives supplied with the standard PDQ-3 do not support double-sided floppy disks.

NOTE - Switching between single and double density floppy disks is performed automatically by the system.

When the system is started, all disk drives are configured for single-sided PDQ-3 format floppy disks, with error messages disabled. Drives are reconfigured by typing <control-D>, followed by the two character sequence:

<drive number><command>

where

<drive number> ::= "0" or "1" or "2" or "3"
<command> ::= "s" or "S" for single-sided disks
"d" or "D" for double-sided disks
"f" or "F" for Western Digital format ("flipped") disks
"i" or "I" for DEC format ("interleaved") disks
"n" or "N" enables floppy disk error messages ("noisy")

NOTE - The "f", "i", and "n" commands are toggles; i.e., they switch the current state to its opposite.

NOTE - The Mapper utility (section 9.0.2) performs explicit remapping of floppy disks between PDQ, WD, and DEC formats. This capability may seem redundant in light of the disk type key's ability to read all of these disk formats; however, disk accesses to WD and DEC disks are considerably slower than disk accesses to PDQ disks because of the translation which takes place in the disk drivers. Thus, while the disk type key is useful for occasional communications with WD and DEC disks, it is more efficient in the long run to remap frequently-used disks than to disk-type them every time they are used.

II. THE OPERATING SYSTEM

The operating system initiates the execution of other system parts and user programs, implements the file system and I/O subsystems, reports hardware and software errors, and provides runtime support for Pascal programs.

Section 2.0 describes the actions performed in response to various kinds of system errors. Section 2.1 describes the file system, which includes file naming conventions and the I/O device organization. System commands and operation are described in section 2.2. Details on the Pascal runtime support routines are contained in the Programmer's Manual.

2.0 Error Handling

This section describes the system's response to hardware or software errors. Execution errors are caused either by incorrect programs or explicit interruption of programs; they are described in section 2.0.0. Stack overflows occur when a program uses up all available system memory, and are described in section 2.0.1. Error messages generated by the floppy disk drives are described in section 2.0.2. The effects of removing disk volumes during system operation (known as "disk swapping") are described in section 2.0.3.

2.0.0 Execution Errors

When an execution error is detected during program execution, the program is suspended, and the operating system prints a diagnostic message on the console. The message consists of a description of the error and the location in the program code where the error occurred.

The error description is usually a textual message; e.g., "Invalid Index". Occasionally, the operating system is unable to obtain the message; in these cases, only the execution error number is printed. A table of execution error numbers and their corresponding messages is displayed in Appendix B.

When the execution error is a user I/O error, a description of the I/O error is printed adjacent to the execution error message; as with execution errors, the unavailability of I/O error messages causes the I/O error number to be printed. A table of I/O error numbers and their corresponding messages is displayed in Appendix A.

The error location is specified in terms of the code file structure; the displayed "S", "P", and "I" offsets represent the code segment number, procedure number within the segment, and procedure-relative byte offset of the instruction causing the error. This information is used in conjunction with a source program listing to pinpoint the error in the source program. Program listings are described in the Programmer's Manual. Segment and procedure numbers are described in the Architecture Guide.

Once an execution error has occurred, the user has two choices available. "Typing <space> to continue", as is prompted on the console, aborts the currently executing program and reinitializes the system. Typing the escape key causes the system to resume execution of the program, the results of which are somewhat unpredictable and dependent upon the nature of the execution error.

2.0.1 Stack Overflow

Stack overflows occur when a program's code and data use up all of the memory in the system; the program is terminated, and the following message appears on the screen:

STK OFLOW

The system then reinitializes itself and redisplays the system prompt.

NOTE - Stack overflows are not always detected by the processor or operating system; when this happens, the system stops without printing any error messages, and must be rebooted. In other cases, the system halts after displaying the stack overflow message. See the Architecture Guide and Programmer's Manual for more information.

2.0.2 Floppy Disk Errors

The software controlling the floppy disk drives can be directed to issue error messages to the console whenever the hardware indicates that a disk operation caused a transient error (see section 1.3.3.4). This section describes the format of floppy disk error messages.

NOTE - this section contains references to the hardware interface of the PDQ-3 disk controller. See the Hardware User's Manual for details.

Here is an example of a disk error message (which fits on one line when displayed on the console) and a description of its format:

```
Flop_42 [01] 01 Fc-94 Fs-30 T-01 S-19 Dc-01 Ds-01
                                    C-0000 A-0012F8 Vs-001A
```

- 42 - High order byte of the disk select register. Low nibble is the disk number (1,2,4,8). High order nibble is density (4=single).
- [01] - The retry number. It indicates the number of times the operation has been attempted without success.
- 01 - The system I/O result indicating the error condition (see Appendix A).
- Fc - The command that was issued to the FDC when the failure occurred.
- Fs - The FDC status register indicating the error condition.
- T - The FDC track register.
- S - The FDC sector register.
- Dc - The DMA command register.
- Ds - The DMA status register.
- C - The DMA count register (negative number of bytes left in the current I/O operation).
- A - The DMA address register (a byte address).
- Vs - The starting virtual sector (a zero-based logical sector number).

2.0.3 Disk Swapping

This section describes the effects of removing disk volumes from the floppy drives during system operation. Floppy disks are often exchanged during system operation in order to retrieve files from offline volumes, or to copy disk volumes onto backup disks; the system accommodates this by keeping track of the online disk volumes. However, disk swapping during program execution can be hazardous; if a system or user program requires a code segment from a disk volume, and the disk volume is no longer mounted in its original drive, the system crashes.

The system attempts to remedy this situation in a couple of ways.

First, the file handler and disk-copying utility programs do not contain segment procedures; their code remains resident in memory at all times during execution. User programs must do the same in order to survive random disk swapping.

Second, the operating system attempts to protect itself from crashes caused by removing the system disk during program execution. Normally, if the system disk is removed or replaced, it must be remounted in the proper drive before the program terminates; in fact, many of the utility programs issue explicit prompts to remount the system disk before terminating. However, if the system determines that the system volume has been removed or replaced, the following message appears after the program terminates:

Replace <system volume name>:

The system waits until the proper disk is remounted, and then redisplays the system prompt as if nothing unusual had occurred. The method used to detect a disk swap is to monitor all disk directory accesses during program execution; if a directory access is not performed on the system's disk drive after the disk has been swapped, program termination halts the system with an unrecoverable execution error instead of displaying the prompt shown above.

2.1 File System

2.1.0 Overview

In the most abstract sense, a file is merely a sequence of data. A file system exists in order to adapt this abstract definition of a file to the requirements and constraints of a given hardware and software environment. The file system described herein has the following outstanding characteristics:

1) Files can be accessed from Pascal programs with standard Pascal file operators.

2) Files possess types to aid the user in identifying the contents of files and to increase system reliability by preventing invalid operations on files.

3) The file system implements high level concepts such as removable disk volumes and device-independent file I/O.

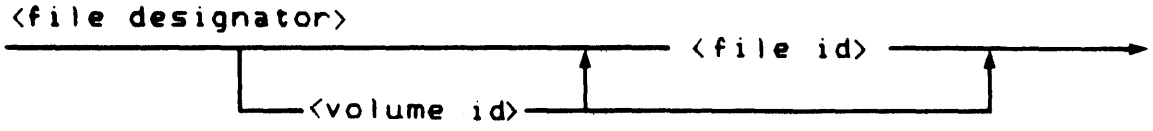
4) The disk file implementation is both time and space-efficient on relatively low performance floppy disk drives.

The following sections comprise a complete user-oriented specification of the file system. Section 2.1.1 presents an overview of file name syntax. Sections 2.1.2 through 2.1.4 describe the syntax and semantics of the file system hierarchy, starting with the lowest levels of device I/O and culminating with file attributes. Section 2.1.5 contains the definitive syntax specification of a file name. Section 2.1.6 describes some system-wide conventions that apply to the file system.

References to file naming conventions and file system terminology throughout this manual (and the Programmer's Manual) refer either implicitly or explicitly to the information presented in this section.

NOTE - In order to present a consistent file system description, this section defines a number of terms intended to describe parts of the file system. New terms are underlined and followed by either an immediate definition or a reference to a defining section; subsequent occurrences of the defined term are not underlined.

2.1.1 Syntax Overview



A valid file designator (informally referred to as file name) consists of a volume identifier and a file identifier. Volume identifiers are described in section 2.1.3. File identifiers are described in section 2.1.4. The complete syntax for a file designator is presented in section 2.1.5.

2.1.2 Physical Units

Physical units correspond to I/O devices; they are addressed by their assigned physical unit number. I/O devices are defined to be either serial devices or block-structured devices (described in section 2.1.2.1). A serial unit is a physical unit assigned to a serial device. A block-structured unit (informally referred to as a disk unit) is a physical unit assigned to a block-structured device.

All physical units may be used as files.

NOTE - Appendix C contains a complete description of the PDQ-3 Computer System's standard device assignments.

Unit Number	device description	unit attribute
1	screen and Keyboard with echo	serial
2	screen and Keyboard without echo	serial
3	graphics	unused
4	disk drive 0	block-structured
5	disk drive 1	block-structured
6	printer	serial
7	remote input	unused
8	remote output	serial
9 - 12	disks 2 - 5	block-structured
13	remote port 0 input	serial
14	remote port 0 output	serial
15	remote port 1 input	serial
16	remote port 1 output	serial
17	remote port 2 input	serial
18	remote port 2 output	serial
19	remote port 3 input	serial
20	remote port 3 output	serial

2.1.2.0 Syntax Overview

The metasymbol <number> may be any positive integer representing a unit number.

2.1.2.1 I/O Devices

I/O devices assumed to be connected to the system include disks, terminals, printers, and remote ports. An I/O device is in one of two states: online or offline. A device is online if it acknowledges status requests from the system and is available for I/O operations.

2.1.2.1.0 Serial Devices

A serial device is defined to either produce or consume a sequence of data. Serial devices assumed to be used with the system are terminals, printers, and remote ports. The software controlling these devices makes some assumptions about the structure of the data sequences handled; in particular, default I/O to serial devices expects human-readable data known as text files. Section 2.1.4.1.0.1.0 provides an overview of text files. Details concerning alternate modes of serial I/O can be found in the Programmer's Manual and Architecture Guide.

2.1.2.1.1 Block-structured Devices

A block-structured device is organized into a fixed number of 512 byte storage areas known as blocks. Blocks are randomly accessible by block number. These devices are usually implemented as fixed or removable disks.

NOTE - Large-capacity (e.g. hard) disks are often partitioned into a number of logical disk devices.

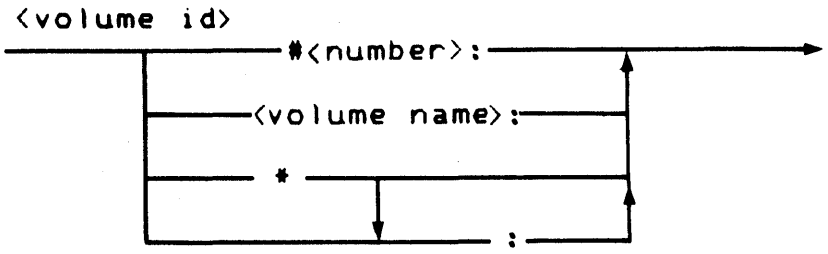
2.1.3 Logical Volumes

Logical volumes correspond to physical units; they are addressed by their assigned volume name (described in section 2.1.5). A serial volume is a logical volume assigned to a serial unit. A block-structured volume is a logical volume assigned to a block-structured unit. Serial volume name assignments are permanent and may not be changed by the user; serial volumes are functionally equivalent to their assigned serial units. Volume name assignments to block-structured units are dynamic and controlled by the user; a block-structured volume is addressable if and only if it resides on an online block-structured unit. Block-structured volumes are described in section 2.1.3.1.

All serial volumes may be used as files. Block-structured volumes should never be addressed as files except when using the file handler to create, examine, and copy entire block-structured volumes.

Volume Name	Assigned Phys. Unit	volume attribute
CONSOLE:	1	serial
SYSTERM:	2	serial
GRAPHIC:	3	unused
<vol name>	4	block-structured
<vol name>	5	block-structured
PRINTER:	6	serial
REMIN:	7	serial
REMOUT:	8	serial
<vol names>	9 - 12	block-structured
REMIN1:	13	serial
REMOUT1:	14	serial
REMIN2:	15	serial
REMOUT2:	16	serial
REMIN3:	17	serial
REMOUT3:	18	serial
REMIN4:	19	serial
REMOUT4:	20	serial

2.1.3.0 Syntax Overview



The volume identifier may either be the system volume "*" (section 2.1.3.3), a unit number, or a volume name. File designators containing either empty volume identifiers or ":" specify the prefixed volume, which is described in section 2.1.3.4.

2.1.3.1 Block-structured (Disk) Volumes

Block-structured volumes (informally referred to as disk volumes) correspond to mass storage devices; the typical case is a floppy disk. A disk volume contains a collection of disk files (described in section 2.1.4). Information describing the files is centralized in a reserved area of the disk known as the disk directory (described in section 2.1.3.5). A disk directory contains the volume name which identifies the disk volume as a whole. A disk volume is online if it resides on an online disk unit; it is addressed by its volume name. Disk volumes may also be addressed by specifying the physical unit containing the disk volume; e.g., a disk volume named "SYSTEM" on unit 4 can be addressed either as "SYSTEM:" or "#4:".

Block-structured units and disk volumes represent two distinct ways of treating disk storage. Disk volumes are implemented on block-structured units; however, they contain a directory and volume name, and are designed to contain a number of disk files. Block-structured units are "bare" disks and have no directory or volume name; they can contain only one file and are addressed by their physical unit number. Section 2.1.4.3.2 describes other differences between disk volumes and block-structured units.

Details concerning the implementation of disk directories and disk files may be found in the Architecture Guide.

2.1.3.2 Disk Volume Usage

Because disk volumes may be referenced by volume name, the system has problems operating when two disk volumes with the same volume name are online. This situation should be avoided as much as possible. When it can't be, all file designators must avoid using volume names as volume identifiers; instead, the physical unit numbers must be used to unambiguously specify files on online volumes.

Disk volume names should always be used in conjunction with a file identifier specifying a disk file on the volume. The only exceptions occur when using the file handler to create, examine, and copy entire disk volumes. Using a disk volume name as a file exposes the volume's disk directory to accidental overwriting by file write operations, thus threatening access to the volume's disk files.

2.1.3.3 System Volumes

The system volume is the disk volume containing the operating system code file; usually, it also contains the code files for the rest of the system parts. The system volume may be specified independently of its assigned volume name by using the volume identifiers "*" or "*:".

2.1.3.4 Prefixed Volumes

Prefixed volumes are used in conjunction with disk file designators. Normally, a disk file designator includes a volume identifier to indicate the volume on which the disk file resides in addition to the disk file identifier itself. Disk file designators lacking a volume identifier are assumed to reside on the prefixed volume; thus, file naming can be simplified by specifying the most frequently accessed disk volume as the prefixed volume. The entire prefixed volume can be addressed with the file designator ":".

The default prefixed volume is the system volume. The P(refix command (in the file handler) is used to specify volumes as the prefixed volume; it designates a volume identifier entered by the user as the prefixed volume name. If the volume identifier matches the name of an online volume, the volume becomes the prefixed volume. The volume identifier can also specify an offline disk volume; when the volume comes online, it becomes the prefixed volume. If the volume identifier specifies a disk unit (as opposed to a volume name), whichever disk volume is mounted in the specified unit becomes the prefixed volume.

Setting the prefixed volume to a serial volume or unit is fruitless, as these devices neither recognize file identifiers nor contain directories.

2.1.3.5 Disk Directories

Disk directories are stored on a disk volume along with disk files. Directories contain the volume name and up to 77 directory entries. A directory entry contains the name, location, and attributes of a disk file on the volume. The file names in a directory must be unique in order to specify a file unambiguously; an existing file is automatically deleted if another file with the same name is entered in the directory. Disk file names are described in section 2.1.4. For more information concerning multiple files with the same name, consult the Programmer's Manual for a description of file operators.

NOTE - When the file system attempts to add a file to a volume containing a full directory, it prints the error message:

    No room on vol

This is somewhat misleading, as the same message is used to indicate a lack of disk space.

2.1.3.5.0 Duplicate Directories

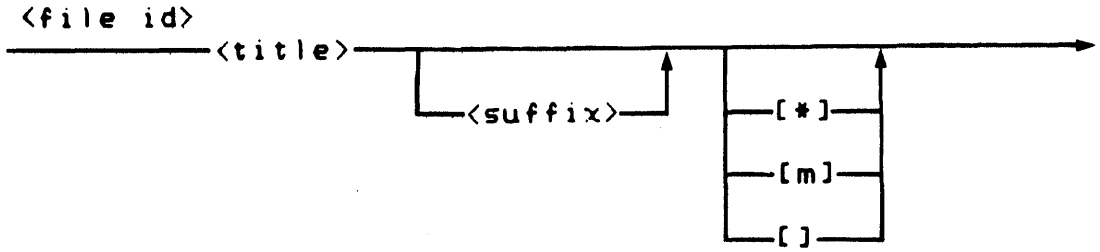
A disk volume may be marked so that the system maintains two disk directories on a disk volume; the second directory is called a duplicate directory and exists as a copy of the main directory. If unforeseen circumstances cause the destruction of the main directory, it can be restored using the information in the backup

directory. The only cost of duplicate directory usage is a slight increase in overhead due to the necessity of updating an extra disk directory during file manipulation. The insurance provided generally outweighs any losses in performance. The utility programs Markdupdir and Copydupdir are used to create duplicate directories and restore deceased main directories (see section 9.1).

2.1.4 Disk Files

Disk files are stored in an integral number of contiguous blocks on a disk and contain either programs or data. File attributes provide useful information about the structure and history of a disk file; they are described in section 2.1.4.1. File names are the most important attribute of a disk file; they uniquely identify a disk file within a directory. File names are described in sections 2.1.4.2 and 2.1.4.3. File length directives control the amount of disk space allocated to a disk file; they are described in section 2.1.4.4.

2.1.4.0 Syntax Overview



File titles distinguish the files in a directory; they are described in section 2.1.4.3. File suffixes allow the system and user to determine the contents of a disk file; they are closely related to file types. File suffixes are described in section 2.1.4.2. The syntactic items delimited by square brackets are length specifiers. Length specifiers serve as directives to the file system to determine the amount of disk space to allocate to a newly created disk file; they are described in section 2.1.4.4.

2.1.4.1 File Attributes

Disk files attributes are used by the system to manipulate the file and by the user to determine the contents and history of the file. From the user's point of view, the prominent file attributes are file type and file date. File types are described in section 2.1.4.1.0. File dates are described in section 2.1.4.1.1. The remaining file attributes visible to the user are file length, starting block, and bytes-in-last-block; these are described in section 2.1.4.1.2.

2.1.4.1.0 File Type

All disk files have an attribute called the file type. File types enable both system and user to determine the contents of a disk file, regardless of its file name. Text file and code file are file types used by the system; files of these types are described in section 2.1.4.1.0.1. Files not containing text or code are assigned the type data file; these are described in section

2.1.4.1.0.2. System restrictions imposed by file types are described in section 2.1.4.1.0.3.

2.1.4.1.0.0 File Type Assignment

When a file is created, the system assigns a file type corresponding to the suffix; subsequent file name changes do not affect the assigned file type. Section 3.5 describes a somewhat underhanded method of changing the type of a file.

2.1.4.1.0.1 UCSD Pascal Files

The two file types described in this section are used to identify files containing specific internal structures; the structures are required (and assumed to be present and correct) by the system parts that operate on typed files. The internal structures of the file types are described in the Architecture Guide.

2.1.4.1.0.1.0 Text Files

Text files are usually created and maintained by the editor; they can also be created by user programs. Text files contain human-readable text that represents either program source files, program data, or written documents suitable for word processing. Serial devices used to display data for human scrutiny (e.g., consoles and printers) recognize text file conventions on output; thus, text files written to serial units or volumes appear as they do in the editor.

2.1.4.1.0.1.1 Code Files

Code files are created by the compiler and manipulated by the linker and the operating system. Code files contain a mixture of P-code and execution information used by the CPU and operating system. Code files may need to be linked before they are executable; when used elsewhere in this manual, the term linked code file refers either to code files not requiring linking to execute or code files that have been linked with the linker.

Attempts to edit a code file with the editor or display a code file on the printer or console will fail; the system misinterprets the code file format as text file information and spews forth a melange of audio/visual garbage for your entertainment. Code files are best examined and modified with the Patch utility program described in chapter 9.

2.1.4.1.0.2 Data Files

Data files are created by programs using files containing data other than text and can have any internal representation. Except for being restricted to lie within an integral number of disk

blocks, data files have no defined internal structure whatsoever; they match the Pascal language's definition of a file as a sequence of arbitrarily structured items.

2.1.4.1.0.3 System Restrictions Imposed by File Types

The operating system does not accept files other than code files for execution, regardless of the file identifier. A weaker form of type checking is performed in some system parts (e.g., the editor) by using the current suffix of a disk file name to guess its file type. This method of checking is sufficient for all practical purposes; however, it can be subverted by changing the suffix of an existing file name or using the file prompt conventions described in section 2.1.6.0.

2.1.4.1.1 File Date

The current system date is assigned to a file when it is created or modified (where "modified" is defined as the replacement of an old file by a new file of the same name). Section 3.5 describes a somewhat underhanded method of changing the file date.

2.1.4.1.2 Size and Location Attributes

The length field indicates the number of blocks allocated to a disk file. The starting block field indicates the absolute block number of the first block of the disk file (block 0 is the first absolute disk block). The bytes-in-last-block field indicates the number of bytes in the last block of the file. This field is always set to 512 for text and code files, because they are created with block-oriented file operators; only data files have interesting values in this field.

2.1.4.2 File Suffixes

File suffixes are separated from file titles by a period. File suffixes treated specially by the system are shown in the following table. Files created with these suffixes are assigned the corresponding file type; otherwise, the file is designated a data file.

Suffix	File Type	System Uses
.TEXT	text file	text file identifier
.CODE	code file	code file identifier
.BACK	text file	editor backup text file
.BAD	data file	damaged area of disk

2.1.4.3 File Titles

File titles uniquely identify disk files within a directory. The system reserves some titles for its own use; these are called system titles. All other valid file titles are user titles.

2.1.4.3.0 System File Titles

System files contain code and data used for system operation; they are identified by the file title "SYSTEM.<system part name>". The following table shows all system file titles and their contents:

System File Title	File Type	Contents
SYSTEM.COMPILER	code	compiler
SYSTEM.ASSMBLER	code	assembler
SYSTEM.EDITOR	code	editor
SYSTEM.FILER	code	file handler
SYSTEM.LIBRARY	code	contains user library routines
SYSTEM.LINKER	code	code file linker
SYSTEM.LST.TEXT	text	default program listing file
SYSTEM.MISCINFO	data	terminal configuration info
SYSTEM.PASCAL	code	operating system
SYSTEM.STARTUP	code	user-defined bootstrap program
SYSTEM.SWAPDISK	data	memory swapped while compiling
SYSTEM.SYNTAX	data	compiler syntax error text
SYSTEM.WRK.TEXT	text	work text file
SYSTEM.WRK.CODE	code	work code file

All code files except for the operating system, compiler, assembler, and library are executable code files and can be invoked from the system prompt with the X(ecute command (see section 2.1.6). SYSTEM.MISCINFO may be examined and modified with the Setup utility (section 9.3.1). Users may add their own library routines to SYSTEM.LIBRARY using the Library utility (section 9.2.0).

SYSTEM.STARTUP is a user-defined program which the system executes during the system bootstrap before displaying the welcome message

or system prompt. It is used for turnkey applications programs which do not require the system.

While bootstrapping, the system searches for SYSTEM.MISCINFO and SYSTEM.PASCAL only on the system volume. To locate the other system parts, the system searches the system volume and then all other online disk units (ordered by increasing unit numbers) for a disk volume containing the system titles.

Work files (SYSTEM.WRK.TEXT and SYSTEM.WRK.CODE) exist to speed up interactive program development; various system parts are automatically invoked when a work file exists. Work files are described in section 2.2.1.

SYSTEM.SWAPDISK is used by the compiler to save memory during the compilation of large programs. If the following conditions hold:

1) A 4-block file named SYSTEM.SWAPDISK resides on the same volume as SYSTEM.COMPILER.

2) A disk directory must be read onto the heap in order to open a file.

3) There is insufficient memory to read the directory, but the heap is larger than 4K bytes.

... then the operating system swaps a section of heap data out to the file SYSTEM.SWAPDISK, read the directory into the resulting section of memory, open the file, and swap the heap data back into memory. See section 9.2.2 for more information.

The default program listing file SYSTEM.LST.TEXT is described in the Programmer's Manual.

2.1.4.3.1 User File Titles

User files may have any valid file title other than the reserved system file titles.

2.1.4.3.2 File Titles with Non-block-structured Volumes

This section describes the consequences of creating files with semantically ambiguous designators; i.e., file names pairing a non-empty file identifier with a volume identifier specifying a non-block-structured volume. When a file identifier is appended to a serial volume name, it is ignored; the file designator is treated as a serial volume identifier. When a file identifier is appended to a disk unit number, the disk unit is assumed to contain a disk volume with its associated directory; block-structured units lacking directories generate the file system error:

No directory on volume

The reason for this apparent discrepancy in behavior is to make

disk file I/O transparent to serial volumes and disk volumes (e.g., the directing of a listing file to either a disk file or a printer). It should be emphasized here that direct file I/O to a block-structured disk unit is only done in rare circumstances; e.g., when it is deemed necessary to dedicate an entire disk unit to the creation and maintenance of a single large-capacity file.

2.1.4.4 File Length and File Length Specifiers

When a disk file is created and made available for subsequent I/O operations, the file system must determine three things: whether the volume specified has an available directory entry for the new file, how much disk space to allocate for the new file, and whether the required disk space is available on the disk. When the I/O operations are completed, the system releases any disk space that was allocated to but not used by the file; however, while the file is available for I/O, it reserves all of its allocated disk space for growing room.

Files created without a length specifier are allocated the largest free space on the volume in order to minimize the possibility of growing files running out of disk space. This causes problems when a program attempts to create a number of new files on a disk volume having only one free space available; though the number of blocks in the free space might easily contain all of the completed files, the first file created is allocated all available disk space and thus prevents the other files from being created.

File length specifiers change the file system's disk space allocation strategy in order to avoid problems such as the one described above. The value of the length specifier is treated as an *estimate* of the eventual maximum size (in blocks) of the file. The file system then allocates the specified amount of disk space for the file in the first free space large enough to contain it. For example, the file specifier "[10]" allocates 10 blocks of disk space in the first 10-block chunk of free disk space.

The file length specifier "[*]" is useful when creating multiple files on a single disk; it allocates either half of the largest space on the disk or the second largest space, whichever is largest.

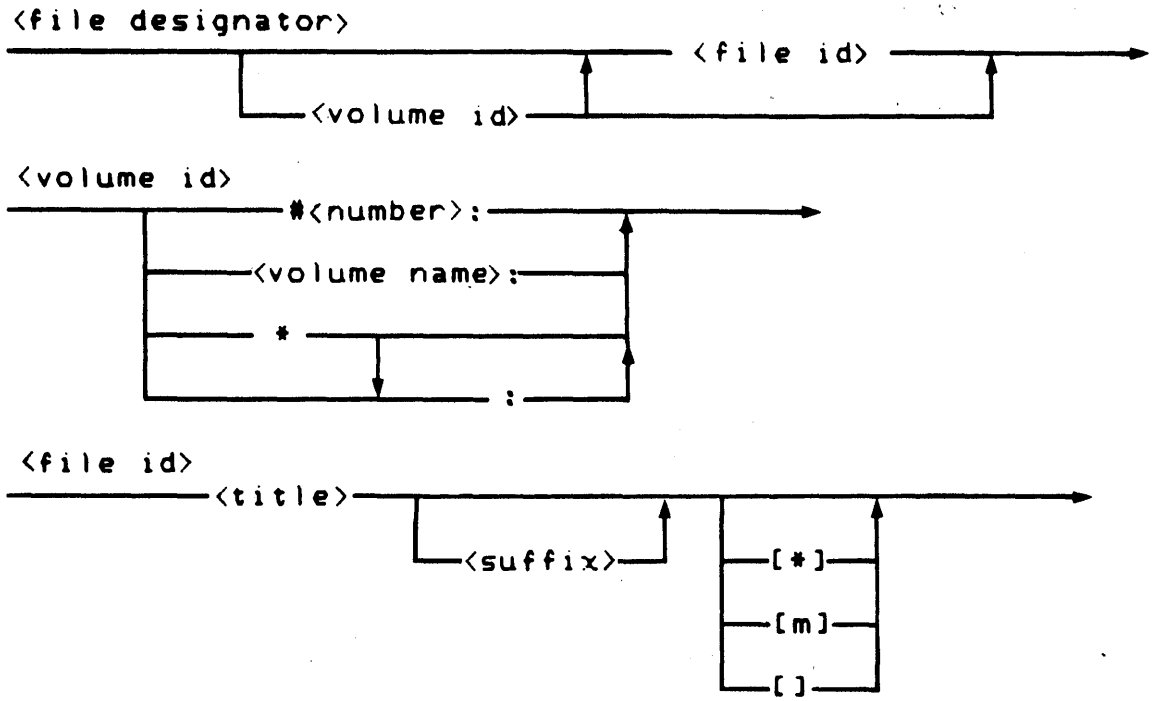
The file length specifiers "[0]" and "[]" are equivalent to a null length specifier; they allocate the largest space available.

If a growing file reaches the end of its initially allocated space, one of two things occurs. If the disk space immediately following the allocated space is used by an existing file, the file system reports a system error; otherwise, the space is part of a free space and the file's allocated disk size is extended into the free space.

Length specifiers may appear in any file designator; however, they are ignored by all file operators other than the file creation operator.

Free spaces are created on disk volumes as a consequence of normal disk file creation and destruction, and the disk file implementation. Disk free space is managed with the Krunch command described in chapter 3.

2.1.5 Syntax Specification



All spaces and control characters are ignored, and all lower case alphabetic characters are mapped into their upper case equivalents. The following characters should not be used in a file designator: "$", "=", "?", and ",". These characters are treated specially by the file handler's file name prompts (see chapter 3 for more details).

The volume identifier may specify a physical unit by its unit number ("#<number>:"), a logical volume by its volume name ("<vol name>:"), the system volume ("*:", "*"), or the prefixed volume (null, ":"). The volume name may contain any printable characters except "#" and ":", and has a maximum length of seven characters.

The file identifier consists of a title followed by an optional suffix and terminated by an optional length specifier. The title and suffix may contain any printable characters except "["; their combined maximum length is fifteen characters. A disk file's directory entry consists of the catenation of title and suffix; this entry must be matched exactly by a file designator's title and suffix in order to locate the disk file.

The file length specifier is delimited by square brackets. The symbol "m" shown as one of the length specifier options denotes a positive integer.

Examples of valid file designators are:

```
*SYSTEM.WRK.CODE[*]
FOON.TEXT
SYSTEM.COMPILER
FLOPPY:SCRUB.BUB.FOTO[10]
:
*
*:
#12:
PRINTER:
DATA
```

2.1.6 File Conventions and Applications

This section describes some system-wide conventions for file name prompts. Programs developed by users should take advantage of these conventions in order to be consistent with the rest of the system.

2.1.6.0 File Name Prompt Conventions

File name prompts accept file names for one of two purposes: locating an existing file to use as an input file, or creating a new file to use as an output file. These operations are implemented with the UCSD Pascal file operators; see the Programmer's Manual for details and examples.

2.1.6.0.0 Input Prompts

Input file prompts appearing in the system are one of two kinds: type checking prompts, and general prompts.

Type checking prompts enforce a weak form of file type checking (see section 2.1.4.1.0) by expecting only the volume identifier and file title for input, appending the input with the suffix corresponding to the desired type, and opening the input file with the resulting file designator. It is assumed that the file suffix is a true indication of the file type; therefore, the file designator should successfully locate the user's input file only if the user's file is of the correct type. Type checking prompts provide a conventionalized "out": a suffix is not appended if the last character in the input is a period (the period is removed). For example, the editor accepts the file name "SYSTEM.SYNTAX." as a valid input text file name identifying the file "SYSTEM.SYNTAX".

General prompts are the more forgiving of the two; they accept any input as a valid file designator and blithely proceed to open the file. If the file system indicates the file was not opened successfully, the proper suffix is appended to the input and the operation is retried. A variation of general prompts is used by the compiler's "include" file mechanism (described in the Programmer's Manual).

2.1.6.0.1 Output Prompts

Output prompts appearing in the system are one of two kinds: good, and bad.

Good prompts expect only the desired file title, catenate the correct file suffix, and create the output file. Example of good prompts include the compiler code file prompt and the editor's output file prompt.

Bad prompts accept any file specification and create the file. Bad prompts have a nasty habit of creating data files (instead of files

with the expected type), because users accustomed to good prompts naively type only a file title as the output file name. Sterling examples of bad prompts exist in the linker and the Library utility program.

2.1.6.1 File Access from User Programs

This section exists solely to stress that all file system features and all file prompt conventions described in the previous section are implemented with the language available to the user; no tricks are involved. This implies that user programs can take full advantage of the file system and prompt conventions for their own prompts.

2.2 Commands and Operation

This section describes the operating system commands and operation. Section 2.2.0 explains how to start the system. Section 2.2.4 describes all commands available in the system prompt. Work files are described in section 2.2.1. The system's state flow is described in section 2.2.3. Automated invocation of system parts is described in sections 2.2.1.1 and 2.2.2.

2.2.0 Starting the System

This section describes the actions taken by the system when it is first started.

The Hardware User's Manual provides instructions for starting the PDQ-3 computer and the operating system. When the system finishes its initialization routines, it displays a welcome message at the center of the screen:

SYSTEM:

12-Apr-81

ACD/VS UCSD Pascal 3.1

The system volume name, current system date, and version are displayed in the welcome message. The system prompt then appears across the top of the screen.

NOTE - The reserved file names PROFILE (chap. 7) and SYSTEM.START-UP (section 2.1.4.3.0) affect the behavior of the system when it is first started.

NOTE - If the welcome message or system prompt seem to be on the wrong part of the screen, consult section 9.3 (terminal configuration).

2.2.1 The Work File

The work file is a special file which is used as a "scratch" or "work" area for the development of programs and documents. It simplifies program development by reducing the number of commands required to edit, compile, link, and execute a program. However, the work file is temporary by nature, and thus susceptible to impromptu removal by certain system actions; therefore, the work file contents can be saved in a named disk file.

Work file operations are described in section 2.2.1.0. The effects of a work file on system operation are described in section 2.2.1.1.

2.2.1.0 Work File Manipulation

The filer commands N(ew, G(et, and S(ave are work file commands. G(et and N(ew create new work files; if a work file already exists, it is removed. N(ew creates an empty work file. G(et creates a work file containing a copy of the contents of a named disk file. S(ave saves the contents of the work file as a named disk file.

The work file consists of two parts: the work text file, and the work code file. The work text file is modified with the editor; the editor command U(pdate saves the results of an edit session as the work text file. The work code file is modified with the compiler or linker; these system parts can be directed to specify their output files as the work code file. The text and code parts of the work file exist separately; thus, the work file may contain a text file, a code file, or both text and code files; in the latter case, the code file is always a direct translation of the current work text file. The work code file is removed whenever the work text file is updated.

When the work file is updated, it is written to a disk file named SYSTEM.WRK. The work text file is named SYSTEM.WRK.TEXT. The work code file is named SYSTEM.WRK.CODE. These files are always written to the system volume.

More information concerning work file manipulation can be found in the sections describing the commands and system parts mentioned in this section.

2.2.1.1 Work File Effects on System Behavior

The editor, compiler, and linker normally request the name of an input file; however, if a suitable work file exists (e.g. work text file for the editor), these system parts proceed automatically using the work file as input.

The system command R(un has the ability to automatically invoke some system parts in order to execute the current work file, regardless of its suitability for execution. The best example of this is to type R(un when only a work text file exists. The system invokes the compiler to compile the source; if the resulting work code file needs linking, the linker is invoked to produce a linked code file. The system then executes the (linked) work code file. All this takes place without requiring the user's attention (though rapturous awe is suggested).

NOTE - typing R(un when no work file exists invokes the compiler, which then prompts for the name of an input file.

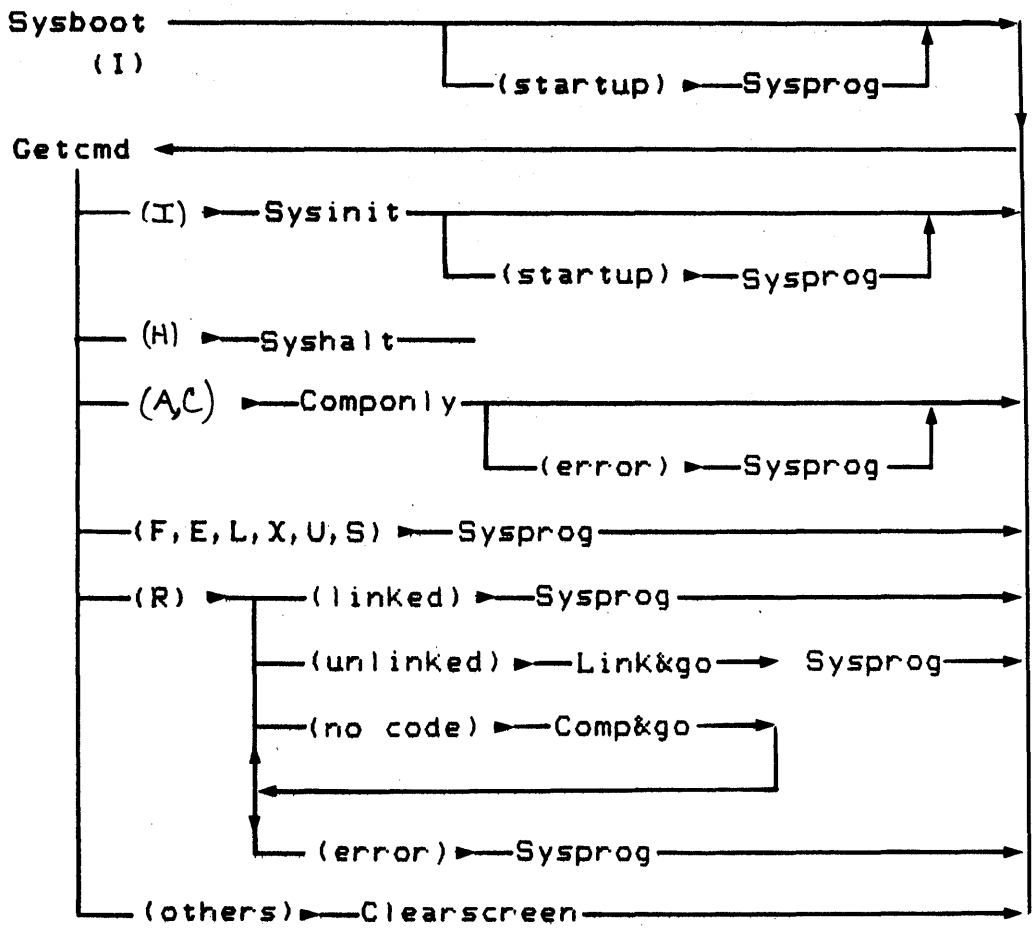
A formal specification of system behavior with respect to work files is presented in section 2.2.3.

2.2.2 Syntax Errors and Editor Invocation

When the compiler detects a syntax error in a source file, the user is given the choice of continuing compilation, aborting compilation, or fixing the error by invoking the editor. If the latter choice is made, the system automatically enters the editor and positions the cursor near the error. If the source file being compiled is not the work file, the editor displays its input file prompt; it is necessary for the user to type the correct file name in order to pinpoint the error in the text.

2.2.3 System State Flow Diagram

This section presents a formal description of all system states along with the actions required to reach them. Words enclosed in parentheses denote conditions that must be satisfied if the ensuing state path is traversed. The list below the diagram contains system action descriptors, system conditions, and definitions relevant to the state diagram. The state flow diagram is on the next page.



Note: "(error) ⟵ Sysprog" sequence invokes the editor.

Descriptor	Definition
Sysboot	system bootstrap
Sysinit	system reinitialization
Syshalt	system halt
Getcmd	system prompt displayed
Clearscreen	console display cleared
Sysprog	system/user program invocation
Componly	invoke compiler or assembler only
Comp&go	invoke compiler and run work file
Link&go	invoke linker and run work file
(startup)	SYSTEM.STARTUP code file on system volume
(<letter>)	system prompt command received
(others)	non-command character received
(linked)	work file is linked code
(unlinked)	work file is unlinked code
(no code)	work file is text only
(error)	compiler syntax error

2.2.4 System Commands

This section describes the commands available from the system prompt. Commands are either completely specified herein or have a partial specification and a reference to another chapter in the manual.

The system promptline has the following form:

```
Command: X(ecute, S(ubmit, R(un, F(ile, E(dit, C(omp,
                                        L(ink, H(alt, ? [3.1]
```

The system's release version is enclosed in brackets at the end of the promptline. Typing "?" displays the remaining commands:

```
Command: A(ssemble, U(ser restart, I(nitialize
```

Typing "?" again returns the original prompt line.

2.2.4.0 Clear Screen

All non-command characters are defined as clear screen commands in the system prompt; typing them clears the screen of all characters and redisplays the system prompt.

2.2.4.1 A(ssemble

Executes the program named SYSTEM.ASSMBLER. Assemblers are not provided with this release. Users may find it convenient to change the name of an oft-used program to SYSTEM.ASSMBLER; it can then be executed by typing "A" from the system prompt.

2.2.4.2 C(ompile

Executes the program named SYSTEM.COMPILER. The compiler translates a Pascal source program into a code file.

If a work text file is present, it is used as the source file; otherwise, the compiler prompts for the source and code file names. Both file prompts expect only the volume and file title to be typed; the file suffixes are automatically appended. The code file prompt has some unique features. Typing <return> updates the work code file with the code file. Typing "$" writes the code file to the work code file and saves it with the same name as the source file.

Compiler operation is described in chapter 5.

2.2.4.3 E(dit

Executes the program named SYSTEM.EDITOR. The editor creates and modifies text files.

If work text file is present, it is used as the input file; otherwise, the editor asks for the name of an input file. Typing <return> enters the editor with an empty file. Typing <escape> aborts the editor.

Editor commands are described in chapter 4.

2.2.4.4 F(ile

Executes the program named SYSTEM.FILER. The file handler is used to manage disk files and disk volumes.

NOTE - Once the filer prompt appears, the system disk can be removed or replaced with another disk volume; however, it must be remounted before leaving the filer.

Filer commands are described in chapter 3.

2.2.4.5 H(alt

Stops the system. The only way to restart the system is to reboot (see chapter 8 and section 2.2.0).

2.2.4.6 I(nitialize

Causes the system to reinitialize all of its state information. This involves initialization of all online I/O devices and system data structures. System programs are searched for and located on online disk volumes. If the code file SYSTEM.STARTUP exists on the system volume, it is executed before the system prompt appears. SYSTEM.STARTUP is described in section 2.1.4.3.0.

All non-fatal execution errors (see Appendix B) cause the system to automatically invoke the I(nitialize command.

2.2.4.7 L(ink

Executes the program named SYSTEM.LINKER. The linker is used to combine user programs with separately compiled library routines to form executable code files.

Linker operation is described in chapter 6.

2.2.4.8 R(un

Executes the work code file. If the work code file does not exist, the compiler is automatically invoked. The behavior of the R(un command with respect to work files is described in sections 2.2.1.1 and 2.2.3.

2.2.4.9 S(ubmit

Executes the program named X.CODE on the system volume. X.CODE is assumed to contain the command file interpreter program, which is used to control the system's operation with a command file.

Command file specification and operation are described in chapter 7.

2.2.4.10 U(ser restart

Reexecutes the last program. This command cannot restart the compiler or assembler, and does not work if the system has been reinitialized.

2.2.4.11 X(ecute

Executes the specified code file.

X(ecute prompts for a code file name. The file suffix ".CODE" is automatically appended to the file name. The file must be a linked code file (section 2.1.4.1.0.1.1).

III. THE FILE HANDLER

The file handler (referred to as the "filer") manages work files, disk files, disk volumes, and disk media. The file system is closely tied to filer operation, and should be thoroughly understood before using the filer; the file system is described in Chapter 2. Section 3.0 describes the filer's prompting peculiarities. Section 3.1 describes the file naming conventions that apply to filer prompts, and introduces the "wildcard" concept; wildcards allow a single file designator to specify several disk files, and thus a single filer operation to manipulate several files at once. Section 3.2 describes the filer commands; the command summary groups the commands by their function, while the alphabetically ordered list describes each command in detail. Sections 3.3 through 3.5 describe methods for recovering inadvertently removed disk files and directories, and also how to change disk file attributes.

3.0 Filer Prompts

The filer's promptline has the following form:

```
Filer: G(et,S(ave,W(hat,N(ew,L(dir,R(em,C(hng,T(rans,D(ate,
                                                    Q(uit[3.0,b]
```

The remaining commands are displayed by typing "?":

```
Filer: B(ad-blks,E(xt-dir,K(rnch,M(ake,P(refix,V(ols,X(amine,
                                                    Z(ero[3.0,b]
```

Typing "?" again causes the original promptline to reappear.

In the filer, responding to "yes/no" questions with any character other than "Y" or "y" constitutes a negative response. Typing <escape> as a response to any data prompt aborts the current command and returns control to the filer prompt.

Many filer commands require one or two file names. Whenever a filer command requests a file name, the user may specify as many files as desired by separating each file name with commas and terminating the list with a carriage return. Commands operating on single files read the names from the list and operate on them one at a time until there are none left. Commands requiring two file names (e.g., C(hange and T(ransfer) take them from the list in pairs until one or none remain; if one file name remains, the filer prompts for the second. If an error occurs while operating on the list (such as an invalid file name), the remainder of the list is not processed.

3.1 File Naming Conventions

3.1.0 General Syntax

The filer accepts standard syntax for file names (see section 2.1.5). All filer commands except for G(et and S(ave require complete file names, including file identifier suffixes; G(et and S(ave automatically append file suffixes to the specified file title.

The "$" character is treated specially when used in a file name; it is applicable only to filer commands which operate on pairs of file names. When used in the second file name, a "$" represents the file identifier in the first file name. For example:

```
Transfer what file? *BUCKS.TEXT,#5:$
```

... transfers the file "BUCKS.TEXT" on the system volume to the disk volume mounted in disk unit 5. The filer substitutes the string "BUCKS.TEXT" for the "$" character.

Volume identifiers normally require a trailing ":" character to differentiate them from file identifiers; however, filer prompts accept volume identifiers of the form "#<number>". This feature applies only to volume identification and not to disk file designation.

3.1.1 Wildcards

The characters "=" and "?" are treated specially when used in a file name; they are called "wildcard" characters because of their ability to make a single file designator specify many disk files. Wildcard characters are used in conjunction with partially specified file identifiers in order to match a subset of all the file names in a given directory. For example, a file designator containing the file identifier "SYS=TEXT" notifies the filer to perform the requested operation on all files whose names begin with the string "SYS" and end with the string "TEXT".

Wildcard file identifiers are constrained to match this form:

```
<string>=<string>
```

The metasymbol <string> represents a sequence of valid file identifier characters. Either or both strings may be empty; thus, "=<string>", "<string>=", and "=" are valid wildcard forms. In the last case, where both strings are empty, the filer acts on every disk file in the specified volume's directory.

The character "?" may be used in place of "=" as a wildcard. "?" is functionally equivalent to "="; however, for each file that matches the wildcard specification, the filer issues a verification prompt before performing the requested operation.

Here are some examples of the use of wildcards:

Transfer what file? #4:SYSTEM.=,ALTDISK:=.CODE

This response transfers all system files to the online volume named "ALTDISK"; in addition, the system files appear as code files on ALTDISK. For instance, SYSTEM.FILER becomes FILER.CODE.

Remove what file? *?

This response generates a series of prompts of the form:

"Remove <file name>?"

... where <file name> is the name of a disk file on the system volume. The number of prompts generated equals the number of disk files on the system volume. For each prompt, typing "y" or "Y" removes the named file; typing any other key except <escape> preserves the file and generates the prompt for the next disk file; typing <escape> aborts the entire R(emove command.

WARNING - In some cases, wildcards may fail to match valid file names. Section 3.2.14.1 describes some other problems associated with the use of wildcards.

3.2 Filer Commands

Section 3.2.0 organizes the filer commands by function and is useful as an overview and cross reference. Sections 3.2.1 through 3.2.18 describe each command in detail; the commands are arranged in alphabetical order.

3.2.0 Filer Command Summary

Q(uit - leave the file handler and return to the system prompt.

3.2.0.0 Work File Commands

Work files are described in section 2.2.1. These filer commands manipulate work files:

G(et - Create a new work file (containing the contents of an existing file).

S(ave - Save the work file contents in a disk file.

N(ew - Create a new work file (empty).

W(hat - Display the name and status of the work file.

3.2.0.1 Disk File & Volume Commands

Disk volumes and files are described in section 2.1. These filer commands manipulate disk files and volumes:

C(hange - Change the name of an existing disk file or volume.

T(ransfer - Transfer a disk file to another location on its disk volume or to another volume. Transfer an entire disk volume to another disk volume.

R(emove - Remove a disk file.

M(ake - Create a disk file.

3.2.0.2 Disk Volume Commands

These commands manipulate disk volumes only:

L(dir - List the contents of a disk directory.

E(xt-dir - List the complete contents of a disk directory.

D(ate - Change the system date.

K(runch - Remove all free disk space between existing disk files.

P(refix - Change the current prefixed volume name.

V(olumes - Display the volume names of all online volumes.

Z(ero - Initialize a disk volume by removing all existing file entries.

3.2.0.3 Disk Media Commands

These commands check for and repair damaged areas of disk media.

B(ad blocks - scan a block-structured unit for damaged disk blocks.

X(amine - Examine and attempt to repair damaged disk blocks.

3.2.1 B(ad blocks scan

Scans a disk for blocks that are not storing information reliably.

The filer prompts for the volume to be scanned. Each block of the named disk is checked for problems; the block number of the block currently under testing is printed out, along with a warning message if the block is bad.

Bad blocks are either repaired or permanently marked bad with the X(amine command.

Bad blocks scanning is performed much more efficiently with the utility program Bad.blocks (section 9.0.4).

3.2.2 CHange

Changes the name of a disK file or disK volume.

This command requires two file names: the name to be changed, and the new name. The first is separated from the second by either a <return> or a comma.

When changing the name of a disK file, a volume identifier or length specifier in the second file name is ignored. A file name is not changed if the new name exceeds 15 characters; instead, an error message is printed.

Wildcard specifications are legal with this command. If a wildcard character is used in the first file name, then it must be used in the second; the strings matched by the first wildcard are substituted for the second wildcard.

Example of changing a disK file name:

```
Change what file? DUMP:=.BACK,=.TEXT
```

This response changes all backup files on the disK volume named DUMP to text files.

When changing the name of a disK volume, a file identifier in the second file name is illegal. A volume name is not changed if the new name exceeds 7 characters; instead, an error message is printed.

Example of changing a disK's volume name:

```
Change what file? #4,WORK:
```

This response changes the name of the disK volume mounted in drive 4 to "WORK".

3.2.3 D(ate

Displays the current system date, and allows the date to be changed.

```
Prompt: Date Set: <1..31>-<Jan..Dec>-<00..99>
        Today is 30-Feb-81
        New date?
```

New date entries have the following form:

```
[<new day>[-<new month>[-<new year>]]]<return>
```

Typing <return> preserves the current date. The metasymbol <new day> is an integer between 1 and 31. <new month> is the first three characters of the month's name (extras are ignored). <new year> is an integer between 0 and 99, denoting the last 2 digits of a year in this century.

NOTE - "/" may be used as an alternate character to the "-" delimiter shown above.

The current date is saved in the system's information file and is displayed in the welcome message and the D(ate command. When disk files are created or modified, the system assigns the current system date to the file; file dates are displayed by the directory listing commands L(dir and E(xt-dir.

3.2.4 E(xtended list

Lists a disk directory in more detail than the L(dir command.

All files and unused areas are listed; the fields displayed (in order) are: file name, file length (in blocks), date of file creation or last modification, starting block address (relative to disk), number of valid bytes in the last block of the file, and file type. Only the block length and starting address fields apply to unused areas of disk.

This command is identical to the L(ist directory command with respect to listing options and wildcards.

Example of an extended directory listing:

```
PROSE:
START.TEXT          4   15-Jan-81     10    512   Textfile
< UNUSED >         18                 14
CHAP3A.TEXT        48    5-Jan-81     32    512   Textfile
PROSE.CODE         33   24-May-80     80    512   Codefile
PROSE3.CODE        35   26-Nov-80    113    512   Codefile
< UNUSED >         32                148
BEST.DATA          16   15-Jan-81    180    314   Datafile
CONT.TEXT          18    5-Jan-81    196    512   Textfile
< UNUSED >        280                214
6/6 files<listed/in-dir>, 154 blocks used, 330 unused,
280 in largest area
```

3.2.5 G(et

Creates a new work file. The work file initially contains a copy of the contents of the specified text file.

If a work file exists, but is not saved, this prompt appears:

    Throw away current workfile?

Typing "y" proceeds with the command. Typing any other character aborts G(et, saving the current work file.

The following prompt appears:

    Get what file?

The file name does not require a suffix; it is appended by the G(et command. The file name designates a text and/or code file as the work file.

NOTE - A disk file is not created by G(et. If the work file SYSTEM.WRK exists, it is removed. The specified disk files become the source of the new work file. Subsequent modifications to the work file are saved in a new disk file named SYSTEM.WRK.

Work files are described in section 2.2.1.

3.2.6 K(runch

Moves all disk files on the specified disk volume to the front of the disk, thus merging all unused disk space into one contiguous area at the end of the disk.

Before crunching a disk volume, be sure to perform a B(ad blocks scan; files can be lost by writing them on top of unmarked bad blocks on the disk. If found, bad blocks must either be fixed or marked with the X(amine command before crunching the disk; the K(runch command carefully avoids disk blocks already marked as "bad".

NOTE - If the file SYSTEM.PASCAL is moved while K(runching the system disk, the system indicates that it must be rebooted.

WARNING - nothing must happen to the system while crunching is in progress. Interrupting a disk crunch may ruin the contents of a disk volume; therefore, the following steps should be taken while crunching:

1) Do not type ahead any system commands during a crunch.

2) Do not disturb any of the online disk volumes.

3) As much as is possible, prevent accidental power-down of the system.

Example of using K(runch:

```
Crunch what vol? #5
```

The user has specified the crunching of the disk volume in drive 5. The system responds with the following question to determine the sincerity of the user's K(runch command invocation:

```
Are you sure you want to crunch <volume identifier> ?
```

A "Y" or "y" answer initiates the crunch. Any other character aborts the command.

3.2.7 L(ist directory

Lists all, or some subset of, the files in the disK directory of the specified disK volume. The directory listing may be displayed on the console or written to a file.

The list command displays this data prompt:

Dir listing of what vol?

Responses have the following form:

[<volume id>[file identifier]][,[<file name>]]

The optional volume field specifies the disK volume whose directory is to be listed; its default value is the current prefixed volume. When the optional file identifier is used, the directory listing contains only the files whose names match the given file identifier (wildcards are used here to designate a group of similar file names).

The optional file name field specifies the name of the file to which the directory listing is to be written; its default value sends the listing to the console.

The directory listing consists of a list of file entries followed by some disK status information. A file entry contains a file's name, length (in blocks), and date. (The E(xt-dir command displays more file information.) The status information includes the number of files listed versus the total number in the directory, the number of blocks used by existing disK files, the total number of unused blocks, and the number of contiguous blocks in the largest unused space.

The most common use of this command is to list an entire disK directory to the console; when the listing is too long to fit on the screen, the following prompt appears after a screenful of file entries:

Type <space> to continue

Typing <space> causes the rest of the listing to be displayed. Typing <escape> aborts the listing command.

Some examples of directory listing responses:

```
Dir listing of what vol? ,
or...
Dir listing of what vol? :
```

... list the directory of the prefixed volume.

```
Dir listing of what vol? *SYSTEM=
```

... lists all of the system files on the system volume.

```
Dir listing of what vol? #4:=.TEXT,MYDISK:DLIST.TEXT
```

... lists all of the text files on the disk volume in drive 4 and writes the listing to the text file "DLIST.TEXT" on the online disk volume "MYDISK".

An example of a directory listing:

```
PROSE:
START.TEXT          4  15-Jan-81
CHAP3A.TEXT        48   5-Jan-81
PROSE.CODE         33  24-May-80
PROSE3.CODE        35  26-Nov-80
BEST.TEXT          16  15-Jan-81
CONT.TEXT          18   5-Jan-81
6/6 files<listed/in-dir>, 154 blocks used, 330 unused,
 280 in largest area
```

3.2.8 MCake

Creates a disk file with the specified file name.

File length specifiers are extremely useful in conjunction with this command; they specify the length of the file to be created, and indirectly determine the location of the file on the disk.

Sections 3.3 through 3.5 describe applications of this command, which include the recovery of lost files and the manipulation of existing disk files and free spaces.

Some restrictions exist with respect to the creation of text files. A text file must be created with an even number of blocks and contain a minimum of four blocks. Text files specifying a length of less than four blocks are not accepted, and odd block lengths are rounded down to the closest even number.

Wildcards are not allowed.

Example of using the Make command:

```
Make what file? *STUFF[7]
```

... creates the data file "STUFF" in the first unused 7-block area on the system volume.

3.2.9 N(ew

Creates a new work file. The new work file is empty.

If a work file exists, but has not been saved, this prompt appears:

    Throw away current workfile?

Typing "y" or "Y" removes the work file; typing any other character aborts the command.

NOTE - If the work file SYSTEM.WRK exists, it is removed. Backups of the work file (i.e. SYSTEM.WRK.BACK) are unaffected by N(ew, and must be manually removed.

3.2.10 Prefix volume

Changes the current prefixed volume to the volume specified.

This prompt is displayed:

    Prefix titles by what vol?

A valid response contains a volume identifier; any associated file identifer is ignored. The volume specified need not be online.

If the volume identifier contains a unit number, the prefixed volume is set to the name of the volume in the specified disk drive. If no volume is online in the disk unit, the prefixed volume is set to the unit number itself, and the prefixed volume is defined to be whatever disk volume is mounted in that unit.

The current prefixed volume can be determined by responding to the data prompt with ":" (this actually sets the new prefixed volume to the current prefixed volume).

3.2.11 Quit

Exits the filer and returns control to the system prompt.

NOTE - The system disk should be remounted in the proper disk drive before typing Quit.

3.2.12 R(emove

Removes files from the directory.

The files specified are removed from the disk; the disk space they occupied is marked as unused space, and their directory entry is erased and made available for future files. Length specifiers are ignored in file names, and wildcards are allowed.

Before completing the removal of files matched by a wildcard file name, the filer displays this prompt:

```
Update directory?
```

Responding with a "y" or "Y" causes all of the matched files to be removed. Typing any other character aborts the command and saves all the files.

NOTE - SYSTEM.WRK.TEXT and/or SYSTEM.WRK.CODE should be removed only by the N(ew command; using R(emove to remove them fails to update the system's work file state variables and may result in confusing system behavior.

NOTE - When a disk file is removed, its data is not destroyed; only the directory entry that locates and protects the file's data is removed. Thus, inadvertently removed disk files may be recovered without harm if immediate actions are taken. See section 3.3 for more information.

3.2.13 S(ave

Saves the work file contents in a disk file.

If the work file originates from a disk file other than SYSTEM.WRK, this prompt appears:

    Save as <file name>?

Typing "y" or "Y" writes the work file contents to the disk file named by the prompt. Typing any other character generates the prompt described below.

If the work file has not been saved (or the user "fell through" from the above prompt), this prompt appears:

    Save as what file?

The specified file name must not contain a file suffix or length specifier; the appropriate suffix (.TEXT or .CODE) is automatically appended to the file name response. Wildcards are not allowed.

NOTE - If the work file contents are saved on the system volume, the file SYSTEM.WRK is C(hanged to the specified file name; the resulting disk file becomes the source of the work file. If the work file contents are saved on a different volume, SYSTEM.WRK is T(ransferred to the volume with the specified file name; the source of the work file remains in the file SYSTEM.WRK.

3.2.14 Transfer

Copies the specified disk file or disk volume to the specified destination.

This command requires two file names: the source file and the destination file. The pair of names may be separated by either a comma or <return>. Complete file names must be provided. Length specifiers are ignored in the source file name, but are recognized in the destination file name as a means of controlling the location of the destination file. Wildcards are allowed.

Transfer is used for the following tasks:

1) Copying disk files onto different disk volumes.

2) Copying entire disk volumes onto different disks (though the Backup utility does a better job of it).

3) Transferring files to and from the console, printer, or remote device.

4) Moving disk files to other locations on the same disk volume.

Transfers from serial units are allowed if the device can generate data; generally, only the console is used in this fashion. Files emanating from a serial device are terminated by the transmission of an end-of-file flag; this is done from the terminal by typing <eof>.

Length specifiers are useful for controlling the location of disk files written to the destination volume. For instance, if a 25-block unused area is at the front of a volume, and a 25-block disk file is to be transferred to the volume, the file can be written directly to the unused space by adding the length specifier "[25]" to the destination file name. Without the length specifier, the filer writes the file into the largest available free space on the destination volume.

NOTE - See section 3.2.14.1 for problems with Transfer.

Examples of disk file transfers:

```
Transfer what file? *system.=,#5:$
```

... transfers copies of all system files on the system volume to the disk volume mounted in unit 5.

```
Transfer what file? stuff.text, stuff.text[25]
```

... transfers the file "STUFF.TEXT" to an unused area of disk containing at least 25 contiguous blocks.

```
Transfer what file? WORK:,BACKUP:
```

... copies the entire disk volume "WORK" onto the disk volume "BACKUP", destroying BACKUP's existing contents. When the transfer is completed, two identical disk volumes named "WORK" are online.

```
Transfer what file? DOCUMENT.TEXT,PRINTER:
```

... prints the text file "DOCUMENT.TEXT" on the printer.

3.2.14.0 Single-drive Transfers

Filer operations involving two distinct disk volumes are easily performed with a system having two disk drives online; however, they can also be performed using a single online disk drive.

Example of a single-drive transfer:

Transfer what file? WORK:IMPORTANT.TEXT

To where? BACKUP:$

The disk volume "WORK" must not be removed until the following prompt appears:

Put in BACKUP: Type <space> to continue

At this point, the disk volume "WORK" is removed from the drive and replaced with the disk volume "BACKUP", and <space> is typed. Transfers of large files or entire disk volumes generate a series of prompts having the form:

Put in <volume name>: Type <space> to continue

... where <volume name> alternates between the name of the source and destination volumes until the transfer is complete. Transferring entire disk volumes in this fashion is a tedious process, as the filer can only buffer as much data as it can fit in memory; the user must suffer through numerous disk swappings.

NOTE - Failure to mount the correct disk volume after a volume prompt jeopardizes the successful transfer of files; keep the disk volumes straight!

3.2.14.1 Transfer Problems and Warnings

WARNING - Unless entire disk volumes are being transfered, the destination's file identifier must not be omitted; otherwise, the directory of the destination volume may be destroyed. Transfers to a destination disk volume are verified with the prompt:

Possibly destroy directory of <volume name> ?

Typing "y" or "Y" commences the disk transfer, and overwrites the existing directory; typing any other character aborts the transfer and spares the directory.

Example of directory destruction:

Transfer what file? MYDISK:DIR.WHAM.CODE,VICTIM:

WARNING - Wildcards should not be used in file names when transferring files to different locations on the same disk volume; the results are unpredictable.

Example of bad wildcards:

Transfer what file? =,=

WARNING - Transfers of entire disks lacking directories may fail, as the filer depends on directory information to determine the number of blocks on the disk to transfer. If the information is not present, the filer transfers data in integral buffer quantities until no more can be transferred; depending on the disk size, the last (remainder) part of the disk may not be transferred. This problem can be avoided by using the Backup utility (section 9.0.1) to copy entire disk images.

NOTE - The filer does not allow any other characters or strings to be associated with "$" in a file name; for instance:

Transfer what file? FOON.TEXT,$[50]

... is not accepted by the filer. An alternative method exists for taking advantage of "$" in this situation:

Transfer what file? FOON.TEXT[50],$

The length specifier is ignored in the source file name, but the "$" carries it over to the destination name where it is recognized.

3.2.15 V(olumes online

Lists all volumes currently online along with their assigned unit numbers.

A typical volume display is:

```
Volumes on-line:
1   CONSOLE:
2   SYSTERM:
4 * MYDISK
5 # EXTRA:
8   REMOUT:
Prefix is - EXTRA
```

An asterisk ("*") marks the system volume. Online disk volumes are indicated by "*" or "#". The current prefixed volume is displayed at the bottom.

NOTE - The presence of a disk volume name in the list indicates that the volume is online. On the other hand, the presence of a serial volume name merely indicates that the system supports the corresponding device; the device itself may be online or offline.

3.2.16 W(hat is workfile?

Identifies the name and state of the current work file. The work file state is either "saved" or "not saved".

3.2.17 X(amine bad blocks

Attempts to physically recover suspected bad blocks, and mark unrecoverable blocks as unusable.

Example of using X(amine:

```
Examine blocks on what volume?
```

After specifying a volume name or unit number, the following prompt appears:

```
Block number range?
```

The user enters the block number(s) of suspected bad blocks (section 3.2.1 describes one method of detecting them). Block number ranges have the following form:

```
<block number>[-<block number>]
```

When the optional part is used, all blocks between the two block numbers specified are examined.

If any files are endangered by containing bad blocks, the following prompt appears:

```
File(s) endangered:
<file name>
Try to fix them?
```

Typing "y" or "Y" starts the fixing process on the named blocks; typing any other character aborts the command. When completed, X(amine returns one of these messages:

```
Block <block number> may be ok
```

... indicating that the block is probably fixed, or ...

```
Block <block number> is bad
```

... indicating that the block is a hopeless case. X(amine offers the user the option of marking hopeless blocks as files of type "bad". These files are not shifted by the K(runch command; their presence prevents regular files from being written over bad areas of the disk.

WARNING - A "fixed" block may contain garbage as data; the fixing process can only ensure the integrity of subsequent write operations to the fixed block. Block-fixing is done by reading up a block, writing it out, and reading it up again. If the two read operations bring in identical data without raising any I/O errors, the block is considered fixed ("may be ok"); otherwise, the block is declared bad.

3.2.18 Z(ero directory

Writes an empty directory on the specified disk.

Z(ero is used to build new disk volumes on either brand new disks or obsolete disk volumes. If an old volume resides on the disk, some of its volume information is assumed to be applicable to the new disk volume; the prompt sequence is changed accordingly.

NOTE - Z(ero automatically marks all disks to contain duplicate directories.

3.2.18.0 New Disks

The following prompt appears:

Zero dir of what vol?

The volume identifier of the disk to be zeroed is specified. The next prompt is:

Number of blocks (S-494 D-988 Q-1976)?:

Any positive integer may be entered. The numbers displayed are standard values for single density (S), double density (D), and double density/double-sided (Q) 8-inch floppy disks. The next prompt is:

New vol name?

Any valid volume name may be entered. The entered volume name is verified by the next prompt:

<volume name> correct?

Typing "y" or "Y" zeroes the disk; typing any other character aborts the command. In both cases, control returns to the filer prompt.

NOTE - Brand-new disks should be formatted with the Format utility (section 9.0.3) before being Z(eroed.

3.2.18.1 Recycling Old Volumes

If the disk specified for zeroing contains an existing disk volume, the following changes occur to the prompt sequence defined in the previous section. Before the block number prompt, the Z(ero command is verified with the prompt:

Destroy <current volume name>?

Typing "y" or "Y" continues the prompt sequence; typing any other character aborts the command.

Instead of requesting the number of blocks on the disk, the filer assumes that the new disk volume has the same number of blocks as its ancestor, and prompts:

<block number> blocks?

... where <block number> is the number of blocks in the obsolete disk volume. Typing "y" or "Y" uses the existing value for the new volume; typing any other character generates the block number prompt described in the previous section.

3.3 Recovering Lost Files

Files may be lost by explicit removal or by creation of a new file having the same name as an existing file; in both cases, the directory entry for the existing file is erased, and the file appears to be permanently lost. This is not always true. This section describes a method for recreating removed files.

When a disk file is removed, the file itself is still on the disk; only its associated directory entry is erased. However, the disk space occupied by the removed file is marked as unused space; any subsequent activity involving the writing of data to the disk may overwrite the file's contents. Therefore, the probability of recovering a lost file is directly related to the disk activity occurring between the removal of the file and the discovery by the user of its nonexistence.

The E(xtended directory list command displays both files and unused areas on a disk volume. The object of this method is to determine which area marked as unused space on the disk contains the missing file, and then to use the M(ake command to create dummy files of various sizes until the position and size of one of the dummy files coincides with the missing file (see section 2.1.4.4 for a description of file space allocation directives). If this stage is reached, recovery consists of removing any other dummy files created during the hunt, and changing the name of the coincident dummy file to the name of the missing file.

NOTE - Files created with M(ake cannot write over the data in the missing file; they are merely directory entries associating a file name with a group of blocks on the disk.

File recovery is easiest when the file's size and location are known beforehand; the following example is a demonstration of this case. The process becomes more difficult when some of the parameters are unknowns; several iterations of creation and removal of dummy files may be necessary before the missing file is located and contained.

Of the various file types, it is easiest to verify the capture of text files; dummy text files viewed in the editor immediately reveal their contents. Data and code files are comparatively difficult to capture; verification of their contents requires a knowledge of their underlying structure and the utility programs Patch, Library, and Libmap (described in chapter 9). Data file structures must be known by the user. Code file structures are described in the Architecture Guide.

Example of recovering a lost text file:

Here is a pre-accident directory listing:

```
PROSE:
START.TEXT          4   15-Jan-81     10    512   Textfile
< UNUSED >         18                  14
CHAP3A.TEXT        48    5-Jan-81     32    512   Textfile
PROSE.CODE         33   24-May-80     80    512   Codefile
PROSE3.CODE        35   26-Nov-80    113    512   Codefile
< UNUSED >         32                 148
BEST.TEXT          16   15-Jan-81    180    512   Textfile
CONT.TEXT          18    5-Jan-81    196    512   Textfile
< UNUSED >        280                 214
6/6 files<listed/in-dir>, 154 blocks used, 330 unused
```

The valuable file BEST.TEXT is now accidentally removed by the creation of a new file BEST.TEXT; fortunately, the user is alert enough to remember the location of the old BEST.TEXT. Here is the current situation:

```
PROSE:
START.TEXT          4   15-Jan-81     10    512   Textfile
< UNUSED >         18                  14
CHAP3A.TEXT        48    5-Jan-81     32    512   Textfile
PROSE.CODE         33   24-May-80     80    512   Codefile
PROSE3.CODE        35   26-Nov-80    113    512   Codefile
< UNUSED >         48                 148
CONT.TEXT          18    5-Jan-81    196    512   Textfile
BEST.TEXT          16   15-Jan-81    214    512   Textfile
< UNUSED >        264                 230
6/6 files<listed/in-dir>, 154 blocks used, 330 unused
```

The dummy files are created with the M(aKe command. DUMMY1.TEXT[18] fills the 18-block unused area at the front of the disk. DUMMY2.TEXT[32] fills the first 32 blocks of the 48-block unused area that contains the missing file. DUMMY3.TEXT[16] fills the last 16 blocks of the 48-block area, and coincides with the old copy of BEST.TEXT. The directory now appears as:

```
PROSE:
START.TEXT          4   15-Jan-81     10    512   Textfile
DUMMY1.TEXT        18   15-Jan-81     14    512   Textfile
CHAP3A.TEXT        48    5-Jan-81     32    512   Textfile
PROSE.CODE         33   24-May-80     80    512   Codefile
PROSE3.CODE        35   26-Nov-80    113    512   Codefile
DUMMY2.TEXT        32   15-Jan-81    148    512   Textfile
DUMMY3.TEXT        16   15-Jan-81    180    512   Textfile
CONT.TEXT          18    5-Jan-81    196    512   Textfile
BEST.TEXT          16   15-Jan-81    214    512   Textfile
< UNUSED >        264                230
9/9 files<listed/in-dir>, 220 blocks used, 264 unused
```

The file has been recovered; only cleanup remains. DUMMY1.TEXT and DUMMY2.TEXT have served their purpose as free space fillers; they are removed. The new copy of BEST.TEXT is saved under a different name, and DUMMY3.TEXT is changed to BEST.TEXT.

```
PROSE:
START.TEXT          4   15-Jan-81     10    512   Textfile
< UNUSED >         18                14
CHAP3A.TEXT        48    5-Jan-81     32    512   Textfile
PROSE.CODE         33   24-May-80     80    512   Codefile
PROSE3.CODE        35   26-Nov-80    113    512   Codefile
< UNUSED >         32                148
BEST.TEXT          16   15-Jan-81    180    512   Textfile
CONT.TEXT          18    5-Jan-81    196    512   Textfile
NBEST.TEXT         16   15-Jan-81    214    512   Textfile
< UNUSED >        264                230
7/7 files<listed/in-dir>, 170 blocks used, 314 unused
```

3.4 Recovering Lost Directories

The loss of a disK directory is a much more serious setbacK than the loss of a single disK file. The best protection against directory mishaps is to maintain duplicate directories on all disK volumes. When a disK volume loses its regular directory, but has a duplicate directory, the Copydupdir utility (section 9.0.1) replaces its deceased regular directory with a copy of the duplicate directory; the volume is then restored.

WARNING - regular disK directories are stored on blocKs 2-5 of a disK volume, while duplicate directories are stored on blocKs 6-9; unfortunately, this implies that some accidents may simultaneously wipe out both directories. The method for recovering from this situation is to Z(ero the directory, and then use the method described in the previous section for fishing the files from the disK; needless to say, this is a tedious and not necessarily rewarding tasK. The best protection for a disK volume is to maintain a copy of the volume on a separate disK.

3.5 Changing the Type or Date of a File

Users occasionally find themselves stuck with a file of the wrong type. A common occurrence of this problem is the the linker's penchant for producing an output data file instead of the desired code file; the data file contains valid linked code, but its file type prevents it from being executed by the system.

One solution to this problem is based on the method presented for recovering lost files (described in section 3.3). The bogus file is removed and a new file is created with the MAKE command such that the new file is coincident with the old file; additionally, the new file is created with the file suffix corresponding to the desired file type.

Example of changing a file's type:

The file PROSE should be a code file, but somehow has ended up as a data file; thus, it is nonexecutable. Here is the directory listing:

```
PROSE:
START.TEXT          4   15-Jan-81     10    512   Textfile
STUFF.DATA         18   32-Feb-80     14    202   Datafile
CHAP3A.TEXT        48    5-Jan-81     32    512   Textfile
PROSE              33   24-May-80     80    512   Datafile
PROSE3.CODE        35   26-Nov-80    113    512   Codefile
< UNUSED >         32                148
BEST.TEXT          16   15-Jan-81    180    512   Textfile
CONT.TEXT          18    5-Jan-81    196    512   Textfile
< UNUSED >        280                214
7/7 files<listed/in-dir>, 172 blocks used, 312 unused
```

The data file PROSE is removed; a 33-block free space now exists where it once resided:

```
PROSE:
START.TEXT          4   15-Jan-81     10    512   Textfile
STUFF.DATA         18   32-Feb-80     14    202   Datafile
CHAP3A.TEXT        48    5-Jan-81     32    512   Textfile
< UNUSED >         33                 80
PROSE3.CODE        35   26-Nov-80    113    512   Codefile
< UNUSED >         32                148
BEST.TEXT          16   15-Jan-81    180    512   Textfile
CONT.TEXT          18    5-Jan-81    196    512   Textfile
< UNUSED >        280                214
6/6 files<listed/in-dir>, 139 blocks used, 345 unused
```

The M(ake command is used to recreate the file:

```
     Make what file? PROSE.CODE[33]
```

PROSE exists once again; it is now an executable code file:

```
  PROSE:
  START.TEXT          4   15-Jan-81     10    512  Textfile
  STUFF.DATA         18   32-Feb-80     14    202  Datafile
  CHAP3A.TEXT        48    5-Jan-81     32    512  Textfile
  PROSE.CODE         33   24-May-80     80    512  Codefile
  PROSE3.CODE        35   26-Nov-80    113    512  Codefile
  < UNUSED >         32                148
  BEST.TEXT          16   15-Jan-81    180    512  Textfile
  CONT.TEXT          18    5-Jan-81    196    512  Textfile
  < UNUSED >        280                214
  7/7 files<listed/in-dir>, 172 blocks used, 312 unused
```

The procedure for changing the file date is similar. Prior to M(aking the file, use the D(ate command in the filer to temporarily change the system date to the desired point in time; M(ake assigns this date to the recreated file. Don't forget to return to the present afterwards.

IV. THE EDITOR

The editor is used to create and modify text files. Editor prompts are similar to filer and system prompts; they are described in section 4.0. Text files may contain either Pascal programs or documents; because these have different formatting conventions, the editor's operation (known as the "environment") can be changed to suit program development or word processing. Editor environments are described in section 4.1. Basic editor commands and features are described in sections 4.2 through 4.9. Section 4.10 describes the remaining editor commands; the command summary provides command overviews grouped by their function, while the detailed command descriptions are organized alphabetically. Problems encountered during regular editor use are described in section 4.11.

4.0 Editor Prompts

Editor prompts display either a promptline of available edit commands or a command line (generated as a result of typing a command from an outer promptline) displaying the available command options. All editor prompts display the current direction (described in section 4.5) in the leftmost character of the prompt. Prompts are almost always present at the top of the screen, but occasionally disappear during some edit commands; depending on the situation, typing either <etx> or a different command redisplays the prompt.

The editor prompt has the following format:

```
Edit: A(djust C(py D(lete F(ind I(nsrt J(mp R(place Q(uit X(chng Z(ap
```

Only the most commonly used commands appear on the prompt. The remaining commands are displayed by typing "?".

4.1 Edit Environments

Edit commands affect the structure of text; edit environments affect the behavior of edit commands. Environment parameter values are saved within text files; unless changed, they control not only the current edit session, but all future edit sessions on the current text file. The most important parameters are "auto-indent", "filling", and "margins". Auto-indent is used to facilitate the indentation of Pascal programs. Margins and filling are used for processing documents; in particular, filling allows the justification of paragraphs of text within the current margins.

Edit environments are described in more detail in section 4.10.11 (the S(et command).

4.2 The File Window

The editor allows the the entire console screen to be used much like a chalkboard; any text displayed on the screen can be directly accessed and modified. At the beginning of an edit session, the editor displays the start of the file in the upper left corner of the screen. Unfortunately, most text files have more lines than can be displayed at once on the console; therefore, when the user moves to a section of text that is above or below the section currently displayed, the screen is updated by shifting some of the existing text off the screen to make room for the display of previously hidden lines of text. The screen can be thought of as a "window" sliding over the text file being edited; the entire text file is accessible using the edit commands, but only the section of text that is currently being changed can be viewed through the window.

4.3 The Cursor

If the screen can be considered a chalkboard, the cursor then serves as eraser, chalk, and pointer. All action takes place around the cursor; it represents the user's exact position in the file, and it can be moved to any position within the text file. The file window automatically follows the cursor; any command which moves the cursor off the current window recenters the window to display the text adjacent to the cursor.

Note that the cursor is never really "at" a character position; it is between the character where it appears and the immediately preceding character. This convention is important; it affects the behavior of the I(nsert and D(elete commands.

4.4 Repeat Factors

Most commands accept repeat factors. A repeat factor is specified by typing a positive integer before typing the command character; the digits of the integer are not printed on the screen, but the integer is internally recorded by the editor for a subsequent command. A repeat factor specifies that a command is to be repeated the number of times determined by its preceding factor. For example, entering "2 <down>" causes the <down> command to be executed twice, moving the cursor down two lines. The default repeat factor value is 1. A slash ("/") typed before the command indicates that the command is to be repeated until the end of the text file is reached. Commands accepting repeat factors are noted as such in their descriptions.

4.5 Direction

The editor maintains an environment parameter named "direction". Direction affects commands involved with cursor movement; for example, typing the space bar normally moves the cursor left-to-right across a line of text, and down when crossing text lines.

After changing the direction, the space bar exhibits the exact opposite behavior. The current direction is indicated by the leftmost character of editor prompts: ">" denotes forward direction, "<" denotes backwards direction. The default direction is forwards. Commands affected by direction are noted as such in their descriptions.

Direction commands may be executed whenever their Key definitions do not conflict with an enclosing command invocation (e.g., typing "<" in I(nsert). The following keys are defined to change direction:

"<" or "," or "-"	Change the current direction to backward
">" or "." or "+"	Change the current direction to forward

4.6 Markers

Markers enable arbitrary cursor positions in a text file to be easily accessible from anywhere within the file. Markers do not appear in the text itself; the only way to locate a marker is to jump to it. Markers are specified by name; names may contain up to eight characters, and are case-insensitive (e.g. the marker names "STUFF" and "stuff" denote the same marker).
Markers are saved across edit sessions in the text file. A file can contain up to ten markers.

The S(et M(arker command creates a marker at the current cursor position. Setting a marker to an existing marker name removes the old marker setting. J(ump M(arker moves the cursor to the specified marker. Existing marker names are displayed with the S(et E(nvironment command.

4.7 Moving The Cursor

This section describes most of the cursor-moving commands. Two alphabetic commands that move the cursor are J(ump and P(age; these are described in section 4.10.

One command not described below is the "equals" command, which is executed by typing "=". Equals causes the cursor to jump to the beginning of the last section of text which was inserted, found, or replaced; subsequent invocations will return to the same location. Equals is unaffected by direction, but is affected by a C(opy or D(elete operation between the start of the file and its current roosting location.

The cursor commands are described in the following table:

Direction insensitive commands-

<down>	Moves cursor down
<up>	Moves cursor up
<right>	Moves cursor right
<left>	Moves cursor left
<back-space>	Moves left

Direction sensitive commands-

<space>	Moves direction
<tab>	Moves cursor to the next tab stop; tab stops are every 8 spaces, starting at the left of the screen
<return>	Moves to the beginning of the next line

Repeat factors can be used with any of the above commands.

The cursor's column position is preserved by the <up> and <down> commands; however, when the cursor is moved outside the text, it is treated as though it were immediately after the last character or before the first in the line.

4.8 The Copy Buffer

The editor maintains a copy of the most recently changed text in its copy buffer. The contents of the copy buffer can be inserted into the text with the C(opy B(uffer command. The copy buffer is used to move or duplicate blocks of text within the file.

The contents of the copy buffer are updated by the following commands:

1) D(elete - the buffer is filled with the deleted text, regardless of whether the deletion is accepted (terminated with <etx>) or escaped (terminated with <esc>).

2) I(nsert - the buffer is filled with the inserted text only when the command is accepted; it is emptied after escaping from an I(nsert.

3) Z(ap - the Z(apped text is moved into the buffer.

The copy buffer is of limited size. Whenever a Z(ap or D(elete command changes more text than can fit in the copy buffer, the user is warned that the text cannot be copied and is asked (with a "yes/no" prompt) to verify acceptance of the command.

4.9 Entering Strings in F(ind and R(eplace

The F(ind and R(eplace commands operate on character strings. This section describes the features unique to these commands, including: syntax for specifying character strings (described in section 4.9.0), editor variables which contain the current target and substitution strings (described in section 4.9.1), and an environment parameter which affects the editor's method of searching for character strings (described in section 4.9.2). More details on this topic can be found in the descriptions of the F(ind, R(eplace, and S(et E(nvironment commands.

4.9.0 String Syntax

Strings may contain any characters (including nonprinting characters); they are delimited by two occurrences of the same character. For example, "/I'm a string/", ".8.", and "*randy*" represent the strings "I'm a string", "8", and "randy", respectively. Delimiting characters may be any non-alphanumeric character other than <space>.

NOTE - This is one of the few places in the system where a <return> is not required at the end of the data typed in; the command is executed immediately after the closing delimiter of the last string parameter is typed. Also, the editor does not allow a completed string parameter to be backspaced over from the prompt.

4.9.1 String Variables

The editor provides two string variables for saving the last string arguments used in a F(ind or R(eplace. The target string (named "<targ>") is used by both commands; the substitution string (named "<sub>") is used only by R(eplace. The string values in both of these variables may be used in subsequent F(inds and R(eplaces by using the letters "S" or "s" in place of an explicit string argument. For example, in F(ind, typing "S" (read as "find same") finds an occurrence of the contents of the <targ> variable in the text file. In R(eplace, typing "SS" (read as "replace same with same") replaces an occurrence of <targ> with the contents of <sub>, while typing ".match.s" replaces occurrences of the string "match" with the contents of <sub>.

The current values of <targ> and <sub> can be examined with the S(et E(nvironment command. No values are displayed if the variables are not assigned values during the edit session.

4.9.2 Search Modes

F(ind and R(eplace both have two different methods of searching for strings in a text file - Token mode and Literal mode. In Literal mode, the editor searches for any occurrences of the target string. In Token mode, it searches for an isolated occurrence, which is defined as a string delimited by spaces or other punctuation. For

example, in the string "now is the time for blisters", a Literal mode search finds two occurrences of the search string "is", while ToKen mode finds only one.

ToKen mode ignores spaces within strings; thus, the two strings ".,." and ". , ." are equivalent.

The search mode is kept as an environment parameter; its name is "Token def", which is short for "ToKen default mode". When this parameter is set true, all searches default to ToKen mode; when set false, they default to Literal mode. The initial parameter value is true, but can be changed by the user with the S(et E(nvironment command.

The current default search mode can be overridden in F(ind and R(eplace by using the letters "L"/"l" (force Literal mode) and "T"/"t" (force ToKen mode). These must appear outside of the string parameters; here are some examples of search mode override:

"L.foon." (find the string "foon" in Literal mode);

"T/foon//yeen/" (replace all toKen occurrences of "foon" with the string "yeen");

",bad,L,good," (replace all literal occurrences of "bad" with the string "good").

4.10 Editor Commands

Section 4.10.0 contains a command overview; the commands are grouped according to their function. Sections 4.10.1 through 4.10.14 describe each command in detail; the commands are alphabetically ordered.

4.10.0 Command Summary

4.10.0.0 Moving Commands

<down>	cursor down
<up>	cursor up
<right>	cursor right
<left>	cursor left
<space>	cursor in direction
<back space>	cursor left
<tab>	cursor to next tab stop
<return>	cursor to next line
"<" "," "-"	backward direction
">" "." "+"	forward direction
"="	cursor to start of last inserted/found/replaced

J(ump: Jump to marker or the beginning or end of the file.

P(age: Move cursor one page in the current direction.

4.10.0.1 Text-Changing Commands

I(nsert: Insert text.

C(opy: Copies last inserted/deleted/zapped text into the file.

D(elete: Delete text.

X(change: Exchange text.

Z(ap: Delete all text between last found/replaced/inserted/ adjusted text and the current cursor position.

4.10.0.2 Pattern Matching Commands

F(ind: Find character string patterns in text.

R(eplace: Locate string patterns in text and replace with a substitute pattern.

4.10.0.3 Formatting Commands

A(djust: Adjusts indentation of the current line.

M(argin: Adjust all text between two blank lines to the current margin settings.

4.10.0.4 Miscellaneous Commands

S(et: Set M(arkers to J(ump to or E(nvironment to change parameters.

V(erify: Redisplay screen with the cursor centered.

Q(uit: Leave the editor.

4.10.1 A(djust

Repeat factors are allowed.

Prompt:

```
>Adjust: L(just R(just C(enter <left,right,up,down-
                              arrows> (<etx> to leave)
```

A(djust changes the indentation of a text line. The <right> and <left> commands move the entire line on which the cursor is located one space right or left, respectively.

"L" and "R" left-justify and right-justify lines to the current margin settings. "C" centers the line between the margins. Margins are described in the S(et E(nvironment command.

A series of lines may be adjusted by adjusting one line the desired amount and then using the <up> and <down> commands to adjust adjacent lines by the same amount. Note that horizontal commands can be intermixed with vertical commands to allow cumulative horizontal offset changes on successive line adjusts; thus, typing "A(djust <left> <left> <down> <left> <down>" moves the current line two spaces to the left, while the two lines below it are moved three spaces to the left.

The <etx> key is typed to finish the command; the cursor is left at the beginning of the last line adjusted. There is no command available to exit A(djust.

4.10.2 C(opy

Prompt:

```
>Copy:  B(uffer F(ile <esc>
```

4.10.2.0 C(opy B(uffer

Typing "B" copies text from the copy buffer. The copy buffer contents are copied into the text, starting at the cursor location prior to invoking C(opy. The cursor is left at the front of the copied text.

The copy buffer is described in section 4.8.

4.10.2.1 C(opy F(ile

Prompt:

```
>Copy:  from what file[marker,marker]?
```

Typing "F" copies portions of text from another text file. The section of copied text is inserted into the current text file starting at the cursor location prior to invoking C(opy. The cursor is left at the front of the copied text.

Any text file may be specified; the file suffix ".TEXT" is optional.

WARNING - The disk containing the editor's code file must not be removed.

The marker specification (including the square brackets) is optional, and is used to copy selected portions of another file. Its form is:

```
<file name>["["[<marker name>],[<marker name>]"]"]
```

The markers specified must be present in the other file. The text copied is that which lies between the first and the second markers specified. An empty marker field indicates one end of the file as the delimiter of the copied text. For example, "[,<marker name>]" indicates that all text between the front of the file and <marker name> should be copied. Markers are described in section 4.6.

C(opy F(ile does not alter the contents of the file being copied.

4.10.3 D(elete

Prompt:

>Delete: < > <Moving commands> (<etx> to delete, <esc> to abort)

The cursor must be positioned at the first character to be deleted. Before entering D(elete, the cursor position is recorded; it is called the "anchor". As the cursor is moved away from the anchor using the moving commands, text in its path disappears. As the cursor is moved back toward the anchor, the previously deleted text is restored.

To accept the deletion, type <etx>; to escape, type <esc>.

NOTE - While the D(elete command itself does not accept repeat factors, the moving commands used within D(elete do accept repeat factors.

Example of using D(elete:

Here is the text before deleting:

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
This sentence of the text is to remain the same.  This
sentence is to be modified by the delete command.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

The cursor is positioned over the letter "t" in the second occurrence of the word "to". Enter D(elete by typing "D", then type <space> six times and <etx>. The text and cursor position now appear as follows:

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
This sentence of the text is to remain the same.  This
sentence is modified by the delete command.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

4.10.4 F(ind

Repeat factors are allowed.

Prompt:

```
>Find[<n>]:  L(it <target> =>
or ...
>Find[<n>]:  T(oK <target> =>
```

... depending on the value of the Token default environment parameter. The metasymbol <n> denotes the repeat factor value passed to F(ind.

F(ind finds the <n>-th occurrence of the target string in the text, starting at the current cursor position and moving in the direction displayed. If the repeat factor is "/", the last occurrence is found.

If an occurrence of the target string is found, the cursor is positioned after the found string; otherwise, the following prompt appears:

ERROR: Pattern not in file. Please press <spacebar> to continue.

Typing <esc> while entering the target string exits the F(ind command.

See section 4.9 for more details on using F(ind.

NOTE - Because F(ind leaves the cursor at the end of a target string, F(inds in the backward direction behave oddly. After a backward F(ind locates an occurrence of the target string, it is necessary to type "=<bs>" to move the cursor in front of the target string before finding the next match; otherwise, F(ind keeps finding the same target occurrence.

Example of using F(ind:

We will attempt to find "rutabaga". The cursor is located at the start of the line.

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
This sentence rutabaga contains an out-of-place word.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

The F(ind command is invoked with an argument of "rutabaga":

```
>Find[<1>]:  L(it <target> =>/rutabaga/
```

The cursor is moved to this position:

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
This sentence rutabaga_contains an out-of-place word.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

4.10.5 I(nsert

Prompt:

```
Insert: Text(<bs> a char, <del> a line)
                    [<etx> accepts, <esc> escapes]
```

Characters (including <return>) are inserted into the text as they are typed in. Any nonprinting characters that are typed are echoed with a "?". Text may be changed while it it is being inserted - typing <backspace> removes the last inserted character, while typing <del> removes the current line of inserted text. Text preceding the inserted text cannot be removed.

To accept the insertion, type <ext>; to escape, type <esc>.

Occasionally, I(nsert may add a blank at the end of the original line into which the insertion occurred. This allows optimization of large character-moving operations; it has no impact on Filling, and is not included in the copy buffer.

I(nsert is affected by the following environment parameters: Auto-indent, Filling, and Margins. These control the text margins as successive lines of text are inserted. See the S(et E(nvironment command for more details.

Example of using I(nsert:

Here is the text before inserting:

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
This sentence of the text is to remain the same.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

The cursor is positioned over the letter "t" in the word "to". Enter I(nsert by typing "I", then type "not <etx>". The text and cursor position now appear as follows:

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
This sentence of the text is not to remain the same.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

4.10.5.0 Using Auto-indent

If Auto-indent is True, a <return> causes the next line to have the same level of indentation as the immediately preceding line. If False, the indentation level for a new line is always zero. When Auto-indent is True, indentation levels are changed by using the <space> and <backspace> keys immediately following a <return>.

Example of Auto-indent:

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Line 1     Original indentation
Line 2     <ret> maintains current indentation level
  Line 3   <ret><space><space> indents by two
  Line 4   <ret> maintains current indentation level
Line 5     <ret><back space><back space> unindents by two
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

4.10.5.1 Using Filling

If Filling is True, all words inserted are forced to lie between the left and right margins. The editor
does this by automatically inserting a <return> between words whenever the right margin would have been exceeded, and by indenting to the left margin before every new line. Any character strings delimited by spaces or by a space and hyphen are considered words.

A paragraph is a series of text lines delimited by blank lines. Filling automatically adjusts the right margins of the remainder of a paragraph that has text inserted into it. However, any line beginning with a command character is not touched; it is considered to terminate the paragraph. Command characters are described in section 4.10.7.0.

The margins of a filled paragraph may be re-adjusted by using the M(argin command.

4.10.6 J(ump

Prompt:

Jump: B(eginning E(nd M(arker <esc>

Typing "B" or "E" moves the cursor to the beginning or end of the file.

4.10.6.0 J(ump M(arker

Prompt:

Jump to what marker?

Typing a marker name followed by a <return> moves the cursor to the marker's location in the file.

If the specified marker does not exist, the following prompt is displayed:

ERROR: Marker not there. Please press <space-bar>
to continue.

Markers associate user-defined names with arbitrary cursor positions within the text file. Section 4.6 describes markers.

Current marker names can be viewed with the S(et E(nvironment command.

4.10.7 M(argin

M(argin reorganizes the paragraph of text currently occupied by the cursor so that its text lines lie within the current margins. A paragraph is defined as a series of text lines delimited by blank lines. M(argin is used strictly for word processing; it cannot be executed unless Filling is True and Auto-indent is False.

The text format produced is similar to the filled format described in the I(nsert command (using Filling): M(argin indents to the paragraph margin on the first line of the paragraph, inserts a <return> between words whenever the right margin would be exceeded, and indents to the left margin before every new line. Any character strings delimited by spaces or by a space and hyphen are considered words.

Margin values are set with the S(et E(nvironment command.

M(argin may take several seconds to reorganize long paragraphs of text. The screen remains blank until the paragraph is finished; the screen is then redisplayed.

Example of using M(argin:

The paragraph before M(argin:

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
The Margin Command is executed
by typing "M" when
the cursor is in the paragraph to be margined.
The
Margin Command deals with
only one paragraph at a time
and realigns the text to the specification set in the
environment.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Set:
Left margin - 5
Right margin - 60
Paragraph margin - 10
Auto-indent - False
Filling - True

The paragraph after M(argin:

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        The Margin Command is executed by typing "M" when
    the cursor is in the paragraph to be margined.  The
    Margin Command deals with only one paragraph at a time
    and realigns the text to the specification set in the
    environment.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

4.10.7.0 Command Characters

For purposes of formatting, a paragraph is defined as a series of text lines delimited by blank lines. However, an arbitrary line of text can be protected from M(argin if a command character appears as the first non-blank character on the line. M(argin treats these lines as though they were blank lines. The character definition of the command character is controlled by the S(et E(nvironment command.

Command characters also affect the behavior of I(nsert.

WARNING - Do not use M(argin within a line that starts with the Command character.

4.10.8 P(age

Repeat factors are allowed.

Displays the screen of text adjacent to the current screen; the current direction determines whether the preceding or following screen is displayed. The cursor is left on the same line of the screen, but is moved to the start of the line.

4.10.9 Q(uit

Prompt:

```
>Quit:
     U(pdate the workfile and leave
     E(xit without updating
     R(eturn to the editor without updating
     W(rite to a file name and return
```

One of the four options must be selected by typing U, E, R, or W; all other characters are ignored.

U(pdate -

Stores the text file as the work file; it is named SYSTEM.WRK.TEXT. Work files are described in section 2.2.1.

E(xit -

Terminates the edit session - unless the W(rite option has already been used, all modifications made to the text during the edit session are lost, as the text file is not saved on the disk.

R(eturn -

Returns to the editor without updating. The cursor is returned to the same position in the file it occupied when "Q" was typed. This command is often used after unintentionally typing "Q".

W(rite -

Prompt:

```
>Quit:
Name of output file (<cr> to return) -->
```

The current text file may be written to any file name; a text file suffix is not required with the file name. Q(uit can be aborted by typing <return> instead of a file name; the text file and the editor prompt then reappear. However, if the file is written to a disk file, the following prompt appears:

```
>Quit:
Writing.......
Your file is <number> bytes long.
Do you want to E(xit from or R(eturn to the Editor?
```

Typing "E" exits from the Editor and returns control to the system prompt. Typing "R" returns control to the editor; the text and prompt are redisplayed and the cursor is returned to its original position. It is a good practice to periodically write the current text file contents out to a disk file in order to save the work invested in a long edit session.

4.10.10 R(eplace

Repeat factors are allowed.

Prompt:

```
>Replace[n]:  L(it V(fy <targ> <sub> =>
or ...
>Replace[n]:  T(ok V(fy <targ> <sub> =>
```

... depending on the value of the Token default environment parameter. The metasymbol <n> denotes the repeat factor value passed to R(eplace.

R(eplace replaces <n> occurrences of the target string in the text with the contents of the substitution string, starting at the current cursor position and moving in the current direction. If the repeat factor is "/", all occurrences of the target string are replaced.

The verify option ("V(fy") permits the examination of each occurrence of the target string prior to its replacement; it is specified (in the same fashion as the Token and Literal modes - see section 4.9) by typing the letter "V" within the prompt.

When V(erify mode is used, each occurrence of the target string found in the text is displayed on the screen, and the following prompt appears:

>Replace: <esc> aborts, 'R' replaces, ' ' doesn't

Typing an "R" replaces the string. Typing a space spares the current target occurrence from replacement.

In V(erify mode, the repeat factor applies to the number of times a target occurrence is found, not the number of times it is replaced.

If the specified number of target occurrences is found, the cursor is positioned after the last replaced string; otherwise, the following prompt appears:

ERROR: Pattern not in file. Please press <spacebar> to continue.

Typing <esc> while entering the string parameters exits the R(eplace command.

See section 4.9 for more details on using R(eplace.

Example of using R(eplace:

We will attempt to make the sentence in this example more palatable by replacing the string "rutabagas". The cursor is located at the start of the line.

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Chilled rutabagas are delicious when served with whipped cream.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

The R(eplace command is invoked with a target string of "rutabagas" and a substitution string of "strawberries":

```
>Replace[1]:  L(it V(fy <targ> <sub> =),rutabagas,,strawberries,
```

The string is replaced and the cursor is moved to this position:

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Chilled strawberries_are delicious when served with whipped cream.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

4.10.11 S(et

Prompt:

>Set: M(arker E(nvironment <esc>

Markers enable arbitrary cursor positions in a text file to be easily accessible from anywhere within the file; they are described in detail in section 4.6. Marker setting is described in section 4.10.11.0.

The editor's environment maintains text file information that is stored separately from the text. The environment is used to display and/or modify editor variables which control the editor's operation or aid the user in editing a text file. The environment is described in section 4.10.11.1.

4.10.11.0 S(et M(arker

Prompt:

Name of marker?

Marker names may contain up to eight characters; they are terminated by typing a <return>.

A maximum of ten markers is permitted in a file at any one time; attempts to set an eleventh marker generate the following prompt:

```
Marker ovflw.
Which one to replace.
0)  <marker name>
1)  <marker name>
  ...
9)  <marker name>
```

Typing a number between 0 and 9 removes the associated marker definition to make room for the new marker.

See section 4.6 for details on markers.

4.10.11.1 S(et E(nvironment

Prompt:

```
>E(nvironment: (options) <etx> or <sp> to leave

     A(uto indent              True
     F(illing                  False
     L(eft margin              0
     R(ight margin             79
     P(ara margin              0
     C(ommand ch               .
     T(oKen def                True

     3120 bytes used, 12345 available

     Patterns:
     target = 'xyz',  subst = 'abc'
```

NOTE - Some of the values shown in this example are arbitrary; they vary from file to file. However, the environment parameter values displayed above are the editor's default values. Though not shown in this example, any existing marker names are displayed.

4.10.11.1.0 Environment Parameters

Environment parameters affect the behavior of some edit commands; particularly I(nsert, M(argin, F(ind, and R(eplace (see the sections describing these commands for more details). Parameter values are changed in the environment by typing the parameter's displayed command character.

The parameters are one of three types: boolean, character, or integer. Boolean parameters are changed merely by typing "T"/"t" or "F"/"f", while character parameters are changed by typing a character; neither of these types require a termination character to complete the prompt. Integer parameters accept a string of digits and are terminated by typing <space> or <return>.

A(uto indent - affects I(nsert. It is a boolean parameter with default value "T".

F(illing - affects I(nsert and M(argin. It is a boolean parameter with default value "F".

L(eft margin
R(ight margin
P(ara margin - affect I(nsert, M(argin, and A(djust. These are integer parameters; values should be between 0 and 84. Default values: L(eft - 0, R(ight - 79, P(ara - 0.

C(ommand ch - affects I(nsert and M(argin. It is a character parameter with default value ",". See section 4.10.7.0 for more information.

T(oken def - affects F(ind and R(eplace. It is a boolean parameter with default value "True". See section 4.9.2 for more information.

4.10.12 V(erify

Redisplays the text window and repositions the window in order to center the cursor on the screen.

NOTE - This command is especially useful in rare situations where the editor is not displaying the cursor in the position it thinks it is in; V(erify usually knocks it back to its senses.

4.10.13 eX(change

Prompt:

eXchange: Text (<bs> a char) [<esc> escapes; <etx> accepts]

Replaces characters in the text file with characters typed in, starting from the current cursor position.

Exchanged text may be backspaced; the original text reappears. Typing <esc> aborts eX(change with no changes made to the original text, while typing <etx> accepts the changes made to the file. The cursor is left at the end of the exchanged text.

NOTE - eX(change does not allow typing past the end of the line or typing a <return>. It is not affected by direction.

Example of using eX(change:

This is the original text (the cursor position is underlined):

```
- - - - - - - - - - - - - - - - - - - -
Boy, I just love this rutabaga pie!!
- - - - - - - - - - - - - - - - - - - -
```

After typing "xdocumentation<etx>", the sentence now appears as:

```
- - - - - - - - - - - - - - - - - - - -
Boy, I just love this documentation!
- - - - - - - - - - - - - - - - - - - -
```

4.10.14 Z(ap

Deletes all text between the start of the previously F(ound, R(eplaced or I(nserted text (Known as the "equals marker" - see section 4.7 for details) and the current position of the cursor.

NOTE - Z(ap is designed to be used immediately after a F(ind, R(eplace, or I(nsert; it should not be used in any other situations.

If more than 80 characters are to be deleted, a prompt is posted to verify the operation. The results of a Z(ap are normally saved in the copy buffer for possible later use; however, if a Z(ap deletes more text than can fit in the buffer, the user is notified with a prompt and asked to verify the command.

4.11 Editor Problems

This section describes some problems that arise from regular use of the editor.

4.11.0 Buffer Overflow

When a text file is too large to fit in available memory, the editor displays the message "Buffer overflow" while reading in the file, and then proceeds to operate in its normal fashion. Unfortunately, the text file in memory is a truncated version of the text file; all text at the end of the file that would not fit into the editor buffer is not present.

This is a serious problem if a large-file editor is not available to split the text file into smaller files; the regular editor is helpless in this situation.

Unless text files are originally created either with a large-file editor or on a system with more memory, the editor normally cannot produce a text file that is too large to edit in a later edit session. See the following sections for details.

4.11.1 Writing Out the File

At the end of an edit session, while the editor writes the modified text file from its buffer to a disk file, the message "Error in writing out the file - type <space> to continue" sometimes appears; typing <space> returns the user to the editor with no updating to disk performed. This message can arise from many different error conditions. The most common are:

a) The output file name was invalid.

b) There is insufficient space on the specified disk volume to hold the output file.

c) The text file has grown too large for the editor to handle.

Though they generate identical warning messages, these problems are quite distinct and must be handled differently. The following sections describe some handy solutions, along with pitfalls arising from user actions contrary to those dictated by the solutions.

4.11.1.0 Invalid File Names

Once identified, this problem is easy to solve. The editor attempts to open a disk file (for the output file) with the file name specified by the user. The file system responds with an I/O error that is mapped into the editor's standard error message. The typical problem is either an incorrect volume name or a file name that is too long; once this is confirmed by inspection of the file name, it is sufficient to Q(uit W(rite from the editor with a

correct file name.

4.11.1.1 Insufficient Space on Volume

This problem is trivial if multiple disk volumes can be placed online simultaneously; if the specified disk volume lacks the necessary disk space, it is sufficient to Q(uit W(rite the text file to an online volume which can spare the disk space.

The problem can be serious (see the warning below) if the possiblity exists of unmounting the disk volume containing the editor's code file in order to mount a volume having the disk space needed for the output file. Systems having only one drive are an obvious example, but the problem is more subtle on multiple-drive systems if the editor's resident volume is unknown.

WARNING - If it becomes necessary to unmount the editor's resident volume, a specific sequence of actions is required - the price of nonconformance with this sequence is the loss of all work done during the edit session via a system crash. Here is the required command sequence: The error message has appeared, and the space bar is typed to return the user to the editor. The user must type Q(uit W(rite BEFORE removing the editor's disk volume. At this point, the editor is waiting for the name of the output file; it is now safe to replace the editor's volume with another volume, specify the output file name using the new volume's name, and type <return> to start a successful disk write.

4.11.1.2 File Too Large

Text files become too large during an edit session by an overabundance of insert and copy operations. The editor has three methods of notifying the user of its buffer status: the used and unused space listed in the environment, rather devious prompting behavior during an I(nsert, and the output file message described in section 4.11.1. These are described in the following paragraphs.

It is wise to periodically examine the number of bytes left in the edit buffer (displayed in the environment). When a text file gets down to about one thousand (1000) unused bytes, the user should split the text file into two smaller files before adding more text.

When the editor gets below a thousand unused bytes, it begins to have some trouble managing the text file. In the I(nsert command, the prompt "Please finish up the insertion - type <space> to continue" starts appearing when the first character is typed; the underlying problem can be confirmed by checking the number of unused bytes in the environment. Once confirmed, it is high time to split the text file.

If the warnings described in the past two paragraphs go unheeded, the editor does not complain until the file is written to disk; then, "Error in writing out the file" appears. At this point, the user must delete enough of the text file in memory to enable the

remainder to be written to disK.

V. THE COMPILER

This chapter describes compiler operation from the system user's point of view. Compiler usage is described in section 5.1. System-level problems encountered during compilation are described in section 5.2.

The UCSD Pascal language implementation is described in the Programmer's Manual.

5.0 Introduction

The compiler is a one-pass recursive descent compiler for the UCSD Pascal language. It is based on the P2 compiler developed at ETH Zurich.

The compiler reads a text file containing a Pascal program, and produces a code file (containing P-code) and an optional text file (containing a program listing). The code file is executable if the program does not reference separately compiled library routines; otherwise, the code file (containing a mixture of P-code and linker information) must be linked before it may be executed. Library routines and linking are described in chapter 6. P-code and linker information are described in the Architecture Guide. Program listings are described in the Programmer's Manual.

The following sections contain passing references to compiler options; because these options are set by directives embedded in Pascal programs rather than by compiler prompts, they are described in the Programmer's Manual.

5.1 Using the Compiler

The compiler is invoked from the system prompt by typing C(ompile. Typing R(un invokes the compiler if the work code file doesn't exist.

5.1.0 Setting Up Input and Output Files

If a work file exists, the compiler uses it as the input file, and names the output file "*SYSTEM.WRK.CODE[*]"; otherwise, the following prompt appears:

```
Compile what text?
```

The specified input file name should not contain the suffix ".TEXT"; it is automatically appended by the compiler unless the file name ends with a period (which is stripped off).

The next prompt asks for the output file name:

```
To what codefile?
```

Typing <return> causes the output file to become the work code file. Typing "<esc> <return>" aborts the C(ompile command. A "$" in the output file name is substituted with the input file title; thus, compiling "STUFF" to "$1" names the output file "STUFF1.CODE".

If an output file name is specified, it should not have the suffix ".CODE"; it is automatically appended by the compiler unless the file name ends with a period (which is stripped off). Length specifiers are sometimes necessary in the output file name - see section 5.2 for details.

5.1.1 Console Display

During compilation, a running account of the compiler's progress is written to the console; however, this can be inhibited by a couple of methods: the "quiet" compile option can be asserted, or a program listing may be directed to the console by the "list" compile option. The former leaves the screen blank during compilation, while the latter uses the screen to display the program listing.

NOTE - On CRT terminals, suppressing the console display speeds up the compiler approximately thirteen percent.

Example of a console display:

```
PASCAL Compiler [III.H0]
--> SYSTEM.WRK.TEXT
<   0>.........................................
LAINIT [4710]
<  43>.........
GETFILE [4692]
<  52>...................
WRITEIT [4674]
<  71>..............
NEWLINE [4634]
<  84>.....................................................
< 134>.....................................................
< 184>.........
COPYIT [4616]
< 192>...............
SEND [4627]
< 205>......

211 lines, 6 secs, 2110 lines/min
Smallest available space = 4616 words
```

The compiler's release version is delimited by square brackets at the start of the display. The name of each routine in the program is displayed; the adjacent number delimited by square brackets indicates the current amount of memory available (# of words). Numbers delimited by angle brackets indicate the current line number in the source program. Each dot represents one source line compiled. The file name following the symbol "-->" indicates a new source file. A file name following the symbol "--" indicates the current source file.

5.1.2 Syntax Error Handling

If the compiler detects a syntax error, the current source line is printed on the screen; the symbol causing the error is pointed at by "<<<<". Below this, the following prompt is displayed:

Line <n>, error <m>: <sp>(continue), <esc>(terminate), E(dit

... where <n> is the current source line, and <m> is the error number.

Typing <space> skips the erroneous symbol and resumes compilation. Typing <esc> aborts the compiler and returns control to the system prompt. Typing "E" automatically invokes the editor. If the current input file is not the work file, the editor first prompts for the name of the current input file. Once the file is specified, the editor reads it in, positions the cursor over the error, and prints the error number or message.

A list of syntax error numbers and their corresponding error messages is provided in Appendix D.

NOTE - If the wrong input file is given to the editor, the editor reads in the file and positions the cursor where the error would be if the correct file were read in (see section 5.2.1).

When the "list" compile option is asserted, syntax error messages are also written to the listing file. However, if both the "list" and "quiet" compile options are asserted, error messages are only written to the listing file; compilation continues without interruption, as no error message or prompt is displayed on the console.

NOTE - If syntax errors are detected in the program, the compiler does not produce an output code file.

5.2 Compiler Problems

This section describes strictly system-related problems caused by using the compiler. Problems concerning the correct compilation of Pascal programs are described in the Programmer's Manual.

5.2.0 X(ecuting the Compiler

The compiler can only be invoked by typing C(ompile. Unlike the other system parts (which are X(ecutable as regular code files), the compiler must remain as the system file "SYSTEM.COMPILER" in order to set up its input and output files correctly. Attempts to X(ecute the compiler's code file unfailingly result in the compiler issuing syntax error 401 at the start of compilation.

NOTE - U(ser restart also does not work with the compiler.

5.2.1 Syntax Errors and the Editor

In some situations, the communication between compiler and editor (described in section 5.1.2) seems muddled after syntax errors; the editor positions the cursor in a location far removed from the actual site of the error.

This problem arises when a Pascal source program is spread across a number of text files that are "included" into the compiler's input file (see the Programmer's Manual for details). For reasons discussed below, the editor reads in a file other than the current input file, and places the cursor at the file position set by the compiler - i.e., the right place in the wrong file.

This can occur by explicitly typing the wrong file name into the editor's prompt - it is the user's responsibility to keep track of the current input file (the console display provides this information). However, if the program being compiled resides in the work file and includes other files, the editor always enters the work file after a syntax error. This is incorrect if the error occurs in an "include" file; worse, it cannot be prevented by user actions. The only way around this problem is to avoid using the work file when a program uses "include" files.

5.2.2 Insufficient Memory

Compiling large programs may cause the system to "stack overflow". Programs containing a large number of identifiers use large amounts of memory during compilation - sometimes more than the system can provide. Here, in increasing order of severity, are some ways to avoid running out of memory:

1) Make a four-block data file named "SYSTEM.SWAPDISK" on the system volume. This can save one thousand words of memory during disk directory accesses; directories are accessed while opening "include" files for compilation.

2) Assert the "swapping" compile option. This can save four thousand words of memory, but the compile speed is cut in half.

3) Reorganize the program to minimize memory usage. Minimize the use of global variables and/or divide the program into separately compiled units (see the Programmer's Manual for details).

4) Buy more memory!

5.2.3 Insufficient Space on Volume

When the compiler is directed to write a program listing to a disk file, the output code file competes for disk space with the program listing file - adversely, in some circumstances. Here is a typical scenario:

The program listing file and output code file are to be written to the same disk volume, which has a single area of available disk space. The output code file is opened first, with a default length specifier of "*"; it reserves one half of the available disk space. The listing file is opened next, entitling it to the rest of the disk space. (Note - these defaults are assigned by the operating system and compiler - not the file system).

Unfortunately, program listing files are usually much larger than their corresponding code files; if the listing file needs any more than half of the total available space to be completed, compilation aborts because of a "no room on vol" error from the file system. By adding an explicit length specifier to the file name entered at the compiler's output file prompt, the user can limit the amount of disk space allocated for the code file, and thus maximize the amount of disk space available for the listing file.

VI. THE LINKER

The linker links together separately compiled programs and library routines. It produces an executable code file containing the host program's code and a copy of each library routine referenced by the host program. Section 6.0 introduces the concept of separate compilation; the description of library routines and library files provided in this section is sufficient for using the linker. Section 6.1 describes how to use the linker. Problems encountered during regular use of the linker are described in section 6.2.

NOTE - The utility programs Library and Libmap perform tasks related to linking (see section 9.2 for details).

6.0 Separate Compilation

Separate compilation, also known as "external compilation" or "modular programming", allows programs to be created from individually compiled parts. Here are some advantages arising from separate compilation:

1) New parts can be written, compiled, and combined with existing parts to create new programs. The new parts themselves might later be used in other programs; thus, a growing catalog of useful software parts becomes available for use in general software development.

2) Large programs constructed from separate parts are easily modified; changes are isolated to individual parts, allowing fast and reliable program maintenance.

3) By breaking them into separately compiled parts, programs can be developed that are larger than could be compiled in one piece by the system.

Library routines are created in UCSD Pascal with the "unit" construct. A unit is defined as a collection of routines and data that is accessable to programs. Units are stored in libraries; at the system level (as opposed to the program level), a unit is addressed by the file name of the library containing it. Units are described in section 6.0.0. Libraries are described in section 6.0.1.

NOTE - This section provides only a system-level description of units. The Programmer's Manual describes how to construct units and use them in programs.

6.0.0 Units

A unit contains a group of related routines and data; part of the group is exported for use by programs, while the rest remains hidden inside the unit. Programs import all of a unit's exported routines and data by using the unit. Units are stored in libraries; within a library, they are addressed by unit name.

6.0.1 Libraries

A library is a non-executable form of code file containing between one and sixteen units. Libraries are created by the compiler and managed with the Library utility (section 9.2.0); they are addressed by their file name. The file name "*SYSTEM.LIBRARY" denotes the system's default library; units residing in this file do not require a library name to be located by the system. Both the compiler and linker reference library files when a program uses units.

6.1 Using the Linker

If the work file is being used for program development, the R(un command automatically invokes the linker if the compiled work file requires linking. Moreover, rather than prompting for a library file name, the linker automatically searches the file *SYSTEM.LIBRARY for the referenced units; if they are not present, the linker aborts with an error message. Therefore, the user must manually link his files in the following cases:

1) The program requiring linking is not in the work file.

2) The units required for linking reside in library files other than SYSTEM.LIBRARY.

The linker is invoked manually by typing L(ink. The following prompt appears:

Host file?

The host file is the code file containing a program which references units residing in one or more library files. Typing <return> or "*<return>" specifies the work file as the host; otherwise, the linker appends the ".CODE" suffix to the file name unless it ends with a period (which is stripped off). The Linker then prompts for the library files containing the required units:

Lib file?

Up to eight library file names may be entered; the prompt reappears until <return> is typed. Typing "*<return>" specifies the file *SYSTEM.LIBRARY as one of the library files. The linker prints "Opening <lib file name>" after successfully opening each library file.

After the library files have been specified, the following prompt appears:

Map name?

Typing <return> skips the map file option; otherwise, a map file is created with the specified file name. The ".TEXT" suffix is automatically appended to the name unless it ends with a ".". The map file lists linker information used to resolve the procedure and data references between host and unit (see chapter 9 - Libmap for a description of map files and the Architecture Guide for a description of linker information).

The linker then prompts for the output file name:

Output File?

Typing <return> writes the output file to the work code file; otherwise, the file is written to the specified file name.

NOTE - this prompt does not append a ".CODE" suffix; the user must explicitly type this suffix to create an executable code file.

After the output file is specified, linking commences. During linking, the names of the host and each used unit is printed.

Linking is aborted if any required units are missing or undefined; the following message appears: "Unit <identifier> undefined".

6.2 Linker Problems

Unlike other file name prompts in the system, linker prompts do not recognize escape characters. To abort the linker from one of its file name prompts, type "<escape><return>". The linker attempts to open a file named "<escape>", fails, and issues this prompt: "type <sp> (continue), <esc> (terminate)". Typing <escape> then aborts the linker.

VII. COMMAND FILE INTERPRETER

The command file interpreter is used to automate system operation; it reads a command program from a text file (known as a "command file"), translates the program into a series of system commands and input data, and queues the commands and data in the Keyboard type-ahead buffer for eventual use by the system. Command interpreter operation and command file names are described in section 7.0. Command language syntax is described in section 7.1. Examples of command programs appear in section 7.2. The file "X.DEMO" is a command file that presents an overview of the command interpreter.

7.0 S(ubmitting Command Files

Typing S(ubmit from the system prompt automatically executes the code file "X.CODE" residing on the system volume; this file contains the command interpreter. The following prompt appears:

```
Filename?
```

The specified command file name must not contain the file suffix ".TEXT".

The command interpreter also accepts "targets" as valid responses to its file name prompt; targets specify a command file and the label or line number within the command program where execution should commence. Targets are described in section 7.1.1.

Typing <return> aborts the command interpreter and returns control to the system prompt.

7.0.1 Command File Execution

If the command interpreter discovers an error in a command program, it halts without notifying the user of the problem; control is returned to the system prompt. If a command program contains an infinite loop, the command interpreter must be halted by rebooting the system.

When the execution of a command program finishes, its output is queued in the Keyboard type-ahead buffer (as if it had been typed from the Keyboard), and the command interpreter terminates. Control is returned to the system prompt, but the type-ahead buffer contains queued input; the system then begins to read characters out of the type-ahead buffer and process them as system commands and data.

NOTE - the Keyboard type-ahead buffer contains a maximum of 64 characters.

WARNING - Command files are written with the assumption that the various system parts behave in a predetermined fashion; i.e., that

the order of commands and data in the type-ahead buffer match the order of the generated promptlines. If an unexpected system condition causes an unplanned-for prompt to appear, the queued commands and data may lose their synchronization with the system prompts; chaos then presides until the type-ahead buffer is emptied. It is theoretically possible for the resulting series of randomly generated commands to destroy the contents of online disk volumes. The user can terminate out-of-control command files by typing <ctrl-X>; this clears the type-ahead buffer of all queued characters.

7.0.2 Reserved Command File Names

Two command file titles are reserved by the system for special uses: "PROFILE" and "$EXEC". A command file named "PROFILE.TEXT" is automatically S(ubmitted when the system is bootstrapped. A command file named "$EXEC.TEXT" is automatically submitted when the S(ubmit command is invoked.

NOTE - Automatic execution of "$EXEC" may be subverted by typing ahead a command file name after typing S(ubmit. If the command interpreter detects characters queued in the type-ahead buffer, it will use them to build a command file name rather than opening "$EXEC".

NOTE - "PROFILE" and "$EXEC" are expected to reside on the prefixed volume.

WARNING - The file title "$EXEC" causes problems in the filer, as it violates the restriction on using the "$" character in a file name. The best way to change a command file title to/from "$EXEC" is to edit the file and write it out with the desired file name.

7.1 Command Language

The command language described in this section is named "eXec". An eXec program is stored as a series of commands and labels in a text file; a single text line contains at most one eXec command or label. Command lines start with a reserved command word; all other lines are treated as comments. Commands are described in section 7.1.0. Commands take either "targets" or "textlines" as arguments. Targets are used as arguments by the flow-of-control commands; they are described in section 7.1.1. Textlines contain text that is either immediately written to the screen or queued in the type-ahead buffer; they are described in section 7.1.2.

When dealing with alphabetic characters, the command interpreter is case-insensitive for commands and labels; however, case is preserved for screen I/O.

Blank characters are usually ignored by the command interpreter, with the following exceptions:

Blanks are significant after these commands: READ, WRITE, WRITELN, and T.

Blanks should not occur in targets.

7.1.0 Commands

Commands must appear as the first token on a text line. Commands may be classified by their time of "execution":

Immediate commands (READ, WRITE, CALL, etc.) cause the command interpreter to execute the command upon processing the line.

Deferred commands (STK, S, RUN) cause the command interpreter to save characters for subsequent use by the system.

7.1.0.0 Immediate Commands

WRITE

Form: WRITE <textline>

Writes <textline> to the console (without writing <return>).

WRITELN

Form: WRITELN <textline>

Writes <textline><return> to the console.

T

Form: T <textline>

Synonymous with the WRITELN command, but allows a longer "textline" argument because of its abbreviated form.

READ

Form: READ <textline>

Writes <textline> to the console; then, reads text from the keyboard until <return> is typed. The text read is stored in an interpreter variable named "Answer"; its contents are accessable with the special character "?" (described in section 7.1.2).

GOTO

Form: GOTO <target>

Command interpretation continues at <target>.

CALL

Form: CALL <target>

Command interpretation continues at <target>, but returns to the command following the CALL after a RUN command is executed.

VERBOSE

Form: VERBOSE

Verifies each command before executing it; the command is written to the console, and the user may type either <return> to execute it or <escape><return> to abort the command interpreter. VERBOSE is used to debug command programs.

QUIET

Form: QUIET

Disables the VERBOSE command.

7.1.0.1 Deferred Commands

STK

Form: STK <textline>

Saves <textline> on the command interpreter's internal stack.

S

Form: S <target>

STKs a S(ubmit command for <target>.

RUN

Form: RUN

If CALL commands are extant, command interpretation continues at the command following the last CALL; otherwise, RUN puts all text saved on the command interpreter's internal stack into the system's type-ahead buffer, and terminates the command interpreter.

7.1.1 Targets

Form: <target> ::= [<filename>] ["/<label>" or "\<line#>"]

Targets are used as arguments to the GOTO and CALL commands; they indicate the location in a command file where command interpretation is to continue. Targets denoting a specific location within a command file contain either a zero-origin line number (e.g., "\004") or a label (e.g., "/beginloop") which is the first token on a line.

NOTE - Care must be taken to ensure that labels have names distinct from command names. For instance, "shell" is not a valid label; it is interpreted as s<target>, where <target> = "hell".

Targets can specify locations in other command files with the optional file name field; e.g., "profile/subroutine". File suffixes must not be used in the file name. If only the file name field is specified, command interpretation continues at the first line in the named command file.

NOTE - Targets may also be used in the command interpreter's initial file prompt to specify the location in a command file where interpretation is to commence.

7.1.2 Text Lines

Within "textline" arguments, key commands are prefixed with the escape character "!"; they are denoted as follows:

" "	<space>	"r"	<return>
"!"	! (single "!")	"b"	<bs>
"^"	<up>	"e"	<escape>
"v"	<down>	"d"	<delete>
"<"	<left>	"t"	<tab>
">"	<right>		

Two special characters definitions have special properties: "k" and "?". An occurrence of "!?" in a textline is substituted with the text read in by the last READ command.

WARNING - Occurrences of "!?" are replaced with garbage if no READ command is performed beforehand.

The special character "!k" should only be used in textlines passed as arguments to the STK command. All occurrences of "!k" are replaced by special tokens as they are put in the type-ahead buffer. Later, when the system encounters one of these tokens while reading characters from the type-ahead buffer, it requests direct keyboard input until a <null> is typed, and then resumes reading from the type-ahead buffer. Thus, a series of queued system commands and data can be punctuated with requests for input directly from the keyboard, allowing automated tasks to possess interactive capabilities. (See the example in section 7.2).

7.2 Example eXec Programs

Example from command file "X.DEMO":

```
- - - - - - - - - - - - - - - - - -
writeln line 0 executing
s /target
run

target
writeln target executing
writeln calling /t2
call /t2
writeln /t2 returned
writeln going to /t3
goto /t3

t2
writeln /t2 running
run

t3
writeln /t3 gone to
writeln
read Enter Text :
writeln You Typed "!?"
writeln
writeln end of test
run
- - - - - - - - - - - - - - - - - -
```

Example of listing a disk directory:

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
t
t Once S(ubmitted, this program runs forever...
t
loopstart
read directory listing of what volume?
stk f e !? !r ! ! ! q
s /loopstart
run
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

This command program repeatedly prompts for a volume name, invokes the filer, lists the directory of the specified volume, and returns to the system prompt. The three blanks are added in case the directory listing is longer than the screen; otherwise, the blanks are consumed by the filer's promptline. Note that the title message is printed only once; subsequent invocations of the command file jump to the label "loopstart". Note also that the command interpreter automatically expands the specified target to include the name of the enclosing command file.

Another example of listing a disK directory:

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
stK f e !k !r ! ! ! q
run
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

In this example, the volume name is not specified until the actual filer prompt is displayed; at this point, the system requests direct input from the Keyboard (bypassing the queued <return>, three blanKs, and "q"). The volume name must be terminated by typing <null>. The listing is then made and control is returned to the system prompt.

VIII. SYSTEM MONITOR

The system monitor is named HDT, short for "Hexadecimal Debugging Tool". HDT is capable of: examining and modifying the contents of memory words and I/O device registers, starting/suspending/resuming system operation, and recovering from power failures.

HDT does not display a promptline; instead, the prompt character ("#") is printed on the console. HDT commands are described in section 8.1. Examples of using HDT appear in section 8.2.

NOTE - HDT is implemented as a Pascal program resident in PROMs. Its code occupies memory addresses F400-F7FF hex. Its data occupies memory in 100-200 hex and 22-25 hex; using HDT to modify the contents of these areas disrupts monitor operation and thus is not recommended. The Hardware Reference Manual describes the memory layout of the PDQ-3 system, including memory addresses reserved for I/O devices and other system functions.

8.0 Entering The Monitor

HDT is activated in these situations:

1) Pressing the RESET button on the front panel.

   HDT prompts for a command. Typing "R" causes HDT to boot the system from the system volume. The PDQ-3 can be configured to automatically boot the system after RESET is pressed - see the Hardware User's Manual for details.

2) System power-up.

   HDT checks for a power fail restart in progress. If a restart is in progress (and battery backup exists for the system memory), HDT restarts the system at the point where a power failure interrupted it; otherwise, HDT acts as if the RESET button was pressed.

3) Typing the monitor key (<control-P>) during system operation.

4) Calling the predefined procedure HALT in a Pascal program.

HDT is invoked as a high priority process, suspending normal system operation; HDT then prompts for a command. During monitor operation, all interrupts are latched and any outstanding I/O operations continue. System operation is resumed by typing "P".

8.1 Monitor Commands

HDT commands examine and modify memory contents, boot the system from the system volume, and resume execution of a currently suspended system or user program. All numbers used in HDT are hexadecimal (hex digits: 0..9, A..F); all memory addresses are word addresses; all data quantities are 16-bit words. Hex numbers are entered as a string of hex digits; if a number contains more than four digits, only the last four are significant.

HDT commands are all single key commands; lower-case alphabetic characters are mapped into their upper-case equivalents. Commands and numbers are echoed on the console as they are typed. Typing an invalid command or number causes HDT to print "?" and redisplay the prompt character.

The commands are:

R

Form: R

Reboot the system from the system volume. Invoking this command when the system volume is not mounted causes HDT to continually retry until the volume is mounted.

P

Form: P

Resume execution of a suspended user or system program. Invoking this command if a program is not currently suspended halts the monitor.

/

Form: [<number>]/

Set current address.
Display contents of current address.

If <number> is typed, it becomes the current address. HDT then displays the contents of the current address.

<return>

Form: [<number>]<return>

Set contents of current address.
Redisplay prompt.

If <number> is typed, it is stored into the word at the current address. HDT then displays the prompt character. No warnings are generated for invalid memory writes; e.g., storage into ROM.

<line feed>

Form: [<number>]<line feed>

Set contents of current address.
Increment current address and display contents.

If <number> is typed, it is stored into the word at the current address. HDT then increments the current address, and displays the contents of the current address.

^

Form: [<number>]^

Set contents of current address.
Decrement current address and display contents.

If a number is typed, it is stored into the word at the current address. HDT then decrements the current address, and displays the contents of the current address.

@

Form: [<number>]@

Set current address indirect and display contents.

If the number is typed, it is stored into word at the current address. HDT then sets the current address to the contents of the current address, and displays the contents of the current address.

8.2 HDT Examples

In the following examples, the user's responses are underlined.

Starting the system with the system disk mounted:

```
#R
```

Zeroing memory locations 2000-2002 hex:

Memory beforehand:

```
#2000/2937 <line feed>
2001/A1A1 <line feed>
2002/ABCD <line feed>
2003/FEFE <return>
#
```

Zeroing memory:

```
#^
2002/ABCD 0^
2001/A1A1 0^
2000/2937 0<return>
#
```

Memory afterwards:

```
#/0000 <line feed>
2001/0000 <line feed>
2002/0000 <cr>
#
```

Chaining through memory pointers starting at 1000 hex:

```
#1000/234E @
234E/3EFC @
3EFC/0000 1000@
1000/234E <return>
#
```

IX. UTILITIES

The programs described in this chapter perform useful system functions; they are known as "utility programs". Unlike the system parts described in the previous chapters, utility programs are invoked as user programs with the Xecute command.

9.0 Disk Management

This section describes the utility programs used to manage disk media: Booter, Backup, Mapper, Format, and Bad.blocks.

Booter copies the bootstrap software from one disk to another. Track 0 and disk blocks 0 and 1 can contain bootstrap code required for bootable system disks. Booter is described in section 9.0.0.

Backup copies entire disk images from one disk to another. Its most common use is to make backup copies of disks containing valuable data. Backup is described in section 9.0.1.

Mapper converts entire disk volumes to different disk formats, thus allowing floppy disks to be read by UCSD Pascal systems running on different machines. Mapper is described in section 9.0.2.

Format writes formatting information on blank disks so they may be used on the PDQ-3 system. Format is described in section 9.0.3.

Bad.blocks performs high-speed scanning of disks for bad blocks; it is described in section 9.0.4.

9.0.0 Bootstrap Copier

The utility program Booter (BOOTER.CODE on the utilities disk) copies bootstrap information (i.e., all of track 0 plus blocks 0 and 1) from a source volume to a destination volume.

9.0.0.0 Using Booter

X(ecute BOOTER. The following prompt appears:

```
Copy Boot From #4: to #5:  ?
<cr> to Copy, <esc> <cr> Exits
```

The source disk must occupy unit 4, and the destination disk must occupy unit 5. Typing <esc><return> exits Booter; typing almost any other character(s) (including <return>) starts the copy.

Booter always generates one last message before terminating:

```
Insert System Diskette in #4: and Hit <cr>, Please
```

Obey the prompt and type <return>; Booter then terminates.

9.0.1 Disk Copying

The utility program Backup (BACKUP.CODE on the utilities disk) copies the entire contents of a disk volume (called the "master" or "source" volume) onto another disk (called the "backup" or "destination" volume). Although there are other ways to copy disks (e.g., the T(ransfer command in the filer), Backup has the following features:

1) Backup checks that the backup volume is an exact copy of the source volume by repeatedly reading the finished copy and comparing its contents with those of the source volume.

2) Backup copies any bootstrap information contained on the source volume.

9.0.1.0 Using Backup

X(ecute BACKUP. The following prompt appears:

```
Master in #4:  Backup in #5:  ?
```

Typing "Y" designates unit 4 as the master volume and unit 5 as the backup volume. Typing <esc> generates the exit prompt described below. Typing "N" switches the unit number assignment:

```
Master in #5:  Backup in #4:  ?
```

Typing "Y" now designates unit 5 as the master volume and unit 4 as the backup volume (<esc> is same as above).

NOTE - The following prompts assume the master is in unit 5; specifying the other case generates similar prompts, but with interchanged unit numbers.

A verification message then appears:

Master on #5: Volume <source volume name>

If the designated backup disk possesses a volume name other than "BACKUP", the following prompt appears:

Destroy #4: Volume <destination volume name> ?

Typing "N" exits the Backup program; typing "Y" prints the following message:

Backup on #4: Volume <destination volume name>
Master has <# of blocks on source volume> blocks

Backup then proceeds to copy the source volume; it writes a series of dots to the screen to indicate its progress. When copying is successfully completed, this prompt may appear (it is omitted if the backup volume's initial volume name is already "BACKUP") :

May I rename <source volume name> to BACKUP: ?

This message is potentially confusing, as the master and backup volumes have the same volume name at this point. Typing "Y" changes the backup volume's name to "BACKUP" (the master volume name is not changed).

The exit prompt then appears:

E(xit to Boot Diskette in #4 ?

Typing "E", "Y", or <esc> returns the user to the system prompt; as implied by the prompt, the system disk is assumed to be mounted. Typing "N" (or any of the remaining characters) redisplays the original Backup prompt:

Master in #4: Backup in #5: ?

... allowing a new set of disks to be copied.

9.0.2 Disk Format Conversion

The utility program Mapper (MAPPER.CODE on the utilities disk) changes floppy disk formats; this allows disk volumes to be transported between systems with different hardware configurations. Mapper operates on disks having the following standard formats: Digital Equipment (DEC), Western Digital, and PDQ-3. The contents of a source disk are written ("mapped") onto a destination disk in the format requested by the user; the source disk is not affected.

NOTE - Disks having Western Digital or DEC format can be read by the PDQ-3 without being remapped. See section 1.3.3.4 for details.

9.0.2.0 Using Mapper

X(ecute MAPPER. The following prompt appears:

```
Source  D(ec W(d P(dq :
```

The choices available are: "D", "W", "P", and <escape>. The first three specify the corresponding disk format; <escape> generates Mapper's exit prompt (described below).

NOTE - Mapper cannot verify the source disk's format; incorrectly specifying the source disk's format yields a scrambled destination disk. Mapper will not map a disk to the same format (i.e., a straight copy); use the Backup utility to do this.

The next prompt is treated similarly:

```
Target  D(ec W(d P(dq :
```

Once the source and destination formats are specified, the following prompt appears:

```
Map #4:[ <source format> ] ---> #5:[ <target format> ] OK ?
```

The choices available are: "Y", "N", and <escape>. Typing "Y" starts the mapping process; typing <escape> terminates Mapper; typing "N" generates the following prompt, which is treated similarly:

```
Map #5:[ <source format> ] ---> #4:[ <target format> ] OK ?
```

While Mapper maps, information detailing its progress is displayed in the upper right-hand corner of the screen. Typing a <blank> during mapping causes Mapper to skip the current track, and continue mapping on the next track. Typing <escape> interrupts mapping and generates the exit prompt.

NOTE - On some systems, error messages appear while mapping a DEC-formatted disk into another format; certain incompatibilities can arise while mapping track 0 of a DEC-format disk. If the error messages persist, skip track 0 by typing a <blank>. Mapping should resume without problems on Track 1.

When mapping is completed, the exit prompt appears:

```
Mapping completed
R(epeat or <cr>
```

Typing "R" restarts Mapper; typing <return> exits Mapper. Be sure to replace the system disk in unit 4 before typing <return>.

9.0.3 Disk Formatting

The utility program Format (FORMAT.CODE on the utilities disk) formats floppy disks in the PDQ-3 disk format. Disk formatting is used for:

1) Preparing new disks (8" soft-sectored floppys only - we recommend Dysan disks).

2) Recycling old disks with different formats.

3) Fixing disks which have been rendered unreadable by unfortunate circumstances.

WARNING - When a disk or an area of a disk is reformatted, its original data is irretrievably lost.

9.0.3.0 Using Format

X(ecute FORMAT. The following prompt appears:

Enter unit number containing disk to be formatted [0,4,5]

Typing "0" exits Format; typing any of the other numbers generates the following prompt:

Format single or double density? (S or D)

Typing "S" specifies single density formatting; typing "D" specifies double density.

The next prompt is:

Format single or double sided? (S or D)

Typing "S" specifies single-sided disks; typing "D" specifies double-sided.

NOTE - Before choosing double density, be sure that your floppy disks can handle double density formatting. Before choosing double-sided, be sure that your disks AND disk drives support it; standard PDQ-3 disk drives do not support double-sided disks.

The next prompt is:

Skewing? (Y or N)

Typing "Y" directs Format to skew the placement of disk sectors in order to improve disk performance. Typing "N" suppresses sector skewing. See the Architecture Guide for more information on disk sector skewing.

The next prompt is:

    Format all tracks? (Y or N)

Typing "Y" initiates formatting of the entire disk; typing "N" generates the following prompt:

    Enter starting track number

The starting track number is typed in, followed by a <return>; The final track number is handled similarly:

    Enter final track number

Once the track range is specified, formatting commences. The screen displays the following messages detailing Format's progress:

    Formatting <starting track #> - <track # being processed>
    Verifying <starting track #> - <track # being processed>

9.0.3.1 Reformatting Bad Blocks

This section describes how to reformat bad blocks that cannot be fixed with the X(amine command in the filer. It is necessary to determine which tracks the bad blocks occupy; only these tracks need reformatting. Here are the formulae for determining the track and sectors used by an arbitrary block:

    (<block #> * 4 DIV 26) + 1 = <track #>
    (<block #> * 4 MOD 26) + 1 = <starting sector #>

There are four sectors per block. If the starting sector is 25, the next track should be reformatted also, for it contains the rest of the block.

NOTE - The above formulae and information are for single density disks. For double density, "4" => "2". For double-sided, "26" => "52".

NOTE - reformatting entire tracks to fix a bad block destroys the contents of adjacent blocks.

9.0.4 Fast Bad Blocks Scanning

The utility program Bad.blocks (BAD.BLOCKS.CODE on the utilities disk) checks a disk file or disk volume for damaged blocks. Bad blocks scanning can also be performed with the filer's B(ad blocks command; however, Bad.blocks is much faster. Bad blocks are repaired with the filer's X(amine command or the Format utility (section 9.0.3).

9.0.4.0 Using Bad.blocks

X(ecute BAD.BLOCKS. The following prompt appears:

File to scan?

Typing <return> exits Bad.blocks; typing a volume id (e.g. "#5:" or "MYDISK:") scans an entire disk volume; typing a file name scans a single file on a disk volume. The next prompt is:

Scan all <# blocks in file> blocks [y/n]

Typing "Y" scans all blocks occupied by the specified file; typing "N" generates this prompt:

Start scanning at block:

Type the number, followed by a <return>. The starting block number is relative to the start of the specified file; e.g., a starting block of 0 initiates bad blocks scanning on the first block of the file, even if the file itself starts at block 45 on the disk volume.
The following prompt is defined similarly:

Stop scanning after block:

Once the block range is specified, scanning begins; Bad.blocks indicates its progress by writing a series of message having the following form:

Scanning blocks <block number> to <block number>

When scanning a single disk file, the block numbers indicated are relative to the start of the file; when scanning a disk volume, the block numbers displayed correspond to the actual disk block numbers. Bad.blocks checks 40 blocks at a time.
If a bad block is detected, the following message appears:

Block <block number> is bad

When Bad.blocks is finished, it indicates the total number of bad blocks detected:

<number> bad blocks

Before terminating, Bad.blocks writes the following prompt:

Insert system disk and type <CR>

Typing <return> returns control to the system prompt.

9.1 Duplicate Directory Management

This section describes two utilities that manage duplicate directories: Markdupdir and Copydupdir.

Markdupdir (MARKDUPDIR.CODE on the utilities disk) modifies a disk volume currently maintaining only a primary directory so that it maintains a duplicate directory. This is usually done with the filer command Z(ero; Markdupdir is used to add a duplicate directory to an existing disk volume without destroying its contents.

Copydupdir (COPYDUPDIR.CODE on the utilities disk) copies the duplicate directory into the location of the primary disk directory; it is used after unfortunate circumstances destroy the main directory.

Primary and duplicate disk directories are described in section 2.1.3.5 and the Architecture Guide.

9.1.0 Using Markdupdir

X(ecute MARKDUPDIR. It first prompts for the disk drive (4 or 5) containing the volume to be marked.

If the disk volume already has a duplicate directory, the user is notified; typing <return> then exits Markdupdir. Otherwise, blocks 6-9 on the disk volume are checked to see if they are currently occupied by a disk file; if so, the user is asked to verify the mark, as the disk file would be overwritten by a duplicate directory. Typing "Y" proceeds with the marking; typing any other character exits Markdupdir.

The status of blocks 6-9 can be checked with the filer command E(xtended list. If the first disk file in the directory starts at block 6, or if it starts at block 10 and is preceded by a four-block unused area, then the disk has not been marked. However, if the first file starts at block 10 and there are no unused blocks at the beginning, the disk has been marked.

Examples of directory listings of unmarked volumes:

```
SYSTEM.PASCAL        31        10-Jan-79      6      Codefile

<unused>              4        10-Jan-79      6      Codefile
SYSTEM.PASCAL        31        10-Jan-79     10      Codefile
```

Example of a directory listing of a marked volume:

```
SYSTEM.PASCAL        31        10-Jan-79     10      Codefile
```

9.1.1 Using Copydupdir

X(ecute COPYDUPDIR. It first prompts for the disk drive (4 or 5) in which the copy is to take place.

The user is notified if the disk is not currently maintaining a duplicate directory. If a duplicate directory is found, a prompt is issued to verify that the current primary directory is to be destroyed. Typing "Y" copies the directory; typing any other character exits Copydupdir.

9.2 Library Management

Libraries are managed with the utility programs Library and Libmap (LIBRARY.CODE and LIBMAP.CODE on the utilities disk).

Library transfers units between library files. It is used to create and maintain the system library and user-defined libraries. Library is described in section 9.2.0.

Libmap lists library file information in symbolic form; among other things, it displays the units residing in a library, and the names of exportable routines and data in each unit. Libmap is described in section 9.2.1.

See chapter 6 for a system-level description of units and libraries and the Programmer's Manual for a program-level description of units and libraries.

9.2.0 Using Library

X(ecute LIBRARY. The following prompt appears:

```
Output Code File ->
```

The file name entered becomes the name of the library file produced by Library.

Typing only a <return> exits Library.

NOTE - Library does not append a suffix to the specified name; libraries function equally well as code files or data files.

The following prompt then appears:

```
Link Code File ->
```

Enter the file name of the library to be modified. Library then lists the name (and code size in words) of each unit in the library. Note that sixteen slots are shown; a library file contains a maximum of sixteen units.

NOTE - When adding units to an existing library (such as SYSTEM.LIBRARY), output and link file names can be identical; otherwise, it becomes necessary later on to remove the old library file and change the output file's name back to the original library name.

After the library's contents are displayed, this prompt appears:

Segment # to link and <space>, N(ew file, Q(uit, A(bort

Typing a displayed unit's slot number followed by <return> indicates that the unit is to be copied into the output file. Library then requests an output file slot for the unit:

Seg to link to?

After typing a slot number and <return>, library moves the unit into the output file; the other slots in the output file also appear. The remaining units in the link file are copied across in a similar manner. Library displays the current number of blocks in the output file at the bottom of the library display.

NOTE - When expanding an existing library, be sure to preserve its units by copying them into the output file before adding the new units.

N(ew file redisplays this prompt:

Link Code File ->

Enter the name of a file containing new units; as before, the units in the file are displayed, and can be copied into the output file.

NOTE - The new file is usually a code file produced by the compiler. It could be used as a library file; "merging small libraries with a larger library" is a more precise description of Library's task than "adding units to a library".

Q(uit displays this prompt:

Notice?

Up to eighty characters of text may be typed before typing <return>. The text is moved into the segment dictionary of the output file. This is used for embedding copyright notices in the library file.

A(bort exits Library; the output file is not saved. A(bort works everywhere except after typing Q(uit.

NOTE - Library can be used to view and rearrange code segments within an executable code file. See the Architecture Guide for details.

9.2.1 Using Libmap

Xecute LIBMAP. The following prompt appears:

```
enter library name:
```

Typing <return> exits Libmap; typing the file name of a library generates this prompt:

```
list linker info table (Y/N)?
```

Typing "Y" directs Libmap to print a textual representation of the linker information embedded in each unit; it also generates the next prompt:

```
list referenced items (Y/N)?
```

Typing "Y" directs Libmap to print a symbolic list of all external references contained in the linker information.

The following prompt appears regardless of the choices made for linker information:

```
map output file name:
```

Typing "#1:" or "console:" directs the listing to the console; otherwise, Libmap automatically appends ".TEXT" to the output file name.

When the map file is completed, this prompt reappears:

```
enter library name:
```

NOTE - Libmap can also be used to list the linker information and code segments of any code file.

Example of a library unit and its map listing:

Here is a UCSD Pascal unit:

```
unit mapexample; interface
  uses extraref;

  var i,j,K: integer;

  procedure map1;

implementation

  var m,n:boolean;

  procedure private;
  begin
    writeln ;
    m := true;
    n := false;
  end;

  procedure map1;
  begin
    i := 1;
    j := 2;
    K := 55;
  end;

end;
```

Here is its map listing:

```
Segment # 1:  MAPEXAMP  library unit

  uses extraref;

  var i,j,K: integer;

  procedure map1;


    MAPEXAMP unit byte reference (0 times)
    UNITVAR  public big reference (0 times)
    N        private big reference (once)
    EXTRAREF unit byte reference (0 times)
    I        public big reference (once)
    J        public big reference (once)
    K        public big reference (once)
    M        private big reference (once)
```

The segment's name, number, and type are displayed on the first line, followed by a list of the unit's exported routine and data names.

The linker information shows all external variable and unit references made by the unit. An external object's name and link type are always printed. External references display their reference format and number of references. External definitions (usually seen in host programs) display their assigned data offset.

Linker information is described in the Architecture Guide.

9.3 Terminal Configuration

This section describes the system parts used to create and maintain a standard interface between system software and the terminal. These parts enable the system to use many different terminals with a minimum of effort.

Two system parts define the system's current terminal interface: GOTOXY and SYSTEM.MISCINFO.

The operating system procedure GOTOXY implements random (i.e., X-Y coordinate) addressing of the cursor position.

The data file named "SYSTEM.MISCINFO" resides on the system volume. It contains three kinds of information: miscellaneous system data, terminal screen control characters, and key definitions for the special commands. Its contents are read into a system data structure named SYSCOM after booting or I(nitializing the system (see the Architecture Guide for details on SYSCOM). The system uses the values in SYSCOM to perform various screen control operations.

Three system parts are used to reconfigure the system's terminal interface: Config, Setup, and Binder.

The utility program Config (CONFIG on the system disk) reconfigures the system for the following terminals:

1) DEC VT-52 compatible terminals (such as the Zenith Z19).

2) Soroc IQ-120

3) VC 404

4) Teleray

Config simplifies system configuration for the listed terminals; it renames an existing data file (which already contains system information for the specified terminal) as SYSTEM.MISCINFO, and modifies the existing GOTOXY procedure in the operating system. Config is invoked by the default command program when the system is booted for the first time - see the Hardware Reference Manual for details. Config's operation is described in section 9.3.0.

Terminals not supported by Config require manual reconfiguration of the system. A new SYSTEM.MISCINFO must be created from scratch and assigned the proper terminal parameters. A new GOTOXY procedure must be written and bound into the operating system. These tasks are performed with the utilities Setup and Binder.

The utility program Setup (SETUP.CODE on the utilities disk) is used to create a new MISCINFO file. Setup is described in section 9.3.1. The utility program Binder (BINDER.CODE on the utilities disk) binds a compiled GOTOXY procedure into the operating system's code file. Details on creating, compiling, and binding a new GOTOXY are presented in section 9.3.2.

9.3.0 Using Config

A copy of the necessary MISCINFO file (SOROC, VC404, TRAY, VT52) must reside on the system volume. An online drive is assumed to contain a bootable system disk requiring reconfiguration.

X(ecute "CONFIG." (note - the "." is necessary because Config's file name lacks the ".CODE" suffix). The following prompt appears:

```
What is the destination drive [4,5,9,10]?
```

Enter the drive containing the disk to be configured. Config then displays the following menu:

```
The terminals for which ACD has constructed drivers include:

A) Zenith/Heathkit (or any VT-52 compatible terminal)
B) Soroc IQ-120
C) VC 404
D) Tray

Type the letter for your terminal ([RETURN] for neither):
```

Type the appropriate letter; typing <return> exits Config. Config reads the corresponding MISCINFO file from the system volume, and writes it to SYSTEM.MISCINFO on the destination disk. The code file on the destination disk containing the operating system (SYSTEM.PASCAL) is located, and its default GOTOXY procedure is modified to work for the indicated terminal.

If the disk is successfully reconfigured, this message appears:

```
Done.
```

If any problems occur, one or both of these messages appear:

```
File error: Configuration not done

Consult the SETUP section of the user manual for instructions.
```

These appear if <return> was typed instead of a letter - the reference to Setup is printed because Config assumes that manual reconfiguration is necessary, as none of the supported terminals was specified.

Conditions causing problems include:

```
Wrong or missing MISCINFO file on the system volume
Faulty or off-line destination disk
Destination volume is write-protected
No room on destination volume
No file SYSTEM.PASCAL on destination volume
Bad block in SYSTEM.PASCAL
```

9.3.1 Using Setup

X(ecute SETUP. Setup spends a few moments copying the contents of SYSCOM into its own buffer, and then displays the following prompt line:

SETUP: C(HANGE T(EACH) H(ELP) Q(UIT)

H(ELP describes the currently available commands.

T(EACH describes how to use Setup.

NOTE - Please ignore the section references to the WD Manual displayed in T(EACH. GOTOXY binding is described in this manual in section 9.3.2.

C(HANGE is used to display and modify screen control and special command information in Setup's edit buffer.

Q(UIT displays the following prompt:

QUIT: D(ISK) OR M(EMORY) UPDATE, R(ETURN) H(ELP) E(XIT)

D(ISK UPDATE saves the contents of Setup's edit buffer in the data file "NEW.MISCINFO". This must be changed to "SYSTEM.MISCINFO" to be used by the system.

M(EMORY UPDATE writes the contents of Setup's edit buffer to the SYSCOM data structure in memory; the new values may be tested immediately, but are lost if the system is rebooted or I(nitialized.

R(ETURN returns the Setup promptline.

E(XIT exits Setup.

9.3.1.1 Fields in Setup

This section describes the fields accessed by the C(HANGE command. The fields represent three kinds of system information: keys, characters, and parameters.

Keys map character sequences from the keyboard into the system's various Key commands (e.g. <control-F> from the keyboard is recognized as the flush command). Key fields in Setup have the word "KEY" in their field names.

Characters are character sequences that the system writes to the terminal in order to manipulate the screen display (e.g. writing the ERASE LINE character to the terminal erases the characters displayed on the current line).

Parameters are various integer or Boolean values which control the system's operation (e.g. the HAS CLOCK field is a Boolean parameter indicating the presence of a system clock).

Section 9.3.1.2 lists field values for some common terminals. The terminal functions (and related character sequences) referred to in this section should be documented in the terminal's functional specification. Key command defintions for some common terminals are listed in Appendix F.

NOTE - The ASCII character names used in some fields are defined in Appendix E.

BACKSPACE

Writing this character to the console moves the cursor one space to the left. This must be a single character. Suggested value: ASCII BS

DISK READ RATE
DISK SEEK RATE
DISK WRITE RATE

These fields were introduced by Western Digital to control the disk characteristics in their system; the PDQ-3 system does not use them.

EDITOR ACCEPT KEY

This key is used in the editor to conclude commands. save the text changes. Suggested value: ASCII ETX (ctrl-C or ctrl-J)

EDITOR ESCAPE KEY

This key is used in the editor to exit from commands. Suggested value: ASCII ESC (ctrl-[)

ERASE LINE

Writing this character to the console erases all characters on the line that the cursor is on, and positions the cursor at the start of the line.

ERASE SCREEN

Writing this character to the console erases the entire screen and positions the cursor at the top left of the screen.

ERASE TO END OF LINE

Writing this character to the console erases all characters from the current cursor position to the end of the line, and leaves the cursor at its current position.

ERASE TO END OF SCREEN

Writing this character to the console erases all characters from the current cursor position to the end of the screen, and leaves the cursor at its current position.

HAS 8510A

This should always be set to FALSE on PDQ-3 systems; it is set to TRUE only on Terak machines.

HAS CLOCK

This indicates the presence of a system clock; it should always be set to TRUE on PDQ-3 systems.

HAS LOWER CASE

This is set to TRUE if the terminal supports lower-case characters; otherwise, FALSE.

HAS RANDOM CURSOR ADDRESSING

This is set to FALSE only when using hard-copy terminals; otherwise, TRUE.

HAS SLOW TERMINAL

This field is intended for terminals operating at less than 600 baud. It is not used by the PDQ-3 system.

KEY FOR BREAK

This Key is intended to terminate the current program. It is not used by the PDQ-3 system.

KEY FOR FLUSH

This is the console output cancel Key. When the FLUSH Key is typed, console output is discarded until FLUSH is typed again or the system reads from the terminal. This field is not used by the PDQ-3 system, as the flush Key is hard-wired to (ctrl-F).

KEY FOR STOP

This is the console output stop Key. When the STOP Key is typed, the system halts on the next console output operation. This field is not used by the PDQ-3 system (see section 1.3.3.1).

KEY TO BACKSPACE

This Key moves the cursor one space to the left. Default value: ASCII BS

KEY TO DELETE CHARACTER

This Key deletes the character where the cursor is, and moves the cursor one character to the left. Suggested value: ASCII BS (control-H or "backspace")

KEY TO DELETE LINE

This Key deletes the line occupied by the cursor. Suggested value: ASCII DEL ("rubout")

KEY TO END FILE

This Key sets the Boolean intrinsic EOF to true when it is typed while reading from the predeclared files INPUT or KEYBOARD. Suggested value: ASCII ETX (control-C or "home")

KEY TO MOVE CURSOR UP
KEY TO MOVE CURSOR DOWN
KEY TO MOVE CURSOR LEFT
KEY TO MOVE CURSOR RIGHT

These keys are used by the editor for cursor control. If the terminal keyboard has a vector pad, it should be used to define these keys. Otherwise, four keys may be chosen in the pattern of a vector pad and be assigned the control codes that correspond to them (e.g., ctrl-K, ctrl-O, ctrl-;, ctrl-.).

LEAD-IN CHAR FROM KEYBOARD

Some terminals contain keys that generate two-character sequences. If the prefix character is the same for all of these keys, it is used to set the value of the field LEAD-IN CHAR FROM KEYBOARD. The PREFIX[<field name>] field for each two-character key must then be set to TRUE.

LEAD-IN TO SCREEN

Some terminals require two-character sequences to activate certain functions. If the prefix character is the same for all of these functions, it is used to set the value of the field LEAD-IN TO SCREEN. The PREFIX[<field name>] field for each two-character function must then be set to TRUE.

MOVE CURSOR HOME

Writing this character to the console "homes" the cursor; i.e., moves it to the upper left hand corner of the screen.

NOTE - If the terminal does not have such a character, the field should be set to ASCII CR ("return"); as a consequence, the editor will be unusable. Use YALOE (section 9.4) instead.

MOVE CURSOR RIGHT

Writing this character to the console moves the cursor one space to the right without erasing any characters.

NOTE - If the terminal does not have such a character, the editor will be unusable. Use YALOE (section 9.4) instead.

MOVE CURSOR UP

Writing this character to the console moves the cursor vertically up one line without erasing any characters.

NOTE - If the terminal does not have such a character, the editor will be unusable. Use YALOE (section 9.4) instead.

NON-PRINTING CHARACTER

This character is displayed whenever a non-printing character is written to the console by the editor. Standard value: ASCII "?"

PREFIXED[<field name>]

The system will recognize any two-character sequences generated by a key or sent to the console if the PREFIXED field corresponding to the appropriate field is set to TRUE. See the descriptions of the LEAD-IN TO SCREEN and LEAD-IN CHAR FROM KEYBOARD fields for more details.

SCREEN HEIGHT

The number of text lines displayable on the console. Standard value: 24 decimal. Value for hard-copy terminals: 0.

SCREEN WIDTH

The number of characters on one line on the console. Standard value: 80 decimal.

VERTICAL DELAY CHARACTER

This character is intended for implementing vertical move delays on slower terminals. This field is not used by the PDQ-3 system. The vertical delay character is hard-wired to <null>.

VERTICAL MOVE DELAY

This field can take integer values between 0 and 11. Many types of terminals require a delay after certain cursor movements to enable the terminal to complete the movement before the next character is displayed. The delay is implemented by sending a series of <null> characters to the terminal; the value in this field determines the number of characters to be sent.

9.3.1.2 Sample Setups For Some Popular Terminals

Terminals:	LSI ADM-3A	SOROC IQ120	ZENITH Z19
Fields:			
BACKSPACE	l-arrow	ctrl-H	ctrl-H
EDITOR ACCEPT KEY	ctrl-C	home	ctrl-J
EDITOR ESCAPE KEY	esc	esc	esc
ERASE LINE	NUL	NUL	l
ERASE SCREEN	ctrl-Z	"*"	E
ERASE TO END OF LINE	NUL	T	K
ERASE TO END OF SCRN	NUL	Y	J
HAS LOWER CASE	TRUE	TRUE	TRUE
HAS RAND CURS ADDR	TRUE	TRUE	TRUE
HAS SLOW TERM	FALSE	FALSE	FALSE
KEY FOR BREAK	ctrl-B	break	break
KEY FOR FLUSH	ctrl-F	ctrl-F	ctrl-F
KEY FOR STOP	ctrl-S	ctrl-S	ctrl-S
KEY TO BACKSPACE	BS	BS	BS
KEY TO DELETE CHAR	ctrl-H	l-arrow	backspace
KEY TO DELETE LINE	DEL	DEL	DEL
KEY TO END FILE	ctrl-C	ctrl-C	ctrl-C
KEY TO MOV CURS DOWN	ctrl-J	d-arrow	B
KEY TO MOV CURS LEFT	ctrl-H	l-arrow	D
KEY TO MOV CURS RGHT	ctrl-L	r-arrow	C
KEY TO MOV CURS UP	ctrl-K	u-arrow	A
LEAD IN FROM KBD	NUL	NUL	ESC
LEAD IN TO SCREEN	NUL	ESC	ESC
MOVE CURSOR HOME	ctrl-^	ctrl-^	H
MOVE CURSOR RIGHT	ctrl-L	r-arrow	C
MOVE CURSOR UP	ctrl-K	u-arrow	A
NON-PRINTING CHAR	"?"	"?"	"?"
PREF [ED ACCEPT KEY]	FALSE	FALSE	FALSE
PREF [ED ESCAPE KEY]	FALSE	FALSE	FALSE
PREF [ERASE LINE]	FALSE	TRUE	TRUE
PREF [ERASE SCREEN]	FALSE	TRUE	TRUE
PREF [ERASE TO EOLN]	FALSE	TRUE	TRUE
PREF [ERSE TO EOSCN]	FALSE	TRUE	TRUE
PREF [KEY DEL CHAR]	FALSE	FALSE	FALSE
PREF [KEY DEL LINE]	FALSE	FALSE	FALSE
PREF [KEY MV CRS DN]	FALSE	FALSE	TRUE
PREF [KEY MV CRS LT]	FALSE	FALSE	TRUE
PREF [KEY MV CRS RT]	FALSE	FALSE	TRUE
PREF [KEY MV CRS UP]	FALSE	FALSE	TRUE
PREF [MOV CURS HOME]	FALSE	FALSE	TRUE
PREF [MOV CURS RT]	FALSE	FALSE	TRUE
PREF [MOV CURS UP]	FALSE	FALSE	TRUE
PREF [NONPRINT CHAR]	FALSE	FALSE	FALSE
SCREEN HEIGHT	24	24	24
SCREEN WIDTH	80	80	80
VERTICAL MOVE CHAR	NUL	NUL	NUL
VERTICAL MOVE DELAY	5	10	0

9.3.2 GOTOXY Binding

First, a Pascal program containing the GOTOXY procedure must be written and compiled to a code file. If the system has not been configured for the terminal being used, it might be necessary to create the program with the line-orinted editor YALOE; the regular editor may be unusable.

Here is an example of a complete GOTOXY program for the Soroc IQ 120 terminal:

```
{$U-,S+}
program NewGotoXY;

  procedure SorocIQ120GotoXY(X,Y: integer);
  const
        NumChars = 4;     { chars in sequence }
        LeadIn   = 27;    { Soroc lead-in char }
        CmdChar  = '=';   { Soroc command char }
        XBias    = 32;    { Soroc X & Y bias }
        YBias    = 32;
        XMax     = 0;     { Screen Width Parameters }
        XMin     = 79;
        YMin     = 0;     { Screen Height Parameters }
        YMax     = 23;
        terminal = 1;     { Terminal I/O Unit }
        noSpec   = 12;    { No special chars }
  var
        CharSeq: packed array [1..NumChars] of char;

  begin
    if X > XMax then X := XMax
    else if X < XMin then X := XMin;
    if Y > YMax then Y := YMax
    else if Y < XMin then Y := XMin;

    CharSeq[1] := chr(LeadIn);
    CharSeq[2] := CmdChar;
    CharSeq[3] := chr(YBias + Y);
    CharSeq[4] := chr(XBias + X);

    unitwrite(terminal,CharSeq,NumChars,,noSpec);
  end {SorocIQ120GotoXY};

begin {dummy program}
end.
```

This example demonstrates most of the requirements and restrictions imposed on new GOTOXY procedures. Most terminals use similar character sequences for cursor addressing; the parameters most likely to vary are the prefix and command chars, and the biases applied to the X and Y coordinates. These should be documented in the terminal's functional specification.

The compiler directive at the top of the program is required; "U-" directs the compiler to create the program at the system level, while "S+" is merely to save space.

NOTE - Programs compiled with the "U-" directive are not executable.

The name "GOTOXY" cannot be used as a procedure or variable name; it is reserved for standard calls to the GOTOXY intrinsic. Binder recognizes the new GOTOXY procedure by its position in the code file; hence, there must be only one procedure in the program, and the main program must be an empty block.

The UNITWRITE intrinsic is used to make the GOTOXY as efficient as possible. Using the standard procedure WRITE slows down terminal response. The UNITWRITE option to suppress special character processing is asserted to prevent the system from interpreting any of the characters in the command sequence as special characters (e.g., DLE expansion).

GOTOXY must ensure that its X and Y argument values are in the proper range; if not, they must be truncated. Also, be sure that the X parameter controls horizontal cursor movement and the Y parameter vertical cursor movement. If there are any doubts about a GOTOXY, it is worthwhile to embed the GOTOXY procedure in a test program and test it out before running Binder.

9.3.2.0 Using Binder

X(ecute BINDER. The following prompt appears:

Enter name of file with GOTOXY procedure:

Enter the name of the code file containing the compiled GOTOXY. The prefix volume should be set to the volume containing the operating system code file. Binder reads the operating system code into memory, binds in the new procedure, and writes the modified system code back to the disk. Rebooting the system reloads the operating system with the new GOTOXY installed and ready for action.

NOTE - Binder removes the old operating system code file; be sure to make a copy of it on a backup disk before running Binder. Binder can be run successfully on the same file many times; however, this is not suggested, as the operating system code becomes progressively larger (and thus wastes memory). This occurs because Binder merely adds the new procedure code to the end of the existing code, and then updates the GOTOXY procedure pointer.

NOTE - A newly bound in GOTOXY is correct if:

1) The welcome message appears in the center of the screen when the new system is booted (section 2.2.0).

2) The editor seems to work correctly.

9.4 Line-Oriented Text Editor

YALOE is a line-oriented text editor designed for use in systems having a hard-copy device (e.g., teletypewriter) for a terminal, or on unconfigured systems (see section 9.3); YALOE works in these situations, while the regular editor does not.

Section 9.4.9 contains a summary of all YALOE commands.

9.4.0 Entering YALOE

YALOE is invoked by X(ecuting YALOE.CODE; however, if YALOE is to be used extensively, it can assume the role of the standard system editor. Change the screen editor's code file to a different file name (e.g. SCREEN.EDITOR), and then change YALOE.CODE to SYSTEM.EDItyping E(dit from the system prompt now invokes YALOE.

If a work file exists, the editor prints:

    Workfile <file name> read in

... where <file name> is the name of the current work file.

If the workfile is empty, this message appears:

    No workfile read in.

9.4.1 Entering Commands and Text

The editor operates in either Command mode or Text mode. The editor is in Command mode when it is first entered; in Command mode, all keyboard input is interpreted as edit commands. Commands may be invoked individually or as part of a command string specifying the execution of a sequence of commands. Text mode is entered whenever a command is typed that must be followed by a text string; when the text string is terminated, the editor returns to Command mode.

Examples of command and text strings appear in the sections describing the edit commands.

NOTE - unlike other parts of the system, YALOE does not display promptlines automatically; instead, an asterisk ("*") is printed to indicate that commands may be entered. Commands are entered by typing command characters; they are displayed on the screen as they are typed. The "?" command lists the available commands on the screen.

9.4.1.0 Command Arguments

Some edit commands allow a command argument to precede the command character. The argument usually specifies the number of times the command should be performed or the particular portion of text to be affected by the command. The definitions listed below are used in the command descriptions.

Command arguments are:

n — Any integer, signed or unsigned. Unsigned integers are assumed to be positive. In a command that accepts an argument, the default value is 1; if only a minus sign is present, the value is -1. Negative arguments imply backwards cursor movement.

m — An integer between 0 and 9.

0 — The beginning of the current line.

/ — Denotes the number 32700. A "-/" denotes -32700. "/" is used as an "infinite" repeat factor.

= — Equivalent to the signed integer argument "-n", where n equals the length of the last text string argument used. Applies only to the J(ump, D(elete, and C(hange commands.

9.4.1.1 Command Strings

Commands may be entered singly or in strings; they are not executed until <esc><esc> is typed. Command strings consist of a sequence of single character commands. Commands requiring text strings are separated by the <esc> terminating the command's text string; commands not requiring text strings may optionally be separated by <esc>.

NOTE - <esc> echoes a dollar sign ("$") when typed. The <esc> terminates the text string and returns control to Command mode. The examples in this section display <esc> in its echoed form "$".

Spaces, carriage returns and tabs within a command string are ignored unless they appear in a text string. When the execution of a command string is complete, the Editor prompts for the next command with an asterisK ("*").

If an error is encountered while executing a single command, execution of the command string is terminated; the results of the preceding commands in the string remain, but subsequent commands in the command string are discarded.

9.4.1.2 Text Strings

In Text mode, all Keyboard input is treated as text until <esc> is typed. Commands requiring text strings are F(ind, G(et, I(nsert, M(acro define, R(ead file, W(rite to file, and eX(change.

9.4.2 The Text Buffer

The text file being modified by the editor is stored in the text buffer. Files must fit in the text buffer to be successfully edited.

9.4.3 The Cursor

The cursor is the position in the file where the next command will be executed. Most edit commands use the cursor position as a starting point in their operations on the text file.

9.4.4 Special Commands

Various keys on the keyboard have special functions when used in YALOE. These commands are described below:

<esc>

Echoes a dollar sign ($) on the console. A single <esc> terminates a text string. A double <esc> executes a command string.

CTRL H
<chardel>

Deletes a character from the current line. On hard-copy terminals, it echoes a percent sign ("%") followed by the character deleted. Deletions are done right to left, with each deleted character erased by the %, up to the beginning of the command string. CTRL H may be used in both Command and Text Modes.

CTRL X

CTRL X causes the editor to ignore the entire command string currently being entered; YALOE responds with an asterisk ("*") to accept new commands. If the command string covers several lines, all lines back to the previous command prompt are ignored.

NOTE - The Operating System currently reserves CTRL X for its own purposes; this command does not work.

CTRL O

CTRL O causes the Editor to switch to the optional character set (bit 7 turned on).

NOTE - If strange characters start appearing on the terminal, CRTL O may have been accidentally typed. Typing CRTL O again should fix the problem.

9.4.5 Input/Output Commands

The commands that control I/O are: L(ist, V(erify, W(rite, R(ead, Q(uit, E(rase, and O(ption.

9.4.5.0 L(ist

Format:

nL

Prints the specified number of text lines on the terminal without moving the cursor. Variations of this command are illustrated in the examples below.

*-3L$$ Prints all characters starting at the third preceding line and ending at the cursor.

*5L$$ Prints all characters beginning at the cursor and terminating at the fifth carriage return (line).

*0L$$ Prints from the beginning of the current line up to the cursor.

9.4.5.1 V(erify

Format:

V

Prints the current text line on the terminal. The position of the cursor within the line has no effect on the command and the cursor is not moved. No arguments are used. VERIFY is equivalent to a "*0L$$" list command.

9.4.5.2 W(rite

Format:

W<file title>$

... where <file title> is a text string containing a valid file title. The editor appends the text file suffix ".TEXT" unless the title ends with ".", "]" or ".TEXT". If the title ends in ".", the dot is removed.

This command writes the entire text buffer to the specified disk file. It does not move the cursor or alter the contents of the text buffer.

If the specified volume has insufficient room to hold the disk file, the following error message is printed:

OUTPUT ERROR. HELP!

The text buffer can be written to another volume.

9.4.5.3 R(ead

Format:

R<file title>$

... where <file title> is a text string containing a valid file title.

The editor attempts to locate the specified file. If no file is found having the given title, a ".TEXT" suffix is appended and the editor makes another attempt at finding the file.

The contents of the specified file are copied into the text buffer starting at the cursor position.

WARNING - If the file read in does not fit, the entire text buffer contents become undefined. This is an unrecoverable error.

9.4.5.4 Q(uit

The Q(uit command can have these forms:

QU	Quit and update by writing to the work file.
QE	Quit and exit YALOE; the text is not saved.
Q	Issue a prompt requesting one of the following options: U, E, or R. R returns to the edit session.

The "QU" command writes the file to the work text file; it is similar to the W(rite command. "R" is often used to return to the editor after a "Q" has been accidentally typed.

9.4.5.5 E(rase

Format:

E

Erases the screen; this command only works with video display terminals.

9.4.5.6 O(ption

Format:

nO

Automatically display the text surrounding the cursor each time the cursor is moved; this option only works with video display terminals. The argument specifies the number of lines to be displayed. This option is disabled when the editor is entered; it is enabled by typing O(ption, and disabled by typing O(ption again. The cursor location is indicated by a split in the displayed text line.

9.4.6 Cursor Moving Commands

The commands that move the cursor are: J(ump, A(dvance, B(eginning, G(et, and F(ind. They are described in the following sections.

The direction of cursor movement is specified by the sign of the command argument; e.g., when applied to the J(ump command, the arguments (+n) and (n) move the cursor forward n characters, while the argument (-n) moves the cursor backwards n spaces.

Carriage returns are treated as a single text character.

Examples of the moving commands are given in section 9.4.6.4.

9.4.6.0 J(ump

Format:

nJ

Moves the cursor a specified number of characters in the text buffer.

9.4.6.1 A(dvance

Format:

nA

Moves the cursor a specified number of lines. The cursor is positioned at the beginning of the line to which it moved. A command argument of "0" moves the cursor to the beginning of the current line.

9.4.6.2 B(eginning

Format:

B

Moves the cursor to the beginning of the text buffer. A logical complement to this command would be "End"; this can be simulated with "/J".

9.4.6.3 G(et and F(ind

Format:

nF<target string>$ nG<target string>$

These commands are synonymous. Starting at the current cursor position, the text buffer is searched for the n'th occurrence of the specified text string; the sign of n determines the search direction. If the search is successful, the cursor is positioned immediately after the text string if n is positive, or immediately before the text string if n is negative. If the string is not found, an error message is printed, and the cursor is left at the end of the buffer if n is positive, or at the beginning if n is negative.

9.4.6.4 Examples of Cursor Moving Commands

In these examples, the cursor position is indicated by an underscore character; the cursor does not appear on a hard-copy device.

Here is the original text:

```
- - - - - - - - - - - - - - - - - - - - - -
The time has come
          the walrus said
          to balk at many things
- - - - - - - - - - - - - - - - - - - - - -
```

*8J$$ Moves the cursor forward 8 characters:

```
- - - - - - - - - - - - - - - - - - - - - -
The time has come
          the walrus said
          to balk at many things
- - - - - - - - - - - - - - - - - - - - - -
```

*-A$$ Moves the cursor up one line:

```
- - - - - - - - - - - - - - - - - - - - - -
The time has come
          the walrus said
          to balk at many things
- - - - - - - - - - - - - - - - - - - - - -
```

*BCcome$=J$$ Moves the cursor to the beginning of the text buffer and searches for the string "COME". When the string is found, the cursor is positioned at the start of the string:

```
- - - - - - - - - - - - - - - - - - - - - -
The time has come
          the walrus said
          to balk at many things
- - - - - - - - - - - - - - - - - - - - - -
```

9.4.7 Text Changing Commands

The commands that change text are: I(nsert, D(elete, K(ill, C(hange, and eX(change. These are described in the following sections. Examples of these commands are given in Section 9.4.7.5.

9.4.7.0 I(nsert

Format:

I<text string>$

Starting at the current cursor position, the characters in the specified text string are added to the text. YALOE enters Text mode after typing the "I", Text mode is terminated by typing "$". The cursor is left immediately after the last inserted character.

Occasionally, large insertions may fill the temporary insert buffer; before this happens, the editor prints "Please finish" on the console. Typing <esc><esc> finishes the current command. To continue, type "I" to re-enter Text mode.

9.4.7.1 D(elete

Format:

nD

Starting at the current cursor position, the specified number of characters are removed from the text buffer; negative arguments indicate backwards cursor movement. The cursor is left at the first character following the deleted text.

9.4.7.2 K(ill

Format:

nK

Starting at the current cursor position, the specified number of lines are deleted from the text buffer. The cursor is left at the beginning of the line following the deleted text.

9.4.7.3 C(hange

Format:

nC<text string>$

Starting at the current cursor position, n characters are replaced with the specified text string. The cursor is left immediately after the changed text.

9.4.7.4 eX(change

Format:

nX<text string>$

Starting at the current cursor position, n lines are replaced with the specified text string. The cursor is left at the end of the changed text.

9.4.7.5 Examples of Text Changing Commands

*-4D$$	Deletes the four characters immediately preceding the cursor (even if they are on the previous line).
*/K$$	Deletes all lines in the text buffer after the cursor.
*0CAAA$$	Replaces the characters from the beginning of the line to the cursor with "AAA" (same as *0XAAA$$).
*BCA$=CB$$	Searches for the first occurrence of "A" and replaces it with "B".
*-3XNEW$$	Exchanges all characters beginning with the first character on the third line back and ending at the cursor with the string "NEW".
*B$CTWINE$=D$$	Moves the cursor to the beginning of the text buffer, searches for the string "TWINE", and deletes it.

9.4.8 Other Commands

Miscellaneous commands include: S(ave, U(nsave, M(acro, N (macro execution), and "?".

9.4.8.0 S(ave

Format:

nS

Starting at the current cursor position, the specified number of text lines are copied into the save buffer. The cursor position and the text buffer contents are not affected. Each time a S(ave is executed, the previous contents of the save buffer are destroyed. If the execution of a S(ave command would overflow the save buffer, the editor generates a warning message and does not perform the S(ave.

The contents of the save buffer are accessed with the U(nsave command.

9.4.8.1 U(nsave

Format:

U

Starting at the current cursor position, the current contents of the save buffer are inserted into the text buffer. The cursor is left in front of the inserted text. If the text buffer does not have enough room for the contents of the save buffer, the Editor generates a warning message and and does not execute the U(nsave.

The save buffer can be removed by typing the command "0U".

9.4.8.2 M(acro

A macro is a single command that executes a user-defined command string. Macros are created with the M(acro command. A macro can invoke other macros (including itself recursively).

Format:

mM%<command string>%

... where m is an integer between 0 and 9 which is used to specify the macro definition. The default macro number is 1. The command string delimiter ("%" in the example above) is always the first character following the "M". The delimiter may be any character that does not appear in the macro command string itself. The second occurrence of the delimiter terminates the macro definition.

All characters except the delimiter are legal command string characters, including a single <esc>. All commands are legal in the command string.

If an error occurs when defining a macro, the following error

message is generated:

Error in macro definition.

The macro will have to be redefined.

Example of a macro definition:

```
*4M%FPREFACE$=CEND PREFACE$V$%$$
```

This example defines macro number 4. When macro 4 is executed (using the "N" command), the editor looks for the string "PREFACE", changes it to "END PREFACE", and displays the change.

NOTE - A maximum of 10 macros may exist at one time.

9.4.8.3 N (Execute Macro)

Format:

nNm$

Executes the specified macro definition. "m" is the macro number (between 0 and 9 that identifies the macro; its default value is 1. Because m actually represents a text string of commands, the N command must be terminated by <esc> (echoed as $).

Attempts to execute undefined macros generate the following error message:

Unhappy macnum.

Errors encountered during macro execution generate:

Error in macro.

9.4.8.4 ? (Display Info)

Format:

?

Prints a list of all commands, the current size of the text buffer and save buffer, the numbers of the currently defined macros, and the amount of memory available for expansion of the text buffer.

9.4.9 Command Summary

n - integer argument m - macro number

```
 ?:     Display command list and file information.
nA:     Advance the cursor to the beginning of the
        n'th line from the current position.
 B:     Go to the Beginning of the file.
nC:     Change by deleting n characters and inserting
        the following text.  Terminate text with <esc>.
nD:     Delete n characters.
 E:     Erase the screen.
nF:     Find the n'th occurrence from the current cursor.
nG:     position of the following string.  Terminate
        target string with <esc>.
 I:     Insert the following text.  Terminate text
        with <esc>.
nJ:     Jump cursor n characters.
nK:     Kill n lines of text from the current cursor
        position.
nL:     List n lines of text.
mM:     Define macro number m.
nNm:    Perform macro m, n times.
nO:     On, off toggle.  If on, n lines of text will be
        displayed above and below the cursor each time
        the cursor is moved.  If the cursor is in the
        middle of a line then the line will be split into
        two parts.  The default is whatever fills the screen.
        Type 0 to turn off.
 Q:     Quit this session, followed by:
            U:(pdate        Write out a new SYSTEM.WRK.TEXT
            E:(scape        Escape from session
            R:(eturn        Return to editor
 R:     Read file into buffer starting at cursor;
        format is: R<file name><esc>.
        WARNING:  If the file will not fit into the
        buffer, the buffer contents become undefined!
nS:     Put the next n lines of text from the cursor
        position into the Save Buffer.
 U:     Insert (Unsave) the contents of the Save Buffer into the
        text at the cursor; does not destroy the Save Buffer.
 V:     Verify:  display the current line.
 W:     Write file (from start of buffer);
        format is: W<file name><esc>.
nX:     Delete n lines of text, and insert the following text;
        terminate with <esc>.
```

9.5 Byte-level File Editor

The utility program Patch (PATCH.CODE on the utilities disk) is used to view and alter the contents of a disk file. Files are addressed as a series of 512-byte blocks; the contents of each block can be displayed on the console either in hex format or as a mixture of hex and ASCII characters. The contents of a displayed block can be modified by moving the cursor to the desired position, typing in the new data, and writing the modified block back to disk. Patch can examine and modify text and code file information; because it is a low-level utility, it is generally avoided by users who are not extremely curious or desperate.

9.5.0 Using Patch

X(ecute PATCH. The following promptline appears:

```
Patch [H0]: F(ile, Q(uit
```

Q(uit exits Patch; F(ile generates the prompt:

```
Filename: <cr for unit i/o>
```

Enter the name of the file to be edited. Patch expects complete file names; suffixes are required. Specifying a disk file limits Patch to the blocks used by the file. Blocks are referenced by relative block number (e.g., first block in the file is block 0).

Typing <return> generates this prompt:

```
Unit to patch [4,5,9..12]:
```

Type the number corresponding to the unit containing the disk to be examined (note - typing "0" exits the prompt). Specifying a disk unit allows Patch to access all blocks on the mounted disk. Blocks are referenced by absolute block number (e.g., the first block on the disk is block 0).

When either a file name or a unit number has been entered, the original prompt reappears with an added command:

```
Patch [H0]: G(et, F(ile, Q(uit
```

G(et generates the prompt:

```
BLOCK:
```

A block number is entered. The specified block is read into memory; it becomes the current block. The current block is affected only by G(et and the Alter commands. Patch maintains only one current block.

NOTE - No range checking is provided on block numbers. If a block number is out of range, Patch accepts the command, but does not change the current block.

When a current block exists, the original prompt reappears with two new commands:

Patch [H0]: G(et, H(ex, M(ixed, F(ile, Q(uit

H(ex displays the contents of the current block in hexadecimal characters. M(ixed attempts to display the block in ASCII characters; bytes not containing valid ASCII characters are displayed in hex.

H(ex and M(ixed generate the following prompt after displaying the current block:

Alter: pad vector 1,5,3,0 0..F hex characters, S(tuff, Q(uit

The cursor is initially positioned at the first byte in the block; the vector keys and space bar control its movement. Typing a hex character changes the character at the current cursor position.

NOTE - The promptline commands "1,5,3,0" are obsolete and unimplemented; they should have been removed by Western Digital a long time ago.

S(tuff is used to set a series of bytes to the same value. The following prompt appears:

Stuff for how many bytes:

Enter a number (between 0 and 512, depending on the current cursor position). The next prompt is:

Fill with what hex pair:

Enter two hex characters.

Starting at the current cursor position, Patch assigns the specified value to the number of bytes indicated, and updates the display.

NOTE - a <return> is not required after the hex pair is entered. Patch starts stuffing immediately after the second hex character is typed.

WARNING - The system may crash if S(tuff is asked to change more bytes than are displayed between the cursor and the end of the current block. DO NOT stuff past the displayed bytes.

In Alter mode, Q(uit redisplays the original Patch prompt with an added command:

Patch [H0]: G(et, P(ut, H(ex, M(ixed, F(ile, Q(uit

P(ut writes the current block to its proper disk location. It is not possible to write the current block to any other disk block than the one it was read from.

9.6 Code File Disassembly

The utility program Disassembler (DISASM.CODE on the utilities disk) is used to display the contents of a code file in symbolic form. The information available includes:

1) The number of code segments in the file.

2) The symbolic name of each code segment.

3) The number of procedures in a code segment.

4) Symbolic displays of a procedure's code and constant data.

NOTE - The disassembler uncovers many details of the UCSD Pascal implementation; therefore, much of the terminology used to describe its output is not defined in this manual. See the Architecture Guide for a definition of the following terms: P-code, constant pools, exit ic's, data segments, procedure bodies, and code segments.

NOTE - If a code file contains a program having library references, the disassembler can display the referenced library routines only if the code file has been linked.

9.6.0 Using Disassembler

X(ecute DISASM. After a few seconds, the following prompt appears:

input file:

The input file name does not require a suffix (if the disassembler cannot open the file by appending ".CODE", it trys again sans suffix). Typing only a <return> exits the disassembler.

The next prompt is:

listing file:

The list file name requires a ".TEXT" suffix if the listing is sent to a disk file. Typing "#1:" or "console:" directs the listing to the console. Typing only a <return> exits the disassembler.

The Segment Guide appears next; its prompt is:

```
Segment Guide: A(ll, #(of segment, Q(uit
```

Below this prompt is a table displaying the following information for each code segment in the file: segment name, segment number, and number of procedures.

At the bottom of the Segment Guide is the prompt:

```
Segment:
```

A(ll generates a disassembled listing of every procedure in every code segment in the code file. Q(uit exits the disassembler. Typing one of the segment numbers displayed in the Segment Guide sends the user into the Procedure Guide for the specified code segment:

```
Procedure Guide: A(ll, #(of procedure, Q(uit
```

Below this prompt, the disassembler indicates the number of procedures in the current code segment. Procedures are addressed by their procedure number (range for a given segment is 1 to <procs in seg>).

At the bottom of the Procedure Guide is the prompt:

```
Procedure:
```

A(ll generates a disassembled listing of every procedure in the current code segment. Q(uit exits the Procedure Guide and reenters the Segment Guide. Typing a procedure number generates a disassembled listing of the corresponding procedure; when the listing is complete, the following prompt appears:

```
press spacebar to continue...
```

Typing <space> reenters the Procedure Guide.

Example of a disassembled listing:

Here is the sample program:

```
program example;

  procedure target;
  var i,j : integer;
      s : string;
  begin
    i := 1;
    j := 18;
    if i >= j then
      s := 'right'
    else
      s := 'wrong';
  end;

begin
  target;
end.
```

Here is a disassembled listing of procedure "target":

```
SEGMENT=  1  PROCEDURE=   2  BLOCK=  1   BLOCK OFFSET=   2
CONSTANT POOL:
1:   0572.  r! 6967.ig! 6874.ht! 0577.  w! 726F.ro! 6E67.ng!
EXIT IC:  0031                      DATA SEGMENT SIZE:  002B
          block #  1   offset in block=  18
SEG PROC     OFFSET                                HEX CODE
  1    2     0(000):  SLDC          1              01
  1    2     1(001):  STL           2              A402
  1    2     3(003):  SLDC         18              12
  1    2     4(004):  STL           1              A401
  1    2     6(006):  SLDL          2              21
  1    2     7(007):  SLDL          1              20
  1    2     8(008):  GEQI                         B3
  1    2     9(009):  FJP          22              D40B
  1    2    11(00B):  LLA           3              8403
  1    2    13(00D):  LCA           1              8201
  1    2    15(00F):  LDCB         80              8050
  1    2    17(011):  CXG      ASSIGN              940311
  1    2    20(014):  UJP          31              8A09
  1    2    22(016):  LLA           3              8403
  1    2    24(018):  LCA           4              8204
  1    2    26(01A):  LDCB         80              8050
  1    2    28(01C):  CXG      ASSIGN              940311
  1    2    31(01F):  RPU          43              962B
```

BLOCK and BLOCK OFFSET respectively indicate the block number and byte offset of the procedure body in the code file.

The constant pool is displayed only if it exists. The number at the start of each line of constant pool data indicates the pool-relative word offset of the first word on the line. Each word of constant data is displayed in hex; if it exists, the ASCII representation is printed alongside.

EXITIC is a decimal value displaying a code-relative byte offset. DATA SEGMENT SIZE is a hex value indicating the number of words in the local data segment.

The "block #" and "offset in block" fields denote the beginning of the procedure code in the file.

Procedure code offsets are given in hex and in decimal. P-code mnemonics and their hex eqiuvalents are displayed for each instruction. System calls are recognizable by the substitution of a system call's procedure name for its segment and procedure number; this helps the user match the code in the dis-assembled listing with source statements in the corresponding program listing.

9.7 Printer Spooler

The utility program Printer (PRINTER.CODE on the utilities disk) starts the printer spooler, which writes text files to an I/O device concurrently with normal system operation. The spooler allows users to edit, compile, and run programs while text files are being printed on the line printer. The printer spooler is a background task that executes while the system is suspended (e.g. waiting at a promptline). Printer is described in section 9.7.0. The utility program Spoolgen (SPOOLGEN.CODE on the utilities disk) removes the printer spooler from the system, freeing up 400 words of memory for situations where the extra memory is needed more than the spooler is. Spoolgen is described in section 9.7.1.

9.7.0 Using Printer

X(ecute PRINTER. The following prompt appears:

    What is the output unit (0,1,<online units>)?

... where <online units> is a list of unit numbers for all online serial units (1 is the console unit). Typing "0" exits Printer. Typing any other number designates the corresponding unit as the output unit.

The next prompt is:

    File to print?

File names in Printer have the following form:

    [\]<filename>

A "\" preceding the file name indicates that the file is printed without pagination; otherwise, all files are paginated (at 60 lines per page).

Up to three files may be queued for printing; the file name prompt reappears after each file name is typed in. Typing only <return> to the file name prompt indicates that no more files are to be queued for printing; Printer then terminates.

NOTE - Printer has the following restrictions:

A) Files queued for printing must not be modified, moved, or removed until they are finished printing; the same restrictions apply to the disk volumes containing them. Be wary of K(runch. The best way to avoid problems of this nature is to move files to an unused online disk volume before printing them.

B) The output device used by the spooler should not be accessed by the system until the spooler is finished.

C) If the output unit is unit 8, and the printer operates at a different baud rate than the terminal, then Printer loses its concurrent capabilities. The system prompt reappears during printing, but the system does not accept any commands until all printing is finished. This restriction is imposed by the hardware; see the Hardware User's Manual for details.

9.7.1 Using Spoolgen

X(ecute SPOOLGEN. The following message appears:

The printer spooler is currently <spoolstate>.

... where <spoolstate> is "ENABLED" or "DISABLED".

The following prompt then appears:

Do you wish to ENABLE or DISABLE it (E/D) ?

Typing "E" enables the spooler. Typing "D" disables it. Typing <escape> exits Spoolgen.

NOTE - The system must then be rebooted to acutally enable or disable the spooler.

Spoolgen modifies a parameter stored in SYSTEM.MISCINFO. When the system is booted, the parameter value is checked. If spooling is enabled, the spooler is allocated 400 words of memory, and is available for use; otherwise, the memory is not allocated, and the spooler displays the message "queue full" when executed.

9.8 Calculator

The utility program Calc (CALC.CODE on the utilities disk) simulates a desktop calculator.

9.8.0 Using Calc

X(ecute CALC. The following prompt appears:

```
->
```

Calc expects a one-line expression in algebraic form as a response. Up to 25 different variables are available. Variable names are significant only to eight case-insensitive characters. Variables having a value may be used as constants. Two predefined variables are PI (3.141593) and E (2.718282).

The remainder operator (specified by the dyadic operator "\") rounds its result to an integer.

WARNING - Because the remainder operator is based on Pascal's MOD operator, it should not be used with negative arguments.

Arguments of the factorial function (form: FAC(x)) are rounded to integer values; all arguments X : (0 <= X <= 33) cause the expression to be rejected.

The uparrow is used for exponentiation (form: x^y). The result is calculated using the formula: e ^ y ln (x); operands must be positive or the expression is rejected.

The predefined variable LASTX is always assigned the value of the previous correct expression.

Arguments of the trigonometric functions are expected to be in radians. Degree-to-radian conversion is accomplished with the formula: RADANGLE = (PI/180) * DEGANGLE.

Calc generates an execution error if an overflow or underflow occurs. If this happens, all user-assigned variables and their values are lost.

Typing <return> in response to a prompt exits Calc.

Example of a Calc session:

```
-> PI
      3.14159

-> E
      2.71828

-> A = (FAC(3)/2)
      3.00000

-> 3 + 6
      9.00000

-> A + 6
      9.00000

-> <return>
```

APPENDIX A: I/O RESULTS

0	No error
1	Bad Block, Parity error (CRC)
2	Bad Unit Number
3	Bad Mode, Illegal operation
4	Undefined hardware error
5	Lost unit, Unit is no longer on-line
6	Lost file, File is no longer in directory
7	Bad Title, Illegal file name
8	No room, insufficient space
9	No unit, No such volume on line
10	No file, No such file on volume
11	Duplicate file
12	Not closed, attempt to open an open file
13	Not open, attempt to access a closed file
14	Bad format, error in reading real or integer
15	Ring buffer overflow
16	Write Protect; attempted write to protected disk
17	Illegal block number
18	Illegal buffer address

APPENDIX B: EXECUTION ERRORS

0	System error
1	Invalid index, value out of range
2	No segment, bad code file
3	Exit from uncalled procedure
4	Stack overflow
5	Integer overflow
6	Divide by zero
7	Invalid memory reference <bus timed out>
8	User Break
9	System I/O error
10	User I/O error
11	Unimplemented instruction
12	Floating Point math error
13	String too long

APPENDIX C: I/O UNIT ASSIGNMENTS

This section describes the hardware devices assigned to the system's physical unit numbers. The operating system contains software drivers to support I/O to the indicated devices. See the Hardware User's Manual for details on the devices listed below. Physical units are described in section 2.1.2. The Programmer's Manual describes Unit I/O operations.

Unit Number	PDQ-3 Device Assignment
1	Console port (echo)
2	Console port (no echo)
3	unassigned
4	Floppy Drive 0
5	Floppy Drive 1
6	LPV-11 (FFA0 hex) parallel printer
7	unassigned
8	serial printer port
9	RP-02 (FEE4 hex) Logical Disk 0
10	RP-02 (FEE4 hex) Logical Disk 1
11	RP-02 (FEE4 hex) Logical Disk 2
12	RP-02 (FEE4 hex) Logical Disk 3
13	DLV-11J (FEA0 hex) Port 0 Input
14	DLV-11J (FEA0 hex) Port 0 Output
15	DLV-11J (FEA4 hex) Port 1 Input
16	DLV-11J (FEA4 hex) Port 1 Output
17	DLV-11J (FEA8 hex) Port 2 Input
18	DLV-11J (FEA8 hex) Port 2 Output
19	DLV-11J (FEB8 hex) Port 3 Input
20	DLV-11J (FEB8 hex) Port 3 Output
128	Keyboard Type-Ahead Buffer (write only)
129	Fast console output

NOTE - Hex numbers displayed with I/O device names indicate the memory address used to communicate with the device.

NOTE - The assignments shown here may change in future versions of the system.

APPENDIX D: COMPILER SYNTAX ERRORS

```
  1:  Error in simple type
  2:  Identifier expected
  3:  'PROGRAM' expected
  4:  ')' expected
  5:  ':' expected
  6:  Illegal symbol
  7:  Error in parameter list
  8:  'OF' expected
  9:  '(' expected
 10:  Error in type
 11:  '[' expected
 12:  ']' expected
 13:  'END' expected
 14:  ';' expected
 15:  Integer expected
 16:  '=' expected
 17:  'BEGIN' expected
 18:  Error in declaration part
 19:  Error in <field-list>
 20:  ',' expected
 21:  '*' expected
 22:  'Interface' expected
 23:  'Implementation' expected
 24:  'Unit' expected

 50:  Error in constant
 51:  ':=' expected
 52:  'THEN' expected
 53:  'UNTIL' expected
 54:  'DO' expected
 55:  'TO' or 'DOWNTO' expected in for statement
 56:  'IF' expected
 57:  'FILE' expected
 58:  Error in <factor> (bad expression)
 59:  Error in variable
 60:  Must be semaphore
 61:  Must be processid

101:  Identifier declared twice
102:  Low bound exceeds high bound
103:  Identifier is not of the appropriate class
104:  Undeclared identifier
105:  Sign not allowed
106:  Number expected
107:  Incompatible subrange types
108:  File not allowed here
109:  Type must not be real
110:  <tagfield> type must be scalar or subrange
111:  Incompatible with <tagfield> part
112:  Index type must not be real
113:  Index type must be a scalar or a subrange
114:  Base type must not be real
115:  Base type must be a scalar or a subrange
```

116: Error in type of standard procedure parameter
117: Unsatisfied forward reference
118: Forward reference type identifier in variable declaration
119: Re-specified params not OK for a forward declared procedure
120: Function result type must be scalar, subrange or pointer
121: File value parameter not allowed
122: Forward declared function result type can't be re-specified
123: Missing result type in function declaration
124: F-format for reals only
125: Error in type of standard function parameter
126: Number of parameters does not agree with declaration
127: Illegal parameter substitution
128: Result type does not agree with declaration
129: Type conflict of operands
130: Expression is not of set type
131: Tests on equality allowed only
132: Strict inclusion not allowed
133: File comparison not allowed
134: Illegal type of operand(s)
135: Type of operand must be boolean
136: Set element type must be scalar or subrange
137: Set element types must be compatible
138: Type of variable is not array
139: Index type is not compatible with the declaration
140: Type of variable is not record
141: Type of variable must be file or pointer
142: Illegal parameter substitution
143: Illegal type of loop control variable
144: Illegal type of expression
145: Type conflict
146: Assignment of files not allowed
147: Label type incompatible with selecting expression
148: Subrange bounds must be scalar
149: Index type must be integer
150: Assignment to standard function is not allowed
151: Assignment to formal function is not allowed
152: No such field in this record
153: Type error in read
154: Actual parameter must be a variable
155: Control variable cannot be formal or non-local
156: Multidefined case label
157: Too many cases in case statement
158: No such variant in this record
159: Real or string tagfields not allowed
160: Previous declaration was not forward
161: Again forward declared
162: Parameter size must be constant
163: Missing variant in declaration
164: Substitution of standard proc/func not allowed
165: Multidefined label
166: Multideclared label
167: Undeclared label
168: Undefined label
169: Error in base set
170: Value parameter expected
171: Standard file was re-declared

172: Undeclared external file
174: Pascal function or procedure expected
175: Semaphore value parameter not allowed
182: Nested units not allowed
183: External declaration not allowed at this nesting level
184: External declaration not allowed in interface section
185: Segment declaration not allowed in unit
186: Labels not allowed in interface section
187: Attempt to open library unsuccessful
188: Unit not declared in previous uses declaration
189: 'Uses' not allowed at this nesting level
190: Unit not in library
191: No private files
192: 'Uses' must be in interface section
193: Not enough room for this operation
194: Comment must appear at top of program
195: Unit not importable

201: Error in real number - digit expected
202: String constant must not exceed source line
203: Integer constant exceeds range
204: 8 or 9 in octal number

250: Too many scopes of nested identifiers
251: Too many nested procedures or functions
252: Too many forward references of procedure entries
253: Procedure too long
254: Too many long constants in this procedure
256: Too many external references
257: Too many externals
258: Too many local files
259: Expression too complicated

300: Division by zero
301: No case provided for this value
302: Index expression out of bounds
303: Value to be assigned is out of bounds
304: Element expression out of range

398: Implementation restriction
399: Implementation restriction
400: Illegal character in text
401: Unexpected end of input
402: Error in writing code file, not enough room
403: Error in reading include file
404: Error in writing list file, not enough room
405: Call not allowed in separate procedure
406: Include file not legal

APPENDIX E: ASCII CHARACTER SET

0	000	00	NUL	32	040	20	SP	64	100	40	@	96	140	60	`
1	001	01	SOH	33	040	21	!	65	101	41	A	97	141	64	a
2	002	02	STX	34	042	22	"	66	102	42	B	98	142	62	b
3	003	03	ETX	35	043	23	#	67	103	43	C	99	143	63	c
4	004	04	EOT	36	044	24	$	78	104	44	D	100	144	64	d
5	005	05	ENG	37	045	25	%	69	105	45	E	101	145	65	e
6	006	06	ACK	38	046	26	&	70	106	46	F	102	146	66	f
7	007	07	BEL	39	047	27	'	71	107	47	G	103	147	67	g
8	010	08	BS	40	050	28	(	72	110	48	H	104	150	68	h
9	011	09	HT	41	051	29	)	73	111	49	I	105	151	69	i
10	012	0A	LF	42	052	2A	*	74	112	4A	J	106	152	6A	j
11	013	0B	VT	43	053	2B	+	75	113	4B	K	107	153	6B	k
12	014	0C	FF	44	054	2C	,	76	114	4C	L	108	154	6C	l
13	015	0D	CR	45	055	2D	-	77	115	4D	M	109	155	6D	m
14	016	0E	SO	46	056	2E	.	78	116	4E	N	110	156	6E	n
15	017	0F	SI	47	057	2F	/	79	117	4F	O	111	157	6F	o
16	020	10	DLE	48	060	30	0	80	120	50	P	112	160	70	p
17	021	11	DC1	49	061	31	1	81	121	51	Q	113	161	71	q
18	022	12	DC2	50	062	32	2	82	122	52	R	114	162	72	r
19	023	13	DC3	51	063	33	3	83	123	53	S	115	163	73	s
20	024	14	DC4	52	064	34	4	84	124	54	T	116	164	74	t
21	025	15	NAK	53	064	35	5	85	125	55	U	117	165	75	u
22	026	16	SYN	54	066	36	6	86	126	56	V	118	166	76	v
23	027	17	ETB	55	067	37	7	87	127	57	W	119	167	77	w
24	030	18	CAN	56	070	38	8	89	130	58	X	120	170	78	x
25	031	19	EM	57	071	39	9	89	131	59	Y	121	171	79	y
26	032	1A	SUB	58	072	3A	:	90	132	5A	Z	122	172	7A	z
27	033	1B	ESC	59	073	3B	;	91	133	5B	[	123	173	7B	{
28	034	1C	FS	60	074	3C	<	92	134	5C	\	124	174	7C	|
29	035	1D	GS	61	075	3D	=	93	135	5D	]	125	175	7D	}
30	036	1E	RS	62	076	3E	>	94	136	5E	^	126	176	7E	~
31	307	1F	US	63	077	3F	?	95	137	5F	_	127	177	7F	DEL

APPENDIX F1: ADM 3-A TERMINAL

Key Command Definitions

The key definitions shown below fall in one of two classes: "hard" (fixed definition in system) or "soft" (user-redefinable). Hard keys are described in section 1.3. Soft keys are described in chapter 9.3.1.

Function	Key
escape	ESC
return	RETURN
delete line	RUBOUT
EOF	control-C
backspace	l-arrow key
tab stop	control-I
accept	control-C
cursor down	d-arrow key
cursor up	u-arrow key
cursor left	l-arrow key
cursor right	r-arrow key
Stop	control-S
Stop (alt.)	control-Q
Flush output	control-F
HDT	control-P
Flush input	control-X
Set disk shape	control-D

APPENDIX F2: SOROC IQ-120 TERMINAL

Key Command Definitions

The Key definitions shown below fall in one of two classes: "hard" (fixed definition in system) or "soft" (user-redefinable). Hard keys are described in section 1.3. Soft Keys are described in section 9.3.1.

Function	Key
escape	ESC
return	RETURN
delete line	RUBOUT
EOF	control-C
backspace	l-arrow key
tab stop	TAB
accept	home
cursor down	d-arrow key
cursor up	u-arrow key
cursor left	l-arrow key
cursor right	r-arrow key
Stop	control-S
Stop (alt.)	control-Q
Flush output	control-F
HDT	control-P
Flush input	control-X
Set disk shape	control-D

APPENDIX F3: ZENITH Z19

Key Command Definitions

The key definitions shown below fall in one of two classes: "hard" (fixed definition in system) or "soft" (user-redefinable). Hard keys are described in section 1.3. Soft keys are described in section 9.3.1.

Function	Key
escape	ESC
return	RETURN
delete line	RUBOUT
EOF	line feed
backspace	back space
tab stop	TAB
accept	LINE FEED
cursor down	d-arrow key
cursor up	u-arrow key
cursor left	l-arrow key
cursor right	r-arrow key
Stop	control-S
Stop (alt.)	control-Q
Flush output	control-F
HDT	control-P
Flush input	control-X
Set disk shape	control-D