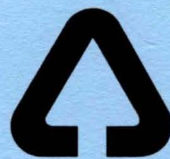


Advanced Computer Design

AOS System User's Manual

TABLE OF CONTENTS

SECTION PAGE

I INTRODUCTION 1

0 Scope of this Manual 1
1 Notation and Terminology 2
2 System Organization 3

3 Command and Data Overview 4
0 Prompt Lines 4
1 File Names 4
2 Data Prompts 5

4 Key Commands 6
0 Accept and Escape 6
1 Console End of File 6
2 Cursor Movement 6
3 User Interrupt Commands 6
0 Monitor Trap 6
1 Stop and Start 7
2 Console Output Flush 7
3 Keyboard Type-ahead Flush 7
4 Disk Type 8

II OPERATING SYSTEM 9

0 Error Handling 10
0 Execution Errors 10
1 Stack Overflow 11
2 Floppy Disk Errors 12
3 Disk Swapping 13

1 File System 14
0 Overview 14
1 Syntax Overview 15

2 Physical Units 16
0 Syntax Overview 17
1 I/O Devices 18
0 Serial Devices 18
1 Block-structured Devices 18

3 Logical Volumes 19
0 Syntax Overview 20
1 Block-structured (Disk) Volumes 21
2 Disk Volume Usage 21
3 System Volumes 21
4 Prefixed Volumes 22
5 Disk Directories 22
0 Duplicate Directories 22

II OPERATING SYSTEM (continued)

1 File System (continued)

4 Disk Files 24
0 Syntax Overview 24

1 File Attributes 24
0 File Type 24
0 File Type Assignment 25
1 UCSD Pascal Files 25
0 Text Files 25
1 Code Files 25
2 Data Files 25
3 Restrictions Imposed by Types 26
1 File Date 26
2 Size and Location Attributes 26

2 File Suffixes 27

3 File Titles 27
0 System File Titles 27
1 Other Reserved Titles 29
2 User File Titles 29
3 Titles with Non-block-structured Volumes . . 29

4 File Length and File Length Specifiers 29

5 Syntax Specification 31

6 File Conventions and Applications 33
0 File Name Prompt Conventions 33
0 Input Prompts 33
1 Output Prompts 33
1 File Access from User Programs 34

2 Library System 35
0 System Library 36
1 Intrinsics Library 36
2 Program Library 37
3 User Library 37
4 Library Configuration Examples 37

3 System Configuration 40
0 Operating System Libraries 40
1 I/O System Configuration 41
2 Terminal Configuration 42
3 System Shell 42
4 Execution Error and Breakpoint Processing 43
5 Performance Optimizations 43

II OPERATING SYSTEM (continued)

4 Commands and Operation 45
0 Bootstrapping the System 45
1 The Work File 48
0 Work File Manipulation 48
1 Work File Effects on System Behavior 48
2 Syntax Errors and Editor Invocation 49
3 System State Flow Diagram 49

4 I/O Redirection Options 51
0 Execution Option Lists 53
1 Output Redirection Options 54
2 Input Redirection Options 55
3 T-File Options 55
4 Prefix Options 56
5 Library Options 56
6 System I/O Redirection 57

5 System Commands 58
0 Clear Screen 59
1 C(ompile 60
2 E(dit 61
3 F(ile 62
4 H(alt 63
5 I(nitialize 64
6 M(emory 65
7 R(un 66
8 S(ubmit 67
9 U(ser restart 68
10 X(ecute 69

III THE FILE HANDLER 71

0 Filer Prompts 71

1 File Naming Conventions 72
  0 General Syntax 72
  1 Wildcards 72

2 Filer Commands 73

  0 Command Summary 73
    0 Work File Commands 73
    1 Disk File & Volume Commands 74
    2 Disk Volume Commands 74
    3 Disk Media Commands 74

  1 B(ad blocks scan 75
  2 C(hange 76
  3 D(ate 77
  4 E(xtended list 78
  5 G(et 79
  6 K(runch 80
  7 L(ist directory 81
  8 M(ake 83
  9 N(ew 84
  10 P(refix volume 85
  11 Q(uit 86
  12 R(emove 87
  13 S(ave 88
  14 T(ransfer 89
  15 V(olumes online 94
  16 W(hat is workfile? 95
  17 X(amine bad blocks 96
  18 Z(ero directory 98

3 Recovering Lost Files 100

4 Recovering Lost Directories 103

IV THE ADVANCED SYSTEM EDITOR 104

0 Basic Concepts 106
0 Prompt Lines 107
1 Commands 107
2 File Name Prompts 108
3 The Edit Environment 110
4 The File Window 111
5 The File Buffer 111
6 The Cursor 112
7 Backup Files 113

1 Using the Editor 114

0 Entering the Editor 114
1 Repeat Factors 115
2 Direction 116
3 Markers 116
4 Moving The Cursor 118
5 The Copy Buffer 120
6 Entering Strings in F(ind and R(eplace 121
7 Nested Editing 123
8 Change Logging 126
9 User-defined Functions 128

IV THE ADVANCED SYSTEM EDITOR (continued)

2 Commands 135

0 Command Summary 135
0 Moving Commands 135
1 Text-Changing Commands 136
2 Pattern Matching Commands 136
3 Formatting Commands 136
4 Buffer Managing Commands 136
5 Function Defining Commands 136
6 Miscellaneous Commands 137

1 A(djust 138
2 B(eginLine 139
3 C(opy 140
4 D(elete 142
5 E(dit 143
6 F(ind 144
7 G(etch 146
0 <GetAgain> 147
8 I(nsert 148
9 J(ump 150
10 K(olumn 151
11 L(ineEnd 152
12 M(argin 153
13 N(ext 155
14 O(ppositePage 156
15 P(age 157
16 Q(uit 158
17 <record> 160
18 R(eplace 161
19 S(et 163
20 <takeup> 168
21 T(oDisk 169
22 U(ptop 170
23 V(erify 171
24 W(ordMove 172
25 eX(change 173
26 Z(ap 176

3 Sample Edit Session 177

4 Problems 183

V COMPILER 187

0 Introduction 187

1 Using the Compiler 188
0 Setting Up Input and Output Files 188
1 Console Display 189
2 Syntax Error Handling 190

2 Compiler Problems 191

0 Syntax Errors and the Editor 191
1 Insufficient Memory 192
2 Insufficient Space on Volume 192

VI COMMAND FILE INTERPRETER 193

0 S(ubmitting Command Files 193
0 Command File Execution 193
1 Reserved Command File Names 194

1 Command Language 194
0 Commands 195
0 Immediate Commands 195
1 Deferred Commands 197
1 Targets 197
2 Parameters and Variables 198
3 Text Lines 198

2 Example eXec Programs 200

VII SYSTEM MONITOR 203

0 Entering The Monitor 203
1 Monitor Commands 204
2 HDT Examples 207

VIII UTILITIES 209

0 Disk Management 209

0 Bootstrap Copier 210
0 Using Booter 210

1 Disk Copying 211
0 Using Backup 211

2 Disk Format Conversion 213
0 Using Mapper 213

3 Disk Formatting 215
0 Using Format 215
1 Reformatting Bad Blocks 216

4 Fast Bad Blocks Scanning 217
0 Using Bad.Blocks 217

5 Hard Disk Management 219
0 Using Drive.Con 219

6 Changing Volume Size 223
0 Using Change.Dir 223

1 Data Recovery 224

0 Using Markdupdir 224
1 Using Copydupdir 225
2 Using Recover 225

2 Library Management 227

0 Using Library 227
1 Using Libmap 230

3 Terminal Configuration 232

0 GOTOXY Binding 234
0 Using Binder 236

1 Using Setup 237
0 Fields in Setup 238

2 Using Advanced System Setup 243

4 System I/O Configuration 248
0 Using Drvr.Info 248

VIII UTILITIES (continued)

5 Line-Oriented Text Editor 251

0 Entering YALOE 251
1 Entering Commands and Text 251
0 Command Arguments 252
1 Command Strings 252
2 Text Strings 252
2 The Text Buffer 253
3 The Cursor 253
4 Special Commands 253
5 Input/Output Commands 254
6 Cursor Moving Commands 256
7 Text Changing Commands 259
8 Other Commands 260
9 Command Summary 263

6 Byte-level File Editor 264
0 Using Patch 264

7 Printer Spooler 270
0 Using Printer 270

8 Calculator 272
0 Using Calc 272

9 Bootstrap Creation 274
0 Using Make.Boot 274

APPENDICES 277

Appendix A: Standard I/O Results 277
Appendix B: Standard Execution Errors 279
Appendix C: Standard I/O Unit Assignments 281
Appendix D: Compiler Syntax Errors 283
Appendix E: ASCII Character Set 287
Appendix F: Terminal Configurations 289
Appendix F1: ADM 3-A Terminal 291
Appendix F2: SOROC IQ-120 Terminal 293
Appendix F3: ZENITH Z-19 Terminal 295
Appendix F4: DEC VT-100 Terminal 297

INDEX 301

I. INTRODUCTION

1.0 Scope of this Manual

This is the reference manual for the UCSD Pascal Advanced Operating System, version 1.0, running on the PDQ-3 Computer System. 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.

AOS Programmer's Manual - Describes the Pascal language implementation used with the Advanced Operating System.

AOS Library User's Manual - Describes the library modules available with the Advanced Operating System.

AOS Architecture Guide - Provides details of the system software to experienced programmers. (Available in the indeterminate future.)

PDQ-3 Subsystem Documents - Describes the physical characteristics and operating procedures for the various hardware subsystems available with the PDQ-3.

1.1 Notation and Terminology

This section describes the notation and terminology used in this manual to describe various system concepts.

A variant of Backus-Naur form (BNF) is used as a notation for describing the form of system constructs. Meta-words are words which represent a class of words; they are delimited by angular brackets ("<" and ">"). Thus, the words "trout", "salmon", and "tuna" are acceptable substitutions for the meta-word "<fish>"; here is an expression describing the substitution:

<fish> ::= trout | salmon | tuna

The symbol "::=" indicates that the meta-word on the left-hand side may be substituted with an item from the right-hand side. The vertical bar "|" separates possible choices for substitution; the example above indicates that "trout", "salmon", or "tuna" may be substituted for <fish>.

An item enclosed in square brackets may be optionally substituted into a textual expression; for instance, "[micro]computer" represents the text strings "computer" and "microcomputer".

An item enclosed in curly brackets may be substituted zero or more times into a textual expression. The following expression represents responses to jokes possessing varying degrees of humor:

<joke response> ::= {ha}

In many instances, the notation described above is used informally to describe the form required by a language construct. Here are some typical examples:

START(<process statement> [,<pid> [,<stacksize> [,<priority>]]])

CONCAT(<string> {,<string>})

The syntax for Pascal's IF statement is:

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

1.2 System Organization

The Advanced Operating System 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 multiprocessing capabilities, an asynchronous I/O system, I/O redirection facilities, and a programming library system. It is especially well suited as a program development and runtime environment for large realtime and multiuser 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 produces either executable Pascal programs or library routines.

System 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.

1.3 Command and Data Overview

This section describes 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.

1.3.0 Prompt Lines

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

```
Command: X(ecute, S(ubmit, R(un, F(ile, E(dit, C(omp [1.0]

Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans [1.0]
```

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. Prompt lines displaying alphabetic character commands accept both lower and upper case characters. With some prompt lines, typing a "?" redisplays the prompt line with a different set of commands. This is done to accommodate wide prompt lines on narrow screens. Prompt lines are usually referred to as "prompts"; thus, the prompt line for the operating system is called the "system prompt", and for the file handler, the "filer prompt".

Many system parts display a version number in their prompt lines; it is usually delimited by square brackets.

1.3.1 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.

1.3.2 Data Prompts

Data prompts are used to obtain input data needed by 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: a single character response to a "yes/no" question (such as the second example), or an input data response requiring a string of input characters terminated 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 characters other than the affirmative ones as negative responses.

Input data responses are usually file names, but can be other items such as the system 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.

1.4 Key Commands

This section describes some key commands used throughout the system. Key command definitions are described in section 8.3 (terminal configuration). Key command definitions for some common terminals are listed in Appendix F.

1.4.0 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.

1.4.1 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 3.2.14 and the Programmer's Manual for more details.

1.4.2 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).

1.4.3 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.

NOTE - User interrupt command processing may be suspended from within a program. See Appendix D of the Programmer's Manual.

1.4.3.0 Monitor Trap

The monitor key interrupts the currently executing user or system program and passes control to the system monitor (described in chapter 7); program execution may be resumed from the monitor. The monitor key is defined to be <control-P>.

1.4.3.1 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>.

1.4.3.2 Console Output Flush

The flush key causes the system to discard all console output until either the flush key is retyped or a keyboard read operation is initiated. 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 prompt line; it displays the prompt line then waits for another command from the keyboard. Since screen output is still being flushed when the prompt line is displayed, the prompt line doesn't appear. The keyboard read cancels flushing; typing <space> causes the prompt line to reappear, and normal system operation is resumed. The flush key is defined to be <control-F>.

1.4.3.3 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 before an input prompt is displayed. 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 32 characters. When it is full, subsequent keyboard input is ignored.

1.4.3.4 Disk Type

The disk type key allows on-the-fly reconfiguration 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. The disk type key also controls the generation of floppy disk error messages (see section 2.0.2).

NOTE - Double-sided floppy disks require double-sided disk drives.

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

When the system is bootstrapped, all floppy disk drives are configured for single-sided PDQ-3 format floppy disks (unless specified otherwise -- see section 2.4.0), with error messages disabled. Floppy 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). Their values are all reset when either the "s" or the "d" commands are issued.

NOTE - The Mapper utility (section 8.0.2.0) 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 and the Library User'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 presented 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 "Segment" name, the "P" number, and the "I" number represent the code segment name, procedure number within the segment, and procedure-relative byte offset of the instruction causing the error. This information should be used in conjunction with a source program listing to pinpoint the error in the source program. The locations of procedure calls leading up to the execution error may be obtained by changing the "Error List Length" field using the Setup utility described in section 8.3.1. Program listings, code segments, and procedure numbers are described in the Programmer's Manual.

Once an execution error has occurred, two choices are available to the user. "Typing <space> to continue", as is prompted on the console, aborts the currently executing program. Typing <escape> 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.

NOTE - When the standard exception handler is installed in the system, execution errors are processed as described above. Execution errors may be processed differently when a custom exception

handler is installed. See section 2.3.4 and the Programmer's Manual for details.

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

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 may be directed to issue error messages to the console whenever the hardware indicates that a disk operation caused a transient error (see section 1.4.3.4). 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 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.

This situation is remedied both at the program level and at the system level.

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 may 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.

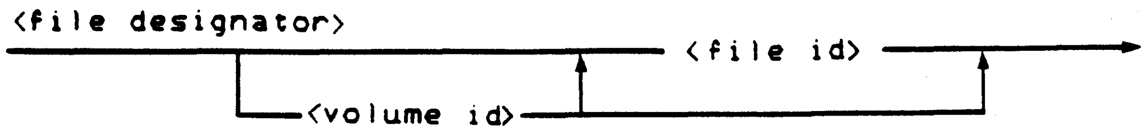
4) The disk file implementation is both time and space-efficient on relatively low performance floppy disk drives.

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 - The device assignments discussed in this manual are the standard device assignments for the PDQ-3. The mapping between physical unit numbers and devices may be defined by the user. Appendix C contains a list of the PDQ-3 Computer System's standard device assignments. Section 2.3.1 describes how device assignments can be modified.

Unit Number	device description	unit attribute
0	system clock	serial
1	screen and keyboard with echo	serial
2	screen and keyboard without echo	serial
3	keyboard type-ahead	serial
4	disk drive 0	block-structured
5	disk drive 1	block-structured
6	printer	serial
7	remote port input	serial
8	remote port output	serial
9 - 12	disks 2 - 5	block-structured
13	remote port 1 input	serial
14	remote port 1 output	serial
15	remote port 2 input	serial
16	remote port 2 output	serial
17	remote port 3 input	serial
18	remote port 3 output	serial
19	remote port 4 input	serial
20	remote port 4 output	serial
21	fast screen output	serial
22	standard input	serial
23	standard output	serial
24	null input and output	serial
25-28	disks 6-9	block-structured

2.1.2.0 Syntax Overview

<unit number>

——————— #<number>: ———————→

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 either produces or consumes a sequence of data. Serial devices used with the system include 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. Management of hard disks is performed by the Drive.Con utility described in section 8.0.5.0.

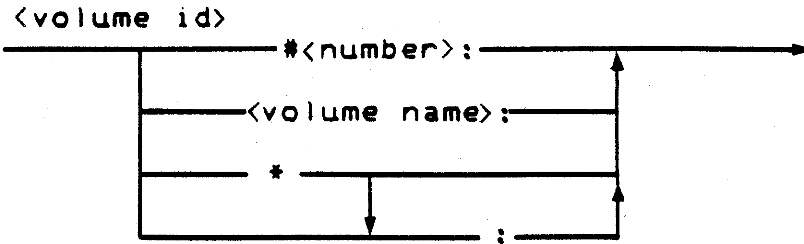
2.1.3 Logical Volumes

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
CLOCK:	0	serial
CONSOLE:	1	serial
SYSTERM:	2	serial
KEYBUFR:	3	serial
<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
FASTCON:	21	serial
STANIN:	22	serial
STANOUT:	23	serial
BUCKET:	24	serial
<vol names>	25 - 28	block-structured

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. 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 this is unavoidable, all file designators should 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 from which the system was bootstrapped (see section 2.4.0); it contains the operating system and usually 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) and the prefix redirection options (section 2.4.4.4) are used to specify volumes as the prefixed volume; they designate a user-specified volume identifier 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.

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

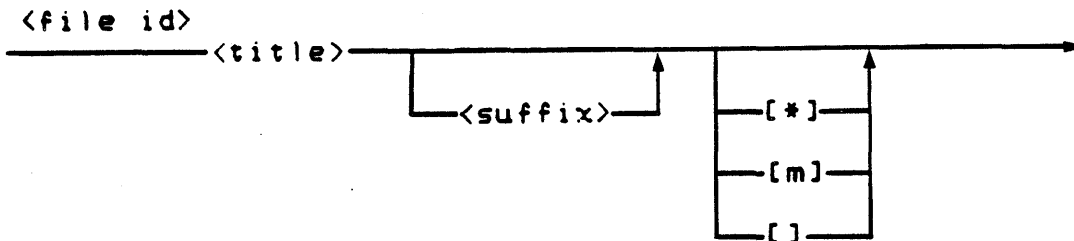
A disk volume may be marked so that the system maintains two disk directories on a disk volume; the second directory is called a duplicate directory and exists as a copy of the main directory. If unforeseen circumstances cause the destruction of the main directory, it can be restored using the information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

rally outweighs any losses in performance. The utility programs Markdupdir and Copydupdir are used to create duplicate directories and restore deceased main directories (see section 8.1).

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.

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 operating system and system utilities. Code files contain a mixture of P-code and execution information used by the CPU and operating system.

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 and Libmap utility programs described in chapter 8.

2.1.4.1.0.2 Data Files

Data files are created by programs and may have any internal representation. Except for being constrained 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 editor does not accept files other than text files for editing. It uses 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).

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. A file created with one of these suffixes is 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

All .BACK files are created by the system editor. They are discussed in section 4.0.7. .BAD files are created by the filer X(amine command. They are discussed in section 3.2.17.

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.DRIVERS	code	system I/O drivers
SYSTEM.DRVINFO	code	I/O drivers information
SYSTEM.EDITOR	code	editor
SYSTEM.FILER	code	file handler
SYSTEM.INTRINS	code	contains intrinsic routines
SYSTEM.LIBRARY	code	contains user library routines
SYSTEM.LST.TEXT	text	default program listing file
SYSTEM.MISCINFO	data	terminal configuration info
SYSTEM.PASCAL	code	operating system
SYSTEM.SHELL	code	system prompt line processor
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, intrinsics, drivers, 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 and the ASS utilities (section 8.3). Users may add their own operating system extensions to the SYSTEM.INTRINS or their own library routines to SYSTEM.LIBRARY using the Library utility (section 8.2.0). The library system is described in section 2.2.

The SYSTEM.DRIVERS file contains the code for all system I/O drivers. The SYSTEM.DRVINFO file contains the mapping between physical unit numbers and the drivers contained in the SYSTEM .DRIVERS file. These files are modified by the Library and Drvr.Info utilities described in sections 8.2.0 and 8.4. System I/O drivers are discussed in section 2.3.1.

SYSTEM.SHELL is the program which the system automatically executes at system bootstrap time. It performs all system prompt line command processing. Replacement of the system shell by a user-defined program is discussed in section 2.3.3.

SYSTEM.STARTUP is a user-defined program which the system shell executes during the system bootstrap before displaying the welcome message or system prompt. It is used for turnkey applications programs which do not require other parts of the system.

While bootstrapping, the system searches for SYSTEM.MISCINFO, SYSTEM.PASCAL, SYSTEM.DRVINFO, SYSTEM.INTRINS, and SYSTEM.DRIVERS on the system volume only. 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 2.4.1.

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) An "include" file directive is being processed; therefore, a disk directory must be read in order to open the "include" file.

3) There is insufficient memory to read the directory, but the program's symbol table occupies more than 4K bytes.

... then the operating system swaps a section of the symbol table out to the file SYSTEM.SWAPDISK, reads the directory into the resulting section of memory, opens the "include" file, and swaps the symbol table back into memory. See section 5.2.1 for more information.

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

2.1.4.3.1 Other Reserved Titles

The file names X.CODE, PROFILE.TEXT, $EXEC.TEXT, and USERLIB.TEXT are reserved for system use in addition to the system file titles enumerated in section 2.1.4.3.0. The S(ubmit command processor (chapter 6) is called X.CODE. The PROFILE.TEXT and $EXEC.TEXT files are used by the command processor as defaults in certain operations. The USERLIB.TEXT contains a list of user library file names. It is discussed in section 2.2.3.

2.1.4.3.2 User File Titles

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

2.1.4.3.3 File Titles with Non-block-structured Volumes

The file system allows the use of serial volume identifiers in conjunction with non-empty file titles (i.e. Console:.Text) even-though serial volumes have no directories. In this case, the file title is ignored. This convention allows a system part to append a standard file suffix to a file prompt response without first having to determine whether or not the suffix is appropriate.

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 complete, 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, the system 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. Although the number of blocks in the free space might easily contain all of the completed files, the first file created is allocated all of the available disk space, thus preventing the creation of other files.

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.

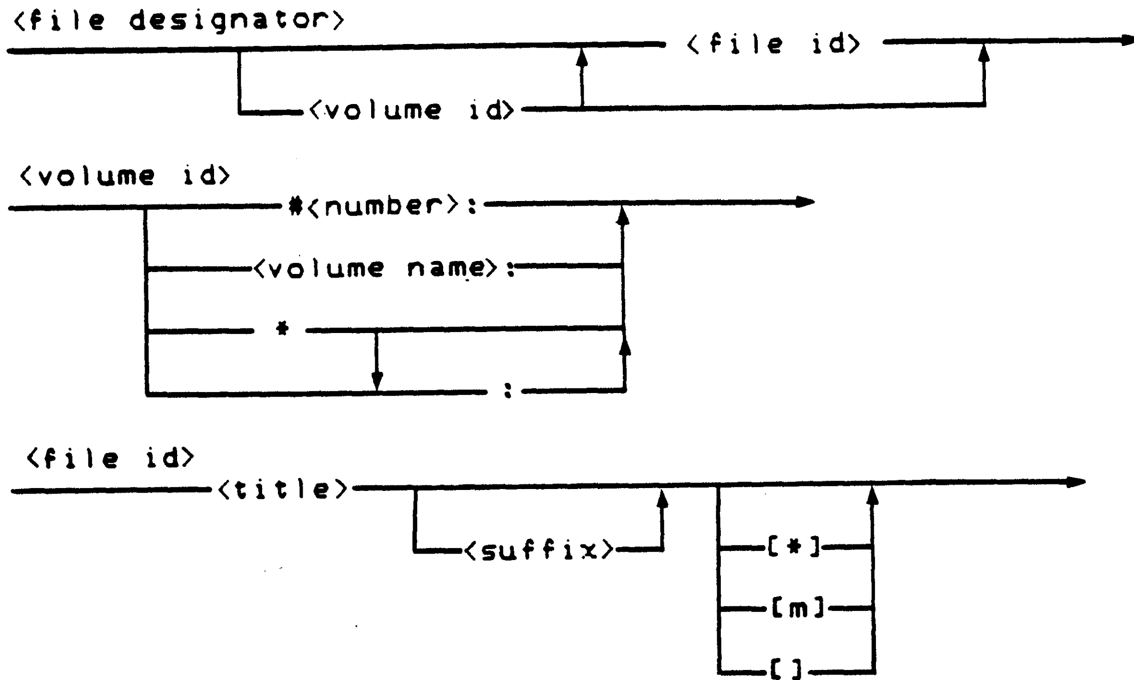
The file length specifiers "[0]" and "[]" are equivalent to a null length specifier; they allocate the largest space available.

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. Disk free space is managed with the K(runch 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 concatenation 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 X(ecute command accepts the file name "SYSTEM.EDITOR." as a valid input text file name identifying the file "SYSTEM .EDITOR" rather than trying to X(ecute "SYSTEM.EDITOR..CODE".

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, concatenate the correct file suffix, and create the output file. Examples 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.

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 Library System

The library system is a collection of pre-programmed routines available for use by user programs. Groups of related routines are packaged as units, identified by unit names of up to eight characters. Units are are maintained in files called libraries. There are four libraries in the library system, each with a different function. They include the system library (section 2.2.0), **intrinsics library (section 2.2.1), the program library** (section 2.2.2), and the the user library (section 2.2.3). Sample library system configurations are presented in section 2.2.4. A program-level description of units appears in section 3.2 of the Programmer's Manual.

A program must import a unit (with the USES statement) before it may call any of the routines contained within the unit. The compiler and the operating system must be able to locate each imported unit in the library system.

At compile time, the compiler searches for each unit imported by the program. The intrinsics library is searched first. If the unit is not found there, the search is continued in the program library, then in the system library, and finally in the user library. This ordering is called the library search path. If a unit is found on the library search path, the compiler uses it in compiling the program, and a record of its use is imbedded in the program's code file; otherwise, a compile error occurs.

At program load time, the operating system searches the library system for each unit imported by a program. If an imported unit cannot be found on the library search path, a diagnostic message, "<unit name> not found", is displayed. If a unit is found, a check is made to determine if the unit has been modified since the program was compiled. If the unit's current version number does not match the version number required by the program, a diagnostic message, "<unit name> is wrong version", is displayed. An additional check is made to determine whether the dynamic variable allocation mechanisms used in the unit conflict with the mechanisms used by other units. If there is a conflict, a diagnostic message, "<unit name> uses wrong heap", is displayed. If all imported units are found, the program and its units are loaded and executed.

A program requiring an obsolete version of a unit may be "updated" by recompiling the program source. An alternate solution is presented in section 2.2.2. Unit version numbers and dynamic variable allocation mechanisms are described in the Programmer's Manual.

Libraries are maintained using the Library utility described in section 8.2.0. They may be examined using the Libmap utility described in section 8.2.1.

NOTE - More than one copy of a unit may exist in the library system. The first copy found in a library search is used. All other copies are ignored.

NOTE - A unit may be stored in a library either with or without its interface text (described in the Programmer's Manual). A unit's interface text must be present if the unit is imported during a program compilation. It need not be present if the unit exists solely for runtime importation. The Library utility can remove interface texts from library files, thereby saving disk space.

2.2.0 System Library

The system library is contained in the SYSTEM.LIBRARY file. It may reside on any online volume. The system determines and fixes its location when the system is bootstrapped or reinitialized. The system library contains both user-defined units and system-related units. They are loaded into memory at program load time, and are unloaded when the program terminates.

The system library should contain units that are stable and reliable. Because of the relatively high overhead incurred in using the Library utility to replace units in the system library, the user library (section 2.2.3) should be used to maintain units requiring frequent modification.

2.2.1 Intrinsics Library

The intrinsics library is contained in the SYSTEM.INTRINS file on the system volume. Units residing in the intrinsics library are treated as user-defined extensions to the operating system. They are loaded into memory at system bootstrap time, and are resident throughout the life of the system. Intrinsic units may allocate data and start tasks that also exist throughout the life of the system.

Since intrinsic units are located at the beginning of the library search path and are permanently resident in memory, programs that use them are compiled and loaded very quickly.

Further information on the intrinsic library is presented in section 2.3.0.2.

WARNING - Introduction of faulty units into the intrinsics library may result in an unbootable system disk.

NOTE - All units imported by intrinsic units must themselves be intrinsic units.

NOTE - The intrinsics library should always contain the PROGOPS unit.

NOTE - The system should be rebooted whenever the intrinsics library is modified.

2.2.2 Program Library

The program library is contained in the current program's code file. Units found in the program library are loaded into memory at program load time, and are unloaded when the program terminates.

A unit installed in a program library is informally referred to as a "private copy" of the unit. Maintaining private copies of imported units assures that when a code file is transferred to another disk volume, all of its imported units are transferred as well.

Since the program library is searched second in a library search, programs that maintain private copies of imported units are compiled and loaded faster than programs that use units found in the system and user libraries.

Recompilation of a program requiring an obsolete version of a unit may be avoided by installing a copy of the obsolete version in the program library. Assuming that the current version of the unit is not installed in the intrinsics library, the obsolete version will be found in the program library during a library search.

A program library is constructed by either the compiler (in the case of inline units -- see the Programmer's Manual for details) or the Library utility.

2.2.3 User Library

The user library consists of a collection of individual library files and code files. They are listed by name (including any '.CODE' suffix) in the user library text file. The user library is searched by searching each file named in the user library text file. File names that cannot be opened are ignored. The default name of the user library text file is *USERLIB.TEXT; it may be changed using the "L=" and "PL=" library redirection options (section 2.4.4.5). User library units are loaded into memory at program load time, and are unloaded when the program terminates.

The user library should contain units that are frequently updated. Since the library may consist of individual code files, units may be compiled directly into the library by naming the unit's code file in the user library text file. This is valuable during unit development since each execution of a program that imports a unit (i.e. a unit test suite) uses the current copy of the unit; no further binding is required. Note that this flexibility carries a time penalty since the user library is at the end of the library search path.

The user library text file is maintained by the system editor.

2.2.4 Library Configuration Examples

The library system may be configured in a number of ways. Some configurations involve tradeoffs of dedicated memory space, program load times, disk space economy, and functionality. Other configurations provide unique capabilities. Sample configurations are presented in sections 2.2.4.0 and 2.2.4.1.

2.2.4.0 Resource Tradeoffs

Several system utilities (e.g. Library, Patch, Drive.Con, etc) make heavy use of units, particularly the PROGOPS, SCCNTRL, PATTERNMATCH, NUMCON, SPOOLER, DIRINFO, SYSINFO, and APPPROCS units. Various tradeoffs are made in placing these units in one library instead of another.

The library configuration on the Pascal system release disk appears as follows:

SYSTEM.INTRINS	SYSTEM.LIBRARY
PROGOPS	COMMANDIO
	SYSUTIL
	SPOOLER

The rest of the units reside in the program libraries contained in each utility's code file. Although the logistical advantages of program library usage are realized, the utility code files occupy much more disk space than is necessary.

Disk usage can be reduced by using the Library program to copy each unit to either the intrinsics library or the system library, then to remove each copy of the unit from the utilities' code files. If the units are placed in the intrinsics library, they occupy memory space during system operation, but the utility programs are loaded even faster. If the units are placed in the system library, utility programs are loaded somewhat slower, but disk space is still liberated.

NOTE - The PROGOPS unit _must_ reside in the intrinsics library.

2.2.4.1 Unique Capabilities

Certain units have properties that manifest themselves as operating system extensions when these units are installed in the intrinsics library. The SPOOLER unit is a good example.

The SPOOLER unit accepts a list of text files and prints the contents of each file on a printer. When the SPOOLER unit is imported by a program, and it is installed in the system library, a program library, or the user library, it is resident in memory only until the program terminates. The operating system prevents program termination until the SPOOLER unit is no longer active. Thus, all printing must be complete before another program may be executed.

When the SPOOLER unit is installed in the intrinsics library, it is said to be imported by the operating system; the operating system may not terminate until the SPOOLER is inactive. Programs that import the unit, however, may terminate while the SPOOLER is active. Thus, other programs may execute while printing is in progress.

2.3 System Configuration

Many aspects of the Advanced Operating System may be customized or configured by the user. Users may configure the system to access new devices by installing either pre-existing or user-programmed device drivers (section 2.3.1). The system may be configured to operate with various terminals by installing either pre-existing or user-defined terminal information files (section 2.3.2). The system user interface may be customized by modifying the prompt line processor (section 2.3.3) or the execution error and breakpoint processors (section 2.3.4). Various performance optimizations may also be applied (section 2.3.5).

2.3.0 Operating System Libraries

Many operating system customizations involve changes in units resident in _operating system libraries_. These libraries include the _system support library_, the _drivers library_, and the intrinsics library. (The intrinsics library is also a part of the library system described in section 2.2). These libraries contain routines used to control program execution, provide both high- and low-level I/O, and miscellaneous system functions.

All operating system libraries must reside on the system disk. Units are loaded from these libraries in order to boot the system. All operating system libraries are maintained using the Library utility described in section 8.2.0.

2.3.0.0 System Support Library

The system support library is contained in the SYSTEM.PASCAL file. It contains major parts of the operating system including the system GOTOXY procedure, the execution error and breakpoint processors, and overlays that implement floating point I/O, transcendental function evaluation, extended precision integer arithmetic, and extended memory management.

Most of the units contained in this library are memory-resident throughout the execution of the Pascal system. The system overlays are not memory-resident unless they are called by a running program (see section 2.3.5). The GOTOXY, execution error, and breakpoint processors may be replaced by the user (see sections 2.3.2 and 2.3.3).

2.3.0.1 Drivers Library

The system drivers library is contained in the SYSTEM.DRIVERS file. It contains I/O driver units used by the system in communicating with system devices. I/O driver units are discussed in section 2.3.1.

NOTE - System I/O drivers may also reside in the intrinsics library. See section 2.3.0.2 for details.

2.3.0.2 Intrinsics Library

The intrinsics library is contained in the SYSTEM.INTRINS file. It contains routines common to both the operating system and user programs. Each unit contained in the intrinsics library is loaded and initialized when the system is bootstrapped.

This library must contain the PROGOPS unit. It may also contain user units. Such units are treated as extensions to the operating system. See section 2.2.1 for details.

NOTE - System I/O drivers may reside in the intrinsics library instead of the drivers library. Such drivers may be imported both by operating system units and by user programs.

2.3.1 I/O System Configuration

Physical units (section 2.1.2) correspond to I/O devices. Communication with I/O devices is performed by routines organized into _I/O driver units_ and installed in the drivers library (section 2.3.0.1). The system may be configured to access new devices by installing I/O drivers capable of communicating with those devices.

I/O driver units for several standard devices exist in the ALL.DRIVERS library file. A partial list of devices supported in this library includes the DEC RL-02 hard disk, DEC RX-02 floppy disk, DEC TM-11 magnetic tape, PRIAM hard disk, 5 1/4" Winchesters, DEC DLV-11 serial port, DEC LPV-11 parallel printer, PDQ-3 console, PDQ-3 system clock, and the PDQ-3 floppy driver. Instructions for programming drivers not contained in this library are provided in Programmer's Manual.

The Pascal system is configured to access a new device by installing the device's I/O driver unit in the driver library and specifying its physical unit number in the SYSTEM.DRVINFO file. The I/O driver unit is installed in the driver library using the Library utility.

New physical unit numbers may be assigned without regard to device type or function starting at 29 and continuing through 63. Since unit numbers between 0 and 28 are in common use in many programs, devices assigned to these numbers should be functionally compatible with currently assigned devices. A table of existing devices may be found in section 2.1.2.

Most I/O drivers are capable of communicating with several devices of the same type (i.e. a floppy driver communicates with several floppy disk drives). Some I/O drivers may partition a single physical device into many logical devices. Each device with which an I/O driver communicates is identified by a _logical device number_. Each driver maintains its own list of valid logical device numbers, usually beginning with 0. One physical unit number may be allocated for each logical device number recognized by an I/O

driver.

The correspondence between a physical unit number and a logical device number of an I/O driver is established using the Drvr.Info utility (section 8.4.0). This mapping is maintained in the SYSTEM.DRVINFO file.

WARNING - Using the Drvr.Info utility to establish a correspondence to an I/O driver not installed in the driver library renders a disk unbootable.

2.3.2 Terminal Configuration

The Pascal system may be used with any terminal that accepts screen formatting commands and generates keyboard sequences containing either 1 or 2 characters. The configuration process proceeds in three steps:

1) Create a SYSTEM.MISCINFO file containing screen formatting commands used by system utilities. This is done using the Setup utility described in section 8.3.1.

2) Construct the GOTOXY procedure appropriate for the terminal. This process is described in section 8.3.0.

3) Add to the SYSTEM.MISCINFO file the additional information necessary to use the system editor. This is done using the ASS utility described in section 8.3.2.

NOTE - The SYSTEM.STARTUP program on the AOS release disk attempts to create a work disk containing the correct terminal configuration for the system terminal. The procedures described above should only be necessary if the system terminal is one for which no configuration has been provided.

2.3.3 System Shell

A shell is a program the performs user interface functions and is capable of starting programs in response to user commands. The SYSTEM.SHELL program, located on the system disk, performs all system-level user interface and program invocation processing. It displays the bootstrap welcome message (section 2.4.0) and the system prompt line. The shell program invokes system programs and user programs in response to the commands described in section 2.4.

The system shell is executed when the system is bootstrapped or reinitialized. The standard shell immediately checks for the existence of the SYSTEM.STARTUP program on the system disk. If this program exists, the shell executes it before displaying the system prompt line. If a startup program does not exist, the shell prints a welcome message and checks for the existence of the PROFILE.TEXT command file on the system disk. If the profile exists, it is S(ubmitted to the command interpreter (see chapter 7). Otherwise, the system prompt line is displayed. The shell

implements program chaining, workfile execution, and system program execution.

The standard shell may be replaced by a user-programmed shell simply by replacing the SYSTEM.SHELL file on the system disk and either rebooting or reinitializing the system. Details on programming shells are provided in the Programmer's Manual.

2.3.4 Execution Error and Breakpoint Processing

The system execution error and breakpoint handlers are programmed as units and installed in the system support library. The execution error processor is contained in the EXCEPTION unit, and the breakpoint processor is contained in the HALTUNIT unit. The execution error processor is responsible for notifying the user of an execution error (section 2.0.0). The breakpoint processor implements the HALT intrinsic described in the Programmer's Manual. Custom execution error and breakpoint processing may be provided by reprogramming and replacing these units. Further details are provided in the Programmer's Manual.

2.3.5 Performance Optimizations

System performance may be improved substantially by judicious organization of the system files. While some of the optimizations presented in this section result from reduced compute time, most of the optimizations involve minimizing the amount of time spent waiting for the completion of disk I/O. Most disk I/O time is spent waiting for the disk read head to become positioned over the desired disk file; there is a direct relationship between the distance the disk read head must travel and the duration of a disk I/O. Since accesses to the bootstrap volume's directory accompany most disk operations, files closest to the volume's directory (located in blocks 2-5) are accessed in the shortest amount of time.

The system bootstrap accesses the system support library, the drivers library, the intrinsics library, and the SYSTEM.DRVINFO file. The time required to bootstrap the system is minimized when these files occupy the blocks closest to the bootstrap volume's directory. In addition, the drivers library should be ordered so that the driver unit most frequently mentioned in the SYSTEM .DRVINFO file is at the top of the Library utility display. Subsequent entries should appear in decreasing frequency of usage. This reduces the compute time necessary to bootstrap the system.

Before displaying the system prompt, the system shell loads two overlays from the intrinsics library. For speedy display of the system prompt after program termination, the SYSTEM.SHELL file should be located close to the SYSTEM.INTRINS file on the boot disk. Program load time is reduced when the SYSTEM.SHELL and SYSTEM.INTRINS are located close to the bootstrap volume's directory.

The system support library contains overlays that may be loaded automatically when a program is executed. Program load time is reduced when the system support library is in close proximity to the intrinsics library and system shell. Note that these overlays, which include the LONGINTS, HEAPOPS, TRANSCEND, and PASCALIO units, may be installed in the intrinsics library instead of the system support library. This causes the overlays to be memory-resident throughout system execution, thus reducing program load time even further.

The recommended ordering of system files on the bootstrap disk is:

SYSTEM.MISCINFO
SYSTEM.DRVINFO
SYSTEM.INTRINS
SYSTEM.SHELL
SYSTEM.DRIVERS
SYSTEM.PASCAL
SYSTEM.FILER
SYSTEM.EDITOR
SYSTEM.COMPILER
SYSTEM.LIBRARY
SYSTEM.SYNTAX

2.4 Commands and Operation

This section describes the operating system commands and operation. Section 2.4.0 explains how to start the system. Work files are described in section 2.4.1. The system's state flow is described in section 2.4.3. Automated invocation of system parts is described in sections 2.4.1.1 and 2.4.2. Section 2.4.4 describes all commands available in the system prompt.

2.4.0 Bootstrapping the System

This section describes how to bootstrap the UCSD Pascal system on the PDQ-3. Bootstrapping starts by applying power to the PDQ-3 and ends when the system prompt line is displayed.

The UCSD Pascal system may be bootstrapped from either a floppy disk drive or a hard disk drive, depending on the system hardware configuration. The following steps are taken to bootstrap the system:

1) Both the system console and the PDQ-3 must be powered-up, and the system console must be connected to the PDQ-3. The PDQ-3 Hardware User's Manual and the system console operator's manual should be consulted for instructions on first time operation.

2) A '#' should appear on the system console screen when the PDQ-3 RESET button is depressed. This is the system monitor prompt. It indicates that the PDQ-3 is ready to accept commands. (The system monitor is described in chapter 7).

3) The bootstrap command may be issued from the system console. A bootstrap command consists of two digits followed by 'R'. The first digit indicates the type of bootstrap device. If the system is configured with a hard disk drive, '0' indicates a hard disk drive, and '1' indicates a floppy disk drive. If the system has no hard disk drives, the first digit should be '0'. The PDQ-3 Hardware User's Manual may be consulted for the exact meaning of this digit for a given hardware configuration. The second digit is the bootstrap drive number (0 for hard disk drive 0 or the left floppy drive, 1 for hard disk drive 1 or the right floppy drive).

   The system automatically distinguishes between floppies recorded in single- and double-density formats. The floppy drives are normally configured for single-sided operation at system bootstrap time. They may be configured for double-sided operation by adding 4 to the second digit. Note that PDQ-3 system software is distributed on single-sided media only. Attempts to access single-sided disks (or double-sided disks recorded on only one side) in double-sided mode result in fatal errors. The floppy drives may be reconfigured during system operation by using the disk type key described in section 1.4.3.4.

Examples of bootstrap commands are given below. The hardware configuration is assumed to contain two floppy drives and a hard disk drive.

Command	Meaning
R	Boot from hard drive 0 (floppies configured single-sided)
4R	Boot from hard drive 0 (floppies configured for double-sided)
10R	Boot from floppy drive 0 (floppies configured for single-sided)
15R	Boot from floppy drive 1 (floppies configured for double-sided)

NOTE - Leading zeroes in the bootstrap command may be omitted.

The system bootstrap is complete when the welcome message is displayed in the center of the console screen:

Welcome SYSTEM: to

ACD's U.C.S.D. Pascal Version AOS 1.0

Current date is 30-May-82

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

NOTE - The reserved file names PROFILE.TEXT (chapter 7) and SYSTEM.STARTUP (section 2.1.4.3.0) affect the behavior of the system at system bootstrap time.

NOTE - If the welcome message or system prompt seem to be on the wrong part of the screen, the system may need to be reconfigured for use with the system console. See section 2.3.2.

NOTE - The system may be rebooted either by pressing the RESET button or using the monitor key to reenter the system monitor. A bootstrap command may then be issued.

2.4.0.0 Bootstrap Failure

Bootstrap failure may result from several causes. Both system hardware malfunctions and corrupted system software should always be suspected in cases of bootstrap failure. More common causes include:

Symptom	Possible Cause
Disk reads data, but stops abruptly with no message	An attempt has been made to bootstrap the Pascal system on a PDQ-3 system for which the software has not been configured. Consult the factory.
	An attempt has been made to bootstrap a single-sided floppy disk with the drives configured for double-sided operation.
	An attempt has been made to bootstrap a disk that does not contain a SYSTEM.SHELL file.
An error message "Fatal I/O Error" appears	An attempt has been made to bootstrap a single-sided floppy disk with the drives configured for double-sided operation.
An error message "Need xxxxxxxx" appears	An attempt has been made to bootstrap a disk that either does not contain a SYSTEM.DRVINFO, SYSTEM.DRIVERS, SYSTEM.PASCAL and SYSTEM.INTRINS file or those files have been corrupted.
	The drivers library does not contain all drivers enumerated in the SYSTEM.DRVINFO file.

2.4.1 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, 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 may be saved in a named disk file.

Work file operations are described in section 2.4.1.0. The effects of a work file on system operation are described in section 2.4.1.1.

2.4.1.0 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 may be modified as a result of compiling a program; the compiler's output may be the new 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 may be found in the sections describing the commands and system parts mentioned in this section.

2.4.1.1 Work File Effects on System Behavior

The editor, and compiler 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 executes the current work file. If only a work text file exists, the R(un command invokes the compiler to compile the work text file into the work code file. The new work code file is then executed. 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.

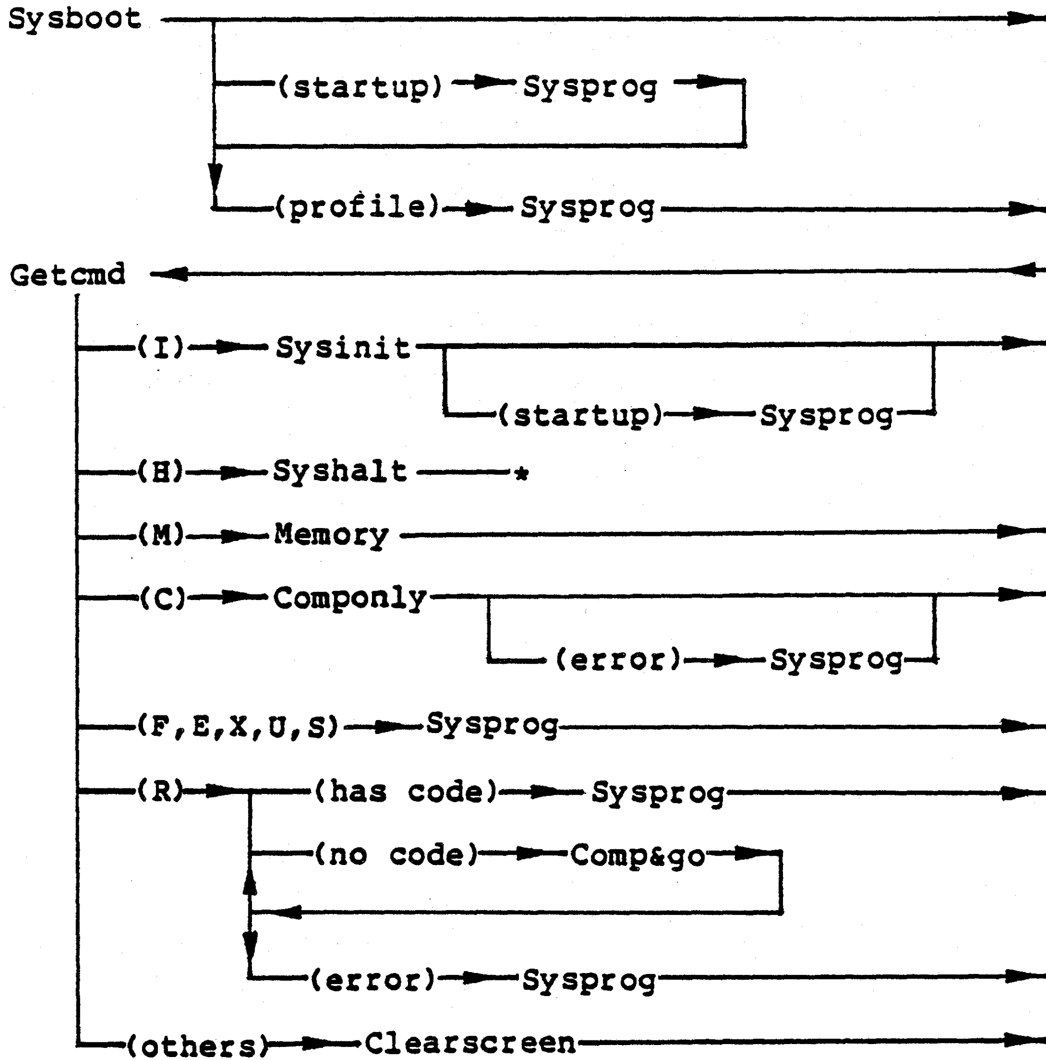
A formal specification of system behavior with respect to work files is presented in section 2.4.3.

2.4.2 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 displays the name of the work file. Responding with a <carriage return> informs the editor that the file name is correct. It then allows the user to jump to the site of the compilation 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.

2.4.3 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.

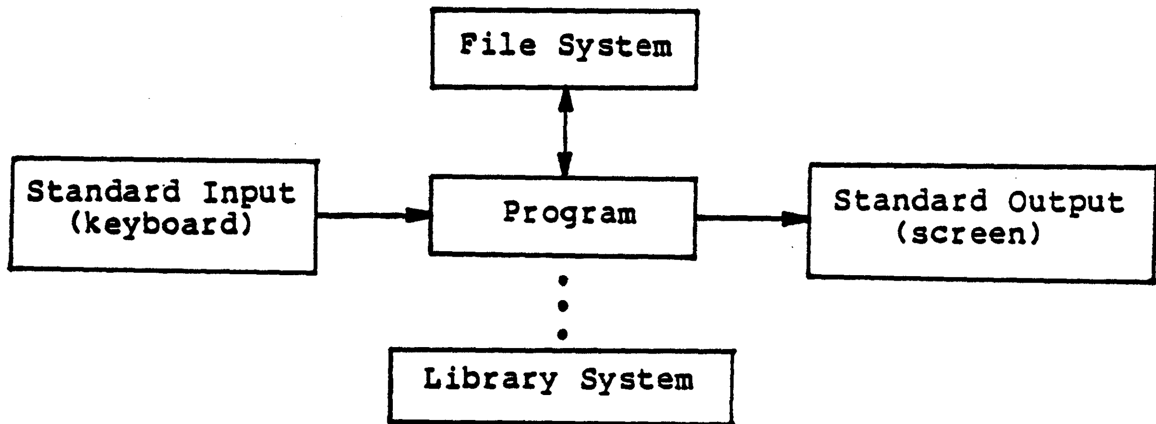


Note: "(error)—►—Sysprog" sequence invokes the editor.

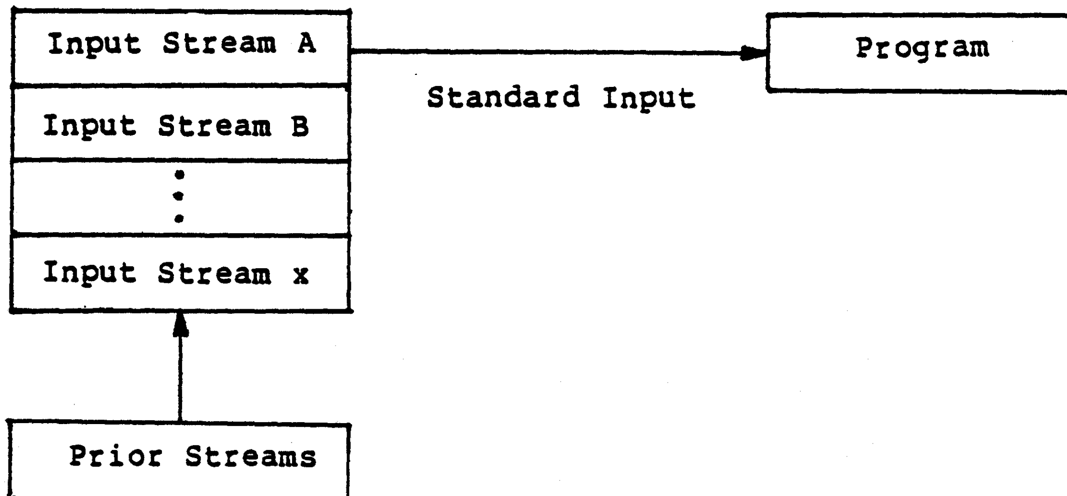
Descriptor	Definition
Sysboot	system bootstrap
Sysinit	system reinitialization
Syshalt	system halt
Getcmd	system prompt displayed
Clearscreen	console display cleared
Sysprog	system/user program invocation
Componly	invoke compiler only
Comp&go	invoke compiler and run work file
Memory	available memory space is reported
(startup)	SYSTEM.STARTUP code file on system volume
(profile)	PROFILE.TEXT command file on system volume
(<letter>)	system prompt command received
(others)	non-command character received
(has code)	work code file exists
(no code)	work file is text only
(error)	compiler syntax error

2.4.4 I/O Redirection Options

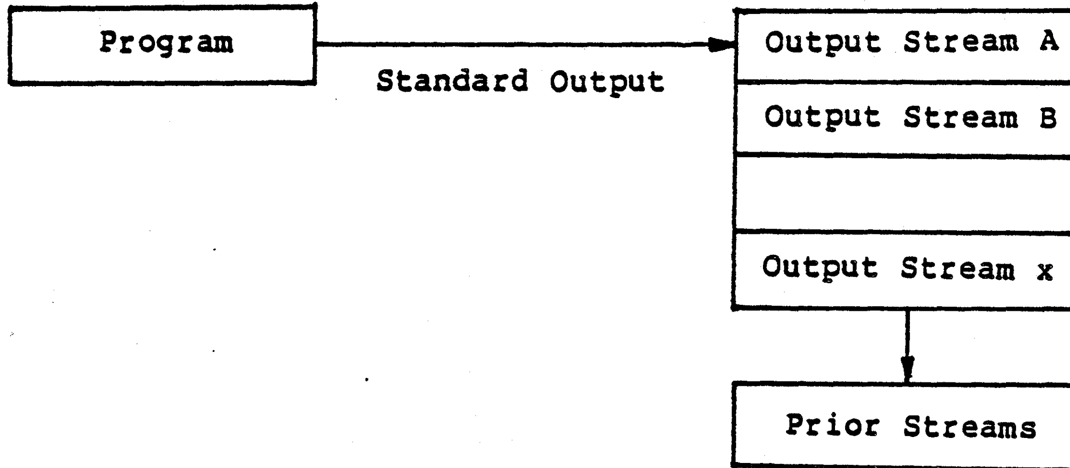
In general, a program accepts input and produces output. A program may receive data from several sources, including the *standard input*, the file system, or system devices. It may send data to several destinations, including the *standard output*, the file system, or system devices. In addition, routines provided by the library system may be used during a program's execution. I/O redirection options are used to modify accesses to the standard input, standard output, file system, and library system without modifying the program itself. They are specified by the user at program execution time.



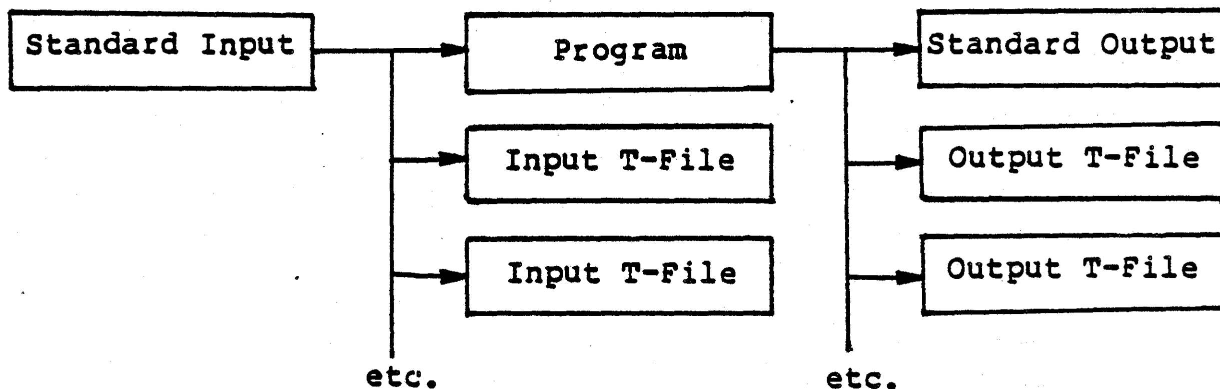
An *input stream* is used to satisfy read requests to the standard input. The default input stream consists of characters read from the system console keyboard. *Input redirection options* may designate disk files and/or serial volumes as input streams which supercede the existing input stream. When an input redirection option specifies more than one file, the resulting input stream consists of the concatenation of all of the specified files (i.e. when one file is exhausted, input is taken from the next file). When an input stream is exhausted, input requests are satisfied by the original input stream.



An _output stream_ is used to satisfy write requests to the standard output. The default output stream consists of the system console screen. _Output redirection options_ may designate disk files and/or serial volumes as output streams which supercede the existing output stream. When an output redirection option specifies more than one file, the resulting output stream consists of the first file, followed by the next file, etc. When one file is filled, output continues to the next file. When an output stream is full, output operations are performed on the original output stream. Note that data read from the standard input is normally echoed to the standard output by the system.



A _t-file_ is used to generate a copy of an input stream or an output stream. T-files connected to the standard input receive a copy of all data read from the input stream. T-files connected to the standard output receive a copy of all data written to the output stream. _T-file options_ designate disk files and serial volumes as t-files. When a t-file option specifies more than one t-file for a given stream, all t-files receive copies of the same data in parallel.



The file system prefix indicates the default volume during accesses to the file system in cases when no volume name is specified (see section 2.1.3.4). _Prefix options_ specify a new file system prefix. The user library contains a list of library file names which the operating system may open while preparing a program for execution (see section 2.2.3). _Library options_ specify a new user library

file name.

2.4.4.0 Execution Option Lists

An *execution option list* is a sequence of I/O redirection options and/or a program name. It may contain input redirection options, output redirection options, t-file options, prefix options, library options, and/or a program name. Execution option lists are used as arguments in the invocation of a program (e.g. with the prompt line X(ecute command, the Chain intrinsic, and the ProgCall and ProgSetup intrinsics -- see the Library User's Manual for details).

I/O redirection options are specified by an option identifier and a list of file names, volume names, and literal strings. A list may contain a single blank at the beginning (to separate it from the option identifier). List elements are separated by either commas or semi-colons, and the list is terminated by one or more blanks. Literals are delimited by '"'s, and imbedded commas are converted into carriage returns. A '"' may be specified inside a literal by '""'. Literals are useful only in input redirection lists; they are ignored elsewhere. In cases where an option identifier list occurs more than once in an execution option list, the associated file name lists are concatenated, separated by commas.

Examples of execution option lists:

```
p= #5:
prose p=#5: o=foon.text,*foon.text i=script.text to=#1:
*System.shell.  pi=inp.text po=bucket: ti=script.text;remout:
```

Syntax for execution option lists:

```
<execution-option-list> ::=  [<program-name>][<option-list>]

<program-name> ::=  UCSD Pascal program name (without .Code)

<option-list> ::=  <redir-option>{[" "]<redir-option>}

<redir-option> ::=  <option-name>=[" "]<file-list>

<file-list> ::=  <file-name>{<file-delim><file-name>}

<file-name> ::=  UCSD Pascal file name

<file-delim> ::=  ,  |  ;

<option-name> ::= O  |  I  |  TO  |  TI  |  P  |  L  |
                  PO |  PI |  PTO |  PTI |  PP |  PL
```

The following table lists each redirection option. Options are listed in the order in which they are processed before program execution.

Option Name	Redirection Option
P, PP	Prefix volume (section 2.4.4.4)
L, PL	User library name (section 2.4.4.5)
I, PI	Input redirection (section 2.4.4.2)
TI, PTI	T-file on input (section 2.4.4.3)
O, PO	Output redirection (section 2.4.4.1)
TO, PTO	T-file on output (section 2.4.4.3)

2.4.4.1 Output Redirection Options

The O= and PO= redirection options accept lists containing file names and volume identifiers. At program execution time, each file is opened for output. The resulting output stream consists of all files that were successfully opened; files that could not be opened are ignored. All files comprising the output stream are closed after the program terminates.

The O= and PO= options each create an output stream. The stream specified by the O= option is created before the stream specified by the PO= option. Thus, the stream created by the PO= option supercedes the stream created by the O= option.

Example of output redirection option use:

```
O=Foon.Text,#10:Farkle.Text  PO=Freep.Text[5],
                                  3CharactersTooLong
```

An output stream consisting of the files Foon.Text and #10:Farkle .Text is constructed as a result of the O= option. Next, an output stream consisting of the file Freep.Text is constructed (3CharactersTooLong cannot be opened, so it is ignored). The first five blocks of program output are written to Freep.Text, then output is directed to Foon.Text, and then to Farkle.Text. If Farkle.Text becomes full, the output stream existing before the creation of the O= stream is used. This is normally the system console screen unless the standard output was redirected prior to program execution (see section 2.4.4.5).

NOTE - A serial volume cannot be filled. Therefore, files following a serial volume in the output stream will never be used. Output to the BUCKET: serial volume is thrown away. This volume may be used to execute a program without viewing its output.

NOTE - Programs normally write to the pre-declared file variable, OUTPUT, in order to write to the standard output. The same effect may be achieved by writing to the STANOUT: serial volume. Output redirection may be performed only on programs that write to either the pre-declared file variable, OUTPUT, or to the STANOUT: serial volume. Note that including the STANOUT: serial volume in an output stream has no effect. See the Programmer's Manual for

details.

2.4.4.2 Input Redirection Options

The I= and PI= redirection options accept lists containing file names, volume identifiers, and literals. At program execution time, each file is opened for input. Temporary files containing literals are also created. The resulting input stream consists of all files that were successfully opened; files that could not be opened are ignored. All files comprising the input stream are closed after the program terminates.

The I= and PI= options each create an input stream. The stream specified by the I= option is created before the stream specified by the PI= option. Thus, the stream created by the PI= option supercedes the stream created by the I= option.

Example of input redirection option use:

```
I=Krap.Text  PI="FirstInp,",Kook.Text,"5",Kreep.Text
```

An input stream consisting of the contents of the Krap.Text file is created as a result of the I= option. Next, an input stream consisting of the literal "FirstInp<cr>", the contents of Kook .Text, the literal "5", and the contents of Kreep.Text is created. When the input stream constructed for the PI= option is exhausted, the contents of Krap.Text is used. If Krap.Text is exhausted, the input stream existing before the creation of the I= stream is used. This is normally the system console keyboard unless the standard input was redirected prior to program execution (see section 2.4.4.5).

Files comprising an input stream may be generated by a program, by a t-file option (section 2.4.4.3), or by the system editor.

NOTE - A serial volume cannot be depleted. Therefore, files following a serial volume in the input stream will never be used. Reads from the BUCKET: serial volume return <eof>. This volume may be used to provide a constant null input.

NOTE - Programs normally read from the pre-declared file variables, INPUT and KEYBOARD, in order to read from the standard input. The same effect may be achieved by reading from the STANIN: serial volume. Input redirection may be performed only on programs that read from either the pre-declared file variables, INPUT and KEYBOARD, or from the STANIN: serial volume. Note that including the STANIN: serial volume in an input stream has no effect. See the Programmer's Manual for details.

2.4.4.3 T-File Options

The TI=, TO=, PTI=, and PTO= t-file options accept lists containing file names and volume identifiers. At program execution time, each file is opened for output; files that could not be opened are

ignored. All t-files are closed after the program terminates.

The TI= and PTI= options create t-files attached to the standard input. The TI= option is processed before the PTI= option. The TO= and PTO= options create t-files attached to the standard output. The TO= option is processed before the PTO= option.

Example of t-file option use:

```
TI=Script.Text TO=Printer:,Copy.Text
```

All data read from the standard input is copied to the file Script.Text. This copy may be used to create an input stream (using the I= option) in order to duplicate the current input stream for subsequent executions of the current program or for later analysis. All data written to the standard output is written both to Printer: and to Copy.Text. Note that this includes all data read from the standard input and echoed to the standard output. Therefore, output t-files contain an exact record of program execution.

NOTE - Note that naming the STANOUT: serial volume as a t-file has no effect.

WARNING - When a t-file is full, the program is interrupted with I/O execution error 8 ("No room on vol"). A serial volume cannot be filled.

2.4.4.4 Prefix Options

The P= and PP= prefix options set the file system prefix to a specified volume identifier (similar to the P(refix command in the Filer -- see section 3.2.10). The volume identifier may contain up to seven characters; a trailing ':' is optional. The PP= option sets the file system prefix only for the duration of the program execution. The prefix then reverts back to its original value. The P= option sets the prefix permanently. When both the P= and PP= options are used, the prefix is determined by the PP= option during program execution, and by the P= option thereafter.

Example of prefix option use:

```
P=#10: PP=*
```

NOTE - An execution option list may contain a P= option without containing a program name. In this case, the file system prefix is set according to the P= option and no program is executed.

2.4.4.5 Library Options

The L= and PL= library options set the user library name (section 2.2.3) to a specified file identifier. The file identifier may contain up to 23 characters; a trailing '.Text' suffix must be supplied if the user library is a text file. The PL= option sets

the user library name only for the duration of the program execution. The library name then reverts back to its original value. The L= option sets the library name permanently. When both the L= and PL= options are used, the library name is determined by the PL= option during program execution, and by the L= option thereafter.

Example of library option use:

```
L=MyLib.Text PL=#10:OtherLib.Text
```

NOTE - An execution option list may contain a L= option without containing a program name. In this case, the user library name is set according to the L= option and no program is executed.

2.4.4.5 System I/O Redirection

With the exception of the L= and P=, all redirection options act only during the execution of a single program. Since the system prompt line processor (*System.Shell, see section 2.3.3) is itself a program, I/O redirection may be applied to a series of programs by re-executing the prompt line processor using redirection options.

For example, three programs could be executed sequentially with preprogrammed input and a copy of the output going to both the console and the printer. This is performed using the following execution option list:

```
*System.Shell.  I="xRichProg,",RichInp.Text,
                "xJoelProg,",JoelInp.Text,"H" TO=Printer:
```

The system command processor, *System.Shell, is re-executed. The Printer: is opened as an output t-file. The input 'xRichProg<cr>' invokes the e(Xecute command and executes the RichProg program. The input stream is then provided by the RichInp.Text file. The JoelProg program is then executed in a similar fashion. Finally, the command processor execution is terminated by invoking the H(alt command. Note that in this example the contents of the RichInp .Text and JoelInp.Text files must correspond exactly with the respective program input requirements; otherwise, the intended input stream for the next program is affected. A safer way of calling these programs is:

```
*System.Shell.  I="xRichProg I=RichInp.Text;Console:,
                  xJoelProg I=JoelInp.Text;Console:, H"
                TO=Printer:
```

In this case, if the RichInp.Text file provides more data than RichProg requires, the excess data is ignored. If it does not provide enough data, additional data is accepted from Console:.

2.4.5 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 prompt line has the following form:

```
Command: X(ecute, S(ubmit, R(un, F(ile, E(dit, C(omp,
                                   M(emory, H(alt, ? [1.0]
```

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

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

Typing "?" again displays the original prompt line.

2.4.5.0 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.

2.4.5.1 C(ompile

Executes the program SYSTEM.COMPILER. The compiler translates a Pascal source program into a code file. If the file *SYSTEM.WRK .CODE does not already exist, the resulting code file becomes the new work 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> names the code file *SYSTEM.WRK.CODE[*] and it becomes the new work code file. Each occurance of the "$" wildcard is replaced by the name of the source file (without its volume designator).

Compiler operation is described in chapter 5.

2.4.5.2 E(dit

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

If a work text file is present, it is used as the default input file. Otherwise, the editor prompts for the name of an input file.

Editor commands are described in chapter 4.

2.4.5.3 F(ile

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

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

Filer commands are described in chapter 3.

2.4.5.4 H(alt

Stops the system and shuts down all I/O devices. The only way to restart the system is to reboot (see chapter 7 and section 2.4.0).

2.4.5.5 I(nitialize

Causes the system to reinitialize all of its state information. This involves termination of all nested shells and 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.

2.4.5.6 M(emory

Reports the amount of free memory left over after memory has been allocated for the global data of the most recently executed program. The greatest amount of memory is reported immediately after the system has booted or been reinitialized, when no program globals are allocated.

2.4.5.7 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 2.4.1.1 and 2.4.3.

2.4.5.8 S(ubmit

Executes the program X.CODE on the system volume. X.CODE is assumed to contain the command file interpreter program, which is used to process command files.

Command file specification and operation are described in chapter 7.

2.4.5.9 U(ser restart

Reexecutes the last program. This command does not work immediately after system bootstrap or reinitialization.

2.4.5.10 X(ecute

Executes the specified code file.

X(ecute prompts for an execution option list (section 2.4.4.0) containing a code file name. The file suffix ".CODE" is automatically appended to the file name.

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 and 3.5 describe methods for recovering inadvertently removed disk files and directories.

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[1.0]
```

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[1.0]
```

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 3.2.14.2 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 do not store information reliably.

The filer prompts for the volume to be scanned. Then it prompts:

Scan for <total blocks on volume> blocks? (Y/N)

Typing 'Y' begins a block-by-block scan of the entire volume; typing 'N' generates the prompt:

Scan for how many blocks?

Typing a number between 1 and the total number of blocks on the volume begins a block-by-block scan starting at block 0 and continuing for the number of blocks specified.

During the scan, each block of the named disk is checked for problems. If the block is bad, a warning message containing the block number is printed out.

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

Scanning for bad blocks is performed much more conveniently with the utility program Bad.Blocks (section 8.0.4).

3.2.2 C(hange

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:

```
STUFF:
WONIT.TEXT        4  15-Jan-81    10    512  Textfile
DIRT.CODE         6  10-Jan-81    14    512  Codefile
< UNUSED >       12               20
SINS3A.TEXT      48   5-Jan-81    32    512  Textfile
UROSE.CODE       33  24-May-80    80    512  Codefile
CROSS3.CODE      35  26-Nov-80   113    512  Codefile
KANT.TEXT        32  12-May-81   148    512  Codefile
SELL.DATA        16  15-Jan-81   180    314  Datafile
< UNUSED >      298              196
7/7 files<listed/in-dir>, 184 blocks used, 310 unused,
298 in largest
```

3.2.5 G(et

Assigns 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, removing the current workfile and its backup. Typing any other character aborts G(et, preserving 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 - G(et does not create a work file. If the work file SYSTEM.WRK exists, it is removed. The specified disk files become the source of the new work file.

Work files are described in section 2.4.1.

3.2.6 K(runch

Merges all unused disk space into a single contiguous unused area. This is done by moving all disk files to one end of the disk or the other, depending on the "starting block" of the crunch. The starting block is specified by the user. Files preceding the starting block are moved to the beginning of the disk; files following the starting block are moved to the end of the disk. The E(xt-dir command is useful in determining the starting block of a file. Note that starting a crunch at the end of the disk moves all files to the beginning 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 SYSTEM.PASCAL, SYSTEM.INTRINS, SYSTEM.DRIVERS, or SYSTEM.SHELL files are moved while K(runching the system disk, the system must be rebooted after the K(runch.

WARNING - Nothing must happen to the system while a crunch 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, generating the following prompt:

```
From end of disk, block <last block> ? (Y/N)
```

A 'Y' response initiates the crunch from the end of the disk. An 'N' response generates the following prompt:

```
Starting at block # ?
```

Typing any block number between 1 and the last block of the disk initiates a crunch which moves all files preceeding that block to the beginning of the disk, and all files following that block to the end of the disk.

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 standard output.

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 last date of access. (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.

NOTE - When listing a directory to a file contained on the volume being listed, the list file appears as a very large temporary file (date = 100).

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:

```
STUFF:
WONIT.TEXT         4  15-Jan-81
DRATIT.TEXT       48   5-Jan-81
SPOSE.CODE        33  24-May-80
UROSE3.CODE       35  26-Nov-80
CONT.TEXT         16  15-Jan-81
KARS.TEXT         18   5-Jan-81
SIMPLE.TEXT        8   3-May-81
7/7 files<listed/in-dir>, 172 blocks used, 322 unused,
 280 in largest area
```

3.2.8 M(ake

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 and 3.4 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 P(refix volume

Changes the current file system prefix to the volume specified.

This prompt is displayed:

    Prefix titles by what vol?

A valid response contains a volume identifier; any associated file identifier 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).

NOTE - The prefixed volume may be set at the system prompt using the 'P=' redirection option (see section 2.4.4.4).

3.2.11 Q(uit

Exits the filer and returns control to the system prompt.

NOTE - The system disk should be remounted in the proper disk drive before typing Q(uit.

3.2.12 R(emove

Removes files from the directory.

The specified files 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, the filer displays this prompt:

Update directory?

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

NOTE - R(emove does not remove files whose date contains a year of 100. These are temporary files and may be removed only by reinitializing the system.

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.

NOTE - S(ave only saves the most current version of the text file. If SYSTEM.WRK.BACK exists, it retains the name SYSTEM.WRK.BACK and remains on the system volume as is, no matter what volume the work file is saved on.

3.2.14 T(ransfer

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.

T(ransfer 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 originating 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 3.2.14.2 for problems with T(ransfer.

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 complete, two identical disk volumes named "WORK" are online. Section 3.2.14.1 discusses volume to volume transfers.

    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" should be removed from the drive and replaced with the disk volume "BACKUP", and then <space> should be 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 Volume-to-Volume Transfers

A disk volume may be copied onto another disk volume by specifying volume names for the source and destination files. Volumes may be copied using either one disk drive or two.

Example of a volume-to-volume transfer:

```
Transfer what file? #4:
To where? #5:
```

If the source volume contains a directory, this command generates the following prompt:

```
Transfer <number of blocks on source volume> blocks ? (Y/N)
```

Typing 'Y' indicates that the entire source disk is to be transferred to the destination disk; the <escape> key aborts the transfer. Typing 'N' indicates that the destination disk media contains fewer blocks than the source disk, and the following prompt is generated:

```
Transfer how many blocks ?
```

Entering 0 aborts the transfer. Typing a number between 1 and <number of blocks on source volume> causes only the specified number of blocks to be transferred.

Once the transfer length has been established, the destination disk is checked for the existence of a directory. If a directory already exists, the transfer is verified before the destination is overwritten:

```
Destroy <Volume name> ?
```

Typing 'Y' proceeds with the transfer; typing any other character aborts the transfer.

NOTE - If the source disk contains files beyond the last block of the transfer, those files must be removed from the copy of the directory on the destination disk. In addition, the directory on the destination disk may list unused space beyond the end of the physical medium. The Change.Dir utility (described in section 8.0.6) should be used to align the directory with the size of the media containing it.

3.2.14.2 Transfer Problems and Warnings

WARNING - Unless entire disk volumes are being transferred, 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:

Destroy <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 - The = wildcard 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 - Two volumes with the same name must not be on line at the same time. File commands involving these two volumes may have unpredictable results.

3.2.15 V(olumes online

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

A typical volume display is:

```
Vols on-line:
  0   CLOCK
  1   CONSOLE:
  2   SYSTERM:
  3   KEYBUFR:
  4 # MYDISK:
  9 # PRIAM:
 10 # SYSMAN:
 11 # WORKDSK:
 12 # BACK:
 21   FASTCON:
 22   STANIN:
 23   STANOUT:
 24   BUCKET:
 25 # DOC:
Root vol is - PRIAM:
Prefix is   - SYSMAN:
```

Online disk volumes are indicated by "#". The current system volume (Root vol) and prefixed volume are 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 of the current work file. If the work file is not saved, it is so specified.

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-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 in the specified block range, the following prompt appears:

    File(s) endangered:
    <file name>
    Fix them?

Typing "Y" starts the repair process on the specified 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 hopelessly bad 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 repair 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.

NOTE - A bad block may be permanently fixed using the Format utility (section 8.0.3) to reformat the track containing the bad

block. Be sure to use media appropriate for density and number of sides on the floppies being used (e.g. use double-sided disks only in double sided drives). This is a common cause of apparent bad blocks.

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.

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:

    Duplicate dir ?

If a duplicate directory is desired, "Y" should be typed: typing any other character will not allocate the duplicate directory. The next prompt is:

    # of blocks on the disk ?

Any positive integer may be entered. A single density disk has 494 blocks; a double density disk has 988 blocks; a double sided double density disk has 1976 blocks; see section 8.0.5.0 for the number of blocks contained on a hard disk volume. The next prompt is:

    New vol name?

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

    <volume name> correct?

Typing "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 8.0.3) 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 duplicate directory prompt, the Z(ero command is verified with the prompt:

    Destroy <current volume name>?

Typing "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:

Are there <block number> blks on the disk ? (Y/N)

... 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.

NOTE - The number of blocks on the volume may be changed without Z(eroing the disk by using the Change.Dir utility (section 8.0.6).

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 data written 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 do not 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 8). Data file structures must be known by the user. Code file structures are described in the Architecture Guide.

NOTE - For the recovery of several lost files or in the case of a thrashed directory, the Recover utility is most useful (section 8.1.2).

Example of recovering a lost text file:

Here is a pre-accident directory listing:

```
STUFF:
WASTE.TEXT          4  15-Jan-81    10   512  Textfile
DICE.TEXT          18  15-Jan-81    14   512  Textfile
< UNUSED >         48               32
SPOSE.CODE         33  24-May-80    80   512  Codefile
UROSE3.CODE        35  26-Nov-80   113   512  Codefile
COLOR.DATA         32   5-Jan-81   148   314  Datafile
< UNUSED >         10              180
KIN.TEXT           16  15-Jan-81   190   512  Textfile
SONS.TEXT          18   5-Jan-81   206   512  Textfile
< UNUSED >        270              224
7/7 files<listed/in-dir>, 166 blocks used, 328 unused
```

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

```
STUFF:
WASTE.TEXT          4  15-Jan-81    10   512  Textfile
DICE.TEXT          18  15-Jan-81    14   512  Textfile
< UNUSED >         48               32
SPOSE.CODE         33  24-May-80    80   512  Codefile
UROSE3.CODE        35  26-Nov-80   113   512  Codefile
COLOR.DATA         32   5-Jan-81   148   314  Datafile
< UNUSED >         26              180
SONS.TEXT          18   5-Jan-81   206   512  Textfile
KIN.TEXT           16  15-Jan-81   224   512  Textfile
< UNUSED >        254              230
7/7 files<listed/in-dir>, 166 blocks used, 328 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[10] fills the first 10 blocks of the 26-block unused area that contains the missing file. DUMMY3.TEXT[16] fills the last 16 blocks of the 26-block area, and coincides with the old copy of KIN.TEXT. The directory now appears as:

```
STUFF:
WASTE.TEXT          4  15-Jan-81    10    512  Textfile
DICE.TEXT          18  15-Jan-81    14    512  Textfile
DUMMY1.TEXT        48   5-Jan-81    32    512  Textfile
SPOSE.CODE         33  24-May-80    80    512  Codefile
UROSE3.CODE        35  26-Nov-80   113    512  Codefile
COLOR.DATA         32   5-Jan-81   148    314  Datafile
DUMMY2.TEXT        10  15-Jan-81   180    512  Textfile
DUMMY3.TEXT        16  15-Jan-81   190    512  Textfile
SONS.TEXT          18   5-Jan-81   206    512  Textfile
KIN.TEXT           16  15-Jan-81   224    512  Textfile
< UNUSED >        254              230
10/10 files<listed/in-dir>, 240 blocks used, 254 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 KIN.TEXT is saved under a different name, and DUMMY3.TEXT is changed to KIN.TEXT.

```
STUFF:
WASTE.TEXT          4  15-Jan-81    10    512  Textfile
DICE.TEXT          18  15-Jan-81    14    512  Textfile
< UNUSED >         48               32
SPOSE.CODE         33  24-May-80    80    512  Codefile
UROSE3.CODE        35  26-Nov-80   113    512  Codefile
COLOR.DATA         32   5-Jan-81   148    314  Datafile
< UNUSED >         10              180
KIN.TEXT           16  15-Jan-81   190    512  Textfile
SONS.TEXT          18   5-Jan-81   206    512  Textfile
SAVEKIN.TEXT       16  15-Jan-81   224    512  Textfile
< UNUSED >        254              230
10/10 files<listed/in-dir>, 182 blocks used, 312 unused
```

NOTE - The number of valid bytes in the last block of a file is always recovered as 512. If the actual number of valid bytes differs from 512 (e.g. as in COLOR.DATA), the correct number of bytes may be recovered by writing a program which uses the DChangeEnd routine in the Dirinfo unit (see Library Users Manual for details).

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 primary directory, but has a duplicate directory, the Copydupdir utility (section 8.1.1) replaces its deceased primary directory with a copy of the duplicate directory; the volume is then restored.

WARNING - Primary 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 Recover utility (section 8.1.2) is valuable in recovering lost directories. Another method 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.

IV. THE ADVANCED SYSTEM EDITOR

The Advanced System Editor (ASE) is a screen-oriented text editor. Based on the UCSD Pascal L2 large-file editor, ASE provides powerful text-editing capabilities while maintaining the friendly user interface of its predecessors.

ASE capabilities include:

Large-file Editing - ASE is restricted only by disk space in the size of text files that may be edited.

User-defined Functions - ASE supports up to eight user-defined functions. A function key can be "taught" to execute a series of commands when it is typed. ASE also includes facilities for maintaining function key definitions.

Terminal Interface - Any terminal key or sequence of keys can be mapped to any editor command; also, a number of keys can be mapped to the same command. The utility program ASS (Advanced System Setup) facilitates editing of the key definitions for ASE commands to suit users' terminals and personal tastes.

Change Logging - Facilities are provided for implementing change control of text files. At the end of an edit session, the editor (optionally) requests a description of the changes made during the session, and enters the description with the current date into a change log maintained in the file. ASE ensures the security of change control by maintaining a revision number which indicates the number of times the file has been edited.

Paint Mode Editing - The Exchange command can draw vertical and horizontal character vectors in either direction, allowing painless creation of diagrammatic figures and simplified editing of columns of data.

Nested Editing - ASE can be invoked recursively, allowing the user to suspend the current edit session, edit another file, and then return to the suspended edit session. Disk space permitting, nested editing is permitted to a depth of 6 files.

System Interface - ASE's scope of operation extends beyond editing a single text file. System interfaces with the compiler and work file are preserved; however, the user has the option of controlling these interactions. When requested, ASE generates a menu-style list of all text files on a disk volume, allowing the user to select a file for editing without invoking the filer. New files may be created on any volume, and files may be edited from one volume to another. Also provided is the ability to chain together a series of edit sessions without leaving the editor. Together, these features enhance system performance and usability.

This chapter is organized into six sections: _Introduction_, _Basic Concepts_, _Using the Editor_, _Commands_, _Sample Edit Session_, and _Problems_. _Introduction_ presents an overview of ASE along with information needed to use the manual. _Basic Concepts_ describes the basic editing concepts, some of which are unique to ASE. _Using the Editor_ describes editor features. _Commands_ describes the editor commands. _Sample Edit Session_ provides basic instruction in editor operation. _Problems_ presents bug report forms.

NOTE - The following meta-words denote nonalphabetic edit commands peculiar to the ASE: <GetAgain>, <home>, <del>, <Coll>, <Dir-Change>, <record>, and <takeup>. References to these keys appear throughout this chapter, along with some less commonly used references that are (hopefully) self-explanatory. Definitions of keys for these commands are described in section 4.2.0.

4.0 Basic Concepts

The editor is used to create and modify text files.

Prompt lines in the editor are similar to filer and operating system prompts; they are described in section 4.0.0. Editor commands and the keys defined to invoke them are described in section 4.0.1. File prompts and the editor's file naming conventions are described in section 4.0.2.

Text files may contain either programs or documents; and, as these have different formatting conventions, the editor's mode of operation (known as the "environment") may be changed to suit either program development or word processing. Editor environments are described in section 4.0.3.

Two constraints imposed on the task of editing large text files are the size of the terminal screen for viewing text and the amount of memory available for containing the file. Section 4.0.4 describes the file window, which utilizes the screen as a "sliding window" through which sections of the file may be viewed. Section 4.0.5 describes the operation of the file buffer, which provides a virtual editing buffer in which the most recently viewed text is present in memory while the rest of the file is automatically stored on disk.

The cursor that appears on the terminal screen is the center of action for all editing commands; it is described in section 4.0.6.

Because the text file itself is modified by the actions of the file buffer, a separate copy of the file is created at the start of an edit session; this provides the ability to restore the original file contents when an edit session is exited or interrupted by dire circumstances. The copied file is called a backup file; backup files are described in section 4.0.7.

Although the novice user is encouraged to seek introductory tutorial material elsewhere, section 4.3 consists of a simple sample edit session as an example of use.

4.0.0 Prompt Lines

Editor prompts display either a prompt line of available edit commands or a command line (generated by invoking a prompt line command) displaying the available subcommands. Many editor prompts display the current direction (described in section 4.1.2) in the leftmost character of the prompt. Prompt lines normally appear across the top of the screen, but may disappear when an edit command causes the file window to scroll upwards; depending on the command, typing either <etx> or another command redisplays the prompt.

The main editor prompt appears initially as:

```
>Edit: A(djust C(opy D(elete F(ind I(nsert J(ump R(eplace
                                              Q(uit eX(change ?
```

The remaining commands and the titles of the function key definitions are displayed on the prompt by typing "?":

```
>Edit: B(eginLine L(ineEnd G(etch K(olumn P(age O(ppositePage
                                                     S(et V(erify ?

>Edit: T(oDisk N(ext M(argin Z(ap W(ordMove U(ptop E(dit
                                           <record> <takeup> ?

>Edit: <home> <arrows,tab,space,cr,bs> "<", ">", "=" ?

>Edit: <f1>=takeup1 <f2>=takeup2 <f3>=takeup3 <f4>=takeup4
                                                  <f5>=takeup5 ?

>Edit: <f6>=takeup6 <f7>=takeup7 <f8>=03-Oct-81 ?
```

4.0.1 Commands

ASE commands are classified as primary or secondary: primary commands are displayed on the main prompt'line; secondary commands are the commands displayed on a primary command's prompt'line.

Secondary commands are always invoked by typing their editor-defined keys. Typing the first letter of an alphabetic command (e.g. "U" in the U(pdate option of the Q(uit command) invokes the subcommand; both lower and upper case letters are recognized.

Unlike secondary commands, primary commands may be defined by the user to be invoked by any key or key sequence; additionally, a single command may possess a number of key definitions (see section 8.3.2 for details). Despite this flexibility, key definitions for primary commands tend to follow the conventions used by secondary commands (and in the rest of the system); for instance, lower and upper case letters are usually defined as keys for the alphabetic primary commands.

NOTE - A common practice which violates this convention involves the G(etch command; "g" is mapped to G(etch, but "G" is mapped to the related (nonalphabetic) command <GetAgain>.

Keys and key sequences are classified as printing or nonprinting: printing keys (e.g. " " .. "~" in the ASCII character set) print a character on the screen when typed; nonprinting keys (e.g. key sequences and function keys) are used to invoke actions rather than print characters. Depending on the context, printing keys are used either to invoke commands or insert characters; for instance, typing "j" invokes the J(ump command from the editor prompt, but inserts the letter "j" in the text while using the I(nsert command. Nonprinting keys are used to define commands that must be nonprinting (e.g. <etx> and <record>); they are also used for nonalphabetic commands such as <takeup>.

The eX(change command has a feature which requires nonprinting key definitions for the more commonly used editor commands. Any editor command may be invoked within eX(change, but non-printing key definitions are required to distinguish editor commands from the text normally dealt with. I(nsert and D(elete are the most useful commands inside eX(change, and therefore should have nonprinting keys along with their standard alphabetic definitions.

4.0.2 File Name Prompts

File name prompts appear in the E(dit and C(opy F(rom file commands. The editor accepts file names with or without file suffixes; the suffix ".TEXT" is automatically appended if it is not supplied. Along with the standard syntax for file names, the editor accepts special syntax for specifying menu selection of files (4.0.2.0), specification of source and destination files (4.0.2.1), and control over the automatic definition of user-defined functions (4.0.2.2).

4.0.2.0 File Menus

An occurrence of the character "?" in a file name causes the editor to produce a menu of all text files on the specified disk volume. The syntax for a menu-select file name is:

<menu specification> ::= [<partial file specification>]?

Typing a volume identifier followed by "?" (e.g. "MYDISK:?") causes all text files on the specified volume to be displayed for selection. If the file name contains part of a file title (e.g. "MYDISK:CHAP?"), the menu only displays text files whose names match the "wildcard" file title (e.g. "CHAP12.TEXT" and "CHAPTER.TEXT").

The menu displays this prompt line at the top of the screen:

```
Select 'a'..'u'(file), <sp>(specify), ?(more info)
```

A file is selected by typing its associated menu character (the range of character choices is displayed in the menu prompt). If more than 21 files exist, the prompt shown above is extended with the phrase "or <ret> (next page)"; selecting a file then consists of selecting a page of the menu and choosing a file from the currently displayed page. Selecting a file redisplays the original file prompt with the selected file name already typed in; only a <cr> (and possibly a marker specification) is required to complete the file prompt with the selected file name. Typing <space> also redisplays the original file prompt, but without adding a file name; a file must then be specified by typing its name. Typing "?" displays the first text line of each file on the menu. A standard practice of ASE users is to maintain a one line description at the front of each text file; the menu option "?" allows easy viewing of these descriptions.

The following is an example of a text file menu:

```
Select: 'a'..'g'(file), <sp>(specify), ?(more info)
ASE-DOC:
  a ASE3       10 .com command summary
**b ASE4      196 .com commands
  c APXA       22 .com sample session
  d APXB       22 .com configuration
  e APXC       12 .com problem reporting
  f ASE2       78 .com usage
  g ASE1       62 .com title, ToC, intro, baysicks

62 files.  3954 free blocks (194 contiguous; "**" too big).
```

As shown in the example, the specified volume name is followed by a list of the titles of all text files on the volume. The size (in blocks) of each text file is also displayed. A description of the displayed volume's free space situation appears at the bottom of the screen. Files too large to edit given the current free space on the volume are marked with "**"; the volume free space description is then extended as shown to describe the meaning of "**". "too big" implies that the volume does not have enough disk space to contain both the new file and its backup file.

4.0.2.1 Source and Destination Files

ASE allows you to specify both the name of the file to be edited and the name of the file to be created by the edit session; among other things, this allows files to be edited from one disk volume to another. The file to be edited is called the _source file_. The file to be created is called the _destination file_.

The syntax for specifying source and destination files is:

<files specification> ::= [<source file>] [,<dest file>]

The metasymbol <source file> indicate the volume and name of the existing file to be edited. The metasymbol <dest file> indicate the volume and name of the file to be created by the edit session. If a destination file is not specified, the existing file is named "<source file title>.BACK" (thus becoming the backup file); the destination file is named "<source file title>.TEXT" and is created on the same volume. If a source file is not specified, a new (empty) text file is created as specified by the destination file name. If neither source nor destination is specified, a new work file (named *SYSTEM.WRK.TEXT) is created.

NOTE - A destination file may be specified after a source file name has been selected from the file menu.

4.0.2.2 Automated Function Definition

The editor has the ability to automatically define function keys (section 4.1.9) within a text file each time the file is edited. The file marker "$PROFILE" (section 4.1.3) is defined as the default automatic function definition marker; its presence in a text file causes the editor to automatically "take up" a function definition starting from the marker's location in the file.

Automatic definition may be prevented by specifying a marker name after the file name - the syntax is:

<file specification> ::= <filename>[=[<markername>]]

If the "=" is not followed by a marker name, automatic definition is not performed; otherwise, the editor is directed to "take up" a function definition from the text file location determined by the specified marker. The following examples apply to a text file named "MYFILE.TEXT" which contains the markers "$PROFILE" and "MARK1":

-- "myfile=$profile" and "myfile" both specify automatic function definition from "$PROFILE".

-- "myfile=" ignores any existing "$PROFILE"; no automatic function definition is performed.

-- "myfile=mark1" specifies automatic function definition from "MARK1".

4.0.3 The Edit Environment

Edit commands affect text; the edit environment affects 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 entry of program text. Margins and filling are used for processing documents; in particular, filling allows the justification of paragraphs of text within the current margins.

The edit environment is described in more detail in section 4.2.19.3 (the Set Environment command).

4.0.4 The File Window

The editor allows the entire console screen to be used much like a chalkboard; any text displayed on the screen may 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. Most text files contain more lines than can be displayed on the console at once; 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 of the screen to make room for the display of previously hidden lines of text. The screen may 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 currently being changed can be viewed through the window.

4.0.5 The File Buffer

The file buffer serves as intermediary between the file window and the text file: the file window slides across the contents of the file buffer, and the file buffer slides across the contents of the text file. Text files larger than the file buffer overflow onto the disk; thus, during an edit session involving a large file, the contents of the file are split into three sections: text between the front of the file and the front of the file buffer (stored on disk), text in the file buffer (stored in memory), and text between the end of the file buffer and the end of the file (also stored on disk). The disk areas containing the ends of the text file are called <u>stacks</u>; the <u>left stack</u> holds the text preceding the file buffer, while the <u>right stack</u> holds the text following the file buffer.

The editor treates the contents of a text file as a sequence of "pages". The file buffer contains a number of pages, as do the stacks. The file buffer slides across the file in integral numbers of pages; thus, "moving" the file buffer in a given direction consists of writing pages from the trailing edge of the file buffer to the adjacent stack, shifting the current contents of the buffer into the resulting space, and reading pages into the leading edge of the buffer from its adjacent stack. This process is called "paging".

NOTE - Paging can be a time-consuming activity on systems with slow

disks. Experienced ASE users adapt their editing habits to the file buffer and buffer managing commands so as to minimize the amount of time spent in unnecessary paging (see below for details).

ASE has a number of commands which either affect or are affected by paging and the file buffer. The simplest method of moving the file buffer is by moving the cursor through the file. As the cursor moves through the contents of the file buffer, lines of text appear and disappear from the file window; when the end of the file buffer is reached, the message "Paging..." (or sometimes "Moving") is displayed, and the file window movement is momentarily delayed until the new pages are installed in the file buffer. The ability of the cursor to cause automatic paging is controlled by the value of the environment parameter "Auto Buffer". J(umping to a marker may cause paging; when it does, the message "Moving..." is displayed.

The F(ind and R(eplace commands always notify the user that the end of the file buffer has been encountered while searching for target strings:

End of Buffer encountered. Get more from disk? (Y/N)

Typing "Y" continues the search, with paging done automatically; typing "N" terminates the command without paging.

The D(elete and Z(ap commands are restricted in range to the file buffer; they do not have the ability to page the file buffer.

The N(ext command explicitly slides the file buffer across the file. When the environment parameter "Auto Buffer" is set False, N(ext provides the only way to move the file buffer. Because N(ext reads in as many pages as possible in the specified direction, it can occasionally reduce the amount of paging performed while making a single long pass through the file. The tradeoff involved is the time spent making one disk access in N(ext versus the time spent making a number of disk accesses while auto-paging.

The T(oDisk command complements N(ext - it throws all of the pages in the specified direction out of the file buffer. T(oDisk is used to empty the file buffer when an excess of text insertion commands fills it to capacity. When the buffer becomes full, the I(nsert command may automatically write part of the file buffer out to disk; when it does, editing is suspended while a series of dots appears on the current line. The screen is redisplayed and normal editing resumes when paging is completed.

4.0.6 The Cursor

If the screen can be considered a chalkboard (section 4.0.4), 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 - 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 I(nsert and D(elete commands.

4.0.7 Backup Files

Every edit session involves a source file and a destination file (4.0.2.1). At the start of a session, the contents of the source file (if any) are copied to the destination file. During the edit session, the editor manipulates the contents of the destination file; the source file is untouched, protecting its contents from loss of data due to system crashes or user errors. The untouched file is called a backup file.

If an edit session specifes the same file name for source and destination files, the source file is renamed "<file title>.BACK", thus becoming the backup file. For instance, editing the work file "SYSTEM.WRK.TEXT" creates the backup file "SYSTEM.WRK.BACK". A backup file is created only if the edited file is updated; exiting an edit session automatically restores the original file name of the source file.

4.1 Using the Editor

This chapter describes editing features and some basic editor commands. Editor invocation is described in section 4.1.0; editor termination is described in the Q(uit command (section 4.2.16). Commands which move the cursor are described in section 4.1.4; the remaining commands are presented in section 4.2. Basic editing features are presented in sections 4.1.1 - 4.1.3 and 4.1.5 - 4.1.6. Advanced features are described in sections 4.1.7 - 4.1.9.

4.1.0 Entering the Editor

The editor is invoked by typing "E" (for E(dit) from the system prompt line. A prompt with the following form appears on the top of the screen while the editor starts up:

    ASE 0.7n

When the editor finishes its initialization, the following prompt appears:

    Edit: [ASE 0.7n]
    ?<cr> Looks, <cr> Creates, <esc> Exits or Filename:

Typing "?<cr>" displays a menu of the files available for editing on the prefixed volume (see section 4.0.2.0 for details). Typing <esc> exits the editor and redisplays the system prompt. Typing <cr> creates a new work file and enters the editor; the new file's default name is SYSTEM.WRK.TEXT. An existing file is edited by typing its file name; section 4.0.2 describes the file name conventions for this prompt.

NOTE - If the work file exists when the editor is first invoked, the work file name is automatically entered into the file name prompt; it can be removed by typing <del>.

If the editor doesn't have enough disk space to edit the specified file, it prints an error message on the screen and terminates the edit session. If this happens, remove all the old backup files, K(runch the disk volume, and try again; this usually creates enough disk space to edit the file.

If enough disk space exists, the editor displays the following message while it is creating the backup file:

    Copying to <dest name>

NOTE - Copying takes a few moments when editing large files on systems with slow disks.

When copying is completed, the editor normally positions the file window and file buffer at the front of the text file, placing the cursor on the first line of text. In some situations, however, the

editor behaves differently.

Here are some other ways an edit session can begin:

-- If automatic function key definition is specified (4.0.2.2), the editor automatically jumps to the appropriate marker and takes up a function definition for function key <f1>, leaving the cursor at the end of the text form (see section 4.1.9 for details). The taken-up function definition may invoke itself (by the use of "|x", see 4.1.9.2), automatically stepping the editor through a series of edit commands. (All this is exciting to watch if you've never seen it work before -- or have forgotten about it!)

-- If the specified file contains the marker "$LAST", the editor automatically invokes the J(ump M(arker command to the marker "$LAST". Type <cr> to jump to the marker. Type <esc> to exit the command and start the edit session normally. $LAST allows you to return to the last place you were editing. See section 4.1.3 and the Q(uit command (section 4.2.16) for more details on "$LAST".

-- If you are resuming a suspended edit session after nested editing, the edit session may begin with an automatic J(ump M(arker command to the editor-defined marker $CURSOR; this allows you to return to the last place you were editing (even if $LAST is not set).

-- If the editor is automatically invoked by the compiler because of a syntax error, the edit session always begins with an automatic J(ump M(arker command to the magic marker "$SYNTAX". Typing <cr> jumps to the text causing the syntax error and displays a syntax error message at the top of the screen. Typing <esc> exits the command and starts the edit session normally. Throughout the current edit session, you can jump to $SYNTAX and have the syntax error message redisplayed at the top of the screen.

4.1.1 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 the subsequent command. A repeat factor specifies that a command is to be repeated the number of times determined by the preceding factor. For example, typing "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 a text file boundary is reached. Commands accepting repeat factors are noted as such in section 4.2.

4.1.2 Direction

The editor maintains an environment parameter named "direction". Direction affects commands that move the cursor; 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 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 can be executed unless their key definitions conflict with an enclosing command invocation (e.g. typing "<" in I(nsert inserts "<" in the text rather than changing the direction). The following keys are usually defined to change direction:

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

4.1.3 Markers

Markers are arbitrary cursor positions in a text file which are easily accessible from anywhere within the file. Markers do not appear in the text itself; the only way to locate a marker is to J(ump 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).

NOTE - A file can contain up to 26 user-defined markers.

The S(et M(arker command creates a marker at the current cursor position. Setting a marker to an existing marker name replaces the old marker setting. J(ump M(arker moves the cursor to the specified marker. Existing marker names can be displayed with the S(et E(nvironment and J(ump M(arker commands and can be deleted with the S(et D(eleteMarkers command.

The editor reserves the following marker names for its own use:

$CURSOR
$EQUAL

These markers are used to implement the "equals" command (described in section 4.1.4) and Q(uit B(ackup.

$LAST

If this marker is set by the user, the editor resets it to the last cursor position at the end of every edit session. Its presence in a file causes the editor to automatically set up (but not invoke) a J(ump M(arker to "$LAST" at the beginning of an edit session. See section 4.1.0 and the Q(uit command (section 4.2.16) for details.

$LOG

If this marker is set by the user, the editor automatically prompts for a log entry at the end of each edit session, and then writes the log entry into the file at the marker position. See section 4.1.8 for details.

$PROFILE

The presence of this marker in a file causes the editor to automatically take up a function key definition from the marker's position at the beginning of an edit session. See section 4.1.9 for details.

$SYNTAX

The editor creates this marker after being invoked by the compiler because of a syntax error. See section 4.1.0 for details.

$TAG

This marker is set with the S(et T(ag command and jumped to with the J(ump T(ag command. It is used as a fast-access, temporary file marker. See the S(et and J(ump commands for more details.

4.1.4 Moving The Cursor

This section describes the nonalphabetic cursor-moving commands. The remaining commands are described in section 4.2.

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
B(eginLine	Moves cursor to first character on line
L(ineEnd	Moves cursor to end of line
<home>	Moves cursor to upper left corner of the screen
=	Moves to "=" pointer

Direction sensitive commands-

<space>	Moves direction
<backspace>	Moves opposite direction
<tab>	Moves cursor to the next tab stop; tab stops are initially set every 8th column, but can be changed in the S(et E(nvironment command
<return>	Moves to the beginning of the next line
W(ordMove	Moves to start of next word

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

The cursor's column position is preserved by the <up> and <down> commands. When the cursor is moved outside the current text (in the blank space either before or after a line), its behavior depends on the current command.

At the outermost editor prompt, the cursor is treated as though it were immediately after the last character or before the first character in a line; if the cursor lies outside the current text when a command is invoked, it automatically jumps back to its actual position.

In some editor commands (such as eX(change), cursor movement is not limited to the current text.

The "equals" command is executed by typing "=". Equals saves the current cursor position and moves the cursor to the beginning of the last section of text which was I(nserted, F(ound, or R(eplaced; typing "=" again returns the cursor to its original position. For instance, text to be deleted by the Z(ap command can be verified by typing "=="; the cursor is moved to the equals marker (one Z(ap boundary), back to the original cursor position (the other Z(ap boundary), and then the text is deleted with Z(ap. Equals is not affected by direction.

NOTE - The reserved marker "$EQUAL" is reset after an I(nsert, D(elete, F(ind, C(opy, or R(eplace command. The reserved marker "$CURSOR" is used to save the current cursor position.

4.1.5 The Copy Buffer

The editor maintains a copy of the most recently I(nserted or D(eleted text in a magic place called the 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 sections of text within the file. The content of the copy buffer is always maintained, even across nested E(dit and Q(uit options, as long as you do not exit from ASE back to the Command prompt.

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

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

-- I(nsert fills the copy buffer with the inserted text. If you accidentally type <escape> during I(nsert, the text you typed in can be restored by C(opy B(uffer.

-- Z(ap saves the deleted text in the copy buffer.

NOTE - Storage for the copy buffer is taken directly from the text buffer, and therefore may be too small to contain a copy of the text. Whenever a Z(ap, I(nsert, 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. Usually, forcing extraneous text from the buffer to the disk by use of T(oDisk will provide sufficient room.

4.1.6 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 (4.1.6.0), editor variables containing function definitions and the current search and replace strings (4.1.6.1), and an environment parameter which affects the editor's method of searching for character strings (described in section 4.1.6.2). More details on this topic can be found in the descriptions of the F(ind, R(eplace, and S(et E(nvironment commands (sections 4.2.6, 4.2.18, and 4.2.19 respectively).

4.1.6.0 String Syntax

Strings can 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 except <space> or <esc>.

NOTE - Carriage returns are valid in strings; when they are typed, the screen is erased to make room for the entry of search strings consisting of a number of text lines.

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 backspacing over a delimiter.

4.1.6.1 String Variables

The editor provides two string variables for saving the last string arguments entered in the F(ind and R(eplace commands. It also allows function definitions to be used as string variables.

String variables provided by the editor are the search and replace strings. Their values are updated only by a delimited string which is a search or replace argument. The search string (named "<search>") is set by both F(ind and R(eplace; the replacement string (named "<replace>") is set only by R(eplace. The string values of these variables can be used in subsequent F(inds and R(eplaces by using the letter "S" or "s" to denote the contents of the search string and the letter "R" or "r" to denote the replacement string. For example, in F(ind, typing "R" finds an occurrence of the contents of the <replace> variable in the text file. In R(eplace, typing "sr" replaces an occurrence of <search> with the contents of <replace>, while typing ".match.s" replaces an occurrence of the string "match" with the contents of <search>.

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

Using an unassigned string variable results in the following error message:

ERROR: No old pattern. Type <spacebar> to continue.

The editor also accepts the digits "1" through "8" as string variable names. These variables contain the corresponding function key definitions (<f1>..<f8>); typing the variable name is an abbreviated form of typing the function definition as a string argument. (Note that the function keys themselves cannot be invoked while entering string arguments.)

4.1.6.2 Search Modes

F(ind and R(eplace have a number of methods for locating strings in a text file: Case insensitive mode, Token mode, and Literal mode. Case insensitive mode is used with the other two search modes; it indicates that lower and upper case alphabetic characters are to be considered equivalent while looking for occurrences of the search string. In Literal mode, the editor searches for any occurrences of the search 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.

Token and Literal modes are determined by 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.

Case insensitive mode is not determined by an environment parameter; searches always default to case-sensitive searches, and case-insensitive searches must be explicitly specified.

The current default search mode can be overridden in F(ind and R(eplace by using the letters "C"/"c" (force Case insensitive mode), "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 literal string "foon");

"T/foon//yeen/" (replace tokens of "foon" with the string "yeen");

",bad,L,good," (replace literal "bad" with the string "good");

"lc.a..Z." (replace all occurrences of the letters "a" and "A"

with the letter "Z").

4.1.7 Nested Editing

Nested editing allows the user to suspend the current edit session, edit another file, and later return to the suspended edit session at the point of suspension.

A nested edit session is invoked with the E(dit command which appears on the editor prompt line. When invoked, the fileprompt is issued, and, if successfully negotiated, E(dit writes the file from the current edit session out to disk in a temporary file (see the E(dit command (section 4.2.5) for details). The new edit session proceeds normally, but when it is finished, the Q(uit menu is augmented by a list of the suspended edit sessions. Q(uit A(nother creates a new edit session at the current nesting level. E(xit and A(nother restore the previously suspended edit session by reading its temporary file back into the editor; while this occurs, the following message is displayed:

```
Restoring <file name> (from <temp file name>)
```

Editing then proceeds at the point where it was suspended by the nested E(dit invocation. The actions performed by nested editing are nearly equivalent to having the user manually terminate the current session, enter the editor with a new file, terminate the new session, and reenter the editor with the original file. Nested editing reduces the number of keystrokes required to edit both files and alleviates the need to remember the name of the original file.

In nested edit sessions, the E(dit and Q(uit commands list the files involved in suspended edit sessions:

```
Files being edited (most recent first):
   2.  *CREATED1.TEXT
   1.  STUFF.TEXT            STUFF.BACK
```

The numbers on the left-hand side of the list indicate the nesting level of the associated file. If a suspended file has a backup file, the backup file name appears on the right.

Files created in nested edit sessions are named "CREATED<n>.TEXT" instead of the work file title; <n> ranges from 1 to the number of created files.

All files suspended by nested edit sessions are named "name.ASE!". These file names are usually temporary, i.e. they disappear as suspended edit sessions are restored. However, if the editor is aborted during a nested session (by running out of disk room, for instance), the suspended edit sessions are never restored; they appear as a series of disk files with the file suffix ".ASE!". If you want to save the suspended edit sessions, use the filer to change all file names with suffix ".ASE!" to have the suffix ".TEXT".

NOTE - Though many levels of nesting are allowed, it is rarely attempted because of disk space constraints; nested editing uses

large amounts of disk space. Consider the case of a nested editing session with the file FOON.TEXT:

> Initially, only the file FOON.TEXT exists. Entering "FOON <cr>" in an edit session changes its name to FOON.BACK and creates a working copy, FOON.TEXT, on disk. Entering a nested edit session saves the working copy as FOON.ASE!. Restoring the edit session makes a working FOON.TEXT from the FOON.ASE! on disk. Updating the edit session removes FOON.ASE! and saves the working copy as FOON.TEXT, leaving the original source file as FOON.BACK.

4.1.8 Change Logging

Change logging is used to maintain a history of the changes made to a file; it provides version control for oft-modified program sources, and can be an invaluable aid for programs maintained by more than one person.

A change log consists of a series of text lines (called "log entries") kept at a fixed location in a text file. The front of a file is usually chosen as a logging site because of its high visibility. A log entry is only constrained to lie entirely on a single line; however, a standard formatting convention is:

    <log entry> ::= <date> <initials> <comment>

As displayed, a log entry consists of the entry date followed by the logger's initials and a description of the changes made to the file.

Example of a change log:

```
01-Apr-81 RG Deleted all program sources...  April fools!
29-Mar-81 BD Overhauled doodad for optimal performance.
16-Feb-81 BD Completely reorganized entire thingamajig.
29-Jan-81 BD All night fixing bug #84.
23-Jan-81 BD Spent six long hours fixing bug #83.
```

Change logs maintained in program source files must be "commented out" in order to hide them from the language translator; this may restrict the use of some characters in a log entry.

Example of a commented-out change log in a Pascal source file:

```
{
29-Jan-81 BD All night fixing bug #84.
23-Jan-81 BD Spent six long hours fixing bug #83.
}
```

A change log is started by setting the reserved marker name "$LOG" at the desired logging site in a file. When a file is updated at the end of an edit session, the editor checks for the presence of "$LOG"; if found, the following prompt appears:

    Log Entry? (y/n)

Typing "N" terminates the edit session without logging an entry; "Y" generates the following prompt (the cursor position is underlined):

    Entry: 01-Apr-81 _

The user may now type in the desired log entry. Note that the current system date is already typed in by the editor; if desired, it can be erased with backspaces or <del>. The log entry is terminated by typing <cr>. Typing <esc> while entering the log entry aborts logging and restores the edit session.

When the log entry is completed, the editor automatically jumps to the marker "$LOG", inserts the log entry text, and finishes the edit session. This has the effect of entering the most recent log entry at the front of the change log; thus maintaining chronologically ordered log entries.

NOTE - The editor also maintains a revision number in each text file indicating the number of times the file has been updated. The revision number can be viewed with the S(et E(nvironment command.

4.1.9 User-defined Functions

A user-defined function is a character sequence (known as a *function definition*) that is assigned to a console key (known as a *function key*); typing the function key causes the editor to treat the characters in the function definition as keyboard input. For instance, assume that a function definition contains the character string "istuff<etx>"; typing this function key from the editor prompt causes the characters to be treated as if they were typed in from the console. The first character ("i") invokes the I(nsert command; the character sequence "stuff" is entered as text; the <etx> character completes the command. The net result is that the word "stuff" is inserted into the text each time the function key is typed. Function keys and definitions are described in section 4.1.9.0.

Functions can be defined by one of three methods: recording, taking up a text form, or specification. A function is recorded when sequences of commands and data are saved in the function's definition as they are executed by the editor; recording is described in section 4.1.9.1. Text forms are textual representations of function definitions stored in a text file; they are used to save functions across edit sessions (in a terminal-independent fashion). Text forms can be taken up from the text into a function definition, or created by copying an existing function definition. Text forms and text form conversion are described in section 4.1.9.2. The environment command "U(ser def key" reads a series of keystrokes typed from the keyboard into a function definition; this direct specification of functions is described in the S(et E(nvironment U(serkeys command (section 4.2.19.3.0.1).

4.1.9.0 Function Definitions

The editor provides eight user-defined functions; they are named function keys one through eight and are denoted by the symbols <f1>,<f2>,...,<f8> on the main editor prompt line. Here is a sample (obtained by repeatedly typing "?") of the prompt line displaying some of the function keys:

```
<f5>=takeup5 <f6>=recorded <f7>=taken up <f8>=02-Apr-81
```

The current status of each function is displayed on the prompt line:

```
<function status>  ::=  <default>  | taken up        |
                        recorded   | <user-defined>
```

The functions <f1> .. <f7> are initially defined to take up a text form as their definition: e.g. the default definition for function <f5> consists of "<takeup><f5>" and the prompt line entry displays its status as "takeup5". Function <f8> is initially defined to contain the current system date; typing <f8> in the I(nsert command inserts the date in the format displayed in its prompt line entry.

Functions taken up from a text form display the status "taken up".

Recorded functions display the status "recorded". Unfortunately, specified functions lack status, and therefore do not change the function's originally displayed status. Users can specify their own function status with the "keyname" feature included in the text form notation (see section 4.1.9.2 for details).

Active function definitions are stored in the editor's data space; they are preserved between editor invocations, but not across editor sessions. Text forms are used to permanently define functions (see section 4.1.9.2 for details). The literal character sequence stored in a function definition can be viewed and modified with the S(et E(nvironment U(serkeys command (section 4.2.19.3.0.1).

Repeat factors can be applied to function keys invoked from the editor prompt; for example, typing "3<f1>" causes <f1> to be executed three times. Repeat factors can be applied to functions invoked within editor commands only if nonprinting alternate definitions for the digits have been created (see section 8.3.2).

Function definitions are allowed to contain invocations of other functions; i.e., <f1> = "3<f2><f3>". Functions may also take up any function key; for example, <f1> can be defined as "<takeup><f1>" so that it takes up a new definition when typed.

NOTE - Section 8.3.2 describes the designation of terminal keys as function keys.

4.1.9.1 Function Recording

Recording a function key is accomplished with the <record> command (see the <record> command (section 4.2.17) for details). The <record> command must be followed by one of the function keys. After the function key is typed, the <record> command "disappears"; however, subsequent editor commands and data are recorded as the function key's definition. Recording is stopped by typing the <record> key again. For example, typing "<record><f1>istuff<etx><record>" assigns the function definition "istuff<etx>" to function key <f1>.

NOTE - Function keys may be defined at any point in the editor; definition is not restricted to start or end at the editor prompt.

The editor indicates that it is recording a function key by replacing the "?" normally displayed on the right side of the editor prompt with "<N>", where N indicates the number of the function key being recorded.

WARNING - As the <record> command may be invoked virtually anyplace in the editor, accidentally typing <record> will seem to lock the keyboard while it waits for a function key (or <escape>) to be typed. This may occur at places where the <record> prompt doesn't appear; for instance, I(nsert, eX(change, D(elete, Q(uit, y/n, ...

4.1.9.2 Function Text Forms

Function definitions may be permanently stored in a text file as _text forms_. Text forms are a textual representation of the commands and data of a function definition -- alphabetic characters are represented literally, while commands and nonprinting characters are represented in encoded form. For instance, an occurrence of the letter "a" in a function definition appears as the character "a" in the corresponding text form, while the <etx> command appears in the text form encoded as the character sequence "|e".

Encoding the commands and nonprinting characters has the following advantages:

-- Text forms are terminal-independent, thus ensuring that text files and their embedded function definitions are completely portable.

-- Encoded commands are represented by mnemonic character sequences, enabling text forms to be understood by users familiar with text form notation. Text form notation lends itself to direct manipulation of text forms in order to create and modify function definitions.

Two editor commands are used to shift function definitions between their active-but-volatile phase as invokable function keys and their passive-but-permanent phase as text forms. The C(opy <specialkey> command writes function key definitions to text forms. The <takeup> command reads text forms into function key definitions. Details concerning these commands may be found in the command descriptions in section 4.2.

The marker name "$PROFILE" is reserved for automated function definition at the beginning of an edit session; details are given below.

The following expressions describe the syntax for text forms.

```
<text-form> ::=  { <ch> | <EncodedKey> | <newline> } "|" .

<ch> ::=         <all-printing-characters-except-"|">

<newline> ::=    <text-form-continued-on-next-text-line>

<EncodedKey> ::= "|" <key>

<key> ::=        "|"  |  x  |   s <ch>   | f <digit> |
                  l   |  r  |  u  |  d   |  n  |  e  |
                  b   |  h  |  t  |  !   | "<" | ">" |
                  @   |  l  |  g  |  ^   |  *  |
                  " <string-of-ASCII-printing-except-'"'> "
                  ' <string-of-ASCII-printing-except-"'"> '
                 "{" <string-of-ASCII-printing-except-"}"> "}"

<digit> ::=      0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Definitions for the encoded <key>s are as follows.

Key	Meaning
"|"	The character "|" is placed in the text. (i.e. "||" ::= a single "|")
x	Execute function. The key is automatically invoked after being taken up.
s	Set space. The following character replaces " " as the character denoting <space> (e.g. "|s-" sets "-" to denote <space> in the definition, while subsequent " " characters become insignificant).
f	"f1" denotes <f1>; "f8" denotes <f8>.
l	<left-arrow>
r	<right-arrow>
u	<up-arrow>
d	<down-arrow>
n	<return>
e	<etx>
b	<backspace>
h	<home>
t	<tab>
!	<esc>
">"	insert character (used in eX(change)
"<"	delete character (used in eX(change)
@	<del>
l	<coll>
g	Getch
x	<GetAgain>
*	<takeup>
"{"	Comment (e.g. "|{ no effect on text form}").
'	User-defined keyname. The function status displayed on the prompt line is replaced with the delimited character string. Keynames longer than 11 characters are truncated.
"	Alternate user-defined keyname.

The following is an example of a text form using "|s" and comments.

|{ add 'hot stuff' }|s- i |{insert} hot-stuff |e |.

... translates to the function key definition "ihot stuff<etx>"; when invoked, the function inserts the string "hot stuff" into the text. Note that the space character is redefined to be "-". The remaining spaces in the text form are ignored when the text form is translated, but the occurrence of "-" is replaced with " ".

Example of text form using user-defined keyname.

The text form: |"CenterLine"ac|e|.

... translates to the function key definition "ac<etx>"; when invoked, it centers the text line under the cursor between the current margins (using the A(djust C(enter command). Also, it changes the function status displayed on the editor prompt. For example, taking up this text form into <f5> changes part of the editor prompt to:

<f5>=CenterLine

Example of text form using "|x":

The text form: |xifarkle|e|.

... translates to the function key definition "ifarkle<etx>" with the usual result; however, it is automatically invoked after it is taken up. Because the <takeup> command leaves the cursor at the end of a text form, this text form appears after being taken up as: |xifarkle|e|.farkle

Example of a well documented, nontrivial text form.

The text form:

```
(*
      - - - UCSD Pascal Include File Consumer - - -

      Input assertion:  All include file directives
      in the text have the following form (leading
      blanks in file name are ignored):

         {$I <filename>}

      Output assertion:  The text file contains
      text specified by an embedded include file
      directives.  Consumed include file
      directives are neutralized, and appear in the
      text as comments with the following form:

         {   <filename>}

  |"IncludeFile"|{ Key name on prompt line }
  |s-           |{ set space to "-" }
  f 1 .{$I.     |{ find include directive }
  d 2 |b |e     |{ delete directive specifier "$I" }
  d |g} |!      |{ file name into copy buffer }
  |* c |f2      |{ take up copy buffer into f2 }
  2b            |{ move cursor to next line }
  c f |f2 |n    |{ copy file contents into text }
  |.            |{ end of definition }

*)
```

... is used to merge a Pascal source file and all of its specified "include" files into one (large) text file. (The symbols "(*" and "*)" shown in the example are not part of the text form; they exist merely to "comment out" the text form in the Pascal source program.) As shown, the function only processes one directive at a time; however, typing "/<fn>" while editing the source file copies all of the "include" files.

With the information provided so far, it is almost possible for users to embed text forms in a given text file so that each edit session involving the file automatically defines a group of predefined commands. What is lacking is a means of automating the process of taking up text forms into function keys; the editor reserves the marker name "$PROFILE" for this purpose. When a file is first entered, and the marker "$PROFILE" is present, the command sequence "jm$PROFILE<cr><takeup><f1>" is automatically executed. The marker is assumed to be set at the front of a text form; this function definition is automatically taken up. Other function keys can then be taken up by initially defining <f1>'s text form to have the following format:

```
|x|*|f5|*|f4|*|f3|*|f2|*|f1|.
|{function5}|.
|{function4}|.
|{function3}|.
|{function2}|.
|{function1}|.
```

In this example, the marker "$PROFILE" is assumed to be set at the start of the first text form. This text form is automatically executed after being taken up; its sole purpose is to define all desired function keys from the subsequent text forms in the file (in this case, <f1>'s last deed is to redefine itself to something more useful).

NOTE - Section 4.1.0 describes the interactions of $PROFILE with the other automated editing options. Section 4.0.2.2 describes how automated takeup can be overridden or redirected to another marker in the file by using the editor's extended file name syntax.

4.2 Commands

Section 4.2.0 contains a command overview; the commands are grouped according to their function. Sections 4.2.1 through 4.2.26 describe each command in detail; the commands are alphabetically ordered.

4.2.0 Command Summary

4.2.0.0 Moving Commands

<down>	cursor down
<up>	cursor up
<right>	cursor right
<left>	cursor left
<space>	cursor in direction
<backspace>	cursor in reverse direction
<tab>	cursor to next tab stop in direction
<return>	cursor to next line in direction
"<"	backward direction
">"	forward direction
=	cursor alternates between current position and start of last I(nserted/F(ound/R(eplaced/=d text

<home>: Move cursor to the upper left-hand corner of the screen.

U(ptop: Move the screen window so that the line containing the cursor is at
the top.

W(ordMove: Move cursor to the beginning of the next word.

B(eginLine: Move cursor to the beginning of the current line.

L(ineEnd: Move cursor to the end of the current line.

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

N(ext: Move cursor to the beginning or end of the file.

O(ppositePage: Move cursor one screen page in the opposite direction.

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

4.2.0.1 Text Changing Commands

I(nsert: Insert text.

D(elete: Delete text.

eX(change: Exchange text.

C(opy: Copies text from copy buffer, another file, or a function text form into the file.

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

4.2.0.2 Pattern Matching Commands

F(ind: Find character string patterns in text.

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

G(etch: Find the next occurrence on this screen of the specified character.

<GetAgain>: Finds the next occurrence of the last G(etched character.

4.2.0.3 Formatting Commands

A(djust: Adjust indentation of the current line.

K(olumn: Adjust indentation of a column of text.

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

4.2.0.4 Buffer Managing Commands

N(ext: Extend the file buffer in the specified direction.

T(oDisk: Write part of the file buffer out to disk.

4.2.0.5 Function Defining Commands

<record>: Record subsequent editor commands in a function definition.

<takeup>: Read in a function definition from the text.

4.2.0.6 Miscellaneous Commands

S(et: Set M(arkers to J(ump to, D(elete markers, or enter E(nvironment to change parameters.

V(erify: Redisplay screen with the cursor centered.

E(dit: Save the state of the current edit session and start a new edit session.

Q(uit: Leave the current edit session.

4.2.1 A(djust

Prompt:

>Adjust: [<n>] <arrows> L(just R(just C(enter Z(ero
{<etx>,<esc> line}

The <n> delimited by square brackets displays the "adjust factor". The adjust factor indicates the number of columns by which the current line has been adjusted.

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. The <Tab> key moves the line right to the next tabstop. <Space> and <BackSpace> also move the line forwards and backwards respectively. The adjust factor is incremented or decremented as appropriate.

"Z" sets the adjust factor to zero, thus preserving the current indentation of all adjusted lines. "L" and "R" left-justify and right-justify lines to the current margin settings. "C" centers lines between the margins. Margins are described in the S(et E(nvironment command. The adjust factor is incremented or decre ented automatically.

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 <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. Reversing the vertical direction automatically resets the adjust factor to that of any modifications done on the last line.

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

Typing <etx> finishes the command; the cursor is left at the beginning of the last line adjusted. Typing <esc> exits the command, restoring the original indentation of the current line. If a number of lines have been adjusted, only the last line is restored.

4.2.2 B(eginLine

Repeat factors are allowed.

Moves the cursor to the beginning of the current line. When preceded by a repeat factor <n>, B(eginLine moves the cursor to the beginning of the <n - 1>th line from the current line.

4.2.3 C(opy

Prompt:

>Copy: B(uffer F(rom file <specialkey> <esc>

4.2.3.0 C(opy B(uffer

Repeat factors are allowed (before the C(opy).

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 "=" pointer is set to the tail of the copied text.

The copy buffer is described in section 4.1.5.

4.2.3.1 C(opy <specialkey>

Typing a function key copies the text form of the specified function definition into the text file at the current cursor position, and leaves the cursor at the end of the text form. The "=" pointer is set to the start of the text form. This allows recorded function definitions to be saved in the text file for future editing sessions. The <takeup> command reads up function definitions from text forms.

See section 4.1.9 for more details on special function keys, function definitions, and text forms.

4.2.3.2 C(opy F(rom file

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 end of the copied text. The "=" pointer is set to the front of the text.

Any text file may be specified; the file suffix ".TEXT" is optional. Typing <esc> or only a <return> exits the command.

File names containing the editor's wild card character "?" generate a file menu (described in section 4.0.2.0) displaying all text files on the specified volume. Typing a letter corresponding to one of the displayed file names specifies the file for copying; typing <space> redisplays the original prompt; typing "?" displays the first line of text in each of the files displayed.

The marker specification (including the square brackets) is optional, and is used to copy selected portions of another file. 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. For example, "ourfile[yourmark,mymark]" indicates that all text between the markers "YOURMARK" and "MYMARK" in the file "OURFILE.TEXT" should be copied.

Partial marker specifications are allowed; an empty marker field indicates one end of the file as the delimiter of the copied text. For example, "[,MYMARK]" indicates that all text between the front of the file and the marker "MYMARK" should be copied, while "[YOURMARK,]" indicates that all text between the marker "YOURMARK" and the end of the file should be copied. Markers are described in the S(et M(arker command.

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

4.2.4 D(elete

Prompt:

>Delete: < > <Moving commands> {<etx> to delete, <esc> to abort}

The cursor is considered to 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.

The cursor movement commands B(eginLine, L(ineEnd, G(etch, <Get-Again>, and <arrow>s all are provided as subcommands within the D(elete command.

To accept the deletion, type <etx>; to escape, type <esc>. The "=" pointer is set to the anchor position irrespective of whether <etx> or <esc> is used.

The copy buffer is filled with the text between the cursor and the anchor when <esc> or <etx> is issued. If there is insufficient room in the text buffer for a copy of the text, a yes/no prompt is issued to verify before deleting.

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 before the letter "t" in the second occurrence of the word "to". Enter D(elete by typing "DWW<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.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

4.2.5 E(dit

Prompt:

```
>nested Edit: [ASE n.ma]
?<cr> Looks, <cr> Creates, <esc> Exits or Filename:
```

The state of the current text file is saved and the editor is re-invoked. Typing <esc> to the ensuing file prompt returns the user to the current text file; the remaining options are described in section 4.1.0.

Nested editing is described in section 4.1.7.

4.2.6 F(ind

Repeat factors are allowed.

Prompt:

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

See section 4.1.6.0 for details on specifying search strings.

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

After a successful F(ind, the "=" marker ($EQUAL) is set at the first character of the last found string pattern, and the cursor is left at the character following the string.

If the editor reaches the end of the text buffer while looking for a target string, it displays the following prompt:

```
End of Buffer encountered.  Get more from disk? (Y/N)
```

Typing "Y" causes the editor to continue searching for the target; any subsequent paging is done automatically. Typing "N" or <escape> exits the F(ind command, leaving the cursor at its original position (or after the last found target occurrence on multiple searches).

NOTE - On searches for multiple occurrences of a target string, if the editor doesn't find an occurrence of the target and the last found string is no longer in the text buffer (because of paging), the cursor is not left at the last found string; it's left near the end of the file.

If the specified number of target occurrences is not found, the following prompt appears:

```
ERROR: Pattern not in the file.  Press <spacebar> to continue.
```

See section 4.1.6 for more details on using F(ind.

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 C(ase <target>  =>/rutabaga/
```

The cursor is moved to this position:

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

4.2.7 G(etch

Repeat Factors are allowed.

G(etch finds an occurrence of the subsequently typed character on the screen, starting at the current cursor position and moving in the direction displayed. If the character occurrence is found, the cursor is positioned under it; otherwise, the bell is beeped. /G(etch will find the last occurrence of the character on the screen.

NOTE - G(etch and <GetAgain> will not move the cursor off the current window while searching for a target character.

NOTE - G(etch and <GetAgain> are allowed within other commands (e.g. D(elete).

Function key macro writers will find the U(ptop command to be useful for getting a full screen "in front" of the G(etch.

4.2.7.0 <GetAgain>

The <GetAgain> command (not displayed on the promptline) finds another occurrence of the character specified by the previous G(etch invocation.

NOTE - G(etch and <GetAgain> will not move the cursor off the current window while searching for a target character.

NOTE - G(etch and <GetAgain> are allowed within other commands (e.g. D(elete).

Function key macro writers will find the U(ptop command to be useful for getting a full screen "in front" of the <GetAgain>.

4.2.8 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, with <tab>s being expanded to the necessary number of <space>s. Any nonprinting characters that are typed are echoed with a definable character, usually "?". Text may be changed while it it is being inserted - typing <backspace> removes the last inserted character, typing <del> removes the current line of inserted text, and typing <Coll> puts the cursor in column 1. Text preceding the inserted text cannot be removed. This means that <del> and <Coll> may not be able to go back as far as the previous line or column 1, so they just back up to the beginning of the insertion.

To accept the insertion, type <etx>; to escape, type <esc>. The "=" pointer is set to the beginning of the insertion.

If an insertion is aborted with <esc>, the inserted text is saved in the copy buffer, in case one accidentally escapes from a long insertion. This implies that the "no room to put in copy buffer" message may appear, not only when typing <etx> to accept the insertion, but when escaping as well.

Occasionally, there will be insufficient room in the buffer for an insertion. This may be known before any text is actually typed, or after many characters. In these circumstances, the editor will force some text to disk causing a pause and some momentary "..."s to appear on the screen at the cursor position.

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 (section 4.2.19.3) 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.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

4.2.8.0 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 (unless Filling is True). 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
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

4.2.8.1 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 and it is considered to terminate the paragraph. Command characters are described in section 4.2.12.0.

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

Filling with AutoIndent is often a useful mode.

WARNING - Filling without AutoIndent (fondly referred to as "munch" mode by previous victims) should be used cautiously when editing a text file containing both filled and unfilled text. Failure to either disable Filling or enable AutoIndent before I(nserting into text with a different format (e.g. program source or stylish tables) will radically alter the contents of the text file in an unpleasant fashion.

NOTE - Operating in munch mode is not really necessary, as the M(argin command is valid in all of the four possible modes.

4.2.9 J(ump

Prompt:

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

Typing "B" or "E" moves the cursor to the beginning or end of the current file buffer. Typing "T" jumps to the tag marker "$TAG"; the tag marker is usually created with the S(et T(ag command.

4.2.9.0 J(ump M(arker

Prompt:

Jump to what marker?

Typing <esc> or only a <return> escapes the command.

Typing a marker name followed by a <return> moves the cursor to that marker's location in the file. Typing <esc> anywhere in the marker name escapes the command.

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

ERROR: Not there. Press <space-bar> to continue.

Typing "?<return>" displays a menu of all current marker names in the file. Markers outside of the file buffer are preceded by "<" or ">". "<" indicates markers in text on the left stack; ">" indicates markers in text on the right stack. A marker from the menu is selected by typing its associated letter.

Markers associate user-defined names with arbitrary cursor positions in the text file. See section 4.1.3 for more details.

Markers are removed with the S(et D(eleteMarkers command and examined with the S(et E(nvironment command.

4.2.10 K(olumn

Prompt:

Kolumn: <vector keys> <etx>

Adjusts the indentation of text to the right of the cursor. Operates over multiple lines as does A(djust.

All characters between the cursor and the end of the current text line may be moved with the <left> or <right> commands. Typing <left> moves text to the left; characters are deleted as they move past the cursor. Typing <right> moves text to the right; blanks are inserted past the cursor to fill the space created.

The <up> and <down> commands apply the same adjustment to adjacent lines of text. Vertical and horizontal cursor movements may be intermixed; the editor maintains a cumulative adjust factor as a series of text lines is moved.

Typing <etx> finishes the command. K(olumn does not recognize <esc>.

WARNING - Careless use of the K(olumn command can irretrievably delete valuable portions of one or more text lines.

Example of using K(olumn:

Here is the cursor and text before K(olumn:

```
- - - - - - - - - - - - - - - - - - - -
    sturdy column     _    this
    sturdy column          is
    sturdy column          a
    sturdy column          rather
    sturdy column           shaky
    sturdy column           column
- - - - - - - - - - - - - - - - - - - -
```

Type "K5<left>4<down><left><down><etx>". The text and cursor position now appear as follows:

```
- - - - - - - - - - - - - - - - - - - -
    sturdy column     this
    sturdy column     is
    sturdy column     a
    sturdy column     rather
    sturdy column     shaky
    sturdy column     column
- - - - - - - - - - - - - - - - - - - -
```

4.2.11 L(ineEnd

Repeat factors are allowed.

Moves the cursor to the end of the current line. When preceded by a non-zero repeat factor <n>, L(ineEnd moves the cursor to the end of the <n - 1>th line from the current line.

4.2.12 M(argin

M(argin (also known as M(unch) reorganizes the paragraph of text surrounding the cursor so that its text lines lie within the current margins. A paragraph is defined as a series of text lines delimited by either blank lines or a line beginning with the command character (see S(et E(nvironment (section 4.2.19.3)).

M(argin proceeds automatically when the editor environment is set for word processing; in particular, when Filling is set True and Auto-indent is set False ("munch" mode). Otherwise, the following prompt first appears:

Margin: Are you sure? (y/n)

Typing "Y" starts M(argin; typing "N" exits to the editor prompt.

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.

WARNING - Inadvertently typing "M" when the edit environment is set for word processing can ruin the contents of a text file containing program sources. In situations requiring isolated sections of formatted text in a program's source file, it is wise to leave AutoIndent True in order to disable automatic M(unching.

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:
    A(uto indent  False
    F(illing      True
    L(eft margin  5
    R(ight margin 60
    P(ara margin  10
    C(ommand ch   .
```

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.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

4.2.12.0 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 command character is defined with the S(et E(nvironment command.

Command characters also affect the behavior of I(nsert when in "munch" mode.

4.2.13 N(ext

Prompt:

Next: F(orwards, B(ackwards in the file;
S(tart, E(nd of the file. <esc>

S(tart moves the cursor to the front of the file. E(nd moves the cursor to the end of the file.

F(orwards and B(ackwards slide the file buffer across the text file; F(orwards reads in text in front of the buffer, while B(ackwards reads in text behind the buffer. The cursor position is unchanged.

When the Auto Buffer environment parameter is False, these commands are necessary for moving the file buffer to another part of the file. When Auto Buffer is True, use of N(ext F(orwards or B(ackwards is not required; however, judicious use can enhance editor performance on some configurations by minimizing the amount of paging performed as new sections of the text file are edited.

See section 4.0.5 for details on the file buffer and paging.

4.2.14 O(ppositePage

Repeat factors are allowed.

Displays the preceding screen of text if direction is forward; otherwise, the following screen of text is displayed. The cursor is left on the same line of the screen, but is moved to the start of the line.

4.2.15 P(age

Repeat factors are allowed.

Displays the following screen of text if direction is forward otherwise, the preceding screen of text is displayed. The cursor is left on the same line of the screen, but is moved to the start of the line.

4.2.16 Q(uit

Prompt:

```
Quit:
    A(nother file (after Updating)
    B(ackup and re-edit the same file
    C(hange the name of the output file
    E(xit (but workfile not updated)
    R(eturn to the editor without doing anything
    U(pdate the workfile and leave

Output File Name: <filename>
Backup File Name: <filename>
```

One of the five options is selected by typing "A", "B", "C", "E", "R", or "U"; all other characters are ignored.

E(xit -

The editor verifies the E(xit command with the following prompt:

   Are you sure you want to exit? (Y(es, N(o or A(nother)

Typing "N" returns control to the editor without exiting; the cursor is returned to the same position it occupied before "Q" was typed.

Typing "Y" or "A" terminates the current edit session -- modifications made to the text are lost, as the text file is not saved on the disk. A(nother reinvokes the editor after exiting the current edit session. The contents of the copy buffer, function keys, and search/replace strings are retained.

C(hange -

The displayed output file name is erased. The user can then change the name of the output file by typing a new file name. The editor appends the suffix ".TEXT" to the new file name only if was not specified. Typing only <return> or typing <esc> within the file name exits C(hange and restores the original file name. The Q(uit prompt reappears after the C(hange command.

NOTE - C(hange only changes the output file's name; it cannot change the volume to which the output file is written. Volume identifiers in the new file name are therefore ignored. The name change is not saved if R(eturn is used to restore the current edit session.

R(eturn -

Returns to the editor without updating. The cursor is returned to the same position it occupied before "Q" was typed. R(eturn is often used after accidentally typing "Q".

A(nother -
U(pdate -

A(nother reinvokes the editor after completing the current edit session; the contents of the copy buffer, function keys, and search/replace strings are retained for the new edit session. U(pdate redisplays the system prompt after completing the current edit session. When an edit session is completed, the text (and environment information) is written to the destination file. If the marker "$LAST" exists, it is set to the last cursor position. The editor displays the following information on the screen while completing an edit session:

Quit:
Writing all.......**
The workfile, <filename>, is <n> blocks long.
The backup file is <filename>

B(ackup -

The current edit session is completed as in U(pdate or A(nother; the editor then starts a new edit session, automatically specifying the just-updated file as the source file. B(ackup is popular with people saddled with undependable hardware and/or electrical service! The new edit session begins at the front of the file, with a J(ump M(arker prompt to the previous cursor position ($CURSOR). Note that B(ackup is equivalent to typing "qa<name-of-dest-file>=<return>jm$cursor", except that a $LOG request is not issued.

When nested edit sessions exist, the editor lists the names of each file being edited. A(nother reinvokes the editor for a new edit session at the same level; E(xit and U(pdate restore the previous level as the current edit session. See section 4.1.7 for details on nested editing.

If a marker $LOG exists for a Q(uit A(nother or U(pdate command, the user is requested to make an optional entry in a semi-automated change log. The following prompt appears:

Log Entry? (y/n)

Typing <escape> restores the current edit session. See section 4.1.8 for more information on change logging.

NOTE - If the location of the marker $LOG is not within the current text buffer, the logging prompt appears as:

The log is not in the edit buffer. Log Entry? (y/n)

This is nothing important -- the only difference being some automatic paging if a change is to be logged.

4.2.17 <record>

Prompt:

RecordSpecialKey: <specialkey 1..8> <commands> OR <esc>

Record subsequent editor input as a definition for a user-defined function.

The <record> key command must be followed by typing one of the function keys or <escape>. After the function key is typed, the command "disappears"; however, subsequent editor commands and data are recorded as the function key's definition. Recording is terminated either by typing the <record> key again or by typing the function key being recorded. The <record> command may be invoked at any command level in the editor, but its prompt line appears only when it is invoked from the editor prompt.

If a function key is being recorded when the prompt line changes, the number of that key is shown in the upper right corner of the screen as "<n>".

There is a limited buffer space reserved for remembering the recorded or TakenUp function key definitions. If this buffer overflows during function key recording, the bell sounds and recording stops. There may be no other notice that recording has stopped until the prompt line changes and the absence of <n> in the upper right corner of the screen is noted.

NOTE - As the <record> command may be invoked virtually anywhere in the editor, accidentally typing <record> will seem to lock the keyboard while it waits for a function key or <escape>. This may occur at places where <record> prompt doesn't appear, such as I(nsert, eX(change, D(elete, Q(uit, y/n, ...

See section 4.1.9 for details on user-defined functions.

4.2.18 R(eplace

Repeat factors are allowed.

Prompt:

```
>Replace[<n>]: L(it C(ase V(fy <targ> <sub>  =>
or ...
>Replace[<n>]: T(ok C(ase 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. If no repeat factor is specified, then 1 is assumed.

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 between the current cursor position and the file boundary are replaced.

For detail on the specification of search and replace strings, see section 4.1.6.

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

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 4.1.6.2)) 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[<n>]: <esc> aborts, 'R' replaces, ' ' doesn't
```

Typing an "R" replaces the string. Typing a <space> spares the current target occurrence from replacement. The "R" or space themselves may have repeat factors. <escape> terminates the R(eplace command. The metasymbol <n> indicates the current value of the repeat factor; it is counted down from its initial value as strings are replaced. In V(erify mode, the repeat factor applies to the number of times a target occurrence is replaced, not the number of times it is found.

After a successful R(eplace, the "=" marker ($EQUAL) is set at the first character of the last replaced string pattern, and the cursor is left at the character following the string.

If the editor reaches the end of the text buffer while looking for a target string, it displays the following prompt:

```
End of Buffer encountered. Get more from disk? (Y/N)
```

Typing "Y" causes the editor to continue searching for the target; any subsequent paging is done automatically. Typing "N" or <escape> exits the R(eplace command, leaving the cursor at its original position (or after the last replaced string occurrence on multiple searches).

NOTE - On replacements of multiple occurrences of a target string, if the editor doesn't find an occurrence of the target and the last replaced string is no longer in the text buffer (because of paging), the cursor is not left at the last replaced string; it's left near the end of the file.

If the specified number of target occurrences is not found, the following prompt appears:

ERROR: Pattern not in the file. Press <spacebar> to continue.

See section 4.1.6 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 "yams". The cursor is located at the start of the line.

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

The R(eplace command is invoked with a search string of "yams" and a replacement string of "strawberries":

```
>Replace[1]: L(it C(ase V(fy <targ> <sub>  =>.yams.,strawberries,
```

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

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

4.2.19 S(et

Prompt:

>Set: E(nvironment M(arker T(ag D(eleteMarkers <esc>

Markers enable arbitrary cursor positions in a text file to be easily accessible from anywhere within the file; they are jumped to by using the J(ump M(arker command. The T(ag is a marker with a predefined marker name, $TAG; it allows quick definition of and access to a single position in the text. Marker setting is described in section 4.2.19.0, Tag setting in 4.2.19.1, and marker deletion in 4.2.19.2. Markers are described in detail in section 4.1.3.

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. An overview of the environment is presented in section 4.0.3. The environment is described in detail in section 4.2.19.3.

4.2.19.0 S(et M(arker

Prompt:

Set what marker?

Marker names may contain up to eight characters, including embedded blanks; they are terminated by typing <return>. Marker names are case-insensitive; thus, the two marker names "$LAST" and "$last" denote the same marker. Setting a marker to an existing marker name removes the old marker setting.

A maximum of 26 user-defined markers is permitted in a file. Attempting to create a new marker when the maximum number already exists results in the following prompt:

ERROR: Marker Overflow. Press <spacebar> to continue.

The S(et D(eleteMarkers command removes unwanted markers. See section 4.1.3 for more information on markers.

4.2.19.1 S(et T(ag

Defines the tag marker "$TAG" to reside at the current cursor position. This is a convenient abbreviation for the command "sm$tag<return>". The tag may be jumped to with the J(ump T(ag command.

4.2.19.2 S(et D(eleteMarkers

Prompt:

```
>DeleteMarkers: 'A' ..  'L'  [<etx> accepts, <esc> escapes]

A)  $TAG         B)  ?           C)  $EQUAL      D)  $CURSOR
E)  1            F)  2           G)  3           H)  4
I)  7            J)  8           K)  5           L)  6
```

The promptline is followed by a list of all existing marker names. Each marker name is assigned a letter. Typing a letter within the displayed range removes the corresponding marker from the file; typing <etx> completes the command; typing <esc> exits the command, preserving all markers.

4.2.19.3 S(et E(nvironment

Prompt:

```
>Environment: {options} <spacebar> to leave  [ASE m.na]
    A(uto indent  True
    F(illing      False
    L(eft margin  1
    R(ight margin 80        Workfile:     *SYSTEM.WRK.TEXT
    P(ara margin  6         Backup file: *SYSTEM.WRK.BACK
    C(ommand ch   .
    S(et tabstops
    T(oken deflt  True
    U(ser def key
    B(uffer auto  True

4 bytes used, 12284 available.
There are 0 pages in the left stack, and 0 pages in the right stack
You have 87 pages of room, and at most 1 pages worth in the buffer.

  <search>= 'New Orleans', <replace>=  'Going North'

Markers:
   MARKER-A  MARKER-B  MARKER-C  MARKER-D  $GLITCH

Created March 29, 1957; Last updated March 29, 1981 (Revision 24).
```

All environment values except the search and replace strings are saved in the text file for future edit sessions.

"S(etTabStops" and "U(serkey" are not environment parameters, but subcommands of S(et E(nvironment; they are described in section 4.2.19.3.0. Environment parameters are described in section 4.2. 19.3.1. Following the list of environment commands and parameters is a description of the file buffer state. Stacks, pages, and the file buffer are described in detail in section 4.0.5. If search and replace string patterns have been generated by F(ind or R(eplace, they are displayed next, followed by all current marker names. The bottom line displays the creation date, last modified date, and revision number of the current text file; the revision

number is incremented after each editing session.

NOTE - On systems where the screen is not of sufficient size to display the entire E(nvironment screen, the markers, resource information, and revision number are displayed in a subsidiary screen image accessible with the I(nfo subcommand of S(et E(nvironment. This command is not displayed on the promptline unless it is needed.

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.

4.2.19.3.0 Environment Commands

The environment command "S(etTabStops" affects the behavior of the <tab> command by changing the number and position of the tab stops on the screen; it is described in section 4.2.19.3.0.0. The <tab> command is described in section 4.1.4.

The environment command "U(serkey" displays and changes function key definitions; it is described in section 4.2.19.3.0.1. User-defined functions are described in section 4.1.9.

4.2.19.3.0.0 S(et E(nvironment S(etTabStops

Prompt:

```
Set tabs:<right,left,space,tab> C(ol#
                     {N(o T(ab Z(eroAll D(efault} <etx,esc>

--------T-------T-------T-------T-------T-------T-------T

    Column # 1
```

The text from the current screen is displayed below the prompt to aid the user in setting tabs to match any existing text formatting. The current tab settings are displayed on the line of dashes. The cursor moving commands accept repeat factors, but are not affected by direction. Typing <tab> moves the cursor to the next displayed tab stop. The column number of the current cursor position is maintained below the tab line. The C(ol# command moves the cursor to a specified column number. T(ab sets a tab stop at the current column, while N(o (or typing "-") clears the current column. Z(eroAll deletes all defined tab stops. D(efault restores the default tabstop format (which is one tabstop every 8th column).

Typing <etx> completes the command; <esc> exits S(etTabStops without affecting the original tabstop settings.

NOTE - Tab stop settings are stored in the text file, and thus are saved across edit sessions.

When setting repetitive tabstops, it is often convenient to use a function key. With the cursor at the beginning of the repetive sequence {say every fifth position}, do the following:

"<record><fl>t   <record>10<fl>"

This will set eleven tabstops every fifth column. The text form for this is |{tabset 5}|s-t-----|.

4.2.19.3.0.1 S(et E(nvironment U(serkey

Prompt:

Define Special Keys: <specialkey 1..8> <etx>

A list of the current definitions for the eight user-defined functions appears below the prompt. The function definitions in this command always appear in absolute (as opposed to encoded) form and can be used to view the terminal-dependent aspects of the keyboard. Commands and data are displayed as literal character sequences, with nonprinting characters represented by the decimal representation of their integer value and delimited by parentheses. For instance, a command may be mapped to a key generating an escape character sequence; in this case, an occurrence of the command in a function definition causes the key's entire character sequence to be displayed.

Example of a function definition:

The string entered as:

".foon<cr>noof<cr>foon."

... is displayed as:

"foon(13)noof(13)foon"

Note that the carriage returns are displayed in absolute form; the integer value of the nonprinting <return> character is usally 13.

Typing a function key clears the corresponding definition in preparation for a new definition, and the following prompt appears:

Enter delimited string (e.g. /stuff/)

A function definition is entered by typing an arbitrary delimiting character, the keys comprising the definition, and a second occurrence of the delimiting character (the example in the previous prompt uses "/" as a delimiter). The use of delimiters allows all editor commands to appear in the definition (see search and replace string specification in section 4.1.6).

NOTE - The function definitions described here are stored in the editor during an edit session, but are not saved across edit sessions. Functions may be saved as text forms with the C(opy command. See section 4.1.9.2 for details.

4.2.19.3.1 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" (True) or "f" (False), 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 "True".

F(illing

> Affects I(nsert and M(argin. It is a Boolean parameter with default value "False".

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 1 and 132. Default values: L(eft - 1, R(ight - 80, P(ara - 6.

C(ommand ch

> Affects I(nsert and M(argin. It is a character parameter with default value ".". See the M(argin command (section 4.2.12) for details on command characters.

T(oken def

> Affects F(ind and R(eplace. It is a Boolean parameter with default value "True". See section 4.1.6.2 for more information.

B(uffer auto

> Affects automatic paging of file buffer. It is a Boolean parameter with default value "True". See section 4.1.6.2 for more information.

WARNING - F(illing True while A(utoIndent is False is known as "munch" mode (see I(nsert command, section 4.2.8).

4.2.20 <takeup>

Prompt:

>Takeup: <f1>..<f8> or C(opyBuffer <esc>

4.2.20.0 <takeup> <f1>..<f8>

The text form starting at the current cursor position becomes the function definition of the specified key.

Text forms are function definitions encoded as legal text so they may be stored in a text file. To activate a text form by associating it with a function key, the cursor is moved to the front of the text form and <takeup> is typed. The user is prompted for the function key which is to be thus defined; after typing this function key, the definition is loaded. The cursor is left at the end of the text form and the "=" pointer is set to the point where <takeup> was struck.

NOTE - Text forms must be wholely contained in the file buffer in order to be taken up.

NOTE - If the cursor is not on a text form, text is taken up from the file until the character sequence "|." is read, or until the internal buffer for holding function key definitions overflows. Taking up plain text usually causes the following message to appear:

ERROR: Special key overflow. Press <spacebar> to continue.

See section 4.1.9 for details on user-defined functions.

4.2.20.1 <takeup> C(opyBuffer

Typing "C" causes the following prompt to appear:

>Takeup from copy buffer: <f1>..<f8> <esc>

The contents of the copy buffer are copied into the function definition of the specified function key.

NOTE - The text in the copy buffer is taken up literally, i.e. text forms are not recognized.

See section 4.1.9 for details on user-defined functions.

4.2.21 T(oDisk

Prompt:

>ToDisk: F(orwards or B(ackwards <esc>

Write part of the file buffer out to disk.

T(oDisk is used to empty the file buffer when it fills up with text; all text between the disk page which contains the current cursor position and the specified end of the buffer is written out to disk, thus creating some free space in the file buffer. It is common practice to send unneeded text out to disk.

See section 4.0.5 for details on the file buffer and paging.

4.2.22 U(ptop

The screen is redisplayed, positioning the current text line at the top of the screen.

U(ptop is useful when you wish to see as much as possible of the text following a given cursor position. It is often useful after a F(ind, or in function key definitions which use G(etch.

4.2.23 V(erify

Redisplay the text window and reposition the window so that the cursor is centered on the screen.

NOTE - This command is especially useful in those rare situations where one suspects that the editor is not displaying the screen or the cursor correctly; V(erify causes a complete redisplay of the screen and then repositions the cursor. If this feature is frequently necessary, something is probably wrong with the editor; please report the problem to the factory.

4.2.24 W(ordMove

Repeat factors are allowed.

Move the cursor in the current direction to the start of the next word. Words are defined to be delimited by <blank>s and/or <return>s.

WordMove will not move past an illegal character in the text. This can often be used to advantage, as in the following sequence which removes illegal characters from a file:

"/wx<DeleteCh><Etx>"

or as a text form:

|"EatBad"/wx|<|e|.

4.2.25 eX(change

Prompt:

[>] eXchange: Text <vector keys> {<etx>,<esc> CURRENT line}

Replaces characters in the text file with characters typed in, starting from the current cursor position. Single characters may be deleted from or inserted to the right of the cursor by the subcommands <DeleteChar> and <InsertChar>.

eX(change may range over any number of lines by using the vector keys, <tab>, and <return>; these commands move the cursor without affecting existing text. <Backspace> moves the cursor right or left (depending on the current direction), undoing any changes made within the current line. This is similar to its effect in I(nsert.

Cursor movement and text entry are not constrained to lie within existing text and may be used to extend lines either to the left or to the right.

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. If a number of lines have been eX(changed, only the last line is restored.

eX(change discards any blanks on the end of any line through which it passes.

Example of using eX(change:

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

```
- - - - - - - - - - - - - - - - - - - -
Boy, I just love this rutabaga pie!
Pass the groatcakes, dear.
- - - - - - - - - - - - - - - - - - - -
```

After typing "xdocumentation -<cr>such clever examples!!!!!<etx>", the sentence now appears as:

```
- - - - - - - - - - - - - - - - - - - -
Boy, I just love this documentation -
such clever examples!!!!!_
- - - - - - - - - - - - - - - - - - - -
```

4.2.25.0 Commands in eX(change

Any editor command may be invoked during eX(change provided that the key which invokes the command generates a nonprinting character (see section 8.3.2). For instance, if the editor is configured so that the I(nsert command is invoked either by typing "I" or a nonprinting <insertline> key, typing an "I" within eX(change replaces the character after the cursor with the letter "I", but typing the <insertline> key invokes the I(nsert command. When a command which has been invoked from eX(change terminates, the user returns to eX(change, as if the sequence had been "<etx><command> X".

The nonprinting commands <InsertCh> and <DeleteCh> are valid only within eX(change. <InsertCh> inserts a single blank character at the current cursor position; <DeleteCh> deletes the character under the cursor.

<Coll> is available as a subcommand of exchange. It causes the cursor to be placed in column 1 of the current line.

The two most common commands to be entered are I(nsert and D(elete; their use within eX(change allows fast localized editing with minimum effort.

NOTE - Some terminals lack the requisite keys for defining the nonprinting commands described above; in this case, prefixed key sequences serve as acceptable substitute key definitions (see section 8.3.2).

4.2.25.1 Paint Mode Exchange

Paint mode exchange is used to create diagrams containing vertical and horizontal lines (boxes, for instance) and to edit columns of characters. Normal cursor movement in eX(change is left-to-right; paint mode allows the user to dynamically change the cursor movement to up, down, left, or right.

Typing the <DirChange> key in eX(change generates the following prompt:

Xchg Dir: <arrow> <esc>

eX(change mode returns after the vector key corresponding to the desired painting direction is typed, with the new default direction for cursor movement displayed in the stylish little box at the front of the eX(change prompt line. Note that the cursor moving commands override the default cursor direction; only the entering of text is affected.

Example of paint mode:

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

```
- - - - - - - - - - - - - - - - - - -
    a1 -
    a2 -
    a3 -          HOME SWEET HOME
    a4 -
    a5 -
    a6 -
- - - - - - - - - - - - - - - - - - -
```

After typing "x<DirChange><down>01234<etx>", the text becomes:

```
- - - - - - - - - - - - - - - - - - -
    a0 -
    a1 -
    a2 -          HOME SWEET HOME
    a3 -
    a4 -
    a6 -
- - - - - - - - - - - - - - - - - - -
```

4.2.26 Z(ap

Delete all text between the cursor position and the "=" pointer if the "=" pointer is within the current text buffer.

NOTE - The "=" pointer is set by I(nsert, D(elete, F(ind, R(eplace, and <equals>; see section 4.1.4 for details.

NOTE - The "=" command may be used to view the area to be zapped (see section 4.1.4 for details).

NOTE - Z(ap is designed to be used following F(ind, R(eplace, or I(nsert; it should be used with caution in other situations.

If many characters are to be Z(apped, 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.

4.3 Sample Edit Session

The purpose of this section is to describe the actions performed in a typical editing session; it is intended to aid users who are unfamiliar with the UCSD text editor.

The following actions are illustrated:

-- Starting an edit session and creating a new work file for editing.

-- Using basic cursor-moving and text-changing commands to enter new text and change existing text.

-- Finishing the edit session by writing the text to a disk file.

NOTE - No attempt is made to completely describe the commands used in the following examples; detailed command descriptions may be found in sections 4.1 and 4.2.

NOTE - In the examples of console displays, the cursor is represented by an "underline" character. Some of the promptlines are truncated to fit in the document.

We begin at the beginning - here is the Pascal system prompt:

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

Typing "E" (for "E(dit") invokes the editor, which displays the following prompt:

```
Edit: [ASE 0.7n]
?<cr> Looks, <cr> Creates, <esc> Exits or Filename: _
```

To create a new work file, <cr> is typed; the screen then appears as follows:

```
-----------------------------------------------------------------
>Edit: A(djust C(opy D(elete I(nsert J(ump R(eplace eX(change ?
_




-----------------------------------------------------------------
```

The contents of the work file are displayed on the screen. The edit prompt appears across the top line of the screen and the cursor is positioned at the front of the file. Because new work files are empty, the screen is blank.

In this example, the I(nsert command is used to add text to the empty file. After typing "I" (for "I(nsert"), the editor prompt is replaced with the command prompt for I(nsert:

```
---------------------------------------------------------------
>Insert: Text{<bs> a char, <del> a line} [<etx> accepts, <esc>
_




---------------------------------------------------------------
```

Text may now be entered; the following character sequence is typed in: "Now is the time<cr>for all good men<cr>to come to the aid<cr> of the enemy." Each time the return key (<cr>) is pressed, the cursor moves to the start of the next line. The screen then appears as:

```
---------------------------------------------------------------
>Insert: Text{<bs> a char, <del> a line} [<etx> accepts, <esc>
Now is the time
for all good men
to come to the aid
of the enemy._


---------------------------------------------------------------
```

Note that the I(nsert command is still active; the text entered so far could be removed a character at a time by typing <bs>, a line at a time by typing <del>, or removed completely by typing <esc>; however, <etx> is typed to complete the command. The screen then appears as:

```
---------------------------------------------------------------
>Edit: A(djust C(opy D(elete I(nsert J(ump R(eplace eX(change ?
Now is the time
for all good men
to come to the aid
of the enemy._


---------------------------------------------------------------
```

The entered text is now a part of the file, and may be modified with the other editing commands.

Typing the <up> key twice moves the cursor to the following position:

```
------------------------------------------------------------------
>Edit: A(djust C(opy D(elete I(nsert J(ump R(eplace eX(change ?
Now is the time
for all good men
to come to the aid
of the enemy.

------------------------------------------------------------------
```

The eX(change command is used to replace existing text with new text; typing "X" (for "eX(change") replaces the editor prompt with eX(change's command prompt:

```
------------------------------------------------------------------
[>] eXchange: Text <vector keys> {<etx>,<esc> CURRENT line}
Now is the time
for all good men
to come to the aid
of the enemy.

------------------------------------------------------------------
```

Typing the character sequence "employees<etx>" exchanges "employees" for "men"; <etx> terminates the command. The screen now appears as:

```
------------------------------------------------------------------
>Edit: A(djust C(opy D(elete I(nsert J(ump R(eplace eX(change ?
Now is the time
for all good employees_
to come to the aid
of the enemy.

------------------------------------------------------------------
```

Typing "B2<down>6<right>" moves the cursor to the beginning of the current line (with the B(eginLine command), down two text lines, and six columns to the right:

```
--------------------------------------------------------------
>Edit: A(djust C(opy D(elete I(nsert J(ump R(eplace eX(change ?
Now is the time
for all good employees
to come to the aid
of the_enemy.

--------------------------------------------------------------
```

Typing "I" invokes the I(nsert command once again; however, note that part of the text line is temporarily moved to the right to make room for any inserted text:

```
--------------------------------------------------------------
>Insert: Text{<bs> a char, <del> a line} [<etx> accepts, <esc>
Now is the time
for all good employees
to come to the aid
of the_                                                 enemy.

--------------------------------------------------------------
```

Typing "ir<etx> inserts the characters "ir"; the right-hand side of the text line is restored to its original position when the I(nsert command is accepted by the <etx>.

```
--------------------------------------------------------------
>Edit: A(djust C(opy D(elete I(nsert J(ump R(eplace eX(change ?
Now is the time
for all good employees
to come to the aid
of their_enemy.

--------------------------------------------------------------
```

Typing "D" (for "D(elete") invokes the D(elete command:

```
------------------------------------------------------------------
>Delete: < > <Moving commands>{<etx> to delete,<esc> to abort}
Now is the time
for all good employees
to come to the aid
of their_enemy.


------------------------------------------------------------------
```

Typing the spacebar five times followed by <etx> leaves the screen as:

```
------------------------------------------------------------------
>Edit: A(djust C(opy D(elete I(nsert J(ump R(eplace eX(change ?
Now is the time
for all good employees
to come to the aid
of theiry.


------------------------------------------------------------------
```

Typing "I compan<etx>" uses the I(nsert command to finish the text modifications made in this session:

```
------------------------------------------------------------------
>Edit: A(djust C(opy D(elete I(nsert J(ump R(eplace eX(change ?
Now is the time
for all good employees
to come to the aid
of their company.


------------------------------------------------------------------
```

To terminate the edit session, "Q" (for "Q(uit") is typed. The following prompt then appears:

```
Quit: _
    A(nother file (after Updating)
    B(ackup and re-edit same file
    C(hange the name of the output file
    E(xit (but workfile not updated)
    R(eturn to the editor without doing anything
    U(pdate the workfile and leave

Output File Name: SYSTEM.WRK.TEXT
```

The file name "SYSTEM.WRK.TEXT" is the work file name; to save the file under a different name, the C(hange option is invoked by typing "C". The prompt then appears as:

```
Quit:
    A(nother file (after Updating)
    B(ackup and re-edit same file
    C(hange the name of the output file
    E(xit (but workfile not updated)
    R(eturn to the editor without doing anything
    U(pdate the workfile and leave

Output File Name: _
```

The file is saved under the name "TREASON.TEXT" by typing "treason <cr>"; the prompt then appears as:

```
Quit: _
    A(nother file (after Updating)
    B(ackup and re-edit same file
    C(hange the name of the output file
    E(xit (but workfile not updated)
    R(eturn to the editor without doing anything
    U(pdate the workfile and leave

Output File Name: TREASON.TEXT
```

Typing "U" (for "U(pdate") writes the file to a disk file named "TREASON.TEXT" and terminates the edit session. The system prompt displayed at the start of the edit session reappears:

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

This completes the edit session.

4.4 Problems

This section serves two purposes: to enumerate and explain error conditions detected and flagged by the editor, and to explain unexpected operating characteristics of the editor itself.

This section consists of two parts: a symptom list describing editor actions symptomatic of a bug and/or error condition, and a list of detailed problem reports. The symptom list serves as a reference index for the problem reports.

Each symptom list entry is assigned a number corresponding to a problem report entry, e.g. a symptom list entry assigned the number "9" indicates that a detailed description of the underlying problem may be found in the ninth entry in the problem report.

Each problem report entry contains the following fields:

1) Report number - Used to address problem reports from the symptom list. Problem reports are ordered by report number; new problem reports are added to the end of the list and allocated a report number.

2) Overview - A capsule summary of the problem.

3) Severity - One of four values: "cosmetic", "minor", "major", or "lethal". "cosmetic" implies problems of an aesthetic rather than a functional nature. "minor" implies harmless but confusing problems; these usually require explicit user actions to correct. "major" implies that the integrity of the edit session is threatened. "lethal" implies that the integrity of one or more disk files is threatened.

4) Detailed description - Provides information useful in understanding and thus overcoming the problem.

5) Solution - Suggested detours with which users can avoid or work around the stated problem.

4.4.0 Symptom List

Symptom	Problem Report
The editor generates the error message:	
Not enough room for backup!	1
No more editing room.	2
The editor behaves inexplicably when:	
The disk volume containing the editor code file is removed during an edit session.	3

4.4.1 Problem Reports

Report: 1

Problem:

Not enough disk space to create a working copy of the specified file.

Severity: minor

Description:

This error only occurs at the beginning of an edit session, when the editor is attempting to copy the backup file. A free space on disk at least as large as the file to be edited must exist before an edit session can commence. When the error occurs in a nested edit session, the editor leaves behind a trail of "ASE!" files containing the unfinished edit sessions; however, the backups for these files still exist with their original names.

Solution:

Use the menu option to check if a file may be edited before attempting to specify it for an edit session. Maintain the disk volume's free space regularly with the filer commands; a fine way to gain extra disk space is to remove old backup files.

Report: 2

Problem:

Not enough disk space to enlarge the file in current edit session.

Severity: minor

Description:

ASE uses the largest available free space on disk to hold the file in the current session; however, an overabundance of text insertions can cause a file to grow larger than its allocated disk area. At this point, all attempts to insert the offending text are forestalled by the displayed error message.

Solution:

Update the file in its current state; no space problems will be encountered by doing so. Use the filer to create a larger free space, and then resume editing in a new edit session.

Report: 3

Problem:

The editor crashes after removing the disk volume containing the editor's code file in the midst of an edit session.

Severity: major

Description:

ASE contains a number of disk-resident code segments. If the disk containing these code segments (i.e. the disk containing the editor's code file) is not mounted in the expected disk drive, the system will crash when it attempts to read one of the code segments into memory.

Solution:

Do not remove the editor's disk volume during an edit session.

Report: 4

Problem:

Severity:

Description:

Solution:

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 which are unavailable in its current environment. P-code information is 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 text 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 name to default to *SYSTEM .WRK.CODE[*]. 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".

NOTE - "$" does not include the volume identifier. If the textfile is "#4:STUFF", "#4:$" must be entered to put "STUFF.CODE" on volume 4. Otherwise "$" compiles to the prefixed disk.

The suffix ".CODE" is automatically appended by the compiler to any specified output file name 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.

If the current work code file is not named *SYSTEM.WRK.CODE, the work code file is replaced by the new code file.

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 somewhat.

Example of a console display:

```
PASCAL Compiler [AOS 1.0]
--> SYSTEM.WRK.TEXT
<   0>..............+++++........................
LAINIT [28710]
<  43>.........
GETFILE [28692]
<  52>...................
WRITEIT [28674]
<  71>.............
NEWLINE [28634]
<  84>...................++++...........................
< 134>.................................++...............
< 184>........
COPYIT [28616]
< 192>.............
SEND [28627]
< 205>......
211 lines, 6 secs, 2110 lines/min
Compiled to WORKDSK:SYSTEM.WRK.CODE
```

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. The compiler outputs either a '.' or a '+' to the screen for each line compiled. '.' is output for any line not contained in a comment; '+' is output for commented sections. The file name following the symbol "-->" indicates a new source file. A file name following the symbol "--" indicates the current source file. The destination codefile name is printed at the end of the compilation.

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 if the error number is less than 400; otherwise the compilation is aborted. Typing <esc> aborts the compiler and returns control to the system prompt. Typing "E" automatically invokes the editor. The editor first prompts for the name of the current input file. Once the file is specified, the editor reads the input file and prompts:

Jump to what marker? $SYNTAX

Typing <return> causes the Editor to position the cursor over the error, and display the error number or message. See section 4.1.0 for details.

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

NOTE - If the wrong input file name is given to the editor, the editor reads in the file and gives the "Jump $SYNTAX" message as above. However, the Editor may:

1) respond with the message: "Error, marker not there"
2) respond with the message: "Marker all messed up"
3) position the cursor where the error would be if the correct file were read in (see section 5.2.0)

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 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. If a workfile exists, the system may enter the wrong file name into the editor file name prompt.

This problem arises when a Pascal source program is spread across a number of text files that are "included" into the compiler's input stream (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.

If no workfile exists, 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 being compiled (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. To get around this problem, type <del> upon entering the editor (this removes the work file name from the input prompt), and enter the name of the correct file, or avoid using the work file when a program uses "include" files.

5.2.1 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!

5.2.2 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 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. 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 6.0. Command language syntax is described in section 6.1. Examples of command programs appear in section 6.2. The file "X.DEMO" is a command file that presents an overview of the command interpreter.

6.0 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 6.1.1.

A list of parameters may be specified after the command file name. Command file parameters are strings of characters delimited blanks and terminated by the end of the line. Up to 9 parameters may be passed. Parameters are discussed further in section 6.1.3.

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

6.0.1 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 S(ubmit keyboard type-ahead buffer contains a maximum of 256 characters. Data entered into the type-ahead buffer by the command interpreter is read from the buffer before any data actually entered from the keyboard.

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 prompt lines. If an unexpected system condition causes an unanticipated 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.

6.0.2 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 - "$EXEC" must reside on the prefixed volume. "PROFILE" must reside on the system 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.

6.1 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 6.1.0. Commands take either "targets" or "textlines" as arguments. Targets are used as arguments by the flow-of-control commands; they are described in section 6.1.1. Textlines contain text that is either immediately written to the screen or queued in the type-ahead buffer; they are described in section 6.1.3. Parameters and variables contain text that may be manipulated by a command file program; they are described in section 6.1.2.

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.

6.1.0 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.

6.1.0.0 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 6.1.3).

GOTO

Form: GOTO <target>

Command interpretation continues at <target>.

CALL

Form: CALL <target> { <param>}

Command interpretation continues at <target>, but returns to the command following the CALL after a RUN command is executed. Up to 9 parameters may be passed to the <target> routine. They are treated as local inside of the routine. All parameters are passed by value. No variable paremters are allowed.

CALL's may be nested up to 18 levels deep.

SET

Form: SET <variable#> <value>

Sets the specified variable (|variable#) to the specified value. The value may be a string constant or another variable.

EQU

Form: EQU <valuel> <value2> <target>

Performs a conditional jump to <target> based on the comparison of the values <valuel> and <value2>. The values may be either string constants or variables. A GOTO to <target> is performed if the values are equal.

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.

6.1.0.1 Deferred Commands

STK

Form: STK <textline>

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

S

Form: S <target> { <param>}

STKs a S(ubmit command for <target>. Up to 9 parameters may be passed to the <target> routine.

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.

6.1.1 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.

6.1.2 Parameters and Variables

Up to 9 variables are available to a command file program at any time. They are accessed as |1 through |9. Each variable may contain a text string. A new set of variables is allocated each time a CALL is performed. The old set of variables is placed on a stack (up to 18 deep) until the CALLed routine is terminated.

Variables are initialized to the values of the parameters passed to the command interpreter or passed through a CALL. Parameters are passed as sequences of characters separated by spaces. They occur after a file name or target. The first parameter is assigned to |1, the second parameter to |2, etc. The |0 variable contains the number of variables initialized to parameter values.

Variables may also receive values by using them in a SET operation. Their values may be tested by using the EQU instruction. They may be printed by using a WRITE instruction. They may also be passed as parameters to other routines.

6.1.3 Text Lines

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

```
" "     <space>           "n"    <return>
"|"     | {single "|"}    "b"    <bs>
"u"     <up>              "!"    <escape>
"d"     <down>            "@"    <delete>
"l"     <left>            "t"    <tab>
"r"     <right>           "?"    Answer to last READ
"k"     Relinquish control to the console keyboard
"0".."9" variables 0 through 9
```

An occurrence of "|?" in a textline is substituted with the text read in by the last READ command.

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 may be punctuated with requests for input directly from the keyboard, allowing automated tasks to possess interactive capabilities. (See the example in section 6.2).

6.2 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 |? |n | | | 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 prompt line. 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 lk ln l l l 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.

NOTE - Another example of an eXec program appears in the BINDER .TEXT command file. The operation of this program is described in section 8.3.0.

VII. 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 prompt line; instead, the prompt character ("#") is printed on the console. HDT commands are described in section 7.1. Examples of using HDT appear in section 7.2.

NOTE - HDT is implemented as a Pascal program resident in PROMs on the PDQ-3 CPU board. 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.

7.0 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 may 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.

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".

7.1 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 HDT commands are:

R

Form: [<number>]R

Reboot the system from the specified device. If no <number> is specified, the system is bootstrapped from the default bootstrap device, and the system floppies are configured for single-sided operation.

If a <number> is specified, it indicates an alternate bootstrap device or floppy drive configuration. The <number> consists of two digits. The first digit indicates the type of the desired bootstrap device. In versions of HDT capable of bootstrapping from only one device, the value of this digit is irrelevant. In versions capable of bootstrapping from two devices, this digit is normally 0 to indicate a hard disk and 1 to indicate a floppy drive. The second digit selects a particular drive of the device specified by the first digit, and indicates the initial floppy configuration. A 0 designates drive 0; 1 designates drive 1. The floppy drives may be configured for double-sided operation by adding 4 to the drive number.

For an exact interpretation of <number> for a given hardware configuration, consult the Hardware User's Manual.

P

Form: P

Resume execution of a suspended user or system program. Invoking this command if a program is not currently suspended (i.e. if the monitor was entered because of a power failure or a RESET condition) 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.

NOTE - Most HDT PROMs are capable of bootstrapping the Pascal system from only one type of device (e.g. floppy drives). Some HDT PROMs can bootstrap the system from one of two possible devices. These versions of HDT are not capable of executing the memory examination commands. The HDT.DRVR.CODE file on the AOS release disk contains a version of the SYSDRIVER I/O driver which has a software version of the monitor. The software version identifies itself with a "%" prompt. It implements all monitor commands except R. In addition, the H command is provided; it invokes the HDT PROM, from which the R command may be used. Section 2.3.1 describes how to replace a system I/O driver.

7.2 HDT Examples

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

Starting the system with the system disk mounted:

```
#R or #15R
```

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>
#
```

VIII. 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 X(ecute command.

8.0 Disk Management

This section describes the utility programs used to manage disk media: Booter, Backup, Mapper, Format, Bad.Blocks, Drive.Con and Change.Dir.

Booter copies the system bootstrap from one disk to another. Track 0 must contain the bootstrap code required for bootable system disks. Booter is described in section 8.0.0.

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 8.0.1.

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 8.0.2.

Format writes formatting information on blank disks so they may be used on the PDQ-3 system. Format is described in section 8.0.3.

Bad.Blocks performs high-speed scanning of disks for bad blocks; it is described in section 8.0.4.

Drive.Con configures virtual floppies on a hard disk drive; it is described in section 8.0.5.

Change.Dir changes the number of blocks a directory may access; it is described in section 8.0.6.

8.0.0 Bootstrap Copier

The Booter utility copies a system bootstrap (i.e. all of track 0) from a source to a destination. The source bootstrap may exist either in a file or on a volume and it may be copied to either a file or a volume.

8.0.0.0 Using Booter

X(ecute Booter. The following prompt appears:

    Read from F(ile or U(nit?

Typing <cr> terminates the program.

Typing 'F' causes Booter to prompt for a file name; typing 'U' causes Booter to prompt for a unit number:

    Name of file?
        or
    What Unit?

Typing <return> causes the first prompt to reappear; typing the name of a file containing a bootstrap or the number of an online unit completes the read.

After a successful disk read, the following prompt appears:

    Read successful.

    Write to F(ile or U(nit?

Typing <return> exits Booter. The responses to this prompt specify the bootstrap destination and are entered in the same manner as above. Upon completion of a successful transfer, Booter verifies:

    Write successful.

8.0.1 Disk Copying

The utility program Backup 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.

8.0.1.0 Using Backup

X(ecute Backup. The following prompt appears:

```
Enter Master's Unit # :
```

Typing <return> exits Backup. Typing the number of the unit containing the information to be copied generates the prompt:

```
Enter Backup's Unit # :
```

Typing <return> exits Backup. Typing the number of the unit to which the backup information will be copied causes the following verification message to appear:

```
Master on <source unit number> Volume <source volume name>
```

If the destination volume has a directory the following prompt appears:

```
Destroy <dest unit number> Volume <dest volume name> ?
```

Typing <return>, <space>, 'N' or <esc> generates the exit prompt described below. Typing 'Y' causes the following information to be displayed:

```
Backup on <dest unit number> Volume <dest volume name>
Backing up <# of occupied blocks on source volume> blocks
```

Backup then proceeds to copy the source volume to the destination volume; it writes a series of dots to the screen to indicate its progress. Typing <esc> at any time during the copy generates the exit promt which is described below. When copying is successfully completed, this prompt appears:

```
Unit # <dest unit number> is currently named <dest vol name>.
Would you like to rename it?
```

Typing 'Y' or 'y' generates the prompt:

Enter volume name:

Type <return> to avoid renaming the destination volume and continue to the next prompt. Otherwise, type the desired name for the backup volume. All lower case characters are converted to upper case, and any trailing colon is stripped. The following prompt appears:

<unit number> Renamed <new volume name>
Directory contains <# of blocks on source volume> blocks.
Change Size ?

Typing 'Y' generates the prompt:

New number of blocks (<# occupied blks on dest vol - 32767)?

Type <return> to avoid changing the volume block size; otherwise, type the number of blocks desired. Backup verifies:

<unit number> now contains <new block number> blocks

The exit prompt then appears:

E(xit to Boot Diskette in Boot Drive ?

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 other character) redisplays the original Backup prompt:

Enter Master's Unit # :

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

8.0.2 Disk Format Conversion

The utility program Mapper 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 1.4.3.4 for details.

8.0.2.0 Using Mapper

X(ecute Mapper. The following prompt appears:

```
Source unit number ?
```

Enter the number of the unit containing the source disk; <return> exits the program. If the source unit is a floppy disk, the following prompt appears:

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

The choices available are: "D", "W", "P", and <return>. The first three choices specify the corresponding disk format; <return> allows the source unit number to be reentered. If the source unit is a hard disk, the source format defaults to P(dq.

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 destination unit number is entered in the same manner as the source unit number.

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

```
Map #<source unit> : [<source format>]
        ---> #<destination unit> : [<destination format>] OK ?
```

The choices available are: "Y", "N", and <return>. Typing "Y" starts the mapping process; typing either "N" or <return> allows the source and destination units to be reentered.

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> terminates mapping and returns to the source unit prompt.

When mapping is completed, a new source and destination unit may be specified.

8.0.3 Disk Formatting

The utility program Format formats floppy disks in the PDQ-3 disk format. Disk formatting is used for:

1) Preparing new disks (8" soft-sectored floppies 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.

NOTE - To format any disks other than PDQ-3 floppies, please see the subsystem document for that disk.

8.0.3.0 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 either 4 or 5 generates the following prompt:

    Format single or double density? (S or D)

Typing "S" specifies single density formatting; typing "D" specifies double density. Typing <escape> aborts the formatter.

The next prompt is:

    Format single or double sided? (S or D)

Typing "S" specifies single-sided disks; typing "D" specifies double-sided. Typing <escape> aborts the formatter.

NOTE - Before choosing double density, be sure that the floppy disk are rated for double density usage. Before choosing double-sided, be sure that the disks AND disk drives support it.

NOTE - The formatter configures the system floppies for single- or double-sided operation according to the prompt response. This configuration will remain in effect after the formatter terminates.

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. Typing <escape> aborts the formatter. See the Archi-

tecture 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 <escape> aborts the formatter. 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 <track # being processed>
    Verifying <track # being processed>

8.0.3.1 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, because 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 an entire track to fix a bad block destroys the contents of adjacent blocks on that track.

8.0.4 Fast Bad Blocks Scanning

The utility program Bad.Blocks checks a disk file or disk volume for damaged blocks. Bad blocks scanning may also be performed with the filer's B(ad blocks command; however, Bad.Blocks is much more convenient. Bad blocks are repaired with the filer's X(amine command or the Format utility (section 8.0.3).

8.0.4.0 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.

NOTE - For more bad block information during the scan, the <control-D> Noisy option may be invoked before starting Bad.Blocks. See section 1.4.3.4 for more details.

8.0.5 Hard Disk Management

The utility program Drive.Con is used to allocate disk volumes on hard disk drives. Up to 32 disk volumes per drive may be allocated. Drive.Con displays and alters a drive configuration consisting of a volume label, a size and a status for each disk volume.

8.0.5.0 Using Drive.Con

X(ecute Drive.Con. The following promptline appears:

```
Command: R(ead, W(rite, C(hange, D(ismount, M(ount,
                                         N(ew, Q(uit [1.0]
```

Q(uit exits Drive.Con. <Return> aborts any prompt that does not have an explicit escape.

8.0.5.1 Editing Drive.Con Prompts

Some of Drive.Con's prompts display a series of '____' which determine the field size of the requested response. The field may be modified using a protocol similar to that of the eX(change mode in the system editor; entering a printable character causes the character under the cursor to be replaced by the character entered. Utility keys used to alter the fields are:

Key	Action
<bs>	moves cursor left one space
<left>	moves cursor left one space
<del>	moves cursor to beginning of field
<right>	moves cursor right one space
<tab>	moves cursor to end of field
<return>	accepts all input up to cursor position
<etx>	accepts all input in field

Certain commands allow a range of table entries, indexed by letters, to be affected. The response to such a command may be either a single entry index (e.g. A), a closed entry range (e.g. A-C), or an open entry range (e.g. -C means from the beginning to C, B- means from B to the end, - means from the beginning to the end). If the response contains a '?', the operation is verified for each entry of the range before it is carried out.

8.0.5.2 Drive Configurations

A drive configuration describes the allocation of disk volumes on a hard disk drive. It may exist either in a text file or on a track designated for configuration information (see Appendix D of the Programmers Manual). A configuration residing on the configuration track describes the disk volume allocation for that drive. Configurations may also be stored in text files for subsequent use as data.

A drive is a sequence of *segments* consisting of a drive-dependent number of blocks. Segments are numbered beginning with 0; segment 0 is reserved. Disk volumes may contain an arbitrary number of segments and are identified by a 15 character *volume label*. They may be either *mounted* (visible to the system) or *dismounted*.

8.0.5.3 Displaying a Drive Configuration

R(ead generates the prompt:

Channel: F(ile, D(rive, E(xit

This allows the selection of a drive configuration from either a disk file or a drive's configuration track. F(ile generates the prompt:

File name ? ________________________

Type the name of a file containing a drive configuration.

D(rive generates the prompt:

Drive _

Type the number of a drive whose configuration is to be examined; 0 for systems containing only one hard disk drive.

To obtain a blank drive configuration, type N(ew and choose the appropriate drive type from the selection shown.

The drive configuration display includes a heading which reads:

Volume label   Start   Size   Empty

The Volume label describes the disk volume contents. The Volume label has no relation to the volume name. Start gives the starting segment address on the drive. Size indicates the number of segments allotted to the volume, and Empty shows the number of empty segments between the end of one volume and the beginning of the next (0 if none). The number of blocks corresponding to a quantity of segments may be calculated by multiplying the number of segments by the number of blocks per segment (displayed at the bottom of the screen). Volume label and Size may be altered manually; Start and Empty are automatically maintained by Drive .Con.

8.0.5.4 Altering Drive Configurations

The display may be altered by using the M(ount, D(ismount, and C(hange commands.

8.0.5.4.0 M(ount and D(ismount

M(ounting a volume makes it visible to the system, and D(ismounting renders it invisible (although still intact). The maximum number of mountable volumes is determined by the System I/O Configuration (see section 2.3.1). If the number of mounted volumes exceeds the number of physical units allocated to the hard disk; only those volumes that correspond to physical units are visible to the system. Volumes are assigned to physical units starting with the first mounted entry in the configuration table. In multiple drive systems, these volumes are selected starting with the boot drive and then proceeding from drive 0 to the last drive. The first volume mounted is marked with an asterisk (*) to the left of the volume label and is considered the boot volume. Other mounted volumes are marked with a number sign (#).

M(ount and D(ismount generate the prompt:

    Which entry (<char>-<char>)?

Responses are of the form:

    <response> ::= [<char>] ["-"[<char>]] ["?"]

Responses are entered as described in section 8.0.5.1.

8.0.5.4.1 C(hange

C(hange generates the same prompt line as M(ount and D(ismount, but only single character responses are allowed. C(hange is used to create and destroy volumes, and to alter the volume label and size of existing volumes.

Existing volumes are altered by selecting the index of the desired volume and editing the desired field. The Volume label field must be edited and accepted, before the Size field may be edited. The size of a volume may be reduced by specifying fewer segments; a volume may be removed by specifying a size of 0. The size of a volume may be increased by specifying more segments, assuming an adequate number of empty segments follow the volume.

New volumes are created by C(hanging the first empty entry on the display. Drive.Con will allocate the new volume in the first available space of that size, and update the display accordingly. Before a new volume is visible to the system, it must be M(ounted using Drive.Con and Z(eroed using the F(iler.

8.0.5.5 Saving Drive Configurations

Drive configurations are saved using the W(rite command, which generates the prompt:

    Channel: F(ile, D(rive, E(xit

F(ile generates the prompt:

File name ? ________________________

Typing a file name causes the configuration to be saved in a text file. Drive configurations may be saved in text files for archival purposes.

If the drive configuration on display was not originally R(ead from a drive, D(rive generates the prompt:

Drive _

Type the number of the drive on which the configuration is to be written; 0 for systems containing only one hard disk drive.

If the drive configuration on display was originally R(ead from a drive. D)rive generates the prompt:

Write to Drive <number>?

Type 'Y' or 'y' to write the drive configuration to the drive from which it originated. Typing 'N' or 'n' results in a prompt for a drive number as described above.

If an attempt to exit Drive.Con is made, and the current drive configuration has not been saved, the following appears:

Nothing updated: W(rite, R(eturn, Q(uit

Q(uit then exits the program, R(eturn returns to Drive.Con, and W(rite behaves as described above and then exits the program.

WARNING - W(rite D(rive replaces the configuration on a drive. The prior configuration will be lost if it is not saved in a text file, which may result in lost volumes. If the boot volume has been dismounted, the system may not reboot from the hard drive, but it may be rebooted from a floppy disk, and the configuration restored.

8.0.6 Changing Volume Size

The utility program Change.Dir is used to change the number of blocks a directory may access. This program should be used to match the number of blocks a directory may access with the actual volume size. For example, after a volume size is changed using Drive.Con, the size of the directory on the volume should be changed using Change.Dir.

8.0.6.0 Using Change.Dir

X(ecute Change.Dir. The following prompt appears:

    What is the unit number of the directory you wish to change?

Enter the unit number containing the directory to be changed followed by <return>. Change.Dir prompts:

    The current size of <volume name> is <volume size>;
    What is the new size (<lower bound>-<upper bound>)?

<Volume size> is the number of blocks currently accessible from the directory. Choices for a new size are limited to values that do not exclude any files already on the volume; existing files may not be removed with Change.Dir. If the response is outside of these bounds, Change.Dir aborts with the prompt:

    New size must be at least <lower bound>.

If the response is within bounds, and no I/O error occurs in updating the directory, Change.Dir reports:

    Directory write successful.

NOTE - Typing <return> to any prompt in Change.Dir exits the program.

NOTE - The number of blocks allotted to a directory may be different than the actual number of blocks on the volume (as seen in Drive.Con). If the directory is larger, I/O errors occur on accesses beyond the end of the volume; if the directory is smaller, some volume space is inaccessible to the user hence wasted.

8.1 Data Recovery

This section describes utilities involved in the recovery of data lost through unfortunate circumstances; specifically, the recovery of files from trashed directories. The utilities are Markdupdir, Copydupdir, and Recover.

Markdupdir 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 copies the duplicate directory into the location of the primary disk directory; it is used after unfortunate circumstances destroy the primary directory.

Recover searches a volume for lost text and code files; it is used after both primary and duplicate directories have been destroyed. Files are recovered first by searching the existing directory for valid entries, then by scanning the volume for groups of blocks that appear to be files.

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

8.1.0 Using Markdupdir

X(ecute Markdupdir. The following prompt appears:

    Enter Unit # :

Type <return> to exit Markdupdir; otherwise, type the number of the volume to be marked. If the disk volume already has a duplicate directory, the user is notified and Markdupdir is terminated. If no duplicate directory exists, 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 directory entry 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

or

<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

8.1.1 Using Copydupdir

X(ecute Copydupdir. It first prompts for the disk drive in which the copy is to take place.

The user is notified if the disk is not currently maintaining a duplicate directory (see section 8.1.0). If a duplicate directory is found, a prompt is issued to verify that the current primary directory is to be overwritten. Typing "Y" copies the directory; typing any other character exits Copydupdir.

8.1.2 Using Recover

X(ecute Recover. The following prompt appears:

    Enter # of Unit to be Recovered:

Typing <return> exits Recover; typing the number of the desired volume generates the prompt:

    Enter volume name:

Typing <return> exits Recover; otherwise, the first seven characters typed in are entered into the directory as the volume name. All lower case letters are converted to upper case, and trailing colons are deleted.

Recover first attempts to read the volume size from the current directory. If invalid information is received, Recover prompts:

    Number of blocks on the disk?

Any positive integer may be entered. Recover then searches for valid entries in the volume's directory. Each entry found is listed as:

    <file name> found

If no valid entries are found in the volume's directory, Recover reports:

No files found

After the directory search is completed, the following prompt appears:

Are there still IMPORTANT files missing ?

Typing 'N' or <return> discontinues the file search; typing 'Y' continues with a block by block search through the volume for Text and Code files (Data files are ignored). Each file discovered generates a report:

File <file name> inserted at blocks <start blk>-<end blk>

Upon completion of Recover's file search, this prompt is displayed:

Go ahead and update directory?

Typing 'Y' writes the new directory to the volume; typing 'N' or <return> does not.

8.2 Library Management

Libraries are managed with the utility programs Library and Libmap.

The Library utility may construct a library file from other libraries and code files. It may also bind a unit code file with a program code file to make that code file completely portable (i.e. the program code file has its own copy of the unit so that it may be run in an environment where that unit is not normally available). Library is described in section 8.2.0.

The Libmap utility gives detailed information on the contents of a code file. This includes a list of all segment names and sizes, used units and versions, and other attributes of the unit. Optionally, the interface section of a given unit may be listed. Libmap is described in section 8.2.1.

See section 2.2 for a system-level description of units and libraries and the Programmer's Manual for a program-level description of units and libraries.

8.2.0 Using Library

X(ecute Library. The partial terminal display containing the compilation unit header and output file prompt appears:

```
Status Name    #Blk    Vers    Seg    Ref
    .
    .
    .
    .
Output file: *System.Library__________
```

8.2.0.0 Editing Library Prompts

All Library prompts display a series of '____' which determine the field size of the requested response. The field may be modified using a protocol similar to that of the eX(change mode in the system editor; entering a printable character causes the character under the cursor to be replaced by the character entered. Utility keys used to alter the fields are:

Key	Action
<bs>	moves cursor left one space
<left>	moves cursor left one space
<del>	moves cursor to beginning of field
<right>	moves cursor right one space
<tab>	moves cursor to end of field
<return>	accepts all input up to cursor position
<etx>	accepts all input in field

Certain commands allow a range of table entries, indexed by letters, to be affected. The response to such a command may be either a single entry index (e.g. A), a closed entry range (e.g.

A-C), or an open entry range (e.g. -C means from the beginning to C, B- means from B to the end, - means from the beginning to the end). If the response contains a '?', the operation is verified for each entry of the range before it is carried out.

8.2.0.1 Output and Input Files

When Library is first executed, the output file prompt appears at the bottom of the terminal display. To exit Library, type <del> followed by <return>. The default output file is *System.Library. The file name may be changed using the editing features described above. The output file display consists of:

```
Output File: <file name>     Blocks: <current outfile size>
     "<copyright notice>"
```

The output file size is updated after appropriate Library commands. The copyright notice is optional.

After the output file name has been accepted, the input file prompt appears at the top of the terminal display:

```
Input File: *System.Library
```

The input file name defaults to the output file name if the output file already exists. Otherwise, it defaults to '*System.Library'. The input file name is specified in the same manner as the output file name. Library attempts to open the input file as specified. If the attempt fails, ".Code" is appended and the operation is retried. After the input file is opened, the units contained in this library appear as entries in the terminal display and the copyright message (if any) appears beneath the input file prompt.

8.2.0.2 Library Commands

After the output and input files are specified, the following prompt appears at the top of the display:

```
Library: N(ew, K(eep, T(oss, R(efs, S(trip, I(nt, C(opy,
                                   A(bort, Q(uit [1.0]
```

8.2.0.2.0 K(eep and T(oss

The K(eep command is used to mark units to be included in the output file. All K(ept files are marked with a '#' at the left of the index letter. If the input file supplies more than one compilation unit, the K(eep command prompts for a range of entries to K(eep.

The T(oss command is used to cancel the action of the K(eep command. T(oss may be used on any entry marked with a '#'.

Note - The K(eep command simply changes the status of an entry. A unit is not copied from the input file to the output file until

either the N(ew or Q(uit command is invoked.

8.2.0.2.1 S(trip and I(nt

The S(trip command is used to remove the interface text from the library. This action is noted on the display by the disappearance of the 'i' to the left of the unitname and a decrease in the number of blocks occupied by the unit. The I(nterface command restores the text.

8.2.0.2.2 R(efs

The R(efs command is used to K(eep a unit and all of the units it references. Referenced user units not contained on the display are enumerated by entries with an 'r' to the left of the unit name. These units may be found in other libraries. Referenced system units are enumerated by entries with an 's' to the left of the unit name. System units are provided by the system; thus they are unavailable for binding. No operations are allowed on 'r' and 's' entries; they are for informational purposes only.

8.2.0.2.3 N(ew

The N(ew command is used to specify a new input file. At this time, all K(ept entries are copied to the output file. This changes the status of these entries from '#' to '*', which signifies permanent status (i.e. no further operations may be performed on them). N(ew may be aborted by typing <del> <return>.

8.2.0.2.4 C(opy

The C(opy command is used to copy the copyright message from the input file to the output file. The copyright messages associated with these files are displayed immediately below the file name.

8.2.0.2.5 A(bort

The A(bort command purges the output file and aborts Library.

8.2.0.2.6 Q(uit

The Q(uit command is used to close the output file. All entries are processed as in the N(ew command. The cursor is placed at the end of the output file copyright message for possible editing. The output file is closed when the copyright message has been accepted. Note that if the Q(uit command is issued and there are no K(ept entries, Library A(borts.

8.2.1 Using Libmap

X(ecute Libmap. The following prompt appears:

    What is the name of the output file (<cr> for Stanout:) ?

Typing <return> sends the output to the standard output; typing a file name sends paged output to that file name appended with .TEXT. Either response generates the following prompt:

    What is the name of the library file (<cr> to exit) ?

Typing <return> exits Libmap; otherwise, Libmap automatically appends ".CODE" to the library file name unless the file name is followed by a period (which is stripped). Typing "*" is a special case which invokes the "*System.Library" file.

Following the information for each compilation unit, this prompt appears:

    Do you wish to list the interface section for <unit name> ?

Typing 'Y' generates a listing of the interface section for that unit; typing <esc> aborts the Library listing and regenerates the prompt:

    What is the name of the library file (<cr> to exit) ?

This prompt also appears after the map file is completed.

Example of a library map listing:

```
FILE: *library.Code
(c) Copyright Advanced Computer Design, 1982  All Rights Reserved


=====================================================

Program LIBRARY    Version: 0   Machine: PDQ-3
   Global Data Size: 1265 words   Uses II.0 heap
  Segment 128: LIBRARY    Size: 1969 words
  Referenced Unit 130: APPPROCS   Version: 1   Resident
  Referenced Unit 129: SCCNTRL    Version: 0   Resident

=====================================================

Unit SCCNTRL    Version: 0   Machine: PDQ-3
   Global Data Size: 1 words   Uses II.0 heap
  Segment 128: SCCNTRL    Size: 1350 words
  Referenced Unit 8: GOTOXYU    Version: 0   Resident

=====================================================

Unit APPPROCS   Version: 1   Machine: PDQ-3
   Global Data Size: 55 words
  Segment 128: APPPROCS   Size: 846 words
  Referenced Unit 129: SCCNTRL    Version: 0   Resident
```

8.3 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 UCSD Pascal intrinsic, GOTOXY, implements random (i.e. X-Y coordinate) cursor positioning. It is used by the operating system, the system editor, and various utilities for screen formatting. Its use is described in the Programmer's Manual.

The data file named "SYSTEM.MISCINFO" resides on the system volume. SYSTEM.MISCINFO is a two block file containing system and editor information. The first block of SYSTEM.MISCINFO contains three kinds of system information: miscellaneous system data, terminal screen control characters, and key definitions for 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. The second block of SYSTEM.MISCINFO contains information used by the editor for terminal screen control sequences and special function keys.

Three system parts are used to reconfigure the system's terminal interface: Binder, Setup, and ASS.

The Binder utility binds a compiled GOTOXY procedure into the system support library. Details on creating, compiling, and binding a new GOTOXY are presented in section 8.3.C.

The Setup and ASS utilities are used to create a new miscinfo file. Setup (described in section 8.3.1) modifies the first block, which contains system information. ASS (described in section 8.3.2) modifies the second block, which contains editor information. The following table indicates which utility should be used in order to modify a given function.

	Use Setup	Use ASS
Screen Control Functions		
BACKSPACE CHAR	Y	
CURSOR HOME CHAR	Y	
CURSOR RIGHT CHAR	Y	
CURSOR UP CHAR	Y	
ERASE LINE CHAR	Y	
ERASE SCREEN CHAR	Y	
ERASE TO END OF LINE CHAR	Y	
ERASE TO END OF SCREEN CHAR	Y	
INSERT LINE CHAR		Y
NON-PRINTING CHAR	Y	
Keyboard Functions		
BACKSPACE KEY	Y	Y
CURSOR DOWN KEY	Y	Y
CURSOR LEFT KEY	Y	Y
CURSOR RIGHT KEY	Y	Y
CURSOR UP KEY	Y	Y
DELETE CHAR KEY	Y	Y
DELETE LINE KEY	Y	Y
EDITOR ACCEPT KEY	Y	Y
EDITOR COMMAND KEYS		Y
EDITOR ESCAPE KEY	Y	Y
END FILE KEY	Y	
System Information		
LOWER CASE	Y	
RANDOM CURSOR ADDRESSING	Y	
SLOW TERMINAL	Y	
SCREEN HEIGHT	Y	
SCREEN WIDTH	Y	
VERTICAL MOVE DELAY	Y	

8.3.0 GOTOXY Binding

The GOTOXY intrinsic is implemented as a procedure in a UCSD Pascal Unit (described in section 3.2 of the Programmer's Manual). The GOTOXY intrinsic may be modified for use with a given terminal by writing a new GOTOXY unit, compiling it to a code file, and binding it into the system support library. If the system has not been configured for the terminal being used, it might be necessary to create the unit with the line-orinted editor, YALOE (see section 8.5); the regular editor may be unusable.

Here is an example of a complete GOTOXY unit for the Zenith Z-19 terminal:

```
{$R-,I-  No checking needed (makes GOTOXY faster & smaller)}
Unit Goto_XY_U  {for Zenith Z-19};

Interface

  Procedure My_Goto_XY (X, Y : Integer);

Implementation

  Uses Progops;

  Procedure My_Goto_XY {X, Y : Integer};
  Const Esc = 27;
        Out_Unit = 21;
  Var Buf : Packed Array [0..3] Of Char;
  Begin
    If Y > 23 Then Y := 23;
    If X > 79 Then X := 79;
    If Y < 0 Then Y := 0;
    If X < 0 Then X := 0;
    Buf[0] := Chr (Esc);
    Buf[1] := 'Y';
    Buf[2] := Chr (Y + 32);
    Buf[3] := Chr (X + 32);
    If Prog_Redir (True) Then
      Write (Buf)
    Else
      Unitwrite (Out_Unit, Buf, 4);
  End {of My_Goto_XY};

End {of Goto_XY_U}.
```

This example demonstrates most of the requirements and restrictions imposed on new GOTOXY units. Most terminals use similar character sequences for cursor addressing; the parameters most likely to vary are the prefix and command characters, and the biases applied to the X and Y coordinates. These should be documented in the terminal's functional specification.

The name "GOTOXY" cannot be used as an identifier; it is reserved for standard calls to the GOTOXY intrinsic. The unit must be named GOTOXYU. It should be the only unit in the GOTOXYU code file. The procedure implementing the GOTOXY intrinsic should be the only procedure in the GOTOXYU interface section. It should have exactly two integer parameters. The first parameter is the zero-based horizontal coordinate, and the second parameter is the zero-based vertical coordinate. The procedure must ensure that both coordinates are in the proper range for the target terminal; if not, they must be truncated. The unit may be validated by embedding it in a program (as an in-line unit) and calling it with test coordinates.

The GOTOXY intrinsic must execute as quickly as possible. Thus, only the simplest and fastest operations should be used in order to calculate and output the cursor positioning sequence (i.e. long integer and real operations should not be used, and procedure calls should be kept to a minimum). It is recommended that the "R-" and "I-" compiler directives be used in order to minimize execution checks and code size. Use of the UNITWRITE intrinsic to output the cursor positioning sequence incurs less system overhead than using the standard WRITE procedure, thus reducing I/O time. I/O time can be reduced further by sending the cursor positioning sequence to the Fastcon I/O device (unit 21), which locks out all tasks contending for CPU time until the I/O is complete. Compiler directives and the UNITWRITE intrinsic are described in the Programmer's Manual.

Use of the UNITWRITE intrinsic with the Fastcon device is not compatible with I/O redirection options (section 2.4.4); output operations performed in this manner affect only the system console rather than the output streams and t-files designated by redirection options. The Prog_Redir function (documented in the Library User's Manual) may be used to determine whether I/O redirection is in effect. If it is, cursor positioning sequences should be written to the standard output using the WRITE procedure.

8.3.0.0 Using Binder

Binder is invoked using the S(ubmit command (chapter 6) instead of the X(ecute command. The S(ubmit command accepts the name of a command file and a list of parameters, separated by spaces; null parameters are specified by two contiguous spaces or an end of line. The Binder is executed by S(ubmitting the Binder command file with three optional parameters. The first parameter is the name of the code file containing the GOTOXYU unit. If this parameter is not specified, the Binder prompts for the name of the code file:

    Enter name of file with GOTOXY procedure:

The second parameter is the volume identifier of the volume containing the system support library (SYSTEM.PASCAL) to which the GOTOXYU unit is to be bound. The third parameter is the volume identifier of the volume containing a copy of the Library utility (section 8.2.0). The default value of the second and third parameters is the name of the system volume.

Examples of Binder invocations are:

```
Binder
Binder MyGotoXY
Binder MyGotoXY Testvol: Utility:
Binder MyGotoXY  Utility:
```

Binder X(ecutes the Library utility and accesses the code file containing the GOTOXYU unit. It K(eeps the first unit in the unit display (presumably the GOTOXYU unit) and then accesses the system support library. Binder K(eeps all system support units except the first, which is assumed to be the old GOTOXYU unit. The Library utility is then terminated and the binding is complete. The system should be rebooted immediately.

NOTE - A newly bound in GOTOXY is correct if the welcome message appears in the center of the screen when the new system is booted (section 2.4.0) and the editor seems to work correctly.

8.3.1 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.

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" on the system volume. 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 prompt line.

E(XIT exits Setup.

NOTE - The Setup utility may be used to modify system configuration information. The ASS utility (section 8.3.2) should be used to modify editor configuration information.

8.3.1.0 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 are character sequences initiated at the console keyboard to indicate a request for a particular predefined action. 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).

Most character and key sequences may be prefixed by a "lead-in prefix" The terminal's functional specification should be consulted to determine the character sequences required and emitted by the terminal. Configurations for some common terminals are listed in Appendix F.

NOTE - The ASCII character names used in some fields are defined in Appendix E.

BACKSPACE CHAR

Writing this character to the console moves the cursor one space to the left. Suggested value: ASCII BS

BACKSPACE KEY

This key moves the cursor one space to the left. It should not be prefixed. Default value: ASCII BS

CHAIN QUEUE SIZE

This field sets the maximum number of programs which can be chained together using the CHAIN intrinsic (see Library User's Manual). The default value is 1 and may be changed to any number between 0 and 10. Note that each chain queue entry (even if unused) occupies 40 words of memory.

CURSOR DOWN KEY

This key and the corresponding UP, LEFT and RIGHT 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 assigned the control codes that correspond to them.

CURSOR HOME CHAR

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 8.4) instead.

CURSOR LEFT KEY

This key and the corresponding UP, DOWN and RIGHT 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 assigned the control codes that correspond to them.

CURSOR RIGHT CHAR

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 8.4) instead.

CURSOR RIGHT KEY

This key and the corresponding UP, DOWN and LEFT 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 assigned the control codes that correspond to them.

CURSOR UP CHAR

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 8.4) instead.

CURSOR UP KEY

This key and the corresponding DOWN, LEFT and RIGHT 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 assigned the control codes that correspond to them.

DELETE CHARACTER KEY

This key deletes the character where the cursor is, and moves the cursor one character to the left. Suggested value: ASCII BS

DELETE LINE KEY

This key deletes the line occupied by the cursor. Suggested value: ASCII DEL

EDITOR ACCEPT KEY

This key is used in the editor to conclude commands and save the text changes. Suggested value: ASCII ETX or LF

EDITOR ESCAPE KEY

This key is used in the editor to exit from commands. Suggested value: ASCII ESC

NOTE - If any keys are prefixed, the EDITOR ESCAPE KEY should also be designated as prefixed; otherwise, ambiguities result.

END FILE KEY

This key sets the Boolean intrinsic EOF to true when it is typed while reading from the predeclared file INPUT. Suggested value: ASCII ETX

ERASE LINE CHAR

Writing this character to the console erases all characters on the line containing the cursor, and positions the cursor at the beginning 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.

ERROR LIST LENGTH

This field indicates the number of lines of execution error information to print when an execution error occurs. If the value is 0 or 1, the site of the execution error is reported. If the value is greater than 1, the remaining lines report the locations of the procedure calls leading up to the error. The default is 1 and may be changed to any number between 0 and 255. Section 7.8 of the Programmer's Manual describes how to use this information.

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.

LEAD-IN FROM KEYBOARD

Some terminals provide 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.

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 MOVE DELAY

This field accepts 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.

8.3.2 Using Advanced System Setup

This section describes the utility program Advanced System Setup (ASS), which is used to define the mapping of terminal keys to Advanced System Editor commands. ASS stores ASE command definitions in the SYSTEM.MISCINFO file.

X(ecute ASS. The first prompt is:

Press <return> to start or "!" to abort...

Typing "!" exits ASS. Typing <return> commences ASS.

NOTE - ASS aborts at this point if the file SYSTEM.MISCINFO does not reside on the prefixed disk.

The next prompt is:

Defaults from S(etup, previous A(dvSysSetup, or N(one?

This prompt determines whether the command definitions in the current SYSTEM.MISCINFO file are to be carried over to the new miscinfo file.

Typing "A" preserves all ASE command definitions in the current miscinfo. (Note that this option is applicable only to SYSTEM.MISCINFO's previously created by ASS.)

Typing "S" specifies that the following commands are to be copied from the system configuration information portion of SYSTEM.MISCINFO: <left>, <right>, <up>, <down>, <bs>, <etx>, and <escape>. These command definitions are used by the system utilities and are assigned by the Setup utility (see section 8.3.1). Any pre-existing ASE command definitions are lost.

Typing "N" directs ASS to ignore all command definitions in the current SYSTEM.MISCINFO. Commands not specified by ASS's defaults need to be explicitly defined in ASS.

ASS clears the screen and prompts:

Do you want to change the gap? (y/n)

The current gap setting appears above the prompt preceded by some instructions for determining the gap. The gap influences the size of the edit buffer in memory. A large gap (i.e. 30000) yields a small edit buffer; a small gap (i.e. 20000) yields a larger edit buffer. The size of the edit buffer determines the speed at which the editor operates. Various gap values may be tested in order to obtain optimal editor speed and edit buffer size.

If a gap change is desired, type "y" and enter an integer for the new specification. If the integer is negative, ASE will ask for a gap value during initialization, which is useful for tuning the gap.

ASS then displays a table of all defined ASE commands at the top of the screen, followed by a command prompt. The command table and prompt line are described in section 8.3.2.1. Commands are described in sections 8.3.2.1.0 through 8.3.2.1.5.

8.3.2.0 Character Prompts

Character prompts in ASS accept either a single (printing or nonprinting) character, or a character representation. Character representations allow a character to be specified by its numeric value; they have the following form:

<CharRep> ::= (<integer>)

... where <integer> is a decimal integer between 0 and 255. For instance, the character representation "(65)" denotes the character whose integer value is 65 (ASCII "A"). When ASS displays a character definition on the screen, it prints either the character itself or the corresponding character representation (depending on whether the character is printing or nonprinting). The ASCII table in Appendix E is useful in interpreting character representations.

8.3.2.1 The Command Table and Prompt

The table contains definitions for four classes of commands:

1) Default definitions for alphabetic editor commands (e.g. "I" for the I(nsert command).

2) Default definitions for some other commands (e.g. <dir-change>, <space>, and <return>).

3) (Optional) Definitions for cursor-moving commands obtained from the original SYSTEM.MISCINFO.

4) (Optional) ASE command definitions obtained from the original SYSTEM.MISCINFO.

Here is a portion of the command table:

```
c. S    Set         .. /    Slash       .. (32) Space
```

Each command entry has the following form:

<CmdEntry> ::= <case><prefixed> <value> <fieldname>

<case> ::= . | c

<prefixed> ::= . | p

The case field indicates whether the command is case-insensitive

(i.e. whether lower and upper case responses are equivalent); "c" indicates case-insensitivity, "." denotes case-sensitivity. This field may only be set on commands possessing alphabetic character values.

The prefixed field indicates whether the character value shown is the latter part of a 2-character sequence involving a prefix character; "p" indicates a prefixed character sequence, "." denotes a non-prefixed sequence. (Note that the actual prefix character is not shown in the command entry.)

The character value of the command definition is indicated by <value>. The value appears as either a printing character or a character representation.

The editor command is indicated by <fieldname>. The field name is used in ASS commands to specify an editor command definition for modification.

Information pertaining to the table is displayed below the command entries. The characters used as prefix characters by commands in the table are shown, along with the number of empty slots left in the table for new command entries. The empty slots are used for defining any commands that remain undefined, and also for adding multiple command definitions.

NOTE - Several different key sequences may be defined to represent a given command. For example, the "Digit" fields allow alternate definitions of digits, possibly including sequences beginning with a prefix. This enables repeat factors to be applied to commands invoked within the eX(change command.

The command prompt is displayed next:

```
A(dd D(elete K(ey-surrogate L(ook P(rompted-add Q(uit ?:
```

Typing "?" displays a short description of the prompt line commands.

8.3.2.1.0 A(dd

A(dd is used to add command definition entries. The following prompt appears:

```
Command? (?<ret> shows table)
```

Typing only <return> aborts the A(dd command. Typing "?<ret>" displays the field names of all editor commands (defined or undefined). Commands are specified by typing in their field names followed by <return>. Field names are case-insensitive, and do not have to be completely typed in; the first one or two characters of a field name are sufficient for specifying a command. Typing an invalid field name causes the prompt to be redisplayed.

When a valid field name is entered, the next prompt appears:

    Prefix? (e.g. "(13)")

If a prefix character is not desired, type in "(0)"; this value denotes the lack of a prefix (see the K(ey-surrogate command for an easier way of specifying this value). Prefix characters must be nonprinting; otherwise, the command is aborted. Also, prefix characters must be unique with respect to all other nonprefixed command characters.

The next prompt is:

    Character? (e.g. "F" or "(13)")

Any character may be typed in. If the character is alphabetic, the following prompt appears:

    Case insensitive? (y/n)

Typing "y" indicates the command definition is case-insensitive; typing "n" denotes a case-sensitive definition.

At this point, the command definition is completely specified. If the new definition is equivalent to an existing command definition, it is discarded, and the following message appears:

    Conflicts with <fieldname>

8.3.2.1.1 D(elete

D(elete redisplays the command table and adds the following prompt:

    --- u,d,l,r to move; z zaps, a accepts, ! escapes ---

Typing "u", "d", "l", or "r" moves the cursor up, down, left, or right across the command table respectively; the cursor is always located at the front of a command entry. Typing "z" removes the entry under the cursor. Typing "a" completes D(elete, removing all of the zapped command entries. Typing "!" exits D(elete without removing any zapped entries.

8.3.2.1.2 K(ey-surrogate

K(ey-surrogate is used to simplify the entry of characters into character prompts. A terminal key may be redefined in ASS to generate an arbitrary character value which may be used for the remainder of the current ASS execution. The most common use of this command is for defining a single-key alternate for the character representation "(0)"; this speeds up the entry of nonprefixed command definitions.

When K(ey-surrogate is invoked, it first displays a list of all existing key surrogate definitions, and then prompts:

    Surrogate (ch - "!" exits)

Enter the alternate character. The next prompt is:

    Character

The character (representation) to be redefined is entered.

8.3.2.1.3 L(ook

L(ook redisplays the command table.

8.3.2.1.4 P(rompted-add

P(rompted-add cycles through the commands that remain undefined; for each of these, the following prompt appears:

    Add command <fieldname>? (y/n/!)

Typing "n" skips the current command. Typing "!" exits the P(rompted-add command. Typing "y" generates a prompt sequence which is equivalent to the A(dd command's prompt sequence (section 8.3.2.1.2).

NOTE - There is one character prompt which is unique. It is called InsertLine, and is used to define the _output sequence_ which will cause your terminal to insert a blank line pushing down all following lines. It is used by the ASE for upward scrolling. If it is left undefined because it is not supported by the terminal's hardware, upscroll will not occur and full screen refresh will be used in its stead.

8.3.2.1.5 Q(uit

Q(uit asks if a listing of the key definitions (suitable for documentation) is desired. If so, enter the name of the list file (with any required ".TEXT" suffix). If not, just type <return> without entering a filename.

Q(uit then writes the new terminal configuration to the NEW.MISCINFO file on the prefixed disk. The new commands are recognized by the ASE after NEW.MISCINFO is C(hanged to SYSTEM.MISCINFO (using the filer).

8.4 System I/O Configuration

The Drvr.Info utility is used to modify the I/O system configuration. It establishes the correspondence between a physical unit number and an I/O device. A device is specified as a system I/O driver and a logical device number (see section 2.3.1 for details). Up to 64 physical units may be allocated.

8.4.0 Using Drvr.Info

X(ecute Drvr.Info. The following terminal display will appear:

```
     Read from which file ? *System.Drvinfo______________

     Unit#  Driver  LU    Unit#  Driver  LU    Sys Drivers
                                               A) SYSDRIVE
                                    <list of current drivers>
```

Typing <return> displays the current system I/O configuration and the prompt:

```
  DrvInfo: R(ead,A(ctivate,D(eactivate,N(ame,W(rite,P(rint,E(dit,
                                                        Q(uit [1.0]
```

Drvr.Info loads an I/O configuration into the display buffer with the R(ead command. Changes to the configuration are performed using the A(ctivate, D(eactivate, N(ame, and E(dit commands. The current display may either be written to a file using W(rite or printed in hard copy form using P(rint.

8.4.1 Editing Drvr.Info Prompts

All Drvr.Info prompts display a series of '____' which determine the field size of the requested response. The field may be modified using a protocol similar to that of the eX(change mode in the system editor; entering a printable character causes the character under the cursor to be replaced by the character entered. Utility keys used to alter the fields are:

Key	Action
<bs>	moves cursor left one space
<left>	moves cursor left one space
<del>	moves cursor to beginning of field
<right>	moves cursor right one space
<tab>	moves cursor to end of field
<return>	accepts all input up to cursor position
<etx>	accepts all input in field

Certain commands allow a range of table entries, indexed by letters, to be affected. The response to such a command may be either a single entry index (e.g. A), a closed entry range (e.g. A-C), or an open entry range (e.g. -C means from the beginning to C, B- means from B to the end, - means from the beginning to the end). If the response contains a '?', the operation is verified

for each entry of the range before it is carried out.

8.4.2 R(ead

R(ead is used to display a configuration contained in a Drvinfo file. When Drvr.Info is executed, an automatic R(ead is performed. The prompt is:

Read from which file ? *System.Drvinfo______________

For the initial R(ead, typing <del> <return> aborts the program; for any other R(ead, it exits the prompt. *System.Drvinfo is the default input file. The prompt may be edited to specify another file. Once the file name is accepted, the new configuration is read into the display.

8.4.3 W(rite

W(rite allows the contents of the display to be written to a file. This prompt appears:

Write to what file ? *System.Drvinfo______________

The output file defaults to the input file. Typing <del> <return> exits the prompt. The prompt may be edited to specify another file.

P(rint generates a hard copy of the display. P(rint generates the prompt:

Print to what file ? Printer:____________________

The default printfile is Printer:. The prompt may be edited to specify another file. Typing <del> <return> exits the prompt.

8.4.4 Altering I/O Configurations

The display may be altered by using the A(ctivate, D(eactivate, N(ame and E(dit commands.

8.4.4.0 Activation and Deactivation

A unit is not actually visible to the system unless it is A(ctivated (denoted by a "*" to the left of the Unit#). D(eactivate renders units invisible to the system.

8.4.4.1 N(ame

N(ame allows limited changes to the display. N(ame may create entries, and alter the driver and/or logical device number of current entries. See E(dit for more extensive modifications. N(ame generates the prompt:

Which entry (<first entry> - <last entry>) ? ______

Current entries are altered by selecting the index (or subrange of indices) of the desired entry. Any response in the selected range (except <return> which exits the prompt) generates the prompt:

Which system driver name (<first sys drvr>-<last sys drvr>)?

A driver may be chosen from the list of current system I/O drivers at the right of the display. A new driver may be specified by choosing the empty entry. This generates the prompt:

New driver name ? SYSDRIVE

SYSDRIVE is the default. Any other name may be entered by editing the prompt.

The chosen driver name will appear beside each selected index. The driver name field may be edited and must be accepted before the logical device field may be edited.

8.4.4.2 E(dit

E(dit is used for extensive alterations to the I/O system configuration. This command creates a copy of the display in a temporary file and invokes the editor. E(dit generates the prompt:

Temporary file for edit ? Drvr.Temp___________________

The temporary file name may be edited to specify another file name. After the file name is accepted, the editor is invoked. Modifications to the display may be performed using Editor commands (see Chapter 4). When editing is complete, the Editor is exited using the Q(uit U(pdate editor commands. Then the temporary file is automatically read back into the configuration display. Invalid entries are ignored.

8.4.5 Q(uit

Q(uit exits Drvr.Info. If an attempt is made to exit Drvr.Info without saving the current configuration, the following prompt appears:

Nothing written: W(rite, R(eturn, E(xit

E(xit terminates the program, R(eturn returns to Drvr.Info, and W(rite behaves as described in section 8.4.3 and then exits the program.

8.5 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 8.3); YALOE works in these situations, while the regular editor does not.

Section 8.5.8 contains a summary of all YALOE commands.

8.5.0 Entering YALOE

YALOE is invoked by eX(ecuting YALOE; 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.EDIT. Typing 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.

8.5.1 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.

8.5.1.0 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.

8.5.1.1 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.

8.5.1.2 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.

8.5.2 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.

8.5.3 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.

8.5.4 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.

8.5.5 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.

8.5.5.0 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.

8.5.5.1 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.

8.5.5.2 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.

8.5.5.3 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 with 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.

8.5.5.4 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.

8.5.5.5 E(rase

Format:

E

Erases the screen; this command only works with video display terminals.

8.5.5.6 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.

8.5.6 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 8.5.6.4.

8.5.6.0 J(ump

Format:

nJ

Moves the cursor a specified number of characters in the text buffer.

8.5.6.1 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.

8.5.6.2 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".

8.5.6.3 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.

8.5.6.4 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
- - - - - - - - - - - - - - - - - - - - -
```

*BGcome$=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
- - - - - - - - - - - - - - - - - - - - -
```

8.5.7 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 8.5.7.5.

8.5.7.0 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.

8.5.7.1 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.

8.5.7.2 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.

8.5.7.3 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.

8.5.7.4 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.

8.5.7.5 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.
*OCAAA$$	Replaces the characters from the beginning of the line to the cursor with "AAA" (same as *OXAAA$$).
*BGA$=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$GTWINE$=D$$	Moves the cursor to the beginning of the text buffer, searches for the string "TWINE", and deletes it.

8.5.8 Other Commands

Miscellaneous commands include: S(ave, U(nsave, M(acro, N (macro execution), and "?".

8.5.8.0 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.

8.5.8.1 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".

8.5.8.2 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.

8.5.8.3 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.

8.5.8.4 ? (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.

8.5.9 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 O 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>.

8.6 Byte-level File Editor

The Patch utility is used to view and alter the contents of a disk file. Patch operates in either Edit mode or Dump mode. In Edit mode, files are addressed as a series of 512-byte blocks; the contents of each block may be displayed on the console either in hex format or as a mixture of hex and ASCII characters. The contents of a displayed block may be modified by moving the cursor to the desired position, entering the new data, and writing the modified block back to disk. In Dump mode, Patch creates a byte-level hard copy from either memory or a specified input file. The hard copy output may appear in any of several formats: decimal, hexadecimal, ASCII characters (if printable), decimal bytes and octal bytes. 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.

8.6.0 Using Patch

X(ecute Patch. Patch is in Edit mode when first entered. The following prompt line appears:

```
EDIT: G(et, R(ead, S(ave, T(ype, F(or, B(ack, M(ode,
                                       V(iew, Q(uit,? [1.0]
```

Typing '?' displays the remaining commands:

```
EDIT: D(ump, I(nfo, ?
```

Typing '?' again returns to the original prompt line.

8.6.1 Edit Mode

The Edit commands are G(et, R(ead, F(or, B(ack, M(ode, V(iew, T(ype, I(nfo, S(ave, and Q(uit. Edit commands operate only on the current block shown in the display buffer. The D(ump command enters Dump mode.

8.6.1.0 G(et

G(et allows the specification of an input file for Patch to examine. It generates the prompt:

    FILENAME: <c/r 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 occupied 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:

    Unitnumber: 0 .. 255 ([RET] quits)

Typing <return> exits the prompt. Type the number corresponding to the unit containing the volume to be examined. 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). Block zero is automatically read into the display buffer. Block numbers are irrelevant when accessing serial units.

8.6.1.1 R(ead

R(ead is used to load a specified block from the current input file into the display buffer. R(ead generates the prompt:

    Blocknumber:

Enter a nonegative block number.

8.6.1.2 F(or and B(ack

F(orward and B(ackward change the current block in the display buffer. F(orward displays the next block from the file, and B(ackward displays the preceding block.

8.6.1.3 M(ode

M(ode toggles the display format between hexadecimal digits and characters if printable (hex otherwise).

8.6.1.4 V(iew

V(iew refreshes the current display buffer.

8.6.1.5 T(ype

T(ype allows alteration of the display buffer. T(ype generates the promptline:

```
TYPE: C(har, H(ex, F(ill, U(p, D(own, L(eft, R(ight,
                              <vector arrows>, Q(uit
```

U(p, D(own, L(eft and R(ight are vector keys which change the cursor position. U(p moves the cursor up one row; D(own moves the cursor down one row; L(eft moves the cursor left one column; R(ight moves the cursor right one column. <Vector arrows> are the keys used in the screen editor to move the cursor; they also work in T(ype.

C(har, H(ex and F(ill are used to alter the display buffer. All alterations begin at the current cursor position. The cursor should be moved to the desired change site before C(har, H(ex or F(ill are invoked.

C(har is used to enter printable characters into the display buffer. The following prompt appears:

```
CHARACTERS: <printable characters>, <ETX> quits
```

Starting at the current cursor position, type ASCII characters (only printable characters are accepted) to exchange them with the current contents of the display buffer. Type <ETX> to exit C(har.

H(ex is used to enter hexadecimal digits into the display buffer. The following prompt appears:

```
HEXADECIMAL : 0 .. 9, A .. F, a .. f, <ETX> quits
```

Starting at the current cursor position, type hex digits to exchange them with the current contents of the display buffer. Type <ETX> to exit H(ex. Only hex digits (upper or lower case) are accepted.

F(ill is used to set a series of bytes to the same value. The following prompt appears:

```
Number of bytes:
```

Type <return> to exit F(ill. Otherwise, enter a number between 0 and 511 (depending on the current cursor position). The next prompt is:

```
What Pattern :
Valid Format : C<printable char> or H<hexdigit> <hexdigit>
```

<Return> exits F(ill. A valid format consists of a prefix denoting whether the format is char or hex, and then the actual pattern which will fill the bytes (i.e. 'C' or 'c' followed by one character or 'H' or 'h' followed by two hexadecimal digits).

Q(uit exits T(ype.

8.6.1.6 I(nfo

I(nfo displays the current status of Edit mode, giving the following information:

File: <input file>
Length: <# of blocks in file>
Current: <current block in display buffer>
Byte 0: <byte sex of machine>
Open: <is there an input file? true/false>
Unit I/O: <reading from a unit? true/false>
Unitnumber: <what unit? (-1 if none)>

8.6.1.7 S(ave

Any alterations performed using T(ype affect only the display buffer. S(ave writes the contents of the buffer to the current file. If an altered block is not S(aved, the alterations are lost.

8.6.1.8 Q(uit

Q(uit exits Patch.

8.6.2 Dump Mode

Dump mode is used to generate hard copies of given input files in selected radix formats. Dump mode is entered from Edit mode by using the D(ump command. When Dump mode is entered, the following prompt appears:

```
Dump: D(o, Q(uit
```

D(o performs the dump. Q(uit exits Dump mode.

Following the prompt line is the selection menu. To alter a particular item, type the letter preceding that item. <Return> exits any prompt without changing the current value except for prompts with boolean responses. A 'T' or 'F' response must be entered to exit these prompts.

8.6.2.0 I/O Selection

Dumping is allowed from either memory or an input file. The input file selection appears as follows:

```
A) Input file
B) Starting Block    0
C) Number of Blocks  1
```

'A' generates the prompt:

```
FILENAME : <c/r for Unit I/O>
```

Enter the name of the file to be dumped. Patch expects complete file names; suffixes are required.

Typing <return> generates this prompt:

```
Unitnumber: 0 ..  255 ( [RET] quits )
```

Typing <return> exits the prompt. Type the number corresponding to the volume whose blocks are to be dumped.

'B' prompts for the starting block in the input file. 'C' prompts for the number of blocks to be dumped.

The memory dump selection is as follows:

```
E) Read from Memory  False
F) Starting Word     0
G) Number of Bytes   0
```

'E' allows a dump from memory. If 'E' is true, the dump will be from memory, overriding any input file entered in 'A'. 'F' prompts for the starting word in memory. Signed integers are displayed when the address exceeds 32767. 'G' prompts for a non-negative number of bytes to be dumped from memory.

The output file selection is as follows:

H) Output File
I) Width in Words 15

'H' generates the prompt:

Filename:

Enter the name of the file to be dumped on.

'I' prompts for the size of the output line in 8 character fields. Fifteen generates a 132 column output; eight generates an 80 column output.

8.6.2.1 Radix Format Selection

The dump may contain any or all of the radix formats displayed below. The format selection appears as follows:

		Flip	Both
J) Decimal	False	False	False
K) Hexadecimal	False	False	False
L) Characters	False	False	False
M) Octal	False	False	False
N) Decimal Bytes	False	False	False
O) Octal Bytes	False	False	False

Choose the letter preceding the desired format. The first column indicates whether or not that radix is to be displayed. The last two columns determine display format. 'True' in the second column means that the bytes for that radix will be flipped before being dumped. 'True' in the last column yields an output of both flipped and nonflipped bytes, thus overriding the value in the second column.

A further choice of format is provided for inter-line spacing. The choice of output spacing is as follows:

S) Space between Lines False
T) Space between Groups False

'S' prompts for single spacing between lines. 'T' prompts for spacing between output groups (i.e. blocks or 512 byte sections).

8.7 Printer Spooler

The Printer utility starts the printer spooler, which writes text files to an I/O device concurrently with normal system operation. The spooler may be configured to allow 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 prompt line). Printer is described in section 8.7.0.

8.7.0 Using Printer

X(ecute Printer. The following prompt appears:

What is the output unit (1,<online units>)?

... where <online units> is a list of unit numbers for all online serial output units (1 is the console unit). Typing <return> exits Printer; typing a 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 to be printed without page breaks; otherwise, all files are paginated (at 60 lines per page followed by 6 blank lines).

Printer allows the specification of file names by wildcard. A wildcard may appear an arbitrary number of times anywhere in the file name. The wildcards are:

```
=            match any string
?            match any single character
{a-m}        match any single character 'a' through 'm'
{a-m,  g-j, \{-\}, z}
             match any single character 'a' through 'm',
               not including 'g' through 'j', and
               including '{' through '}' and z
```

The ".TEXT" suffix is appended automatically unless a '.' appears at the end of the file name (which is stripped).

Up to ten files may be queued for printing; the file name prompt reappears after each file name is entered. Typing <return> indicates that no more files are to be queued for printing; Printer then terminates and begins to print the files.

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.

The Spooler unit, SpoolUnit, is bound into the Printer program. The Spooler is activated when the Printer program is X(ecuted and terminated when the Printer program is complete. With this configuration, no other programs may be executed until the Spooler has finished printing all files.

A copy of the Spooler unit is also provided in System.Library. By using the Library program to transfer SpoolUnit into the intrinsics library and rebooting, the Spooler may print files independently of program execution. This allows concurrent printing and system use (see section 2.2.4.1 for details).

8.8 Calculator

The Calc utility simulates a desktop calculator.

8.8.0 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>
```

8.9 Bootstrap Creation

The Make.Boot utility allows users to bootstrap the AOS on a user-supplied device. The Make.Boot utility creates an AOS bootstrap by combining the bootstrap core and serial driver with a user-supplied system device driver (see section 6.2 of the Programmer's Manual for details). A copy of the new bootstrap may be placed on the new device, or it may placed in a disk file which may be subsequently transferred to the bootstrap device using the Booter utility described in section 8.0.0.

The bootstrap resides on track 0 of the bootstrap device. The system monitor (chapter 7) must be equipped with drivers capable of reading track 0 of the bootstrap device in order to bootstrap the AOS. If this is not already the case, the new device may be bootstrapped by writing the new bootstrap on track 0 of some device already accessible to the system monitor (i.e. a floppy) and attempting to bootstrap the floppy. The system monitor reads and executes the new bootstrap, which then proceeds to boot from the new device. The factory may be consulted regarding the production of system monitor proms capable of booting directly from the new device.

Logical unit 0 of the bootstrap device must contain a Pascal floppy with the SYSTEM.PASCAL, SYSTEM.MISCINFO, SYSTEM.INTRINS, SYSTEM.SHELL, SYSTEM.DRVINFO, and SYSTEM.DRIVERS files. The SYSTEM.DRIVERS and SYSTEM.DRVINFO files must be configured to access the new device (see section 2.3.1).

8.9.0 Using Make.Boot

X(ecute Make.Boot. The following prompt appears:

```
What is the name of the bootstrap file ?
```

Enter the name of the file containing the bootstrap core (BOOT.CODE on the AOS release disk); typing <return> aborts the program. If the bootstrap file is found, the following prompt appears:

```
What file is unit SERIALDR in ?
```

Enter the name of the file containing the bootstrap serial driver (BOOT.CODE on the AOS release disk); typing <return> aborts the program. If the file is found, the following prompt appears:

```
What is the actual unit name ?
```

Typing <return> specifies the default bootstrap serial driver name, SERIALDR; otherwise a new bootstrap serial driver name may be entered. If the bootstrap serial driver is found in the specified driver code file, the following prompt appears:

```
What file is unit FLOPPYDR in ?
```

Enter the name of the file containing the user-supplied system

device driver; typing <return> aborts the program. If the file is found, the following prompt appears:

What is the actual unit name ?

The name of the user-supplied system device driver may be entered; typing <return> specifies the default bootstrap disk driver name, FLOPPYDR. If the disk driver unit is found in the specified driver code file, the following prompt appears:

The booter contains 3 segments and occupies <size> words.

Unit to write (0 for file) ?

Enter the physical unit number of the bootstrap device; 0 indicates that the bootstrap should be written to a data file. If a physical unit number is entered, the following prompt appears:

Drive number ?

Enter the drive number used by the device's hardware controller to access the bootstrap device.

If the bootstrap is to be written to a data file instead, the following prompt appears:

Output file name ?

Enter the name of the data file to which the bootstrap is to be written; <return> aborts the program.

The Make.Boot utility attempts to write the bootstrap to the designated destination. If it is unsuccessful, it prints an error message and aborts; otherwise, it terminates normally.

NOTE - If the new bootstrap occupies more than 1664 words, a diagnostic message is printed and Make.Boot aborts before the bootstrap is written. Since the bootstrap core uses only the device initialization and read routines, routines not related to these functions need not be compiled into the version of the device driver used by the bootstrap. The conditional compilation mechanisms presented in the Programmer's Manual are suggested as a way of commenting these sections out.

APPENDIX A: STANDARD 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: STANDARD 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
14	Illegal heap operation

APPENDIX C: STANDARD I/O UNIT ASSIGNMENTS

This section describes the devices normally assigned to the system's physical unit numbers. The mapping between unit numbers and operating devices may be changed by the user. In addition, unallocated unit numbers may be assigned to new devices. See section 2.3.1 for details. 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
0	System Clock
1	Console port (echo)
2	Console port (no echo)
3	Keyboard type-ahead buffer
4	Floppy Drive 0
5	Floppy Drive 1
6	LPV-11 (FFA0 hex) parallel printer
7	DLV-11J (FFB8 hex) Port 3 Input
8	DLV-11J (FFB8 hex) Port 3 Output
9	Hard Disk Drive 0
10	Hard Disk Drive 1
11	Hard Disk Drive 2
12	Hard Disk Drive 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 (FFB8 hex) Port 3 Input
20	DLV-11J (FFB8 hex) Port 3 Output
21	Fast Console Port Ouput
22	Standard Input
23	Standard Output
24	Bit Bucket
25	Hard Disk Drive 4
26	Hard Disk Drive 5
27	Hard Disk Drive 6
28	Hard Disk Drive 7
29	Hard Disk Drive 8

NOTE - Hex numbers displayed with I/O device names indicate the memory address used to communicate with the device.

APPENDIX D: COMPILER SYNTAX ERRORS

1: Error in simple type
2: Identifier expected
3: 'PROGRAM' expected
4: ')' expected
5: ':' expected
6: Illegal symbol (maybe missing ';' on the line above)
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: Unsatisified 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 procedure 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 solution
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: Substition 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
173: Fortran procedure or function expected!
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
196: 'USES LONGINT' required

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
407: disk error
408: compiler error

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	ENQ	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 F: TERMINAL CONFIGURATIONS

The terminal configuration for a given terminal consists of the screen and keyboard definitions used by the system utilities and the Advanced System Editor. The terminal configuration used by the system utilities is specified using the Setup utility documented in section 8.3.1. The terminal configuration used by the Advanced System Editor is specified using the ASS utility documented in section 8.3.2. The SYSTEM.STARTUP program on the AOS release disk attempts to create a work disk containing the correct terminal configuration for the system terminal. The procedures described above should only be necessary if the system terminal is one for which no configuration has been provided.

Terminal configurations for the LSI ADM-3A, Soroc IQ-120, Zenith Z-19, and the DEC VT-100 are presented below.

NOTE - In the Advanced System Editor configuration listings, control character values are parenthesized. The ASCII chart given in Appendix E is useful in interpreting these values.

APPENDIX F1: ADM 3-A TERMINAL

System Terminal Configuration

Fields	Values
BACKSPACE CHAR	Cntrl-H
BACKSPACE CHAR PREFIXED	False
BACKSPACE KEY	BS (Cntrl-H)
BACKSPACE KEY PREFIXED	False
CURSOR DOWN KEY	Down-arrow
CURSOR DOWN KEY PREFIXED	False
CURSOR HOME CHAR	Cntrl-^
CURSOR HOME CHAR PREFIXED	False
CURSOR LEFT KEY	Left-arrow (Cntrl-H)
CURSOR LEFT KEY PREFIXED	False
CURSOR RIGHT CHAR	Cntrl-L
CURSOR RIGHT CHAR PREFIXED	False
CURSOR RIGHT KEY	Right-arrow (Cntrl-L)
CURSOR RIGHT KEY PREFIXED	False
CURSOR UP CHAR	Cntrl-K
CURSOR UP CHAR PREFIXED	False
CURSOR UP KEY	Up-arrow (Cntrl-K)
CURSOR UP KEY PREFIXED	False
DELETE CHAR KEY	Left-arrow (Cntrl-H)
DELETE CHAR KEY PREFIXED	False
DELETE LINE KEY	DEL
DELETE LINE KEY PREFIXED	False
EDITOR ACCEPT KEY	Cntrl-C
EDITOR ACCEPT KEY PREFIXED	False
EDITOR ESCAPE KEY	ESC
EDITOR ESCAPE KEY PREFIXED	False
END FILE KEY	Cntrl-C
END FILE KEY PREFIXED	False
ERASE LINE CHAR	NUL
ERASE LINE CHAR PREFIXED	False
ERASE SCREEN CHAR	Cntrl-Z
ERASE SCREEN CHAR PREFIXED	False
ERASE TO END OF LINE CHAR	NUL
ERASE TO END OF LINE CHAR PREFIXED	False
ERASE TO END OF SCREEN CHAR	NUL
ERASE TO END OF SCREEN CHAR PREFIXED	False
LOWER CASE	True
RANDOM CURSOR ADDRESSING	True
SLOW TERMINAL	False
LEAD IN FROM KEYBOARD	NUL
LEAD IN TO SCREEN	NUL
NON-PRINTING CHAR	"?"
NON-PRINTING CHAR PREFIXED	False
SCREEN HEIGHT	24
SCREEN WIDTH	80
VERTICAL MOVE DELAY	5

Advanced System Editor Terminal Configuration

Nonprefixed Keys

Functions	Keys	Functions	Keys
"<"	,	"<"	<
">"	.	">"	>
"?"	?	Columnl	(5)
Del	(127)	Adjust	A and a
BackSpace	(8)	Beginline	(2)
Beginline	B and b	Copy	C and c
Delete	D and d	DeleteCh	(15)
Digit	9	Digit	8
Digit	7	Digit	6
Digit	5	Digit	4
Digit	3	Digit	2
Digit	1	Digit	0
Down-arrow	(10)	Edit	E and e
Equal	=	Etx	(3)
Home	(30)	InsertCh	(1)
Right-arrow	(12)	Up-arrow	(11)
eXchange	X and x	Find	F and f
GetAgain	G	Getch	(7)
Getch	g	Insert	I and i
Jump	J and j	Kolumn	K and k
LineEnd	L and l	Margin	M and m
Next	N and n	OppositePage	O and o
Page	P and p	Quit	Q and q
Replace	R and r	Return	(13)
Set	S and s	Slash	/
Space	(32)	Tab	(9)
ToDisk	T and t	UpTop	(21)
UpTop	U and u	Verify	V and v
WordMove	(23)	WordMove	W and w
Zap	Z and z		

Prefixed Keys

Functions	Keys	Functions	Keys
Delete	(27) D and d	DirChange	(27) H and h
Escape	(27) (27)	Insert	(27) I and i
Left-arrow	(27) L and l	OppositePage	(27) O and o
RecordKey	(27) R and r	Takeup	(27) T and t
Find	(27) F and f	GetAgain	(27) G
Getch	(27) g	f1	(27) 1
f2	(27) 2	f3	(27) 3
f4	(27) 4	f5	(27) 5
f6	(27) 6	f7	(27) 7
f8	(27) 8		

The specified Gap is: 30000

APPENDIX F2: SOROC IQ-120 TERMINAL

System Terminal Configuration

Fields	Values
BACKSPACE CHAR	Cntrl-H
BACKSPACE CHAR PREFIXED	False
BACKSPACE KEY	Left-arrow (Cntrl-H)
BACKSPACE KEY PREFIXED	False
CURSOR DOWN KEY	Down-arrow (Cntrl-J)
CURSOR DOWN KEY PREFIXED	False
CURSOR HOME CHAR	Cntrl-^
CURSOR HOME CHAR PREFIXED	False
CURSOR LEFT KEY	Left-arrow (Cntrl-H)
CURSOR LEFT KEY PREFIXED	False
CURSOR RIGHT CHAR	Cntrl-L
CURSOR RIGHT CHAR PREFIXED	False
CURSOR RIGHT KEY	Right-arrow (Cntrl-L)
CURSOR RIGHT KEY PREFIXED	False
CURSOR UP CHAR	Cntrl-K
CURSOR UP CHAR PREFIXED	False
CURSOR UP KEY	Up-arrow (Cntrl-K)
CURSOR UP KEY PREFIXED	False
DELETE CHAR KEY	Left-arrow (Cntrl-H)
DELETE CHAR KEY PREFIXED	False
DELETE LINE KEY	RUB (DEL)
DELETE LINE KEY PREFIXED	False
EDITOR ACCEPT KEY	Cntrl-C
EDITOR ACCEPT KEY PREFIXED	False
EDITOR ESCAPE KEY	ESC
EDITOR ESCAPE KEY PREFIXED	False
END FILE KEY	Cntrl-C
END FILE KEY PREFIXED	False
ERASE LINE CHAR	NUL
ERASE LINE CHAR PREFIXED	True
ERASE SCREEN CHAR	"*"
ERASE SCREEN CHAR PREFIXED	True
ERASE TO END OF LINE CHAR	T
ERASE TO END OF LINE CHAR PREFIXED	True
ERASE TO END OF SCREEN CHAR	Y
ERASE TO END OF SCREEN CHAR PREFIXED	True
LOWER CASE	True
RANDOM CURSOR ADDRESSING	True
SLOW TERMINAL	False
LEAD IN FROM KEYBOARD	NUL
LEAD IN TO SCREEN	ESC
NON-PRINTING CHAR	"?"
NON-PRINTING CHAR PREFIXED	False
SCREEN HEIGHT	24
SCREEN WIDTH	80
VERTICAL MOVE DELAY	10

Advanced System Editor Terminal Configuration

Nonprefixed Keys

Functions	Keys	Functions	Keys
"<"	,	"<"	<
">"	.	">"	>
"?"	?	Columnl	(5)
Del	(127)	Adjust	A and a
BackSpace	(8)	Beginline	(2)
Beginline	B and b	Copy	C and c
Delete	D and d	DeleteCh	(15)
Digit	9	Digit	8
Digit	7	Digit	6
Digit	5	Digit	4
Digit	3	Digit	2
Digit	1	Digit	0
Down-arrow	(10)	Edit	E and e
Equal	=	Etx	(3)
Home	(30)	InsertCh	(1)
Right-arrow	(12)	Up-arrow	(11)
eXchange	X and x	Find	F and f
GetAgain	G	Getch	(7)
Getch	g	Insert	I and i
Jump	J and j	Kolumn	K and k
LineEnd	L and l	Margin	M and m
Next	N and n	OppositePage	O and o
Page	P and p	Quit	Q and q
Replace	R and r	Return	(13)
Set	S and s	Slash	/
Space	(32)	Tab	(9)
ToDisk	T and t	UpTop	(21)
UpTop	U and u	Verify	V and v
WordMove	(23)	WordMove	W and w
Zap	Z and z		

Prefixed Keys

Functions	Keys	Functions	Keys
Delete	(27) D and d	DirChange	(27) H and h
Escape	(27) (27)	Insert	(27) I and i
Left-arrow	(27) L and l	OppositePage	(27) C and o
RecordKey	(27) R and r	Takeup	(27) T and t
Find	(27) F and f	GetAgain	(27) G
Getch	(27) g	f1	(27) 1
f2	(27) 2	f3	(27) 3
f4	(27) 4	f5	(27) 5
f6	(27) 6	f7	(27) 7
f8	(27) 8		

The specified Gap is: 30000

APPENDIX F3: ZENITH Z-19

System Terminal Configuration

Fields	Values
BACKSPACE CHAR	Cntrl-H
BACKSPACE CHAR PREFIXED	False
BACKSPACE KEY	Backspace (Cntrl-H)
BACKSPACE KEY PREFIXED	False
CURSOR DOWN KEY	Down-arrow (B)
CURSOR DOWN KEY PREFIXED	True
CURSOR HOME CHAR	H
CURSOR HOME CHAR PREFIXED	True
CURSOR LEFT KEY	Left-arrow (D)
CURSOR LEFT KEY PREFIXED	True
CURSOR RIGHT CHAR	C
CURSOR RIGHT CHAR PREFIXED	True
CURSOR RIGHT KEY	C
CURSOR RIGHT KEY PREFIXED	True
CURSOR UP CHAR	A
CURSOR UP CHAR PREFIXED	True
CURSOR UP KEY	A
CURSOR UP KEY PREFIXED	True
DELETE CHAR KEY	Backspace (Cntrl-H)
DELETE CHAR KEY PREFIXED	False
DELETE LINE KEY	Delete (DEL)
DELETE LINE KEY PREFIXED	False
EDITOR ACCEPT KEY	Linefeed (Cntrl-J)
EDITOR ACCEPT KEY PREFIXED	False
EDITOR ESCAPE KEY	ESC
EDITOR ESCAPE KEY PREFIXED	False
END FILE KEY	Cntrl-C
END FILE KEY PREFIXED	False
ERASE LINE CHAR	l
ERASE LINE CHAR PREFIXED	True
ERASE SCREEN CHAR	E
ERASE SCREEN CHAR PREFIXED	True
ERASE TO END OF LINE CHAR	K
ERASE TO END OF LINE CHAR PREFIXED	True
ERASE TO END OF SCREEN CHAR	J
ERASE TO END OF SCREEN CHAR PREFIXED	True
LOWER CASE	True
RANDOM CURSOR ADDRESSING	True
SLOW TERMINAL	False
LEAD IN FROM KEYBOARD	ESC
LEAD IN TO SCREEN	ESC
NON-PRINTING CHAR	"?"
NON-PRINTING CHAR PREFIXED	False
SCREEN HEIGHT	24
SCREEN WIDTH	80
VERTICAL MOVE DELAY	0

Advanced System Editor Terminal Configuration

Nonprefixed Keys

Functions	Keys	Functions	Keys
"<"	,	"<"	<
">"	.	">"	>
"?"	?	Del	(127)
Adjust	A and a	BackSpace	(8)
Beginline	(2)	Beginline	B and b
Copy	C and c	Delete	D and d
Digit	9	Digit	8
Digit	7	Digit	6
Digit	5	Digit	4
Digit	3	Digit	2
Digit	1	Digit	0
Down-arrow	(26)	Edit	E and e
Equal	=	Etx	(10)
eXchange	X and x	Find	F and f
GetAgain	G	Getch	(7)
Getch	g	Insert	I and i
Jump	J and j	Kolumn	K and k
LineEnd	(12)	LineEnd	L and l
Margin	M and m	Next	N and n
OppositePage	O and o	Page	P and p
Quit	Q and q	Replace	R and r
Return	(13)	Set	S and s
Slash	/	Space	(32)
Tab	(9)	ToDisk	T and t
Up-arrow	(1)	UpTop	(21)
UpTop	U and u	Verify	V and v
WordMove	(23)	WordMove	W and w
Zap	Z and z		

Prefixed Keys

Functions	Keys	Functions	Keys
Column1	(27) 1	Delete	(27) M
DeleteCh	(27) N	DirChange	(27) h
Down-arrow	(27) B	Escape	(27) Q
OppositePage	(27) O and o	Find	(27) F and f
GetAgain	(27) G	Getch	(27) g
Home	(27) H	Insert	(27) L
InsertLine	(27) L	InsertCh	(27) @
Left-arrow	(27) D	RecordKey	(27) J
Right-arrow	(27) C	Takeup	(27) P
Up-arrow	(27) A	f1	(27) S
f2	(27) T	f3	(27) U
f4	(27) V	f5	(27) W
f6	(27) 6	f7	(27) 7
f8	(27) R		

The specified Gap is: 30000

APPENDIX F4: DEC VT-100

System Terminal Configuration

Fields	Values
BACKSPACE CHAR	Cntrl-H
BACKSPACE CHAR PREFIXED	False
BACKSPACE KEY	Backspace (Cntrl-H)
BACKSPACE KEY PREFIXED	False
CURSOR DOWN KEY	Down-arrow (B)
CURSOR DOWN KEY PREFIXED	True
CURSOR HOME CHAR	H
CURSOR HOME CHAR PREFIXED	True
CURSOR LEFT KEY	Left-arrow (D)
CURSOR LEFT KEY PREFIXED	True
CURSOR RIGHT CHAR	C
CURSOR RIGHT CHAR PREFIXED	True
CURSOR RIGHT KEY	C
CURSOR RIGHT KEY PREFIXED	True
CURSOR UP CHAR	A
CURSOR UP CHAR PREFIXED	True
CURSOR UP KEY	A
CURSOR UP KEY PREFIXED	True
DELETE CHAR KEY	Backspace (Cntrl-H)
DELETE CHAR KEY PREFIXED	False
DELETE LINE KEY	Delete (DEL)
DELETE LINE KEY PREFIXED	False
EDITOR ACCEPT KEY	Linefeed (Cntrl-J)
EDITOR ACCEPT KEY PREFIXED	False
EDITOR ESCAPE KEY	ESC
EDITOR ESCAPE KEY PREFIXED	False
END FILE KEY	Cntrl-C
END FILE KEY PREFIXED	False
ERASE LINE CHAR	NUL
ERASE LINE CHAR PREFIXED	False
ERASE SCREEN CHAR	NUL
ERASE SCREEN CHAR PREFIXED	False
ERASE TO END OF LINE CHAR	K
ERASE TO END OF LINE CHAR PREFIXED	True
ERASE TO END OF SCREEN CHAR	J
ERASE TO END OF SCREEN CHAR PREFIXED	True
LOWER CASE	True
RANDOM CURSOR ADDRESSING	True
SLOW TERMINAL	False
LEAD IN FROM KEYBOARD	ESC
LEAD IN TO SCREEN	ESC
NON-PRINTING CHAR	"?"
NON-PRINTING CHAR PREFIXED	False
SCREEN HEIGHT	24
SCREEN WIDTH	80
VERTICAL MOVE DELAY	0

Advanced System Editor Terminal Configuration

Nonprefixed Keys

Functions	Keys	Functions	Keys
"<"	,	"<"	<
">"	.	">"	>
"?"	?	Del	(127)
Adjust	A and a	BackSpace	(8)
Beginline	(2)	Beginline	B and b
Copy	C and c	Delete	D and d
Digit	9	Digit	8
Digit	7	Digit	6
Digit	5	Digit	4
Digit	3	Digit	2
Digit	1	Digit	0
Down-arrow	(26)	Edit	E and e
Equal	=	Etx	(10)
eXchange	X and x	Find	F and f
GetAgain	G	Getch	(7)
Getch	g	Insert	I and i
Jump	J and j	Kolumn	K and k
LineEnd	(12)	LineEnd	L and l
Margin	M and m	Next	N and n
OppositePage	O and o	Page	P and p
Quit	Q and q	Replace	R and r
Return	(13)	Set	S and s
Slash	/	Space	(32)
Tab	(9)	ToDisk	T and t
Up-arrow	(1)	UpTop	(21)
UpTop	U and u	Verify	V and v
WordMove	(23)	WordMove	W and w
Zap	Z and z		

Prefixed Keys

Functions	Keys	Functions	Keys
Columnl	(27) l	Delete	(27) d
DeleteCh	(27) S	DirChange	(27) c
Down-arrow	(27) B	Escape	(27) (27)
Insert	(27) I	InsertCh	(27) R
OppositePage	(27) O and o	Page	(27) p
RecordKey	(27) T and t	Takeup	(27) K and k
Find	(27) F and f	GetAgain	(27) G
Getch	(27) g	Home	(27) H
Left-arrow	(27) D	Right-arrow	(27) C
Up-arrow	(27) A	f1	(27) P
f2	(27) Q	f3	(27) 3
f4	(27) 4	f5	(27) 5
f6	(27) 6	f7	(27) 7
f8	(27) 8		

The specified Gap is: 30000

Appendices

Index

$CURSOR 115,116,119
$EQUAL 116,119
$EXEC.TEXT 29,194
$LAST 115,116,159
$LOG 117,126,159
$PROFILE 110,117
$SYNTAX 115,117
$TAG 117,150,163
.BACK 27
.BAD 27
.CODE 27
.TEXT 27
<accept> 6
<backspace> 6,135
<bs> 6
<Coll> 105,148,174
<del> 105,148
<DeleteCh> 174
<DirChange> 105,175
<down> 6,118,135
<eof> 6,89
<esc> 6,148
<escape> 6
<etx> 6
<GetAgain> 105,136,147
<home> 105,118,135
<InsertCh> 174
<left> 6,118,135
<record> 105,129,136,160
<return> 135
<right> 6,118,135
<space> 6,135
<tab> 135
<takeup> 105,128,130,136,140,168
<up> 6,118,135
A(djust 136,138
Accept Key 6
ALL.DRIVERS 41
Anchor 142
APPPROCS Unit 38
Architecture Guide 1
ASE! 124
ASS 27,42,104,232,243
Auto Buffer 112,155,167
Auto-indent 111,148,149,153,167
Available Memory 65
B(ad Blocks 74,75,80
B(eginLine 118,135,139
BACKSPACE CHAR 238
Backspace Key 6,238
Backup 89,209,211,213
Backup File 106,113,114
Backus-Naur Form 2
Bad Block 216,217
Bad Blocks 75

Bad Prompt ... 33
Bad.Blocks ... 75,209,217
Beginner's Guide ... 1
Binder ... 232,234,236
Block ... 18,26,29
Block Number ... 18
Block-structured Device ... 16,18
Block-structured Unit ... 16,19,21
Block-structured Volume ... 19,21
BNF ... 2
Booter ... 209,210,274
Bootstrap ... 210,274
Bootstrap Failure ... 47
Bootstrapping ... 36,42,43,45,204
Breakpoint ... 40,43
BUCKET: ... 54,55
Byte-level File Editor ... 264
Bytes-in-last-block ... 24,26
C(hange ... 74,76,237
C(ompile ... 60,188
C(opy ... 108,119,136,140
C(opy <specialkey> ... 130,140
C(opy B(uffer ... 140
C(opy F(rom file ... 141
Calc ... 272
CALL ... 196
Case Insensitive Mode ... 122,144,161
CHAIN QUEUE SIZE ... 238
Change Log ... 104,117,126,159
Change.Dir ... 209,223
Clear Screen ... 59
Code File ... 24,25
Command Argument ... 252
Command Character ... 149,154,167
Command File ... 193
Command File Interpreter ... 3,67,193
Command Mode ... 251
Command String ... 252
Compiler ... 3,28,35,48,49,50,60,68,115,187,235,283
Copy Buffer ... 120,142,148,176
Copydupdir ... 22,103,224
Cursor ... 106,112,135,142
CURSOR DOWN KEY ... 239
CURSOR HOME CHAR ... 239
CURSOR LEFT KEY ... 239
CURSOR RIGHT CHAR ... 239
CURSOR RIGHT KEY ... 239
CURSOR UP CHAR ... 239
CURSOR UP KEY ... 240
D(ate ... 74,77
D(elete ... 108,112,113,119,120,136,142
D(ISK UPDATE ... 237
Data File ... 24,25
Data Prompt ... 5
Date ... 126,128,164

Index

DEC Format 8,213
DELETE CHARACTER KEY 240
DELETE LINE KEY 240
Destination File 109
Device Driver 274
Direction 107,116,118,135,156,157
Disk Directory 21,22,103,223
Disk Drive 16
Disk File 21,22,24,71
Disk Swapping 13
Disk Type Key 8,45
Disk Unit 16,21
Disk Volume 21,71,92,93
Disk-to-Disk Transfers 92
Dismount 220
Double Density Floppy Disk 8,45
Double-sided Floppy Disk 8,45,97,204
Drive Configuration 219
Drive.Con 18,209,219
Drivers 41
Drivers Library 40,41,43,274
Drvr.Info 42
Dump 264
Duplicate Directory 22,103,224
Dynamic Variable Allocation 35,40,43
E(dit 61,108,114,137,143
E(XIT 237
E(xt-dir 74,78,100
Editor 3,26,37,42,48,49,61,190,240
EDITOR ACCEPT KEY 240
EDITOR ESCAPE KEY 240
END FILE KEY 240
End of File Key 6
Environment 106,110,164
EQU 196
Equals 116,118,135,140,142,144,148,161,
176
ERASE LINE CHAR 240
ERASE SCREEN 240
ERASE TO END OF LINE 241
ERASE TO END OF SCREEN 241
ERROR LIST LENGTH 241
Escape Key 6,10
eX(change 108,136,173
EXCEPTION 43
eXec 194
Execution Error 10,13,40,43,279
Execution Option List 53
Extended Precision Arithmetic 40,43
F(ile 62
F(ind 112,119,121,136,144,167,176
FASTCON: 235
File Attributes 24
File Buffer 106,111,114,155,169
File Date 24,26
File Designator 15,31,33

File Dump 264
File Identifier 15,21,31
File Length 24,26
File Menu 104,108,114
File Name 4,5,15,24
File Suffix 24,26,27,31,33
File System 4,14
File Title 24,27,29,31,33
File Type 24
File Window 106,111,114
Filer 3,31,62,71
Filling 111,148,149,153,167
Floating Point I/O 40,43
Flush Key 7
Format 98,209,215,217
Function Key 104,108,110,115,122,128,166
G(et 48,72,73,79
G(etch 136,146
Gap 243
General Prompt 33
Good Prompt 33
GOTO 196
GOTOXY 40,42,43,232,234
H(alt 63
H(ELP 237
HALTUNIT 43
Hard Disk 219
Hard Disks 18
Hardware User's Manual 1
HAS CLOCK 241
HAS LOWER CASE 241
HAS RANDOM CURSOR ADDRESSING 241
HAS SLOW TERMINAL 241
HDT 203
Heap 35,40,43
HEAPOPS 43
I(nitialize 64
I(nsert 108,113,119,120,136,148,154,167,
176
I/O Device 16,18
I/O Driver Units 41
I/O Error 10,12
I/O Redirection 51,57,235
I/O Result 12,277
I/O System Configuration 41
Include File 191,192
Input Flush Key 7
Input Prompt 33
Input Redirection Options 51,53
Input Stream 51
InsertLine 247
Interface Text 35
Intrinsics Library 35,36,38,40,41,43
J(ump 135,150
J(ump M(arker 150
K(olumn 136,151

K(runch 30,74,80
Key Command 6
Keyboard 16
L(dir 74,78,81
L(ineEnd 118,135,152
LEAD-IN FROM KEYBOARD 241
LEAD-IN TO SCREEN 242
Left Stack 111,150
Length Specifier 24,29,31,83,89,188,192
Libmap 25,35,100,227
Library 28,35,41,100,227,236
Library Options 52,53,56
Library Redirection Option 37
Library Search Path 35
Library System 35
Library User's Manual 1
Literal Mode 122,144,161,167
Log Entry 104,117,126,159
Logical Device Number 41
Logical Volume 19
LONGINTS 43
M(ake 74,83,100
M(argin 136,153,167
M(emory 65
M(EMORY UPDATE 237
M(unch 153
Make.Boot 274
Mapper 8,209,213
Margins 111,148,167
Markdupdir 22,224
Marker 116,150,163
Memory Dump 264
Meta-words 2
Monitor 63
Monitor Key 6,46,203
Mount 220
N(ew 48,73,84,87
N(ext 112,120,135,136,155
Nested Editing 104,124,159
NEW.MISCINFO 247
No Room On Vol 22,56,192
NON-PRINTING CHARACTER 242
NUMCOM Unit 38
O(ppositePage 135,156
Offline 18
Online 18
Operating System 3,25,35
Operating System Libraries 40
Output Flush Key 7
Output Prompt 33
Output Redirection Options 52,53,54,55
Output Stream 52,54,55
P(age 135,157
P(refix 22,74,85
Paging 111,155,167
Paint Mode 104,175

Paragraph 149,153,154,167
PASCALIO 43
Patch 25,100,264
PATTERNMATCH Unit 38
Physical Unit 16
Physical Unit Number 16,41
Prefix Option 85
Prefix Options 22,52,53,56
Prefixed Volume 20,22,31,51,52,56,85,94
PREFIXED[<field name>] 242
Primary Command 107
Printer 16,270
Printer Spooler 3,38,270
PROFILE.TEXT 29,42,46,50,194
PROGOPS Unit 36,38,41
Program Library 35,37,38
Program Listing 10,28,187,189
Program Name 53
Programmer's Manual 1
Prog_Redir 235
Prompt Conventions 33
Prompt Line 4,106,107
Prompts 4
Q(uit 73,86,137,158,237
Q(uit A(nother 159
Q(uit B(ackup 159
Q(uit C(hange 158
Q(uit E(xit 158
Q(uit R(eturn 158
Q(uit U(pdate 159
QUIET 197
R(emove 74,87
R(eplace 112,119,121,136,161,167,176
R(ETURN 237
R(un 48,66,188
READ 196
Repeat Factor 115,118,129,138,139,140,142,144,
146,152,156,157,161,172
Replacement String 121,161
Right Stack 111,150
RUN 197
S 197
S(ave 48,72,73,88
S(et 137,163
S(et D(eleteMarkers 164
S(et E(nvironment 121,138,164
S(et E(nvironment I(nfo 165
S(et E(nvironment S(etTabStops 118,165
S(et E(nvironment U(serkey 128,166
S(et M(arker 141,163
S(et T(ag 163
S(ubmit 236
SCCNTRL Unit 38
SCREEN HEIGHT 242
SCREEN WIDTH 242
Search String 121,144,161

Secondary Command 107
Segment 220
Serial Device 16,18,25
Serial Unit 16,19,39
Serial Volume 19,29
SET 196
Setup 27,42,232,237
Shell 42,43,64
Single Density Floppy Disk 8,45
Single-drive Transfers 91
Single-sided Floppy Disk 8,45,204
Skew 215
Source File 109
Space Key 6
Spooler 3,270
SPOOLER Unit 38,271
Stack Overflow 11,192
Standard Input 16,51,55
Standard Output 16,51,52,54,55,235
STANIN: 55
STANOUT: 54,56
Start Key 7
Starting Block 24,26
State Flow Diagram 45,49
STK 197
Stop Key 7
Subsystem Documents 1
Swapping Compile Option 192
Syntax Error 49,50,115,117,190,283
SYSCOM 232
SYSINFO Unit 38
System Configuration 40
System Customization 40
System File Title 27,29
System I/O Redirection 57
System Library 35,36,38
System Monitor 3,45,46,203,274
System Overlays 40,43
System Shell 42,43
System Support Library 40,43,236
System Utilities 38,42
System Volume 20,21,236
SYSTEM.COMPILER 27,28,44,60
SYSTEM.DRIVERS 27,28,40,44,47
SYSTEM.DRVINFO 27,28,41,43,44,47
SYSTEM.EDITOR 27,44,61,251
SYSTEM.FILER 27,44,62
SYSTEM.INTRINS 27,28,36,41,43,44,47
SYSTEM.LIBRARY 27,28,36,44
SYSTEM.LST.TEXT 27
SYSTEM.MISCINFO 27,28,42,44,232
SYSTEM.PASCAL 27,28,40,43,44,47
SYSTEM.SHELL 27,28,42,43,44,47
SYSTEM.STARTUP 27,28,42,46,50,64
SYSTEM.SWAPDISK 27,28,192
SYSTEM.SYNTAX 27,44

SYSTEM.WRK.CODE 27,28,60,87,188
SYSTEM.WRK.TEXT 27,28,87,114
T 195
T(EACH 237
T(oDisk 112,120,136,169
T(ransfer 74,89
T-File Options 52,53,55
T-Files 52,55,235
Target 197
Temporary File 81,87
Terminal Configuration 6,42
Text File 18,24,25,83
Text Form 110,130,140,168
Text Mode 251
Token Mode 122,144,161,167
TRANSCEND 43
Transcendental Functions 40,43
Type-ahead Buffer 7,16,193
Type-ahead Flush Key 7
Type-checking Prompt 33
U(pdate 48
U(ptop 135,146,147,170
U(ser Restart 68
UCSD Pascal 187
UCSD Pascal System 1,3
Unit 35,234
Unit Number 17,20,31
Units 192
User File Title 29
User Interface 42,43
User Library 35,37,51,52,56
USERLIB.TEXT 29,37
USES 35
Utility Program 3,209
V(erify 137,171
V(olumes 74,94
Vector Keys 6,118,135,138
VERBOSE 197
Version Number 4,35
VERTICAL MOVE DELAY 242
Volume Identifier 15,20,31
Volume Label 219
Volume Name 19,20,31
W(hat 73,95
W(ordMove 118,135,172
Welcome Message 42,236
Western Digital Format 8,213
Wildcard 60,71,72,76,78,81,83,87,89,93
Work File 28,45,48,50,60,66,71,79,84,88,
114,188
WRITE 195
WRITELN 195
X(amine 74,75,80,96,216,217
X(ecute 69
X.CODE 29,67,193
X.DEMO.TEXT 193

Index

YALOE 234,239,251
Yes/No Question 5,71
Z(ap 112,118,120,136,176
Z(ero 74,98,103

ADDENDA FOR THE SYSTEM USER'S MANUAL

Section 2.2.4 (pages 38-39)

The name of the spooler unit is SPOOLUNIT instead of SPOOLER.

Section 2.4.4.1 (page 54)
Section 2.4.4.3 (pages 55-56)

The editor does not write to either the pre-declared file OUTPUT or to the STANOUT: unit. Therefore, output and t-output options may not be used to manipulate the editor's output stream.

Section 2.4.4 (pages 54-56)

Instances of STANOUT: and STANIN: are ignored in the redirection and t-file lists.

Section 8.0.2.0 (page 213)

A floppy should be mounted in the source drive before the source unit prompt is answered. Likewise, floppy should be mounted in the destination drive before the destination unit prompt is answered.

Sections 8.0.5.1 (page 219),
8.2.0.0 (page 227),
8.4.1 (page 248)

The <up> key may be used to insert a space into the field at the current cursor position. The <down> key may be used to delete the character at the current cursor position.