*Personal Computer Computer Language Series*

# BEGINNER'S GUIDE

for the UCSD p-System™ Version IV.0

Written by Kenneth L. Bowles
Edited by Stan Stringfellow

First Edition (January 1982)

Changes are periodically made to the information herein; these changes will be incorporated in new editions of this publication.

Products are not stocked at the address below. Requests for copies of this product and for technical information about the system should be made to your authorized IBM Personal Computer Dealer.

A Product Comment Form is provided at the back of this publication. If this form has been removed, address comment to: IBM Corp., Personal Computer, P.O. Box 1328-C, Boca Raton, Florida 33432. IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligations whatever.

© Copyright International Business Machines Corporation 1982
© Copyright Kenneth L. Bowles 1980
© Copyright SofTech Microsystems, Inc. 1981

UCSD, UCSD Pascal, and UCSD p-System are all trademarks of the Regents of the University of California.

This book is a Revised Edition of BEGINNER'S GUIDE TO UCSD Pascal by Kenneth L. Bowles.

# CONTENTS


CHAPTER 1. AN OVERVIEW ............ 1-1
- Who ............................ 1-3
- What ............................ 1-3
- Pascal ............................ 1-4
- How to Use this Book ............... 1-6

CHAPTER 2. ORIENTATION FOR BEGINNERS ............................ 2-1
- Goals for this Chapter ............... 2-3
- Getting Started ....................... 2-3
- Simple Commands ................... 2-5
- Special Keyboard Characters .......... 2-9
  - ENTER ......................... 2-9
  - CONTROL ..................... 2-9
- The Concept of a Program............. 2-11
- Building Bigger Programs out of Smaller Programs ................... 2-14
- Commands that Ask for Data ......... 2-16
  - I(nsert ......................... 2-17
  - D(elete ......................... 2-18
  - P(osition ....................... 2-19
- Summary of the Data World Example ... 2-19

CHAPTER 3. ORIENTATION FOR EXPERIENCED PROGRAMMERS ...... 3-1
- Goals for this Chapter ............... 3-3
  - Brief Overview .................. 3-4
  - The Workfile ................... 3-6
  - Running the Edited Program ..... 3-7
  - Saving Workfiles for Future Use ... 3-9
- Entering and Testing a Simple Program ........................... 3-9
- The I(nsert Command ............... 3-11
- The D(elete Command ............... 3-15
- Q(uit and U(pdate your Workfile ...... 3-16

R(un your Program .................. 3-17
Coping with a Compile-time "Syntax" Error ................ 3-18
Saving your Workfile in the Disk Directory ................. 3-19
Suppressing Execution of the Maze at Bootload Time .................... 3-25

CHAPTER 4. SCREEN EDITOR ........ 4-1
Goals for this Chapter ............... 4-3
Editor Overview ..................... 4-4
Cursor Movement Commands ......... 4-7
Arrow Commands and Their Relatives ...................... 4-9
Repeated execution of an arrow command ..................... 4-10
Moving the cursor off the screen ... 4-10
Using SPACE, BACKSPACE, and ENTER ............................ 4-14
The TAB key ........................ 4-15
Multiple occurrences of the target ..... 4-23
F(inding backwards .................. 4-24
L(iteral Targets vs Tokens ........... 4-25
Commands that Change the Workfile's contents ................ 4-29
The Q(uit command and its options ... 4-46
Using the Editor for Word Processing ... 4-48

CHAPTER 5. FILE MANAGER (FILER) ... 5-1
Goals for this Chapter ............... 5-3
Overview of Files and the Filer ........ 5-4
Volume Identifiers ................... 5-5
Simplified Titles for disk files ..... 5-6
Naming Conventions to Simplify Work with Groups of Files ...... 5-7
Workfile Commands ................. 5-9
Status checking/setting commands .... 5-17
Shorthand entry of the destination file name ........................ 5-27

Disk to Disk bulk T(ransfer ........... 5-28
T(ransferring only selected files ... 5-29
Rearranging the files on one disk ...... 5-30
Directory Maintenance Commands .... 5-32
Checking for disk errors and repairing them ...................... 5-42

CHAPTER 6. PASCAL COMPILER-SYNTAX ERRORS ...................... 6-1
Goals for this Chapter ................ 6-3
Preliminaries ........................ 6-4
Comments and Compiler Directives ... 6-5
Include Directive .................... 6-7
The Compiler's CRT Display and the List Directive .................. 6-8
Miscellaneous Compiler "Switch" Directives ........................... 6-13
I/O Check Switch .................... 6-13
Quiet Compilation Switch ......... 6-14
Syntax Errors ....................... 6-14
Unmatched BEGIN .. END pairs ...... 6-17
Comment Not Completed with a Closing "*)" Symbol ................ 6-19
Nested IF Statements ................ 6-20
Execution or Run-Time Errors ........ 6-27

CHAPTER 7. PROGRAMMING TO USE DISK FILES ............................ 7-1
Goals for this Chapter ............... 7-3
Overview ............................ 7-4
Physical Description of UCSD Pascal Disk Files .......................... 7-7
Sector Interleaving ............... 7-8
512-Byte Blocks - Universal Units of Disk Transfer ............... 7-9
Structured Logical Records ....... 7-11
Text Files ....................... 7-13
Working with Structured Data Files ... 7-16
File Declarations and the Buffer (Window) Variable ............. 7-19

Sample Program - Sequential File-to-File Copying .................. 7-32
Random Access Handling of Disk Files... 7-35
Sample Program UPDATE ....... 7-35
Indexed Access - Efficiency Considerations .................. 7-41
Text Files ............................ 7-44
READ and WRITE .................... 7-45
EOLN, READLN, WRITELN: End-Of-Line ......................... 7-47
Efficiency Considerations ......... 7-51
Error Recovery ....................... 7-52

**CHAPTER 8. USING LIBRARIES OF SPECIALIZED ROUTINES (UNITS)** .... 8-1
Goals for this Chapter ................ 8-3
The Reason for Having Preprogrammed Units .............................. 8-3
Overview of Units .................... 8-5
A Sample Unit and its Use ............ 8-8
The Librarian ....................... 8-13

**APPENDIX A. SPECIAL KEYS** ............ A-1

**APPENDIX B. SCREEN EDITOR COMMANDS** ......................... B-1

**APPENDIX C. FILER COMMANDS** ....... C-1

**APPENDIX D. OPERATING SYSTEM COMMANDS** ......................... D-1

**APPENDIX E. COMPILER SYNTAX ERROR MESSAGES** ..................... E-1

**APPENDIX F. EXECUTION ERROR MESSAGES** ............................. F-1

**APPENDIX G. I/O ERROR MESSAGES**.... G-1

# CHAPTER 1. AN OVERVIEW



## Contents

Who ........................................ 1-3
What ........................................ 1-4
Pascal ........................................ 1-5
How to Use this Book ........................ 1-7

# NOTES

# Who

This book is intended to be used as an introduction and reference manual for people just beginning to use the UCSD p-System. The book is designed to be used by at least the following three groups of people:

1) College students, high school students, and others who have never used a computer before;

2) Experienced programmers who have not used UCSD Pascal, particularly those who have been using BASIC and those not yet familiar with interactive CRT based programming systems;

3) Non-programmers who intend to use packaged programs designed to run within the UCSD p-System.

Some portions of the book are designed for use by only one or two of these groups, and can readily be scanned or ignored by others.

Our intent is to make it possible, and relatively easy, for you to learn to use the UCSD p-System by working just with the book and an IBM Personal Computer. You may find it useful to obtain assistance from someone already familiar with the p-System, but that help should not be necessary.

# What

If you are a beginner, and do not recognize the terms used in this section, you may wish to skim over the rest of this overview chapter and jump directly to Chapter 2.

The UCSD p-System is a complete general purpose software package for users of the IBM Personal Computer. The p-System is designed to make it easy

to develop and use programs. Though designed for program development, the UCSD p-System can also be used for many special purpose applications. Examples include Word Processing, Computer Assisted Instruction (CAI), interactive business data processing, communications, process control, and scientific analysis. While designed primarily for use with programs written in the Pascal programming language, the UCSD p-System also allows work with FORTRAN.

This book concentrates on features of the UCSD p-System intended for all users, especially beginners and students. A full user-oriented description of the p-System, including its many advanced features, is given in the *User's Guide for the UCSD p-System* which is intended for experienced programmers. The *PASCAL Reference for the UCSD p-System* by Clark and Koehler describes the Pascal language in detail. The *FORTRAN-77 Reference for the UCSD p-System* describes the FORTRAN language. The *Assembler Reference for the UCSD p-System* describes the Assembler and the 8086/87 assembly language. The *Internal Architecture Guide* discusses details of the internal workings of the p-System.

# Pascal

Pascal is a powerful general purpose programming language designed by Professor Niklaus Wirth of the Technical University in Zurich, Switzerland. The language is named in honor of Blaise Pascal, a famous mathematician. However, mathematics is by no means the only field in which the Pascal language is found to be useful.

The "standard" Pascal language, consisting of Wirth's definition published in 1971 and a few corrections made since then, was originally introduced to help in teaching a systematic approach to good program design. You may have heard of a

method known as "Structured Programming", with which professional programmers are able to write large complex programs in a manner that avoids most of the errors that plague programming work in older languages like BASIC, COBOL or FORTRAN. Among the practical and usable programming languages currently in widespread use, Pascal is the best statement of what structured programming is all about.

Pascal is coming into widespread use for writing complex programs in almost all fields where computers are applied to practical problems. For these applications, the standard Pascal language is often extended to provide specialized facilities not present in Wirth's original definition.

Virtually all of the UCSD p-System is programmed using a slightly extended version of the Pascal language that I will call "UCSD Pascal" in this book. UCSD Pascal includes almost all of the standard language. The extensions beyond the standard language in UCSD Pascal have been included to facilitate teaching using non-mathematics-oriented problem examples, and to facilitate writing a variety of large interactive programs.

UCSD Pascal is relatively easy for beginners to learn, as proven by the thousands of students who have completed an introductory problem solving and computer programming course at UCSD. It is probably true that beginners can learn to write very small programs slightly faster using BASIC than they can using Pascal. As soon as the beginner reaches the stage of needing more than a very few "GO TO" statements, learning to solve the same problem using Pascal becomes easier. Thereafter, the larger the program, the greater will be the advantage in using Pascal instead of BASIC. Most people who are familiar with both Pascal and BASIC agree that the extra effort to learn Pascal is easily saved as soon as one tries to write a program larger than "toy" size.

# How to Use this Book

This book is designed to be used both as an orientation guide for people first learning to use the UCSD p-System, and as a reference manual for the same people once they are familiarized with the p-System. As a reference manual, this book contains enough information to assist in a wide variety of advanced applications of the p-System. However, advanced users with a serious interest in the p-System should supplement this book with the detailed reference manuals for the p-System.

If you have not used computers before, or if your experience is with an "old" system such as those using punched cards for input, you probably should start reading this book in Chapter 2, "Orientation for Beginners". If you have written programs before using some language other than Pascal, and if you have used an interactive computer facility, then you probably can start in Chapter 3, "Orientation for Experienced Programmers". To avoid any more duplication of text than necessary, Chapter 3 is also intended to be read by beginners who have already gone through Chapter 2.

Whether you are a beginner or an experienced programmer, you will find this book easier to use if you have ready access to an IBM Personal Computer along with the STARTUP disk which contains several sample disk files designed specifically for use with this book. These include SYSTEM.STARTUP, and the text files EDITDEMO, and COMPDEMO.

Whether you are learning to program using Pascal, or just learning to use the p-System, you will then want to concentrate on the earlier portions of Chapters 4 and 5 on the Screen Editor and the File Manager respectively. These are the portions of the p-System with which you will be spending most of your time. The Screen Editor and the File Manager

are major tools to be used in making your use of the p-System easier. Both chapters are organized to present the most frequently used commands in an order designed to get beginners making practical use of the p-System as quickly as possible. Since this order does not lend itself to convenient reference use of the book, summaries of the commands are also given in Appendix A in alphabetic order.

Chapter 6 gives many helpful insights into the use of the Pascal Compiler. Chapter 7 describes how to write Pascal programs which use disk files under the UCSD p-System.

The UCSD p-System provides a method of augmenting the Pascal language, for specialized programming applications, using separately prepared libraries of routines. Chapter 8 describes how to use these library routines.

# NOTES

# CHAPTER 2. ORIENTATION FOR BEGINNERS

## Contents

Goals for this Chapter ................ 2-3
Getting Started ........................ 2-3
Simple Commands .................... 2-5
Special Keyboard Characters .......... 2-9
ENTER .............................. 2-9
CONTROL .......................... 2-9
Arrow Keys for Moving the Cursor .... 2-10
The Concept of a Program .......... 2-12
Building Bigger Programs out of Smaller Programs .................. 2-14
Commands that Ask for Data ......... 2-16
Summary of the Data World Example ... 2-19

NEW PROGRAMMER

# NOTES

# Goals for this Chapter

The following are the goals for this chapter:

a) Familiarize yourself with the IBM Personal Computer, with its keyboard, and with the method of inserting your floppy disk to get the UCSD p-System started (called "bootloading").

b) Learn what it means to use a "command" directing the computer to do something.

c) Distinguish between single-character commands and commands that ask for data.

d) Distinguish between a series of commands and a "program", which is really just a series of commands stored in the computer's memory in such a way that those commands can be repeated upon request.

e) Learn how to translate the important abstract control commands used throughout this book into actions you need to take using your computer in order to invoke those commands.



# Getting Started

This chapter is designed to be read when you have an IBM Personal Computer next to you and can use the computer immediately to try out the steps described in this book. To do this exactly as described in the book, the STARTUP disk should be used.

If you are like most people when they first start to use a computer, you probably do not know what to expect at this point. If someone who has already used the UCSD p-System is available to help, a fifteen minute demonstration would be a good idea to cover the material of this chapter and the next.

Lacking someone to demonstrate, I will give you step by step instructions so that you can get familiarized with the p-System by yourself.

First, I assume that your computer has been turned on and is functioning correctly. To turn the computer on the system unit switch should be in the up position. Carefully insert your STARTUP: diskette into the left disk drive connected to your computer.

> **CAUTION:** The floppy disk can be damaged or ruined if not handled carefully! Dust picked up off a table, through one of the slots in the protective jacket, can often ruin a disk. The disk can also be ruined if you write on it with a ball point pen, if you fold the disk or jam it under the cover of a three-ring notebook, if you leave fingerprints on the disk itself by holding through one of the slots in the protective jacket, or if you toss it around like a Frisbee! All these precautions may suggest that you should have second thoughts about getting involved with computers in the first place. Actually, it takes just a little effort to take care of your diskettes. When you do, they can last without damage through years of frequent use.

Both the disk and the computer itself might be damaged if you insert the disk into the disk drive in the wrong direction! Out of the eight different ways in which the disk might be inserted into the drive, only one is correct. The protective envelope, within which the actual disk lives, is generally marked on one side with a printed label. The correct orientation of the disk is the one you get by holding the diskette with your thumb holding the printed label (or at least holding the same side of the diskette envelope). You hold the diskette with the labelled side nearest to you, i.e., so that the printing appears upside down to your eyes.

Once the diskette is safely in the disk drive, the IBM Personal Computer will boot automatically. If at any time you wish to re-boot, you may either turn the power off, wait five seconds and turn it on again, or you may hold down the Ctrl and ALT keys and type the DEL key on the number pad.

In the jargon of the computer industry, the process of getting the system started is called "bootstrap loading", or simply "bootloading". What happens, when you bootload, is that essential parts of the System are copied from the diskette into the active memory of the computer's central processor unit. If the bootloading process is successful, the result will be a displayed figure as in Figure 2-1.



What if you do not get this result? You can detect whether bootloading is proceeding correctly by listening to clicking noises from the disk drive. Also, the indicator light should go on while information is being transferred from the disk into the computer's memory. In most cases, you should experience no problem in bootloading the System for the first time. If something does go wrong, it is advisable to go back through the steps that got you to this point, to make sure that you have not forgotten something.

## Simple Commands

When you first unpack the STARTUP disk associated with this book, it will be arranged to produce the display shown in Figure 2-1 as the result of bootloading.

```
Maze: U(p D(own R(ight L(eft B(ack H(elp X(ecute C(lear Q(uit
      @@@@@@@@@@@@@@@@@@@@@@@@@@@
      @ @         @             @
      @ @@@@@@@@@ @ @ @@@@@@@@@ @
      @   @       @ @         @ @
      @ @ @ @@@@@#@@@@@ @ @@@@@ @
      @ @ @ @   @   @ @ @ @     @
      @ @@@ @ @ @@@ @ @@@ @ @ @@@
      @     @ @   @       @ @   @
      @ @@@@@ @ @@@@@@@@@@@@@ @@@
      @ @     @ @           @   @
      @ @@@@@ @ @ @@@@@@@@@ @ @ @
      @     @ @   @       @ @ @ @
      @ @@@@@ @@@@@ @@@@@ @ @ @@@
      @ @     @   @   @   @ @   @
      @ @@@ @@@@@ @ @@@@@@@ @@@ @
      @   @     @   @         @ @
      @@@ @@@@@ @ @ @ @@@@@@@ @ @
      @     @   @ @   @         @
      @@@@@@@@@ @@@@@@@@@@@@@@@@@
```

**Figure 2-1. Initial display for Maze exercise**

Now direct your attention to the character "#" displayed just 5 lines below the character "R" in "R(ight", which is on the top line of the screen. The display consisting of barriers made out of "@" characters is a maze. In the exercise associated with this display, you will go through the steps necessary to find the exit from the maze, starting at the point marked with the character "#". The object of the exercise is to give you concrete examples and practice with what we call a "command" in computer jargon.

Now press the "D" key on your keyboard exactly once. The result should be the appearance of the character "+" immediately below the "#" character. At the same time, the cursor position should move to mark the newly created "+". A single "D" character will also appear near the bottom left side of your screen.

If nothing at all happens on the screen in response to your pressing the "D" key, chances are that you have not pressed it hard enough! Note: If you press the "D" key again at this point (or if you inadvertently

pressed it more than once) the speaker on your computer will beep. If that did not happen, do not be afraid to press "D" again just to see what it sounds like. The beep is frequently used as a signal to warn you that you are attempting to use a command that does not make good sense just when you try to use it.

Next, press the "R" key twice. The cursor should move two boxes to the right, placing additional "+" characters on the screen as it goes. If you press "R" a third time, the speaker will beep again signifying that you are trying to bump into another wall. Press "D" once or twice, and the result should be to move the cursor down an equal number of places, again leaving "+" characters on the screen.

NEW PROGRAMMER

By this time, you can see that you "command" the computer to do something each time you press the "D" or "R" key. Note in the top line of the screen that there are command characters associated with all four directions in which the cursor can move within the maze. "D" stands for "Down", "R" stands for "Right", "L" for "Left", and "U" for Up. The top line is used as a handy reminder about the available command letters, and what they are supposed to do.

The "(" character used with each command word is a reminder that only one character needs to be pressed to initiate the associated command. On some computer systems, you need to type in the whole word to initiate a command.

Try your hand at finding your way out of the maze. The only way out is at the open box along the bottom line of the maze. Figure 2-2 shows the result of taking a shortcut, which is obviously wrong. Do not let the beeps of the speaker bother you. They simply tell you that you can not keep moving in the direction indicated by the command letter you have just used.

```
Maze: U(p D(own R(ight L(eft B(ack H(elp X(ecute C(lear Q(uit
      ███████████████████████████
      █ █         █             █
      █ █████████ █ █ █████████ █
      █   █+++++++█ █         █ █
      █ █ █+█████#█████ █ █████ █
      █ █ █+█   █   █ █ █ █     █
      █ ███+█ █ ███ █ ███ █ █ ███
      █+++++█ █   █       █ █   █
      █+█████ █ █████████████ ███
      █+█     █ █           █   █
      █+█████ █ █ █████████ █ █ █
      █+    █ █   █       █ █ █ █
      █+█████ █████ █████ █ █ ███
      █+█     █   █   █   █ █   █
      █+███ █████ █ ███████ ███ █
      █+++█     █   █         █ █
      ███+█████ █ █ █ ███████ █ █
      █  ++_█   █ █   █         █
      █████████ █████████████████

ULLLLLLDDDDLLLLDDDDDDDDRRDDR
```

**Figure 2-2. A bad way out of the maze**

Next, if you have not already done so, it would be a good time to see what the "B(ack" command does by pressing "B". If you have moved from the starting position of the "#", each press of "B" will remove one "+" from the screen, backing you up toward the starting point. You may already have noticed that none of the commands "U", "D", "R", or "L" will allow you to back over a "+" already placed on the screen.

# H(elp

Press "H" to see what the "Help" command does. The maze will disappear off the screen, and a list of brief explanations of the available commands will appear. You can return to the maze display by pressing the SPACEBAR key (the long thin key at the bottom of your keyboard - i.e. the one closest to you).

The "C" key, for the "C(lear" command restores the maze display to the condition it was in when you first bootloaded.

The "X(ecute" command is explained in Chapter 5.

If at this point you press the "Q" key for "Q(uit", it might be best to start over by bootloading again. I will explain what's going on in a later section.

# Special Keyboard Characters



The IBM Personal Computer keyboard has some characters designed to be used for special control purposes. I am not yet ready to show you all of the special control characters, but we can begin at this point with a few of the characters used for moving the cursor around on the screen.

## ENTER

As a starting exercise, go through the maze as in the previous section leaving at least a dozen or so "+" characters in the maze. As before, the "B" key causes you to back up toward the starting location each time it is pressed. Now press the ENTER key (marked as a bent arrow going down and turning left) and note what happens. We have arranged things for this exercise alone so that the ENTER key is an alternate method for invoking the "B(ack" command. The ENTER key is used commonly for several other purposes throughout the UCSD p-System. For now, I just want you to be familiar with the ENTER key itself.

## CONTROL

Now, notice that your keyboard has a key marked Ctrl. This key is similar in effect to the SHIFT keys, at least in that Ctrl changes the effect you get from pressing many of the keys on the keyboard. In the maze exercise, if you press the "M" key, the computer will simply beep at you signifying that it

has no corresponding command. Now, hold down the Ctrl key, then press "M" while still holding Ctrl down. Note that what happens is the same as what you get when you press "B" or ENTER. Explanation: Each key on the keyboard, when pressed, sends a unique coded message to the computer. If you hold down SHIFT, the message may change, as from lower case "a" to upper case "A". If you hold down Ctrl, the message changes so that each key showing a letter has some special meaning, generally a meaning that cannot be expressed by showing a single character on the screen. One can arrange the computer to interpret Ctrl+letter messages as calling for a command to be invoked, just as we have used simple letter commands in the maze exercise.

It is often confusing to remember the association between a Ctrl+letter combination and the command action it is intended to invoke. Therefore, the IBM Personal Computer keyboard provides a few specially labelled keys which send the same messages as the associated Ctrl+letter combinations. The keyboard has an ENTER key. For that reason, we do not have to remember that the same effect can be obtained using Ctrl+"M".

## Arrow Keys for Moving the Cursor

Now, go back through the maze exercise using the special CONTROL commands for UP, DOWN, RIGHT, and LEFT. These commands are associated with four special control keys marked with arrows pointing in the four directions. (The arrows are located on the number pad keys 2, 4, 6, and 8.)

A few words about context may help you to understand what we have been doing here. You may wonder why we need the special Ctrl+letter combinations at all if the command letters "U", "D",

"R", and "L" will work just as well. The answer is that we have arranged for those letter commands to work as described just within the maze exercise. In using the UCSD p-System, you will see that we go from the context or environment of one "world" to that of another quite frequently. A little later in this chapter, we will switch to another world in order to illustrate how you use commands that require data. (Thus far, the commands we have been using are all invoked just by pushing one key - or the equivalent Ctrl+letter combination.) Since there are only 26 letters in the English alphabet, there are not enough single letter commands to go around to cover all of the things we want to do in different worlds within the UCSD p-System. Even if there were enough letters (as in the Chinese language) you would not want to spend the time to memorize all the letter-command associations. The UCSD p-System has been designed to make use of some of the commonly available special control keys in order to simplify the use of the System as much as possible.



# The Concept of a Program

Computer people use the term "program" with several slightly different shades of meaning from a beginner's point of view, and we shall have to do so in this book. Basically, a program is a sequence of commands stored in the computer in such a way that each command in the sequence can be carried out automatically, i.e. with no help from a human to go from one command to the next. Generally the first command in the sequence is carried out first, then the next, and so on in the order the commands appear. Methods are available to alter the sequence of commands automatically under certain conditions. Discussion of those methods is best left until you get around to studying the Pascal language for writing programs.

In the Maze example, the sequence of command letters appears in the lower part of the screen in the order in which you type them, i.e. left to right. When more than one line is needed to hold a complete sequence, the command letters go from the right end of one line to the left end of the next, as in the presentation of English text. The Maze program can automatically carry out each command shown at the bottom of the screen, since it is also stored in the computer's memory. Once you have several "+" characters deposited in the maze, press the "X" key and wait to see what happens. The cursor first jumps back to its original position at the "#" character. Then the cursor follows the same route that you followed when you first put the series of "+" characters on the screen. The rate at which it does this is deliberately slowed down so that you can see the correspondence between the position of the cursor within the maze, and the command character marked in the command sequence at the same time.

The sequence of command characters at the bottom of the screen is a crude program. When you press "X", for "Execute", the program is executed. To "execute" a program is basically the same as to "run" the program (although the UCSD p-System, like many others, makes a fine distinction between "eX(ecute" and "R(un" as we shall see in the next chapter). Both terms are used to describe what happens when a sequence of stored commands is carried out automatically one by one. Generally, it is possible to cause a program to be executed as many times as one wishes without altering the program as stored in the computer's memory.

In general, the symbols that we use to represent each command are assigned arbitrarily and purely for convenience. If we spoke Spanish rather than English "RIGHT" would become "DERECHA", "DOWN" might become "BAJO", "UP" would be "ARRIBA", and "LEFT" would be "IZQUIERDA".

Thus it would be convenient to change the letter assignments which correspond to movement of the cursor in the maze example to different values. In fact the meaning of the letter "D" would change! Thus the command letters must be regarded simply as "codes" that are assigned to shorten the amount of information the computer must be given in order that a given command "action" should be carried out.

The "program" we have been considering here, in connection with the maze example, is of course a simplified analogy to the programs one finds on most computers. The computer's "hardware", which you can touch or pick up and carry around, generally understands command codes expressed as small numbers. The command actions called for by those codes are typically very simple in concept. Even the simplest of the popular microprocessors now in use has roughly 70 different commands, and their corresponding codes. A program that carries out any useful function usually consists of hundreds or many thousands of these simple commands.

Fortunately, most humans who use computers have no need to work directly with the numbered command codes. Instead, we write our programs in a form that looks much closer to a sequence of English language statements about what needs to be done. A translator program, called a "compiler", then converts the human readable form of the program into the coded sequence of commands that the computer hardware itself can understand. The form that most humans use today for writing programs is called a "higher level language". For example, the form we use is at a substantially higher level than simple coded commands. Pascal, BASIC, COBOL, and FORTRAN are all commonly used higher level languages.

In the UCSD p-System, a command that you tell the computer to carry out from the keyboard is usually expressed by pressing a single key. In Pascal and other higher level languages, a program more often consists of English words mixed with special characters which represent commands. The English words are used to make a program more readable than a tightly packed sequence of single character commands like the "program" displayed at the bottom of the screen in the maze example. When you are issuing commands to the computer from the keyboard, you are generally aware of the context since the result obtained from issuing each command is apparent immediately. Thus effort is saved by not requiring that whole words be typed into the computer to cause the execution of each command to be initiated. When you read a computer program on paper, or on the screen, many commands are lumped together without obvious and immediate connection with the actions they cause when executed. In this context, the readability is much more important than the immediacy afforded by the single letter encoding of the "interactive" commands. Interactive commands are those you use when you interact directly with the computer rather than waiting for a program to run.

# Building Bigger Programs out of Smaller Programs

You may have noticed that we used a single letter command "X" to call for the "program" of moves through the maze to be executed. In effect the maze example is a simulation of a very simple computer designed for a special purpose. It happens that the simulation is itself a program (written in the Pascal programming language) which is arranged to respond to the various command letters we have been describing in this chapter.

There is nothing to prevent us from deciding to assign a different command code letter to each of several different programs. Thus "X" might cause the execution of one sequence of commands taking us off toward the true exit from the maze. "Y" might be another program which goes off toward a dead end in the maze. "Z" might refer to yet another dead end program. In fact, each of the command letters assigned in the maze program actually calls for the execution of a small program designed to carry out a specific simple action.

Obviously, if we can build a program out of any sequence of command codes, and can give that program itself another unique code, it must be possible to build big programs out of small programs. In other words, we can create a set of special purpose commands by writing "low level" programs (i.e. simple ones) to carry out those commands. We can then create a "higher level" program (i.e. one that is larger, more capable, or more complex) by using a sequence of commands each of which calls for execution of one of the low level programs. We can then create an even higher level program by using a sequence of commands from the next lower level (and perhaps also from the lower levels within the same sequence). This point is one of the main study goals of Chapter 2 in the textbook *Microcomputer Problem Solving Using Pascal*, by K. L. Bowles, (Springer Verlag, New York NY, 1977).

# Commands that Ask for Data

Assuming that you are still working within the Maze example on the computer, now press the "Q" key for Quit. The result should be the display shown in Figure 2-3:

```
Data: I(nsert D(elete P(os R(ight L(eft C(lear Q(uit

ROW ROW YOUR BOAT
```

**Figure 2-3. Initial Display of Data Oriented Command Example.**

This example is designed to illustrate how to use commands that ask for data in the UCSD p-System. It also provides a simplified orientation to the use of built-in facilities for working with "Strings" of characters in the UCSD Pascal language.

The general idea of this example is that commands are provided which allow you to alter the phrase "ROW ROW YOUR BOAT" displayed in the middle of the screen. You can I(nsert additional characters wherever the arrow symbol displayed on the next lower line happens to be pointing. You can move the arrow left or right using "L" and "R" as in the Maze example. You can D(elete characters starting at the position where the arrow points by typing one "x" character for each character you want to be deleted from the displayed phrase. Notice that this is a completely different definition for the "D" command character compared with its use in the Maze example. There should be no confusion since we are now in the Data example's "world" rather than that of the Maze example.

Try using I(nsert to obtain the result shown in Figure 2-4.



```
Data: I(nsert D(elete P(os R(ight L(eft C(lear Q(uit

ROW ROW YOUR big old BOAT

Enter string to be inserted:big old
    Then press RETURN

    Note: Use BACKSPACE (BS) to erase characters
```

Figure 2-4. Displayed String with Additional Data Inserted.

When you press "I" for the I(nsert command, a message appears on the display screen asking that you type in the characters you want inserted. The cursor waits immediately following this "prompting" message. When you type characters they appear on the screen starting at that point. You can back over characters typed in error by using the BACKSPACE key (marked as a left arrow at the upper right of the keyboard). Once the characters typed in are equal to what you had in mind, you cause those characters to be transferred to the program controlling the I(nsert command by pressing the ENTER key. You should then notice that a copy of the characters you typed in has now appeared within the phrase displayed in mid-screen.

# D(elete

The sequence of actions you employ to get D(elete to take effect is very similar to that just described. In this case, the D(elete command asks that you type one "x" character for each character you want deleted in the displayed phrase. Again BACKSPACE can be used to erase excess characters from the screen. ENTER causes the D(elete command action to be completed. Figure 2-5 shows the appearance of the screen just before ENTER is typed to cause deletion of the string "YOUR" from the display. Try this same operation with your computer to observe what happens.

```
Data: I(nsert D(elete P(os R(ight L(eft C(lear Q(uit

ROW ROW YOUR big old BOAT
        ^
        XXXXX_

Type an X for each character to be deleted above
   Then press RETURN

   Note: Use BACKSPACE (BS) to erase characters
```

Figure 2-5. Display in D(elete Command Before Pressing ENTER.

# P(osition

The Data example also offers a command for finding the position of a short "pattern" string of characters within the main string displayed in mid-screen. Press "P" to see what happens. The computer will then ask for you to type in the string of characters you want to be found. As an example, type "BOA" followed by ENTER. The pointer arrow on the display should move to point to the "B" at the beginning of "BOAT".

## Summary of the Data World Example

In the UCSD p-System, virtually all commands that require you to supply data are handled like the commands in the Data example. You press the command code key and a new "prompt" appears on the screen asking for data. You type in the string of characters, usually a name or a number, and then you press ENTER. The command action is then carried out.

As pointed out earlier, commands expressed in the Pascal language generally have the appearance of English language words instead of single characters. Remember that a program is basically a sequence of commands stored for later use. In a program, commands that require data must have that data supplied as part of the program. Unless the program is specifically designed to pick up data from the keyboard, it is generally necessary to store the data needed by the commands as part of the program itself.

# NOTES

# CHAPTER 3. ORIENTATION FOR EXPERIENCED PROGRAMMERS

## Contents

Goals for this Chapter 3-3
Brief Overview ........................ 3-4
The Workfile.......................... 3-6
Running the Edited Program........... 3-7
Saving Workfiles for Future Use ....... 3-9
Entering and Testing a Simple Program .. 3-9
Finish fixing the other errors........... 3-16
Coping with a Compile-time "Syntax" Error....................... 3-18
Saving your Workfile in the Disk Directory ........................... 3-19
First Check Your Disk Directory Using L(ist.......................... 3-19
Now S(ave the Workfile ............... 3-21
What to do if you want to change a previously S(aved Workfile ........... 3-23
Suppressing Execution of the Maze at Bootload Time...................... 3-25

EXP PROGRAMMER

# NOTES

# Goals for this Chapter

To make effective use of this chapter, you should either have some experience in using an interactive computer system for program development, or you should have studied Chapter 2 of this book.

For most programmers, the principal working environment of the UCSD p-System is concentrated in three facilities: the Screen Editor, the File Manager, and the Pascal Compiler. This chapter is intended to give you an overall understanding of how the working environment is used. Details on each of the three major facilities are left until Chapters 4, 5, and 6.

Specifically, here is what you should accomplish in going through this chapter:

a) Learn to enter a small Pascal program into the computer, and how to test and run that program.

b) Learn how to make simple modifications in a small program already stored on your diskette, and to test and save the modified program.

c) Learn what is meant by the "Workfile", and how the Editor, Compiler, and File Manager all cooperate with each other to help you handle the Workfile.

d) Distinguish between the human readable "Text" version of a Pascal program, and the "Code" version of the same program which is executable by the computer.

e) Learn how the File Manager is used as a utility with which you can keep track of your library of program files. Specifically, acquaint yourself with the disk Directory as a tool for telling what currently is saved on your disk(s).

f) Use the File Manager to change your copy of the UCSD p-System on the STARTUP: disk so that it no longer starts up the orientation program (Maze and Data examples for Chapter 2) when you bootload the System.

CAUTION: Some of the steps described in this chapter, and involving use of the File Manager, can leave your diskette changed in such a way that it can no longer be used directly with the step-by-step descriptions in this book. If you decide to jump ahead and make random experiments "just to see what will happen", please be prepared for the possibility that you may have to acquire another diskette in order to start over.

# Brief Overview

In this chapter, I assume that you already know what I mean by a single character "command", and have a rough idea of what I mean by a "program". We assume that you know how to "bootload" (Bootstrap Load) the System. If in doubt, it would be best to scan through Chapter 2, even if you are an experienced programmer. We also assume that you will be programming in the Pascal language, even though FORTRAN can also be used with the p-System in much the same manner as described here.

The purpose of this section is to give you a quick description about how the various major pieces of the UCSD p-System fit together. In later sections, I give simple hands-on exercises using the computer with each of those pieces. Depending upon your own personal way of going at things, you may find it most effective to go through either the quick description or the exercises first, then follow by going through

the other. In any event, you will save time in using the rest of this manual if you take time to get "the big picture" by going through this section.

When you prepare a program to be executed by the UCSD p-System, you write program "statements" in a form that can readily be understood by any human who understands the "programming language" that you use. To get the program statements into the computer in a form that the computer itself can understand, you use a big program called the "Screen Editor", which is provided as a built-in part of the UCSD p-System. The Screen Editor is basically a tool used for purposes similar to those for which you use a pencil and eraser when writing English language text on a piece of paper. There is no really practical way for you to write out a program on paper in such a way that your writing can be directly understood by the computer. Instead, it is necessary to use a keyboard like that on a typewriter, and each key pressed transmits an electronic message to the computer. Without a program to make sense out of the sequence of key-press messages that you send to the computer, those messages would be of very little value. The Editor is the general purpose program tool that we use to prepare programs for the computer. (It can also be used for preparing ordinary written text material, such as this book, as I describe later in this Chapter.)

When you use the Screen Editor, the program text (or any other material that you are writing) is saved temporarily in the computer's memory. We say "temporarily" because all of the contents of the computer's memory are lost when you turn the machine off, and arrangements are generally made for more permanent storage of the information on a flexible diskette, or "Floppy Disk". When you finish changing the text of a program with the Editor, and are ready to try it out, you must use the Editor's Q(uit command. The Q(uit command will respond

by asking whether you wish to "Update" the version of your text on the diskette. If you do request an Update, the text stored in the computer's memory will be transferred to the diskette, and stored in an area called the "Workfile".

## The Workfile

To understand the purpose of the Workfile, it will help for you to understand just a little of how information is stored on the diskette. Information is recorded on a floppy disk using microscopic changes in a magnetic coating on the plastic surface of the diskette. These changes are organized in circular tracks whose purpose is very similar to the "grooves" on a home phonograph record. One diskette has a capacity for approximately 163,000 characters of text. This space is enough to allow storage of many different programs, both in the human readable "Text" form and in the computer executable "Code" form. Therefore the UCSD p-System provides a means for keeping a directory of the various programs stored on the same diskette. The disk directory gives the name of each program, its location on the diskette, how much space on the diskette it occupies, and some additional information needed by the System itself. We say that each entry in the disk directory refers to a "file" of information stored on the disk. For each program, there is a "Text" file, and in most cases there will also be a "Code" file. The disk may also be used for storing various other kinds of information referred to separately in the disk directory is called a "file".

The Workfile is just one of many files stored on your diskette, but its entry in the disk directory uses a special naming convention that saves you trouble while you are working on a new program or changing an old one. When you use the Editor's Q(uit command, and ask for an Update, the text you have been working on is saved on the disk under the

directory name "SYSTEM.WRK.TEXT". Any older version of the file having the same directory name is removed from the disk when you Update in this manner. Whenever you start up the Editor, it assumes that there is a Workfile on the disk and that you wish to work with the text stored in the Workfile. The Compiler and File Manager also make assumptions about the Workfile that save you from having to take explicit actions to keep track of the file you are currently working on.

## Running the Edited Program

Once you are finished making changes in the text of a program using the Editor, you will usually want to have that program turned into the form that can be executed directly by the computer. Then you will want to try the program to see whether it works correctly. This cannot be done until the edited text of the program is translated into the form that will run directly on the computer.



The Compiler resides on the IBM1 disk (which you should place in the right drive). It translates programs saved on disk in their "Text" form into the equivalent "Code" form which can be executed directly by the computer.

We are glossing over a fine point here. The UCSD p-System actually executes all programs using a special "interpreter" program, which makes your computer's processor appear to be a processor designed especially for the purpose of executing Pascal programs. This makes it possible to use the same "Code" form of a Pascal program on any one of many different popular processors, including most of those used in microcomputers.

When you bootload the UCSD p-System, you will find yourself in a command "world" labelled

"Command:" at the left of the Prompt Line at the top of the screen. From the Command: world, you use the E(dit command to start up the Editor. When you use the Editor's Q(uit command, the result will be to bring you back to the Command: world.

When you first boot the STARTUP diskette, the Maze: and Data: command worlds will always appear. I will give you instructions in a later section in this chapter on how to avoid having the Maze: and Data: command worlds will always appear. I will give you instructions in a later section in this chapter on how to avoid having the Maze: always appear, once you have finished using it to get oriented to the System. The Command: world is what appears when you use Q(uit to get out of the Maze: world, and again Q(uit to get out of the Data: world.

If you elect to U(pdate the Workfile when you use the Editor's Q(uit command, you can request the Compiler to translate the program text stored in the Workfile in either of two ways. The most obvious way is to use the Command: world's C(ompile command. A shortcut way is to use the Command: world's R(un command. The System keeps track of what you have been doing to the Workfile, and knows whether you have changed the text stored in the Workfile since the last time you used the Compiler. When you use the R(un command, and the Workfile has been changed, the Compiler is automatically told to translate the text in the Workfile. If the Compiler finds no errors in the program, it then saves the "Code" form of the program on the disk, and then tells the System to go ahead and execute the program. The compiler leaves the Code form of the program in a disk file called SYSTEM.WRK.CODE. Thereafter, you can execute the same version of the program over and over again using the Command: world's R(un command, without calling the Compiler into action again until you change the Text form of the Workfile using the Editor.

Each time your program finishes executing on the computer, control of what happens returns to the Command: world where the System waits for your next command.

## Saving Workfiles for Future Use

Once you have finished making changes in a program you probably will want to save that program on the disk for later use. You will also probably want to clear out your Workfile in order to work on another program. To do this, you use the Command: world's F(ile command, which takes you into the File: Manager's world. (Most of us have become lazy and refer to the File Manager simply as the "Filer".) The Filer provides commands for saving a Workfile under a directory name you may designate, for removing old files no longer needed, for transferring files from one disk to another, for displaying the disk directory on the CRT screen, and other file-related commands. As usual, you use the File: world's Q(uit command to get back to the Command: world.

EXP PROGRAMMER

# Entering and Testing a Simple Program

In this section, I will give a step-by-step account of how you enter a simple program into the computer and then compile it and execute it. I will start from the "Command:" world. To get there after bootloading the STARTUP: disk, you should use the Q(uit commands of the "Maze:" world and the "Data:" world. In the last section of this chapter, I will show you how to arrange to get to the "Command:" world directly after bootloading. (It would be best not to jump to that point right away, since some familiarity with the System gained with a little practice will help you to avoid making an error that could be very awkward to correct.)

As subject matter, I will use the sample program STRING1 from Chapter 1, Section 11, of the Bowles text Problem Solving Using Pascal. We reproduce that program in Figure 3-1 as follows:

```
PROGRAM STRING1;
BEGIN
  WRITE('HI');
  WRITE(' ','THERE');
  WRITELN; (*moves to start of next line*)
  WRITE('HI THERE');
  WRITELN('THIS IS A DEMONSTRATION');
  WRITELN('OF PROGRAM EXECUTION');
END.
```

Figure 3-1. Sample program for familiarization with the System .

You should start from the "Command:" world by typing "E" for the E(dit command. The screen will go blank and then, after some clicking by the floppy disk drive, the following will appear:

```
>Edit:
No workfile is present. File? (<ent> for no file)
```

The Prompt line at top of the screen informs you that you have arrived in the "Edit:" world. No command options are shown yet, since no Workfile is stored on the disk, and it is necessary to establish one. The second line on the screen requests that you type in the name of a text file already stored on the disk, and follow by pressing the ENTER key. In the present instance, you have no such file to use, so you simply press the ENTER key without typing in any name. The Editor will respond with the screen display as follows:

```
>Edit: A(djst C(py D(lete F(ind I(nsrt J(mp ...
```

Except for the list of available command characters in the Prompt line at top of the screen, the display is completely blank. This shows that the working space used by the Editor in the computer's memory is completely blank. It is, as it were, a "blank slate" on which you can start writing.

## The I(nsert Command

To begin typing in the text of the STRING1 program, use the I(nsert command of the "Edit:" world. The somewhat cryptic Prompt message that goes with the I(nsert command tells you that you can start typing in the text. Begin with "PROGRAM" and continue typing until you make a mistake. You can erase a character typed in error by using the BACKSPACE key. One character is erased for each BACKSPACE typed, and you can back up all the way to the point where the I(nsert command's world was entered. Continue typing after any erasure until you finish entering a section of text that you wish to retain. You can terminate the I(nsert command, while retaining the text typed in, by pressing Ctrl-C. Ctrl-C is used on the IBM Personal Computer keyboard as the ETX key (which stands for "End of TeXt").

While within the I(nsert command's world, you move the cursor from the end of one line to the begining of the next by using the ENTER key. To obtain the two column indentation in the third line of the STRING1 program example, press the SPACEBAR twice. When you press ENTER to begin

the subsequent lines, the cursor will return to the same column indented two spaces from the left margin. This is just what you want until you arrive at the last line containing "END."

To eliminate the indentation for that line, there are several ways of proceeding. I will mention only the most frequently used method here, and leave other suggestions for Chapter 4. After pressing ENTER at the end of the previous line, the cursor again comes to the third column and waits for further characters to be typed. At that point, you can use the BACKSPACE key to move the cursor to the left edge of the line. Press BACKSPACE only twice to get there. If you press BACKSPACE once more, you will return to the end of the previous line, effectively backing over and erasing the ENTER character from the text as saved in the computer's memory. No harm is done by this action, but when you again press ENTER to get back to the new line, the cursor will again go to the third column.

Now suppose that you have typed in the lines of text shown in Figure 3-2 which contains several errors. The next question is how to go about correcting those errors without having to start all over again.

```
>Insert: Text (<bs> a char, <del> a line)
  PROGRAM STRING1;
  BEGIN
    WRITE ('HI');
    WRITE(' ','TREE');
    WRITELN: (*moves to start of next line*)
    WRITE('HI THERE');
    WRITELN('THISA DEMSTATION');
    WRITELN('OF PROGRAM EXECUTION)
  END.
```

Figure 3-2. End of I(nsertion With Errors In the Text.

# The I(nsert Command

Compare the fourth line in this figure with the fourth line in the STRING1 program in Figure 3-1. In Figure 3-2, the word "THERE" is misspelled "TREE". As a first step to correct this, complete the I(nsertion by pressing Ctrl-C, if you have not already done so. Next, move the cursor so that it points to the character "R" in "TREE". You do this by using the four cursor positioning arrows on the number pad. These are the same keys as used for moving around the maze in the example given in Chapter 2 of this book. With the cursor pointing at the "R", use the I(nsert command again. Once in the "Insert:" world, type in "HE" followed by pressing Ctrl-C. Notice that the I(nsert command in this case moves everything starting with the "R" over to the right hand margin of the screen. This is done in order to leave you blank columns into which the additional characters may be typed. When you press Ctrl-C, the characters moved to the right of the screen will be returned to connect up once again with the portion of the line still on the left side. At this point the misspelled word has been partially corrected, and should read "THEREEE".

Before you continue, it would be worthwhile to review what you can do with the I(nsert command:

a) You enter the "Insert:" world by typing "I" while in the "Edit:" world. You can then type in characters of text starting from the cursor's position (as it was when the "Insert:" world was entered).

b) You can erase unwanted characters (of those typed in so far during this insertion) by pressing the BACKSPACE key once for each character.

c) You are able to erase all the characters typed so far on any whole line, after the first ENTER is typed during the insertion, by pressing the DELETE-LINE key (Ctrl-BACKSPACE). Do not confuse the DELETE-LINE key with the DEL key which performs a different function not described here. Press the DELETE-LINE key repeatedly to remove additional lines typed in during the current insertion.

d) When you decide to keep the text typed in within the "Insert:" world, press Ctrl-C.

e) If you decide just to leave the "Insert:" world without keeping any of the characters already typed in, press the ESC key, for "escape". This will terminate the "Insert:" world, and return you to the "Edit:" world, with the displayed text again just as it was before you entered the insertion.

Try using the DELETE-LINE and ESC keys while using the I(nsert command to observe what happens.

# The D(elete Command

Now we continue from the point where you have "THEREE" on the screen following use of I(nsert. To obtain "THERE", you will have to delete the last "E" (or the one before it). Move the cursor to point to the "E" you wish to remove. Press "D", for D(elete, and observe that the screen now displays what is shown in Figure 3-3.

```
>Delete: <> <Moving commands> (<etx> ...
  PROGRAM STRING1;
  BEGIN
    WRITE('HI');
    WRITE (' ','THEREE');
    WRITELN; (*moves to start of next line*)
    WRITE('HI THERE');
    WRITELN(' THISA DEMSTATION');
    WRTIELN('OF PROGRAM EXECUTION)
  END.
```

Figure 3-3. Display upon entering D(elete.

To delete one or more characters, move the cursor to the right, or to following lines using the regular cursor moving commands. Each character deleted disappears off the screen. For example, you can delete "E');" by pressing the right arrow four times. Just a with the "Insert:" world, you can back up by using the BACKSPACE key. In the "Delete:" world, you restore characters to the screen when you use BACKSPACE.

Finally, when you are ready to terminate the D(eletion, press Ctrl-C. This will return you to the "Edit:" world, with the text redisplayed and the

deleted characters eliminated. You can terminate the deletion using ESC, just as you can escape out of the "Insert:" world. Again, the displayed text returns to the status it had before you entered the "Delete:" world.

## Finish fixing the other errors

Now try your hand with the Editor by fixing the errors on the last two lines in Figure 3-2 containing "WRITELN". The next to last contains spelling errors. The last lacks an apostrophe just before ")".

# Q(uit and U(pdate your Workfile

---

At this point, you should be ready to compile and test the small program displayed on your screen. Press "Q" (for Q(uit), and you will be shown a selection of three or four options, as shown in Figure 3-4.

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

Figure 3-4. Options for Q(uit command of the "Edit:" world.

# Q(uit and U(pdate your Workfile

Press "U" (for U(pdate) to cause the Pascal program displayed on your screen to be saved in your Workfile on the disk. If you press "E", the Editor will terminate and return you to the "Command:" world without saving anything you have done with the Editor! Press "R" to R(eturn back into the "Edit:" world. The R(eturn option saves you from embarrassing problems if you press Q(uit by mistake while in the "Edit:" world. The W(rite command is useful to record a copy of text under a specified name, without leaving the Editor.



# R(un your Program

Now press "R" (for R(un) from the "Command:" world. The result should be a notice at the top of the screen saying:

```
Compiling...
```

followed by much clicking and some additional displayed information. I will defer explaining what is happening here until Chapter 6. For now, these displayed lines tell you that the Compiler is busy trying to translate your Pascal program in the Workfile into the "Code" form that can be executed by the computer. If the Compiler finds no errors, it will save the executable Code form of your Workfile on the disk, and then will cause your program to start executing (i.e. to start "running"). You will be notified that this is happening by the legend "Running..." at the top of the screen.

# R(un your Program

## Coping with a Compile-time "Syntax" Error

If the Compiler does find an error in your Pascal program, it will not be able to save the Code form of your workfile on the disk, nor will it start execution of your program. Instead, after more clicking of the disk, you will find yourself presented with a message and an opportunity to either continue the compilation or go back in the "Edit:" world with the cursor pointing at the end of the logical item where the Compiler found the error. A message displayed in the promptline will give an explanation of the error the Compiler found. An error found by the Compiler is called a "Syntax Error" because it indicates that you have violated one or more of the formal Syntax Rules which describe how a Pascal program should be constructed. See Chapter 6 for more details on this point.

If you decide to enter the Editor by typing "E", you will need to press the SPACE bar to get the "Edit:" world's promptline so that you can fix the problem.

At this point, it would be a good idea to think over the rest of your program, to see if there might be additional errors similar to the one found by the Compiler. When you are satisfied that you have found "all" the errors, you can Q(uit, U(pdate, and R(un once again to see what happens.

# Saving your Workfile in the Disk Directory

Eventually, you will finish editing and testing revised versions of your program. You may then wish to start working on a completely different program, but may also wish to save the program just finished so that it can be used again at a later time. To do this, use the "F" for F(ile command when in the "Command:" world. After some clicking of the disk, you should receive the following display:

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

## First Check Your Disk Directory Using L(ist

Press "L" for L(ist to see the list of titles of all the files currently stored on your disk. The L(ist command requests data input whereby you tell it which disk to refer to. In the UCSD p-System, each disk has a "Volume Identifier", which is the name of the disk itself. For the present example, respond to the prompt requesting a Volume Identifier by pressing ":" (the Colon key) followed by ENTER. The Filer will respond by listing out the directory showing your disk's contents. Figure 3-5 shows approximately the directory listing you should get at this point.

# R(un your Program

```
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans.
STARTUP:
SYSTEM.PASCAL      121  15-Nov-81
SYSTEM.MISCINFO      1  15-Nov-81
SYSTEM.INTERP       26  15-Nov-81
SYSTEM.FILER        32  15-Nov-81
SYSTEM.EDITOR       49  15-Nov-81
SYSTEM.STARTUP      12  15-Nov-81
SYSTEM.SYNTAX       14  15-Nov-81
NAMEFILE             3  15-Nov-81
SCDEMO.CODE          3  15-Nov-81
COPYSCUNIT.CODE      4  15-Nov-81
UPDATE.CODE          4  15-Nov-81
COMPDEMO.TEXT        6  15-Nov-81
EDITDEMO.TEXT        4  15-Nov-81
UPDATE.TEXT          8  15-Nov-81
SYSTEM.WRK.TEXT      4   1-Jan-82
SYSTEM.WRK.CODE      2   1-Jan-82
16/16 files<listed/in-dir>,299 blocks used, 21 unused, 21 in
```

Figure 3-5. Response to the Filer's L(ist Command.

Note: The two directory entries labelled SYSTEM.WRK.TEXT and SYSTEM.WRK.CODE. These are the two files associated with your Workfile. The first is the form saved by the Editor when you use Q(uit followed by U(pdate starting in the "Edit:" world. The second is the executable form of the same Pascal program, which was saved on the disk by the Compiler.

## Now S(ave the Workfile

Press "S" for S(ave, and note the prompt for a file title on the top line. If the Workfile has not yet been S(aved in a previous version, the Filer will prompt with:

**Save as what file ?**

You type in a name followed by ENTER to complete the S(ave command. If you do not wish to lose a previously S(aved workfile, use a name different from any other already in use in the disk's directory. Suppose we respond to the prompt by typing in "NEWNAME" followed by ENTER. Now repeat the L(ist command to see the result, which is shown in Figure 3-6.



```
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans,
STARTUP:
SYSTEM.PASCAL      121  15-Nov-81
SYSTEM.MISCINFO      1  15-Nov-81
SYSTEM.INTERP       26  15-Nov-81
SYSTEM.FILER        32  15-Nov-81
SYSTEM.EDITOR       49  15-Nov-81
SYSTEM.STARTUP      12  15-Nov-81
SYSTEM.SYNTAX       14  15-Nov-81
NAMEFILE             3  15-Nov-81
SCDEMO.CODE          3  15-Nov-81
COPYSCUNIT.CODE      4  15-Nov-81
UPDATE.CODE          4  15-Nov-81
COMPDEMO.TEXT        6  15-Nov-81
EDITDEMO.TEXT        4  15-Nov-81
UPDATE.TEXT          8  15-Nov-81
NEWNAME.TEXT         4   1-Jan-82
NEWNAME.CODE         2   1-Jan-82
16/16 files<listed/in-dir>, 299 blocks used, 21 unused, 21 in
```

Figure 3-6. L(isting of directory after S(ave of "NEWNAME".

# R(un your Program

Notice that the entries which had been shown as SYSTEM.WRK.TEXT and SYSTEM.WRK.CODE are now shown as NEWNAME.TEXT and NEWNAME.CODE respectively.

If the Filer finds that the file had previously been saved under a particular name, it asks if the file should again be saved with that name:

**Save as DEMO:NEWNAME.TEXT ?**

You can respond to this prompt either with "Y", for Y(es, or "N", for N(o. If Y(es, then the previously S(aved files called NEWNAME.TEXT and NEWNAME.CODE will be removed from the disk directory, and your new version of the Workfile(s) will be given the names:

**NEWNAME.TEXT**
**NEWNAME.CODE**

If you respond to the prompt with N(o, the Filer will prompt with:

**Save as what file?**

as described above, and you can then use almost any new name not already in use in the disk directory. Note, however, that a Workfile name cannot be longer than 10 characters (plus ".CODE" or ".TEXT").

## What to do if you want to change a previously S(aved Workfile

You often will S(ave a Workfile only to realize that you need to make changes in that Workfile at a later time. The Filer's G(et command allows you to designate the name of an old file as the current name associated with the Workfile. Press "G", for G(et, and note the prompt message that appears in response. If the response is:

**Get what file?**

then you can type in any name previously used when S(ave was used in connection with a Workfile (that still is stored on your disk). If it finds the directory entry for the file name you give it, the Filer will respond with:

**Text and Code files loaded**

or:

**Text file loaded**

or:

**Code file loaded**

as the case may be. If the Filer prompts with:

**Throw away current workfile?**

you have the opportunity to avoid possible loss of your files named SYSTEM.WRK.TEXT and SYSTEM.WRK.CODE by typing "N" for N(o. If you type "Y" for Y(es, the Filer will discard your old unsaved workfile, and then prompt you for the file name you want to be loaded as your new Workfile as described above.

# R(un your Program

If you change your mind after starting the G(et command, you can get back to the "File:" world by pressing the ESC key, or by typing in the name of a nonexistent file followed by ENTER.

After using G(et to establish the name of a previously S(aved file as the current Workfile, you can leave the "File:" world using Q(uit. This gets you back to the "Command:" world. If you then use E(dit, the Editor's world will be entered, and the first screen- full of the now current Workfile will be displayed. If you use C(ompile, instead of E(dit, the Compiler will proceed to try to translate the .TEXT form of your now current Workfile into executable form. If the Compiler succeeds, it will save the resulting executable file, as usual, as SYSTEM.WRK.CODE. Regardless of whether the Compiler succeeds, use of the Compiler will cause any previous file with the directory name of SYSTEM.WRK.CODE to be removed from the directory.

If, upon reaching the "Command:" world after leaving the "File:" world, you use the R(un command, the System will attempt to execute the currently saved .CODE form of your Workfile without using either the compiler or the Editor. If there is no .CODE form of the Workfile on the disk, the Compiler will be invoked to translate the .TEXT form of the Workfile into executable form.

# Suppressing Execution of the Maze at Bootload Time

When you first receive your IBM Personal Computer, the "Maze:" world will always appear immediately after you Bootload the STARTUP disk. As soon as you feel familiar with the idea of single character commands, you will probably want to dispense with the "Maze:" and "Data:" world exercises. To do this, enter the "File:" world by using the F(ile command while in the "Command:" world.



Now use the L(ist command, as described earlier in this chapter, and note the entry called SYSTEM.STARTUP. That entry is a special "reserved" name used with the code form of a program workfile called ORIENTER. The ORIENTER Workfile contains the programs which create both the "Maze:" and "Data:" worlds. You can retain that file, but suppress its automatic execution at Bootload time, by changing its name back to ORIENTER.CODE. To do this, press "C" for C(hange. The Filer will prompt with:

**Change what file?**

You answer by typing in:

**SYSTEM.STARTUP**

followed by ENTER. The Filer will respond with:

**Change to what ?**

You answer by typing in:

**ORIENTER.CODE**

followed by ENTER.

# R(un your Program

If you have followed these steps without error, the final result will be indicated by a message verifying that the change has been made. Having made that change, you should no longer have a file called SYSTEM.STARTUP on your disk. You might want to check to make sure that this is correct by using the Filer's L(ist command.

It may help to explain what I have been doing here. When you Bootload the System, it is programmed to look through the disk directory for an executable file called SYSTEM.STARTUP. If one is present, the program contained in that file is loaded into memory and executed automatically. If no such file is present on the disk, then bootloading takes you immediately to the "Command:" world. Now try Bootloading again to verify that this indeed is what happens.

You may wish to get rid of the file containing the "Maze:" and "Data:" worlds entirely, in order to release space on your disk for other uses. To do this, see Chapter 5 on the Filer regarding the R(emove command.

# CHAPTER 4. SCREEN EDITOR

## Contents

Goals for this Chapter ................ 4-3
Editor Overview ...................... 4-4
Cursor Movement Commands ......... 4-7
Arrow Commands and Their Relatives .. 4-9
Repeated executon of an arrow
command .......................... 4-10
Moving the cursor off the screen ...... 4-10
Using SPACE, BACKSPACE,
and ENTER ........................ 4-14
The TAB key ........................ 4-15
Multiple occurrences of the target ..... 4-23
F(inding backwards .................. 4-24
L(iteral Targets vs Tokens ........... 4-25
Commands that Change the
Workfile's contents ................ 4-29
The Q(uit command and its options.... 4-46
Using the Editor for Word Processing ... 4-48

# NOTES

# Goals for this Chapter

To use the UCSD p-System effectively, you need to be familiar enough with the Screen Editor to use it as a convenient tool. The main goal of this chapter is to provide you with a reference summary of how the Editor is used. In each section, the order of presentation starts with the Editor's facilities you are likely to use most often. See Appendix B1 for an alphabetic summary of the Editor's commands with references to descriptive text in this chapter.

Many beginners do not bother to learn how to use all the available facilities of the Editor. While you can make extensive use of the UCSD p-System by knowing how to use only a small part of the Editor's facilities, it will probably save you time to become familiarized with each of the Editor's commands.

Specific goals for this chapter include the following:

a) Learn to use each of the principal commands of the Editor to the point where you are comfortable in using them as tools.

b) Edit a file established as the current Workfile by the Filer, one named at the time when the Editor starts up, and a new file not previously stored on the disk.

c) Terminate the Editor by U(pdating the current Workfile, by E(xiting without update, and by W(riting a named file to disk. Check using the Filer to see what happens in each case.

# Editor Overview

The Editor is the UCSD p-System's principal tool for creating, reading, and changing text files, i.e. files of information in the form directly readable by humans when displayed. The Editor is designed to work with the entire contents of a text file in the computer's main memory as one unit. It can usually handle more than 700 lines of Pascal program text in one file. The Compiler provides a convenient means for combining several of these files into a single large program.

Since your CRT screen cannot display the entire contents of most Workfiles, the screen is used as a movable "window" through which you can view the contents of the Workfile. You point at the place in the Workfile you wish to view by moving the cursor up or down with the commands provided. When moving the cursor would have the effect of shifting to a text line not currently displayed on the screen, the Editor automatically causes the window to be moved so as to display the section of text to which the cursor has moved. In addition to various commands provided to move the cursor from place to place in the Workfile, there are also commands with which you can change the content of the Workfile at the place where the cursor points.

The simplest of the cursor movement commands are the up-pointing and down-pointing arrows, and the arrow keys pointing right and left. Though the content of the Editor's window is displayed as a sequence of lines as in a page of printed text, you can think of the Workfile stored in the computer's memory as if it were stored on one long thin continuous strip of paper, with all the lines connected end to end. Therefore, when the cursor is at the right end of one displayed line, pressing the right arrow once moves the cursor to the left end of the next line below on the display. Similarly, when

the cursor points to the left-most nonblank character in a line, pressing the left arrow returns you to the right end of the line above. (If you are running with a forty column screen, see Appendix A for a description of screen handling keys.)

In addition to the four arrow command keys, there are several other commands for moving the cursor. If you know of a word, or other string of characters stored in the Workfile, you can use the F(ind command to scan through the Workfile looking for that word or string rapidly. You can also S(et markers in the Workfile, and use the J(ump command to shift the displayed window suddenly to any one of the markers. Markers at the B(eginning and E(nd of the Workfile are built into the Editor and you do not need to use the S(et command to establish their positions. There is also a P(age command which allows you to shift the displayed window one screen-full at a time. The direction of the shift depends on whether the "direction flag" in the upper left corner of the screen points right (">"), i.e. toward the end of the Workfile, or left ("<"), i.e. toward the beginning.

Press the keys containing the broken brackets ">" and "<" to change the pointing direction at will. Press the TAB key (the key with the left and right arrows pointing into veritcal bars) to shift the cursor 8 columns to the right or left in the workfile depending on which direction the flag indicates. Press the ENTER key (the bent arrow key) to command the cursor to move to the left-most character of the line following the line where the cursor currently points. Type a number before any of these commands (the number will not be echoed on your screen as you type) to cause the command to be repeated that number of times.

Most of the other commands of the Editor are used to change the contents of the Workfile. I(nsert allows you to type text into the Workfile starting at the position just before the character pointed at by the cursor when you enter the I(nsert command's world. D(elete allows you to remove characters from the Workfile, starting where the cursor points when you enter the D(elete world, and ending where the cursor points when you press Ctrl-C. R(eplace is an extension of the F(ind command which allows you to specify a string of characters to substitute for the word or string which is found after scanning through the Workfile. C(opy is used to insert into the Workfile a passage of text that has previously been saved temporarily in a "buffer" area of the computer's memory following an I(nsert or D(elete command. C(opy can also be used to insert a portion of the text stored in another named Workfile. The A(djust command lets you shift the entire line where the cursor is currently located to the left or to the right. The eX(change command lets you type over characters stored in the Workfile on a one-for-one basis, thus simplifying the steps needed to make some changes in the text.

When you finish editing a Workfile, and need to move on to other activities, use the Editor's Q(uit command which offers several options. The U(pdate option causes the text stored in the computer's memory to be saved on the disk under the reserved Workfile name SYSTEM.WRK.TEXT. Any previous version of your Workfile will be lost when this happens! The E(xit option allows you to leave the Editor without changing anything on the disk. In this case, the text stored in the computer's memory is lost! The W(rite option allows saving the text stored in the computer's memory under a name that you can designate. This option permits you to continue editing without having to restart the Editor. The R(eturn option is provided to allow you to continue editing even if you trigger the Q(uit command by pressing "Q" inadvertently.

# Cursor Movement Commands

To provide an example large enough to give you worthwhile practice with the Editor, we will use the Workfile EDITDEMO, which is supplied on the STARTUP disk. This Workfile contains a Pascal program which combines the programs REPEAT1 and REPEAT2, which are presented in Chapter 3, Section 8, of the book *Microcomputer Problem Solving Using Pascal* referred to in Chapter 2 of this book. Each of these two programs has been changed into a procedure in order to produce a Workfile long enough to occupy at least two windows when viewed on the CRT screen. The program contained in the EDITDEMO Workfile can be compiled and executed, but it is supplied to you primarily for use as a starter file for practicing with the Editor.

To get started, enter the Filer from the "Command:" world by pressing "F". Then use the G(et command to establish EDITDEMO as your current Workfile. Next Q(uit from the Filer, and press "E" for E(dit from the "Command:" world. The result should be the display shown in Figure 4-1.

```
>Edit: A(djst C(py D(let F(ind I(nsrt J(mp K(ol R(plc Q(uit X(ch
PROGRAM EDITDEMO;

PROCEDURE REPEAT1;
VAR S,SG:STRING;
    L,N:INTEGER;
BEGIN
  WRITELN(
   'TYPE ANY STRING FOLLOWED BY <ent>'
      );
  READLN(S);
  N:=1;
  L:=LENGTH(S);
  REPEAT
   SG:=COPY(S,1,N);
   WRITELN(SG);
   N:=N+1;
  UNTIL N>L
END (*REPEAT1*);

PROCEDURE REPEAT2;
VAR S:STRING;

PROCEDURE REVERSE;
  (*REVERSE THE ORDER OF CHARACTERS
```

Figure 4-1. The EDITDEMO Workfile.

## Arrow Commands and Their Relatives

The IBM Personal Computer has four arrow keys on the key pad intended for moving the cursor around on the screen. The up arrow moves the cursor up one line on the screen, the down arrow one line down. The right and left pointing arrow keys similarly move the cursor one position to the right or left. If you want a more detailed introduction to the use of the cursor positioning arrow keys, please see the sections of Chapter 2 which present the Maze example.

As an exercise at this point, note a specific place in the displayed text of the EDITDEMO program, and move the cursor to that place using the arrow keys. Notice that movement to the right or left will only place the cursor within the group of characters starting with the left-most nonblank character on a line, and ending with the blank following the right-most nonblank character. This is intended to be a convenience to users, since the long runs of blank characters displayed elsewhere on the screen are not actually stored in the computer's memory. Vertical movement through the runs of blanks is permitted however. For example, start with the cursor pointing at the "G" in STRING within the long line just 3 lines below "BEGIN" in the REPEAT1 procedure, then press the down arrow 6 times to reach ";" in the line

```
SG:=COPY(S,1,N);
```

Now press the up arrow once, leaving the cursor in the line just above the ";". Now press the left arrow just once, and note that the cursor jumps to the "T" in REPEAT.

## Repeated execution of an arrow command

There is a facility which allows you to simulate repeated pressing a key, such as any of the arrow keys. This is accomplished just by holding down the key to be repeated for about one-half second or more. The Editor program also provides a way to move the cursor a repeated number of positions. To see how it works, place the cursor again on the "G" in STRING in the line:

```
'TYPE ANY STRING FOLLOWED BY <ent>'
```

within the REPEAT1 procedure. Now press the "6" key followed by the down arrow. The cursor should again jump to the ";" in:

```
SG:=COPY(S,1,N);
```

You can cause repeated execution of many Editor commands by first typing in the number of repetitions you want.

## Moving the cursor off the screen

Try moving the cursor to the bottom line of the screen. Now press the down arrow, noting that the entire content of the screen shifts up one line. This is equivalent to moving the displayed "window" down in the text by one line, thus revealing an additional line at the bottom of the screen, and hiding a line at the top. Continue pressing the down arrow until the line:

```
PROCEDURE REPEAT2;
```

appears on the top line of the screen. The screen should appear as shown in Figure 4-2.

```
PROCEDURE REPEAT2;
VAR S:STRING;

PROCEDURE REVERSE;
  (*REVERSE THE ORDER OF CHARACTERS
   IN S*)
VAR NB,NE:INTEGER;
    (*BEGIN AND END POINTERS*)
    SAVE:CHAR;
BEGIN
  NB:=1;
  NE:=LENGTH(S);
  REPEAT
    (*EXCHANGE CHAR'S NB & NE,
      SHIFT NB & NE *)
    SAVE:=S[NE];
    S[NE]:=S[NB];
    S[NB]:=SAVE;
    NB:=NB+1;
    NE:=NE-1
  UNTIL NB>=NE;
END (*REVERSE*);

BEGIN (*REPEAT2*)
```

Figure 4-2. Display Following Multiple Use of the Down Arrow Key.

The upward shifting of the screen contents is called "scrolling", as if the displayed text were actually on a scroll of paper being pulled upwards behind the screen's "window". You cause the screen content to scroll upwards by one line with the Editor, if the cursor is located in the bottom line of the screen and you press the down arrow. This keeps the cursor within the displayed window. Continue pressing the down arrow (or use the repeat feature) causing the text to scroll upwards until it stops scrolling. The cursor will then be in the last line of text in the Workfile, presenting the display shown in Figure 4-3.

```
>Edit: A(djst C(py D(let F(ind I(nsrt J(mp K(ol R(plc Q(uit X(ch Z(ap [E.7h]

BEGIN (*REPEAT2*)
  WRITELN(
  'TYPE ANY STRING FOLLOWED BY <ent>'
      );
  READLN(S);
  WHILE LENGTH(S)>0 DO
  BEGIN
    REVERSE;
    WRITELN(S);
    WRITELN;
    WRITELN('TYPE ANOTHER STRING');
    READLN(S);
  END;
END (*REPEAT2*);

BEGIN (*MAIN PROGRAM*)
  WRITELN('START EDITDEMO');
  WRITELN;
  REPEAT1;
  WRITELN;
  REPEAT2;
END.
```

Figure 4-3. Display After Scrolling to the End of the Workfile.

Shifting the cursor off-screen in the other direction is more awkward. To see what happens, move the cursor upwards carefully until it rests in the top line displayed on the screen. Now press the up arrow just one more time. The result should be as shown in Figure 4-4.

```
>Edit: A(djst C(py D(let F(ind I(nsrt J(mp K(ol R(plc Q(uit X(ch Z(ap [E.7h]
BEGIN
  NB:=1;
  NE:=LENGTH(S);
  REPEAT
    (*EXCHANGE CHAR'S NB & NE,
      SHIFT NB & NE *)
    SAVE:=S[NE];
    S[NE]:=S[NB];
    S[NB]:=SAVE;
    NB:=NB+1;
    NE:=NB-1;
  UNTIL NB>=NE;
END (*REVERSE*);

BEGIN (*REPEAT2*)
  WRITELN(
    'TYPE ANY STRING FOLLOWED BY <ent>'
      );
  READLN(S);
  WHILE LENGTH(S)>0 DO
  BEGIN
    REVERSE;
    WRITELN(S);
    WRITELN;
```

Figure 4-4. Display Following Use of Up Arrow in the Top Line.

In this situation as in several others, the Editor solves the problem of displaying the new cursor position by clearing the screen and then re-displaying to show a window with the cursor in the middle line of the screen.

## Using SPACE, BACKSPACE, and ENTER

The SPACE bar can be used to substitute for both the right arrow and the left arrow (when you are in the "Edit:" or "Delete:" world). When the Editor's direction flag, located in the upper left corner of the screen, points forward (">") the SPACE bar substitutes for the right arrow. When the direction flag points backwards ("<"), i.e. toward the beginning of the Workfile, the SPACE bar substitutes for the left arrow.

The BACKSPACE key is equivalent to the left arrow when you are in the "Edit:" or "Delete:" world. The Editor's direction flag has no effect on the operation of the BACKSPACE key.

The ENTER key causes the cursor to jump to the beginning of the next line. If the Editor's direction flag points forward, i.e. ">", then the ENTER key moves the cursor to the first nonblank character on the next lower line in the Workfile. The displayed window is scrolled upwards if necessary to display the next line. If the Editor's direction flag points backwards, i.e. "<", then the ENTER Key moves the cursor to the first nonblank character on the previous line in the Workfile. The screen window is re-displayed if the ENTER key is pressed when the cursor is located in the top line of text on the screen (and when that line is not the first line in the Workfile).

You might wonder why no special provision has been made to cause the cursor to jump easily to the end of the next or previous line in the Workfile. This can be done by the simple expedient of jumping to the beginning of the line following the line whose end you wish to jump to. Then press the BACKSPACE (or left arrow) key just once.

## The TAB key

The TAB key is used as an "express" version of the SPACE key in the Editor. Each time you press TAB, the cursor is moved until it coincides with a column at which a new group of 8 columns starts. Thus the TAB "stops" are located at columns 1, 9, 17, 25, 33, etc. (It is possible to change the positions of the TAB-stop columns using S(et E(nvironment. See the *User's Guide for the UCSD p-System* for a description of this). If the Editor's direction flag points forward (">"), then the TAB moves the cursor toward the end of the Workfile, jumping from the end of one line to the beginning of the next if necessary. If the flag points backwards ("<"), then the TAB moves the cursor towards the beginning of the Workfile.

# The P(age command

The P(age command is an "express" equivalent of the up arrow and down arrow commands of the "Edit:" world. It is similar in concept to the TAB key command, but moves the cursor whole lines up or down in the Workfile depending upon the current status of the Editor's direction flag. If the direction flag points forward (">"), then the P(age command causes the display and the cursor to move forward in the Workfile as many lines as the screen window is high. Thus, the displayed window will show the next 24 lines in the Workfile. The cursor's position on the screen will remain the same, but its logical position will be moved forwards by 24 lines. Similarly, if the direction flag points backwards ("<") then the P(age command will jump to the previous group of screen-height lines. At the end of the Workfile, the P(age command may not display a complete window full if there are not enough additional lines available in the Workfile to fill the screen. In that case, the cursor will be placed at the end of the file, and only the top half of the window will be filled.

# The J(ump command and its relatives

The Editor's J(ump command provides a way to move the cursor quickly from one place in the file to another without having to use the up or down arrow commands repeatedly. Here is the promptline displayed by the J(ump command:

**>JUMP: B(eginning E(nd M(arker <esc>**

Respond to this prompt with B(eginning, and the cursor will be moved suddenly to the beginning of the Workfile. Similarly, E(nd places the cursor at the end of the Workfile. In both cases, the screen window will be re-displayed if necessary. If you respond with M(arker, the Editor will respond with the following prompt:

**Jump to what marker?**

This refers to "markers" that you can place anywhere in the Workfile using the S(et command. As used with many commands in the Editor, you can press the ESC(ape key to simply terminate the J(ump command's world without doing anything.

# The S(et Command Used For Setting Markers

The S(et command has several different purposes, mainly related to setting "switches" which control how the Editor operates. It can also be used to read the current values of those switches. For purposes of this section on cursor movement commands, we will only be concerned with setting markers into the Workfile for use with the J(ump command. Various other switches that can be reached with the S(et command are intended mainly for use in Word Processing applications and are discussed later in this chapter.

To establish a marker, use the S(et command when in the "Edit:" world. The result will be the promptline:

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

If you respond with M(arker, the Editor's prompt will be:

**Set what marker?**

to which you can respond with a number, name, or other short identifier terminated by ENTER. The position of the marker will be the position of the cursor at the time when you enter the S(et command.

# The S(et Command Used For Setting Markers

As an example, go through the sequence of steps needed to display the portion of the EDITDEMO file shown in Figure 4-4. Place the cursor in the blank line between "END (*REVERSE*);" and "BEGIN..." which is two lines below. Now use S(et to establish a marker simply called "1" (do not type in the quotes when responding to the command). You can check to see the result of doing this by using the E(nvironment option of the S(et command. The response from this option should be as shown in Figure 4-5 if you have successfully established a marker called "1".

```
>Environment: (options) <spacebar> to leave
    A(uto indent         True
    F(illing             False
    L(eft margin         1
    R(ight margin        80
    P(ara margin         6
    C(ommand ch          ^
    S(et tabstops
    T(oken def           True

    989 bytes used, 21539 available.

Created June 9, 1981; Last updated June 9, 1981 (Revision 0).
Editor Version E.7h iv.0.
```

Figure 4-5. S(et E(nvironment display.

# J(umping to markers

Now use J(ump to move the cursor to the B(eginning of the Workfile. The resulting display should be as shown in Figure 4-1. Next, J(ump to the E(nd of the Workfile, getting essentially the display shown in Figure 4-3. In this case, the cursor will be at the very last character position in the Workfile, rather than at the beginning of the last line, as resulted from the sequence of steps that led to Figure 4-3.

Neither of the displays reached by J(umping to the B(eginning or the E(nd of the Workfile shows the text which includes the marker "1" which was S(et in the previous subsection. Now use J(ump, respond to the prompt with M(arker, and then with 1 followed by ENTER. The Editor will respond with a display like that shown in Figure 4-4, and with the cursor at the same position it had when you established the marker.

You can establish 20 markers within a workfile. The Editor will keep track of the logical position of each marker, even when you change the contents of the Workfile. Of course, if you D(elete a section of text containing a marker, it does not make sense to maintain the position of the marker. In this case, the position of the marker may show up almost anywhere in the text that remains. If you use more than two or three markers, it will generally be difficult to remember their logical positions in the text unless you give them names that suggest their locations. However, the Editor will remember only the first 8 characters of a long marker name. If you want to reuse a marker name at a new location, simply S(et it again. If you try to S(et too many different markers, the Editor will prompt you on

# J(umping to markers

steps to follow in replacing one of the markers already established. You can always get a listing of the markers currently established (but not their locations) by using the E(nvironment option of the S(et command.

# The F(ind command

Quite often you will want to jump to a place in a Workfile where you have not previously thought to leave a marker. If you remember at least a small part of the contents of the text near that place, you can easily get there using the F(ind command. To see how the F(ind command works, using the EDITDEMO file as an example, first J(ump to the B(eginning of the Workfile (again leading to the display of Figure 4-1). Now press "F" to enter the F(ind command's world, with the promptline:



>**Find[1]: L(it <target>** =>

The Editor now waits for you to type in a "pattern" string of characters which will be the "target" of a fast search through the Workfile. Before proceeding further, make sure that the Editor's direction flag points forward (">") as shown in the promptline above. If it does not, press ESC, change the direction flag by pressing ">", and again enter the F(ind command.

As an example, respond to the prompt shown above by typing in:

**/BEGIN/**

The two characters "/" serve to bracket or "delimit" the string of characters which are to be found in the Workfile, and they are not included in the target. You can use any special character as a delimiter, including either the single or double quote symbols. We use the right slash "/" because it is conveniently located on the keyboard, and it rarely is included in the target of a F(ind command.

Note that the F(ind command distinguishes between upper case and lowercase characters. If you typed in "begin" or "Begin" rather than "BEGIN", the command will respond by telling you that it could not find any occurrence of the target string.

As soon as you press the delimiter key ("/" in this example) for the second time, the F(ind command will start searching through the Workfile looking for an occurrence matching the target string you have typed in. If all goes well in our example, the F(ind command will complete its work and the cursor will be left pointing at the end of the sixth line of the Workfile's text, immediately following the target pattern "BEGIN".

If at any time, while typing in your desired target string, you decide that you wish to terminate the F(ind command so as to start over again, just press the ESC(ape key.

## Multiple occurrences of the target

In many cases, you will pick a target string that occurs more than once in the Workfile. The F(ind command starts searching (in the direction shown by the Editor's direction flag) from the current position of the cursor. After F(inding the first occurrence of the target, the cursor will be displayed immediately following that target. You may well be looking for a later re-occurrence of the same target. It is simple enough to repeat the same F(ind command at this point, again typing in the same target string. However, there is an easier way.

Continuing our example with the target "BEGIN", again press "F" to enter the F(ind command. Now simply press the "S" key, and note what happens. The cursor will jump to the end of the next occurrence of "BEGIN" in the Workfile. Keep doing this several times, noting what happens. Once the last occurrence of "BEGIN" has been found in the Workfile, an "error" message will appear in the promptline at the top of the screen. In further uses of the F(ind command using the same target, the Editor will refuse to move the cursor any further.



Now J(ump back to the B(eginning of the Workfile. This time, press "2" before pressing "F" to enter the F(ind command. Note that the F(ind command's prompt now appears as follows:

>**Find[2]: L(it <target>** =>

with the digit "2" appearing within the square brackets. This is the F(ind command's "repeat factor", showing how many times the search for the target string will be repeated once typing of the target has been completed. You can type in any whole number as a repeat factor before typing "F".

# The F(ind command

This feature is not designed to make it convenient to type in large repeat factors, and the value of the repeat factor will not be shown on the screen until the F(ind command's own promptline is displayed.

You can get the result of using a very large repeat factor without having to think about its value by typing "/" as a repeat factor. Upon entering F(ind the resulting prompt will be:

>**Find[/]: L(it** <**target**> =>

The result of doing this should be to F(ind the last occurrence of the target within the Workfile.

## F(inding backwards

As mentioned earlier, the F(ind command conducts its search in the direction shown by the Editor's direction flag. So far we have been F(inding only in the forward direction. Now J(ump to the E(nd of the Workfile, and then set the direction flag to backwards ("<") by pressing the "<" key. Next, use F(ind for the S(ame target, noticing that the cursor stops at the end of the last occurrence of "BEGIN". Use F(ind followed by S(ame again (without repeat factor) and note that the cursor does not move. This is because the search starts from the current cursor position, and of course the first occurrence of the target found in the backwards direction is the one already adjacent to the cursor's position.

To perform a multiple search in the backwards direction, you may find any of several tactics useful. After stopping at one occurrence of the target, you can get to the next previous occurrence by using a repeat factor of 2. Another possibility is to use the up arrow once, thus placing the cursor in the line above

the one where an occurrence of the target has just been found. This has the effect of putting the cursor at a point in the Workfile before the target's occurrence just found, and another application of F(ind followed by S(ame will no longer encounter that occurrence. Just as one can use the "infinite" repeat factor [/] to F(ind the last occurrence of the target when going forward, you can use the same repeat factor when going backwards to F(ind the first occurrence of the target starting from the end of the Workfile.

## L(iteral Targets vs Tokens

Unless you use the L(it option of the F(ind command (before typing in the first delimiter of your target) the Editor will assume that you want to locate a target consisting of one or more "tokens". A token may be a complete word, a number, a special (punctuation) character, or an "identifier". As in Pascal, and many other programming languages, an identifier is defined as a string of characters which must start with a letter, and thereafter may consist of additional letters or digits. For example,

```
A
abc
n123
X25p
```

are all identifiers in this context. In the F(ind command, the Editor distinguishes between upper case and lower case letters, regardless of the rules on this subject for any programming language. Thus:

```
BEGIN
begin
Begin
beGIn
```

are regarded as four different targets.

# The F(ind command

The F(ind command permits you to string together several different tokens into a single target. Moreover, it is indifferent to the number of blank space characters between tokens in the Workfile. For example, a target typed in as:

```
S:STRING;
```

would be matched in the Workfile by any of the following:

```
S:STRING;

S : STRING ;

S: STRING;

S:        STRING;
```

or even:

```
S:
STRING;
```

In the last of these examples, the target appears in pieces shown on two successive lines. For the purposes of the F(ind command, the end-of-line marker spearating two successive lines is to be regarded as equivalent to one blank character. (This is equivalent to the definition of end-of-line marks in a file of Type TEXT in Pascal.)

# The F(ind command

Now to understand the distinction between a token and a L(iteral target, J(ump back to the B(eginning of the EDITDEMO Workfile. Enter the F(ind command and use the target:

**/PROC/**

noting that the Editor will claim that this target cannot be found. This is because the target typed in does not match any complete token in the Workfile. Now enter F(ind again, this time pressing the "L" key followed by the "S" key. The cursor will come to rest pointing at the character "E" within the first occurrence of "PROCEDURE". The L(iteral option of the F(ind command tells the Editor to look for a target string which exactly matches the target that you type in. In the L(iteral option, blank characters count exactly as they are found, and all of the target string examples shown in the group just above will be regarded as different.

# The "=" key command

After F(inding a target that you want, it may sometimes be more convenient to have the cursor placed where the target begins rather than at its end. Press the "=" key, when in this situation, and the cursor will be moved to the beginning of the target. In fact the "=" key command serves as the equivalent of a J(ump to beginning of target key even after you have used several other cursor movement commands following the F(ind. (However, the destination of the "=" key command will change to the beginning of the most recent insertion if you use the I(nsert command.)

# The V(erify command

The V(erify command is used to re-display the contents of the Editor's window, placing the cursor as near the center of the screen as makes good sense. Occasionally, the Editor will lose track of characters displayed on the screen which should have been moved. If you have any doubt about the correctness of the displayed text following any command that changes the content of the Workfile, use the V(erify command to get a fresh display. The window displayed by the V(erify command will be a correct representation of the text stored in the computer's memory.

## Commands that Change the Workfile's contents

All of the commands described in this section are designed for use in changing the contents of the workfile copy currently stored in the computer's main memory. All of the commands described in the previous section are used for moving the cursor from place to place in the workfile, but they do not change the content of the workfile. Some of the commands described in this section are designed so that you can change your mind after altering the workfile contents, and can return to the status of the workfile as it was just before the command was entered.

Remember that the changes you make using the commands described in this section affect only the copy of the workfile in the computer's active memory. They have no effect on any copies stored on a disk. Changes on the disk are only made through the Q(uit command which is described in the next major section of this chapter. In general it is a good idea to save the results of your editing changes in the workfile on the disk periodically, (e.g. once every ten minutes or so). If instead, you work without saving the workfile for a long period, you leave yourself vulnerable to losing all your work during that period if the electric power should fail. Since the main memory retains its stored information only as long as the electric power is maintained, even a momentary failure of the electric power could result in loss of your work. If you save your work every ten minutes or so, you will only lose a few minutes worth of work if the power fails.

# I(nsert

The I(nsert command puts the Editor in a mode allowing you to type information into the workfile. All text characters typed in while in the I(nsert command's world become part of the workfile stored in main memory if you terminate the I(nsert using Ctrl-C. If, after I(nserting a substantial amount of text, you decide to back up and start over again, the ESC key allows terminating the I(nsert command without saving anything typed in since the command was last entered.

The entry of information typed in while in the I(nsert command's world starts at the position where the cursor points when the I(nsert command is entered. As noted in the previous section, the cursor's logical position can never be to the left of the left-most nonblank character on a line, nor to the right of the position immediately following the right-most nonblank character on a line. If the cursor's position is between those two limits when you enter the I(nsert command, the Editor will split the characters already in the same line. The portion starting at the cursor's position, when I(nsert is entered, will be pushed as far as possible to the right side of the screen. This is illustrated in Figure 4-6.

```
>Insert: Text {<bs> a char,<del> a line} [<etx> accepts, <esc> escapes]
PROGRAM EDITDEMO;

PROCEDURE REPEAT1;
VAR S,SG:STRING;
    L,N:INTEGER;
BEGIN
  WRITELN(
    'TYPE ANY STRI_                          NG FOLLOWED BY <ent>'
       );
  READLN(S);
  N:=1;
  L:=LENGTH(S);
  REPEAT
    SG:=COPY(S,1,N);
    WRITELN(SG);
    N:=N+1;
  UNTIL N>L
END (*REPEAT1*);

PROCEDURE REPEAT2;
VAR S:STRING;

PROCEDURE REVERSE;
  (*REVERSE THE ORDER OF CHARACTERS
```

Figure 4-6. I(nsert command at entry.



Now if you type characters into the gap within the split line, the display will remain stable unless you type in enough characters to fill up the gap. Figure 4-7 shows what happens when you type in more characters than will fit within the gap.

# I(nsert

```
>Insert: Text {<bs> a char,<del> a line} [<etx> accepts,<esc> escapes]
PROGRAM EDITDEMO;

PROCEDURE REPEAT1;
VAR S,SG:STRING;
    L,N:INTEGER;
BEGIN
  WRITELN(
    'TYPE ANY STRIxyzxyzxyzxyzxyzxyzxyzxyzxyzxyzxyzxyzxyz_
                                           NG FOLLOWED BY <ent>'
```

Figure 4-7. After filling the gap on entry line.

Notice that the right side of the line where the cursor started remains on the screen, but it has been moved down one line to make room for additional text to be typed in. Notice also that when this happens all the subsequent lines in the workfile are removed from the screen. This should be of no concern to you. These subsequent lines of text are still stored in the computer's memory. They have been removed from the screen simply to make room so that you can type in as many lines of additional text as you like.

To continue typing at the beginning of the next line below, press the ENTER key. Notice that the result of doing this is to place the cursor immediately below the left-most character on the line from which the ENTER key was pressed. Figure 4-8 shows the result of typing in "xx" immediately after pressing ENTER following the state of the workfile shown in Figure 4-7.

```
>Insert: Text {<bs> a char,<del> a line} [<etx> accepts,<esc> escapes]
PROGRAM EDITDEMO;

PROCEDURE REPEAT1;
VAR S,SG:STRING;
    L,N:INTEGER:
BEGIN
  WRITELN(
    'TYPE ANY STRIxyzxyzxyzxyzxyzxyzxyzxyzxyzxyzxyzxyzxyz
    xx_
                                        NG FOLLOWED BY <ent>'
```

Figure 4-8. I(nsert after ENTER followed by "xx".

In some cases you may not wish the indentation of a new line to be the same as that of the line from which ENTER is pressed. You can change the indentation if you press either SPACE or BACKSPACE immediately following ENTER and before typing in any visible text. Once you have typed in any visible character at the beginning of the line, use of SPACE or BACKSPACE on that line will no longer affect the indentation.

Quite often you will make errors while typing in characters. If you have not typed in too many characters following an error, the easiest remedy is to use BACKSPACE to remove the offending characters from the screen. Each time you press BACKSPACE while in the I(nsert world, one character previously typed in will disappear from the screen. Naturally, after all characters typed in during the present I(nsert command have been removed, no additional characters that were previously there will be removed. You can use the D(elete command to get rid of characters displayed when in the "Edit:" world.

The DELETE-LINE Key (Ctrl-BACKSPACE) can be used when in the I(nsert command's world as an express version of the BACKSPACE key. Each time DELETE-LINE is pressed, you remove the entire line where the cursor is located, and the cursor returns to the end of the previous line. The DELETE-LINE key cannot be used to remove the line where the cursor was located when I(nsert was entered, and an error message will appear on the promptline if you try to do this.

Occasionally, you may have reason to type in enough characters on one line to cause part of the line to be displayed beyond the right limit of the screen, if that were possible. If you do this, the editor will notify you of the problem by displaying an exclamation point ("!") at the right margin of the screen. The portion of the text that cannot be displayed on that line is still stored in the computer's memory. To get it displayed again on the screen, you may wish to split the line into two by I(nserting an ENTER in the middle of the line. Another possibility is to shift the whole line to the left resulting in less indentation. This can be done with the A(djust command, which is explained in a later section.

A common inadvertent error is the attempt to type in nonvisible control characters like the cursor positioning arrows. The result of doing this will be the display of question mark characters. You can erase these characters in the usual manner, as with any other errors.

# D(elete

The D(elete command is used to remove characters from the text stored in the workfile copy in the computer's memory. After entering the D(elete command, you can move the cursor using any of the cursor moving commands described in this chapter, i.e. the arrow commands and their relatives. J(ump and F(ind do not work within the D(elete command's world.

We can refer to the position of the cursor, when the D(elete command is entered, as the "entry postion". Upon moving the cursor to another position, note that all the characters between the new position and the entry position are erased from the screen. As an example, consider the first window displayed for the EDITDEMO workfile, as shown in Figure 4-1. Place the cursor so that it points at the "R" in "REPEAT" in the third line of the program. Now enter D(elete, then press the space bar three times. The result should be as shown in Figure 4-9.

# D(elete

```
>Delete: < > <Moving commands> {<etx> to delete, <esc> to abort}
PROGRAM EDITDEMO;

PROCEDURE  EAT1;
VAR S,SG:STRING;
    L,N:INTEGER;
BEGIN
  WRITELN(
    'TYPE ANY STRING FOLLOWED BY <ent>'
        );
  READLN(S);
  N:=1:
  L:=LENGTH(S);
  REPEAT
    SG:=COPY(S,1,N);
    WRITELN(SG);
    N:=N+1;
  UNTIL N>L
END (*REPEAT1*);

PROCEDURE REPEAT2;
VAR S:STRING;

PROCEDURE REVERSE:
  (*REVERSE THE ORDER OF CHARACTERS
```

Figure 4-9. D(elete after pressing the space bar three times.

Now press the BACKSPACE key several times, noting that characters backed over in this fashion reappear on the screen.

As with the I(nsert command, you can terminate the D(elete command in either of two ways. Press Ctrl-C to complete the job of removing the text enclosed between the entry position and final position of the cursor (as established during use of D(elete). Press ESC to terminate D(elete in such a way as to leave the workfile just as it was before the D(elete was entered.

# R(eplace

The R(eplace command is an extension of the F(ind command. For details on the F(ind command, see "The F(ind Command" in this chapter. Upon entering the R(eplace command, the following promptline is displayed:

**>Replace[1]: L(it V(fy <targ> <sub> =>**

The bracketed number and "L(it" have the same meanings as they do with the F(ind command. The "targ" is an abbreviated reference to the same target as used with F(ind. As in the F(ind command, after typing in the target explicitly once, subsequent uses of the same target can be made by using "S" for S(ame. In fact, you can use F(ind to establish the target, and then use the S(ame option with R(eplace to refer to the same target. Similarly, you can establish the target explicitly using R(eplace, and then use F(ind with the S(ame option to refer to the same target.

After R(eplace carries out the same search as carried out by F(ind, it deletes the found occurrence of the target and then inserts the substitution string indicated by "sub". As an example, you might respond to the prompt shown above by typing in:

**/BEGIN//START/**

with the result that the first occurrence of "BEGIN" will be changed to "START". Although both target and sub strings are of the same length in this case, they need not be of equal lengths. In fact the substitution string can be of zero length, with the result that the target string will simply be deleted from the workfile after it is found.

# R(eplace

As with the F(ind command, you can use a repeat factor with R(eplace. Use the slash repeat factor [/] to change all occurrences of the target to the sub string.

Quite often you will want to change some occurrences of the target string but not all of them in the workfile. You can change the occurrences that are found by R(eplace selectively by using the V(erify option (abbreviated V(fy to keep the promptline as short as possible). The "V" must be included in your response to R(eplace's prompt before you type in the substitution string. A convenient way to go through most of your workfile with R(eplace, selectively changing only some occurrences of the target, is to use the slash repeat factor [/] and also the V(erify option. Each time a new occurrence of the target is found, the following prompt will appear:

>**Replace[/]:<esc> aborts, R replaces, ' ' doesn't**

You then have three options. Press "R" to complete the replacement of that occurrence with the substitution string, and to cause the cursor to move on to the next occurrence of the target. Press the SPACE bar to bypass the substitution, but allow the search to continue for the next occurrence of the target. Press ESC to cause the R(eplace command to be terminated at that point without either making the substitution or continuing the search.

The C(opy command is used to insert passages of text that have previously been saved into the workfile at the cursor's position. The C(opy command's prompt is as follows:

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

showing that the command has two distinct options.

The B(uffer option is used together with a passage of text that is automatically saved in a "buffer" area of the computer's memory whenever you use either the I(nsert or the D(elete command. Each use of I(nsert or D(elete saves the associated passage of text in the buffer area, removing the previously saved contents of the buffer. After entering C(opy, press "B" for B(uffer to have a copy of the buffer's saved contents inserted in the workfile at the place where the cursor points.

It is important to note that the buffer is filled with a new passage of text whether you terminate the I(nsert or D(elete using either Ctrl-C or ESC. Thus it is possible to mark a passage of text within the workfile using the D(elete for later C(opying, but to leave the original passage intact by terminating the D(elete using ESC.

As an example of the use of the B(uffer option, J(ump to the B(eginning of the EDITDEMO workfile, then carry out the following steps. (The display should be the same as shown in Figure 4-1). First, move the cursor to point to the "V" in "VAR" on the fourth line of the display. Now enter D(elete

and press ENTER twice. The result will be to blank out the two lines:

```
VAR S,SG:STRING;
    L,N:INTEGER;
```

from the display. Next, press ESC, with the result that the display will again appear as in Figure 4-1. Now move the cursor up two lines, so that it points at the left end of the blank line between "PROGRAM ..." and "PROCEDURE ...". Next, press "C" for C(opy followed by "B" for B(uffer. The result should be as shown in Figure 4-10.

```
>Edit: A(djst C(py D(let F(ind I(nsrt J(mp K(ol R(plc Q(uit X(ch Z(ap [E.7h]
PROGRAM EDITDEMO;
VAR S,SG:STRING;
    L,N:INTEGER;

PROCEDURE REPEAT1;
VAR S,SG:STRING;
    L,N:INTEGER;
BEGIN
  WRITELN(
    'TYPE ANY STRING FOLLOWED BY <ent>'
        );
  READLN(S);
  N:=1;
  L:=LENGTH(S);
  REPEAT
    SG:=COPY(S,1,N);
    WRITELN(SG);
    N:=N+1;
  UNTIL N>L
END (*REPEAT1*);

PROCEDURE REPEAT2;
VAR S:STRING;
```

Figure 4-10. Display after C(opy of two lines from the B(uffer.

Note that the original two lines following "PROCEDURE ..." still remain in the display. They could have been eliminated in the same operation by terminating the D(elete command using Ctrl-C as usual.

The F(rom file option of the C(opy command is used in a similar manner, but draws its passage of text from another workfile saved on the disk. If you use this option, the Editor will prompt you to type in the name of the file to be copied into the current workfile. You can C(opy just a portion of another workfile starting at one marker and continuing to a second marker saved in that other workfile. Those markers must be established using the Editor while working with the other workfile. It is not possible to set markers in the other workfile without Q(uitting out of the Editor, typically U(pdating the current workfile as you go. The promptline for the F(rom file option is as follows:

>**Copy: From what file[marker,marker]?**

The pair of marker names can be omitted, and the result will be that the entire file whose name is typed in will be C(opied. Enclose two marker names within square brackets, to get only that portion of the named workfile enclosed between the two markers C(opied into the current workfile. Of course, the first marker should be placed earlier in the other workfile than the second marker to make this operation sensible.

# A(djust

The A(djust command is used to shift selected lines of text to the right or left without changing their contents otherwise. Either single lines or groups of lines can be shifted. The A(djust command has several options shown in its promptline as follows:

**>Adjust: L(just R(just C(enter**
**<left,right,up,down-arrows>**

All of the options refer to the line in which the cursor is located. L(just causes that line to be left justified, i.e. to be pushed to the left as far as possible. R(just causes the cursor's line to be pushed as far right as specified by the right margin currently specified for the Editor. (To find out at which columns the right and left margins are currently set, use the E(nvironment option of the S(et command.) C(enter causes the cursor's line to be placed half-way between the left and right margins as currently specified.

The left and right arrows cause the cursor's line to be shifted one position in the indicated direction each time they are pressed. If you want to shift a group of lines by the same amount, start with the top line of the group and shift it the desired amount using the left and/or right arrow keys. Next, press the down arrow once for each additional line that you want shifted. A similar strategy applies by first shifting the bottom line of the group, then using the up arrow for the other lines of the group. The best way to see how this works is to experiment with it.

You terminate the A(djust command using Ctrl-C. There is no means provided whereby you can escape out of the A(djust command in a way that will restore the text to the status it had before the A(djust command was entered.

Sometimes it is necessary to change just a few characters in the workfile on a one-for-one basis. The eX(change command allows you simply to type over characters already in the workfile without going through the complications of using D(elete followed by I(nsert. Press "X" to enter the eX(change command. Figure 4-11 exemplifies how the eX(change command is used.

```
>eXchange: Text <vector keys> {<etx>, <esc> CURRENT line}
PROGRAM EDITDEMO;

PROCEDURE REPEAT1;
VAR S,SG:STRING;
    L,N:INTEGER;
BEGIN
  WRITELN(
    'type any strING FOLLOWED BY <ent>'
        );
  READLN(S);
  N:=1;
  L:=LENGTH(S);
  REPEAT
     SG:=COPY(S,1,N);
     WRITELN(SG);
     N:=N+1;
  UNTIL N>L
END (*REPEAT1*);

PROCEDURE REPEAT2;
VAR S:STRING;

PROCEDURE REVERSE;
  (*REVERSE THE ORDER OF CHARACTERS
```

Figure 4-11. Illustration of the eX(change command.

To reproduce this example, J(ump to the B(eginning of the EDITDEMO workfile as in Figure 4-1. Then move the cursor to point at the "T" at the left end of the long line beginning "TYPE ANY ...". Now enter eX(change by typing "X". Next, type in "type any str".

Each character typed replaces one that had been in the workfile when the eX(change command was entered. You may also use BACKSPACE, ENTER, TAB and the arrows to move the cursor around, non-distructively, in the eX(change mode. When you have no further characters to exchange, press Ctrl-C to make the changes permanent. Press ESC to cancel any changes made so far on the current line and to exit from the eX(change command.

## Z(ap

The Z(ap command is designed to be used following F(ind, R(eplace, and I(nsert commands.

> CAUTION: Do not try to use Z(ap if you follow any one of these three commands with any other command that changes the text of the workfile, or any command that moves the cursor - the results will be hard to predict.

If the most recent text changing command was I(nsert, Z(ap deletes the text that was I(nserted. Thus, if you complete an I(nsertion using Ctrl-C, and then realize that you made a mistake, Z(ap allows you to start over again.

If the most recent command was F(ind, then Z(ap deletes the occurrence of the target string that was found.

If the most recent command was R(eplace, then Z(ap deletes the substitution string from the text of the workfile.

Following Z(ap, you can use the B(uffer option of the C(opy command to restore the text that was deleted by Z(ap. Thus Z(ap provides an "express" method for F(inding a target and then moving it to an alternate place within the workfile. You use F(ind, followed by Z(ap, then move the cursor to an alternate location, and finally use C(opy followed by B(uffer.

If you use a repeat factor with either F(ind or R(eplace, only the most recent target or substitution string will be deleted by the Z(ap command. If you repeat the Z(ap command, it will delete the contents of the C(opy buffer, with the effect that C(opy cannot be used to restore the effect of the first Z(ap of the group. After Z(ap has been used once, repetition will have no effect on the stored text in the workfile (until you use F(ind, R(eplace, or I(nsert again).

# The Q(uit command and its options

The Q(uit command is used to leave the Editor in an orderly way. Use of the Q(uit command is required if you wish to save the results of an editing session in which you have changed your workfile. (You could also terminate an editing session in a more drastic way by Bootloading again, or by withdrawing your disk from the machine and walking away.) The prompt messages that appear for the Q(uit command were shown in Figure 3-6.

The U(pdate option causes the contents of the computer's memory to be saved on the disk in the reserved file SYSTEM.WRK.TEXT, i.e. in the "unnamed" workfile on the disk. Any previously saved file called SYSTEM.WRK.TEXT will be removed from the disk as a result of this action. Having reached the Q(uit command, there is no way for you to change the name of the file SYSTEM.WRK.TEXT, which is already saved on the disk, in order to prevent it from being removed. However, you can use the W(rtie option of the Q(uit command to save the contents of the computer's memory (resulting from the current Editor session) under a different file name. You can then use the E(xit option to prevent the old version of the workfile from being removed from the disk.

The E(xit option terminates the Editor without taking any action at all to save the contents of the computer's memory. You might use the E(xit option after using the Editor to read the contents of a workfile without making any changes. E(xit from the Editor will then have no effect on your disk directory. Since data errors can sometimes be caused in the process of reading information from the disk, or writing to it, it is best to avoid any more disk

operations than necessary. The E(xit option is a facility designed to assist in avoiding disk operations that are not needed.

The R(eturn option is provided for those of us who develop sloppy habits in typing into the Editor. If you hit the "Q" key inadvertently, you may not really have it in mind to terminate the Editor session quite yet. Press "R" for R(eturn to get back into the Editor at the same place you had been just before the Q(uit was invoked.

The W(rite option allows you to save the current contents of the computer's memory in a named disk file. After the disk file has been saved, the Editor session continues.

The W(rite option requests a file name using the following prompt:

**Name of output file (<cr> to return) ->**

You can respond by typing in the name you want the workfile saved under (leaving out ".TEXT") and following with ENTER. If you simply press the ENTER key (referred to here as "*cr*" for Carriage Return), no disk file will be saved and the Q(uit command will be terminated as if you had used the R(eturn option. If you do respond to the W(rite option with a file name, the Editor will notify you when the disk file has been saved, and then will offer you the option of E(xiting or R(eturning to the Editor.

If you are editing a file which existed before the edit session began, the W(rite option will allow you to write the workfile back to disk under the same name. This can be done by simply typing "$" followed by ENTER. If, for example, the name of the file you are

editing is FILE.TEXT, the write option will display the following prompts:

**$ <ret> writes to FILE.TEXT**
**Name of output file (<cr> to return) ->**

You may either respond with "$" followed by ENTER, which will write the workfile to FILE.TEXT (destroying the old FILE.TEXT), or you may specify a file name as described above.

# Using the Editor for Word Processing

The principal difference between the use of the Editor for Word Processing, and for editing programs, is the automatic "filling" of each line of a paragraph. When filling is used, the Editor scans ahead for the right margin, keeping track of the beginning of the last word you typed in. If the Editor detects that the current word you are typing would extend past the right margin, it automatically moves the current word to the beginning of the next line, filling in the rest of the previous line with blanks. To prepare the Editor for paragraph filling, you will need to use the S(et command to change several switches in the Editor's E(nvironment. (A switch is an option which has only two states, such as "Yes" or "No", "True" or "False".) Figure 4-5 shows roughly the display you should get using the E(nvironment option of the S(et command when the Editor is first entered.

As an exercise to see how paragraph filling works, change the A(utoindent option to False, and the F(illing option to True.

Within the E(nvironment sub-command's world, you select an option to be changed by typing its leading character. For example, to change F(illing type "F". The Editor responds by placing the cursor at the first character position of the current value of the option, i.e. over the "F" in "False". (The legend "False" will simultaneously disappear.) Now type "T", and observe that the displayed value of the option changes to "True". Similarly, to change A(utoindent, type "A" followed by "F" (for "False") or "T" (for "True").

Now you can type in a small paragraph. Notice that you do not need to use the ENTER key to get from the end of one line to the beginning of the next. If the last word you try to type into a line cannot fit within the established margins, the Editor will move the word to the beginning of the next line.

With a small paragraph on the screen, now try an insertion in the middle of that paragraph. The Editor will refill all of the lines following the point of the insertion, putting as many words as possible within each line.

Deleting a portion of a paragraph is slightly more complicated. To see how it works, delete several words from the middle of your paragraph. After completing the deletion (with Ctrl-C), notice that the lines within your paragraph have not been refilled to give the appearance of a properly filled paragraph. Instead, you have to call for the filling explicitly by using the M(argin command in the "Edit:" world. After pressing "M" to initiate the M(argin command, the sreen will go blank for several seconds. The paragraph will then be redisplayed with all lines correctly filled.

Within the Editor's E(nvironment switches, the L(eft margin and R(ight margin have the obvious roles of limiting the left and right extent of a paragraph. The P(aramargin (for "Paragraph margin") switch refers to the indentation of the first line of a paragraph. If you want to change the appearance of any single paragraph, use the following steps:

- S(et the E(nvironment switches to the desired values (and leave the S(et command by pressing SPACE).

- Place the cursor at any point within the paragraph to be changed.

- Press "M" for M(argin.

The Editor recognizes the beginning and ending lines of a paragraph by looking for a completely blank line.

As a final note on Word Processing uses of the Editor, be careful not to try using the M(argin command when the cursor is within a table of data, or in some other passage which you do not want to be filled as if it were a paragraph. The Editor will prevent this from happening if F(illing is set False. If you inadvertently work on a Pascal program with F(illing set True, and if M(argin is also inadvertently used, the results become unreadable.

# CHAPTER 5. FILE MANAGER (FILER)

## Contents

Goals for this Chapter ................ 5-3
Overview of Files and the Filer ........ 5-4
Volume Identifiers .................... 5-5
Simplified Titles for disk files ......... 5-6
Naming Conventions to Simplify Work with Groups of Files ................. 5-7
Workfile Commands .................. 5-9
Status checking/setting commands .... 5-17
Shorthand entry of the destination file name .......................... 5-27
Disk to Disk bulk T(ransfer ........... 5-28
T(ransferring only selected files ....... 5-29
Rearranging the files on one disk ...... 5-30
Directory Maintenance Commands .... 5-32
Checking for disk errors and repairing them ...................... 5-42

FILE MANAGER

# NOTES

# Goals for this Chapter

The File Manager (which is affectionately called the "Filer") is the UCSD p-System's principal tool for keeping track of files stored on your disks. The main goal of this chapter is to provide you with a reference summary of how the Filer is used. As in Chapter 4, the order of presentation starts with the facilities you are most likely to use frequently. See Appendix C for an alphabetic summary of the Filer's command , with references to descriptions in this chapter.

The full set of facilities provided with the Filer is extensive, and goes beyond the range of tasks many beginners will need to handle. Following is a list of specific learning goals for beginners. A good grasp of each of these goals will simplify the use of the UCSD p-System considerably.

a) Establish a previously saved file as the current Workfile for use with the Editor and Compiler.

b) S(ave the current Workfile in the disk directory for later use.

c) Create a N(ew clean Workfile without destroying any previous Workfile.

d) R(emove unwanted old files from your disk directory. Eliminate scattered empty areas on the disk if necessary to make room for new files.

e) T(ransfer a file, or group of files, from one disk to another. T(ransfer a .TEXT file to your CONSOLE: display device or to a remote terminal or printer.

f) Establish the current directory D(ate of your disk.

g) Initialize a new or used disk by clearing it of any previous contents using the Z(ero command.

h) C(hange the directory title of an old file so that you can re-use the same title for a new file without losing the old file.

i) Check your disk for possible B(ad-blocks. Use eX(amine to attempt a repair of marginally bad areas of the disk.

j) Use the "wild-card" file title characters "=" and "?" to simplify the use of certain commands applied to whole groups of files.

# Overview of Files and the Filer

A point of confusion for beginning users of the UCSD p-System is that the term "file" has several related but distinct meanings. Within a Pascal program, an identifier declared to be of a Type associated with a file provides a means of referring to an Input/Output device. I will call this identifier the file's "internal" identifier. The device may be a CRT display, a keyboard, a disk drive, a remote terminal, a printer, or any one of many other possible items of peripheral equipment.

If the I/O device is a disk drive (or perhaps a tape drive), it is used for storing and retrieving information recorded on a secondary storage medium such as floppy disk. In this case, there is usually room for storage of many different collections of information, each separately referred to as a "file". A typical file on the disk might be all the Pascal program statements for a single program, or the executable .CODE version of another program, or possibly a collection of data designed to be used by yet another program. To keep track of all the stored files, a "directory" of the files is also stored on

the disk. Each disk has its own directory, which is available to the System only when the disk is actually mounted in a disk drive connected to the computer. The directory is basically a listing of the names of the files on the disk, their locations, the amount of space they occupy, and several other items of importance mainly to the System itself.

## Volume Identifiers

To distinguish between separate disks, each disk is given a "volume identifier", i.e. a name for the disk itself, which is also stored in the directory. The volume identifier should begin with a letter, and may consist of a total of up to 7 letters and digits. The full "external" name of a disk file consists of the volume identifier followed by the directory name of the file. So that the System may distinguish between the volume identifier and the directory name, the two are separated by a colon character (":"). For example, you might have a disk with a volume identifier of "CLASS", and on that disk a file with a directory name of "TESTPROG.TEXT". The full title of the file would therefore be:

**CLASS:TESTPROG.TEXT**

Volume identifiers in the UCSD p-System refer not only to disks but also to peripheral devices which have no directories. For example, the principal CRT display has the volume identifier "CONSOLE:". Since that device has no directory and is not subdivided logically into files, its full title is:

**CONSOLE:**

with no characters to the right of the colon character. Your IBM Personal Computer also has

facilities to handle the predeclared volumes:

**PRINTER:**
**REMOUT:**
**REMIN:**

where REMOUT: and REMIN: are used for communicating with a remote terminal device or telephone line. You can get a listing of the volumes currently available on your machine by using the Filer's V(olumes command.

## Simplified Titles for disk files

The UCSD p-System allows you to refer to a disk file (either by typing-in a response to a prompt, or by executing a Pascal program statement that refers to the file) without specifying the volume identifier explicitly in certain common circumstances.

If the reference to a file title lacks a volume identifier entirely, the System assumes that the volume intended is the "default volume". When you Bootload, the disk volume containing the UCSD p-System operating system (the file SYSTEM.PASCAL) is initially considered to be the default volume. If the volume identifier of the disk from which you Bootload is

**MYDISK:**

and you have a file called

**FIRSTPROG.TEXT**

on that disk, then that file can be referred to by either of the following title strings:

**MYDISK:FIRSTPROG.TEXT**

**FIRSTPROG.TEXT**

with the same results. You can use the Filer's P(refix command to change the default volume identifier to

some disk other than the one from which you Bootloaded. Then it will still be possible to refer to the same file on MYDISK: by either of the following:

**MYDISK:FIRSTPROG.TEXT**

***FIRSTPROG.TEXT**

where the "*" is interpreted by the System as a substitute for the Bootload disk's volume identifier. These conventions are designed to reduce the amount of typing you need to do to refer to files on several different disks, particularly when using the Filer.

## Naming Conventions to Simplify Work with Groups of Files

Since you will often have to work with a disk that has dozens of directory entries, it sometimes becomes desirable to perform similar operations on many directory entries all at once. The Filer provides several naming conventions that simplify these operations. One common tactic is to construct the directory titles of all files belonging to a related group of files using the same prefix. For example, most of the files provided to you as part of the system software of the UCSD p-System are identified by the prefix:

**SYSTEM.**

and include the files:

**SYSTEM.COMPILER**
**SYSTEM.PASCAL**
**SYSTEM.EDITOR**
**SYSTEM.FILER**

and so on. The period character (".") is included to make the file titles more readable to humans, and has

no special significance to the System. Because some users of the System are accustomed to using other separator characters for this purpose, you can also use the characters "/", "\", "-", and "_" in file directory titles.

Some of the Filer's commands permit selective reference to all files which have the same prefix or suffix. If one of these commands prompts for a file title, and you respond with

**SYSTEM=**

the command would refer to all of the files in the list shown above. The character = is a "wild card" which substitutes for any characters in a title following the prefix "SYSTEM". In the list of files shown above, you could equally well have used

**SYS=**

or

**SYSTEM.=**

with the same results. If you have a mixture of files with .TEXT and .CODE suffixes, the generalized title:

**=.TEXT**

or

**=TEXT**

would refer only to the .TEXT files and not to the .CODE files.

As an extension of the wild card concept, the Filer allows you to substitute the question mark ? for the equals sign =. The Filer will then halt upon reaching each directory entry associated with the prefix or

suffix, and ask whether you wish the command to apply to that particular entry. You respond with Y for Y(es if you wish the command to apply, and with N for N(o (or any other character) if not. The Filer will then continue searching through the directory looking for additional titles matching the wild card specification.

If you leave out the prefix and suffix in a file title, using only the = or ? character, the Filer will refer to each and every title in the directory.

Another possibility is to "sandwich" the wild card character between prefix and suffix, as in:

**PREF=SUFF**

or

**MINE?TEXT**

The wild card naming conventions apply to the following commands: L(ist, E(xtended-directory, T(ransfer, R(emove, and C(hange.

# Workfile Commands

The Workfile concept is designed to simplify the number of steps a user of the UCSD p-System has to take in editing, compiling, and testing new programs. The Filer's Workfile commands are tools for handling the disk directory entries associated with the temporary "unnamed" Workfile and with saved Workfiles on the disk.

Each Workfile may have a .TEXT part and a .CODE part. The .TEXT part is an independent disk file containing, in the case of computer programs, the Pascal or other source programming language statements of one program. The .TEXT file is

produced by the Editor as the result of an editing session. The .CODE part of a Workfile is another disk file containing executable code generated by the Compiler based on translation of the source language statements in a .TEXT file.

When you finish an Editor session, using the U(pdate option of the Q(uit command, the Editor leaves on disk a file called

**SYSTEM.WRK.TEXT**

which is what we are calling the "unnamed" or temporary Workfile. The file title "SYSTEM.WRK.TEXT" is reserved by the System for this use. If you then use the Command: world's C(ompile command, the Compiler will be invoked, and it in turn will look for a file called SYSTEM.WRK.TEXT as its input. If the Compiler succeeds in translating the source language statements from the .TEXT file into executable code, it will leave on disk a file called

**SYSTEM.WRK.CODE**

In the typical program development situation, you can then R(un the program from the Command: world, and will probably discover that further alterations are needed in the source statements of the program. You re-enter the Editor using the Command: world's E(dit command, and the Editor automatically assumes that you want to work with the file SYSTEM.WRK.TEXT if it is present on the disk.

As you can see, the two reserved file names, SYSTEM.WRK.TEXT and SYSTEM.WRK.CODE provide a means of communication among the Editor, the Compiler, and the R(un command of the System's Command: world. When these files are present you do not need to respond to the E(dit, C(ompile, or R(un commands with any file name,

because it is assumed that you want to use the files with the reserved titles.

Of course you will eventually reach the stage where you want to save a version of the .TEXT and .CODE files you have been working with in order to develop a different program. At that point, you enter the Filer, and use the S(ave command. The S(ave command asks for a name, which in practice can be up to 10 characters long. The Filer assumes that you want to retain the .TEXT and .CODE suffixes in these file titles respectively, resulting in titles that are up to 15 characters long. You might respond with the name "PROBLEM1", followed by pressing the ENTER key, at which point the directory entry for SYSTEM.WRK.TEXT is changed to show its title as

**PROBLEM1.TEXT**

and similarly the entry for SYSTEM.WRK.CODE is changed to

**PROBLEM1.CODE**

The Filer now retains "PROBLEM1" as the (simplified) title of your current Workfile. You could verify this by using the Filer's W(hat command which displays the title of the current Workfile.

At this point, the disk no longer contains any files called SYSTEM.WRK.TEXT or SYSTEM.WRK.CODE. However you could Q(uit from the Filer, and use R(un in the Command: world, with the result that the System will start execution of the named Workfile, i.e.

**PROBLEM1.CODE**

If you enter the Editor, using the Command: world's E(dit command, the Editor will load the contents of

PROBLEM1.TEXT into the computer's memory in preparation for an Editing session. If you then make editing changes in this Workfile, or even if you don't, use of the U(pdate option of the Editor's Q(uit command will result in the creation of a new text file on disk called:

**SYSTEM.WRK.TEXT**

That file will be separate and independent of the file PROBLEM1.TEXT until or unless you return to the Filer and use the S(ave command. The S(ave command will then offer you the option of retaining the new temporary Workfile under the name PROBLEM1.TEXT. If you respond with Y(es, the old file under that same name will be removed from the disk, and the temporary Workfile will be given that name instead of SYSTEM.WRK.TEXT.

Of course many other scenarios are possible. You should find it helpful to experiment with the Filer's four Workfile commands to gain a better understanding of how the Workfile is used. Use the L(ist directory command, described in Chapter 4, to observe the directory changes that result from using the Workfile commands in conjunction with the Editor and Compiler.

Use the G(et command to establish an existing file (or a .TEXT and .CODE pair) as the current Workfile. If no files called SYSTEM.WRK.TEXT or SYSTEM.WRK.CODE are currently present in the disk directory, G(et will simply ask for the name of the Workfile you want "loaded" into memory. For example, your disk directory might include the files:

**SNAP.TEXT**
**SNAP.CODE**

When the Filer responds to G(et with the prompt:

**Get what file?**

you might answer by typing:

**SNAP**

followed by ENTER. The Filer will respond with:

**Text and Code file loaded**

if it finds the Workfile, or:

**No file loaded**

if not. If SYSTEM.WRK.TEXT or SYSTEM.WRK.CODE already exists in the disk directory, then G(et will respond with:

**Throw away current workfile?**

If you answer with Y(es, then those files will be removed from the disk, and the System tables will be

# G(et

updated to show that the SNAP files are now to be regarded as the current Workfile. If you answer with N(o (or anything other than Y(es), the G(et command will terminate with no effect, and the Filer's main prompt line will reappear.

If you press G for G(et inadvertently, and wish to return to the main level of the Filer, answer the prompt simply by pressing ENTER.

# S(ave

---

Once you have created a new temporary .TEXT Workfile with the Editor (i.e. SYSTEM.WRK.TEXT) and/or a new temporary .CODE Workfile with the Compiler (i.e. SYSTEM.WRK.CODE) the S(ave command can be used to give those files permanent directory names. If the temporary files are actually on the disk, the S(ave command's prompt will be:

**Save as what file?**

to which you might answer:

**WORK2**

followed by ENTER. As a result, a directory entry for SYSTEM.WRK.TEXT will become WORK2.TEXT,

and SYSTEM.WRK.CODE will become WORK2.CODE.

> **Note:** Any old files called WORK2.TEXT and WORK2.CODE will automatically be removed from the disk by this process, and replaced with the new version of the Workfile. So, make sure you create a new name, or are not concerned with any older versions of the files of the same name.

# N(ew

The N(ew command has the effect of clearing out the current Workfile recognized by the System so that you can begin creating a completely new text file with the Editor. If there is a SYSTEM.WRK.TEXT or SYSTEM.WRK.CODE on the disk when N(ew is entered, the Filer prompts with:

**Throw away current workfile?**

If you answer Y(es, then both the .TEXT and .CODE file are removed from the disk. If you answer with N(o, or any key other than "Y", then the N(ew command is terminated without having any effect.

N(ew has no effect on the actual disk files associated with a current Workfile name. As in the examples presented in earlier subsections, you might have a

# N(ew

named Workfile SNAP active and associated with the disk files SNAP.TEXT and SNAP.CODE following use of the G(et command. You might then try to use N(ew, whether inadvertently or intentionally. Even if the temporary version(s) of the Workfile in SYSTEM.WRK.TEXT and SYSTEM.WRK.CODE were thrown away by this process, the original files SNAP.TEXT and SNAP.CODE would not be touched by the N(ew command.

# W(hat

---

The W(hat command is used to display the name of the currently active Workfile. If the files SYSTEM.WRK.TEXT and/or SYSTEM.WRK.CODE are present, but no G(et operation has been done on a named Workfile, then the Filer will respond with:

**Workfile is not named (not saved)**

If a G(et has been performed on a named Workfile called SNAP, then the response to W(hat will be

**SNAP**

If neither named or unnamed Workfiles are present, then the response to W(hat will be

**No workfile**

# Status checking/setting commands

The principal source of information about the status of files in the UCSD p-System is the directory of files stored on each disk. Content of the directory can be displayed using the L(ist directory and E(xtended directory commands of the Filer.

The other commands in this group provide supplementary information on the V(olumes currently accessible to I/O operations, on D(ate stored in the System disk from which you Bootloaded, and on the default. P(refix volume name currently in force.

## L(ist directory

The L(ist command normally is used to display part or all of the directory of a selected disk volume. The prompt to this command is

**Dir listing of what vol?**

You can answer with an abbreviated volume name, or a complete volume name. You can also provide an optional destination name requesting that the directory listing be sent to some device other than the principal console device of your machine. For example, the optional destination might be a printer connected to the REMOUT: I/O port.

# L(ist directory

To L(ist the content of the System disk from which you have Bootloaded, respond to the prompt by pressing the colon key (":") followed by ENTER. The resulting display will have roughly the appearance shown in Figure 5-1.

```
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate,? [C.11]
STARTUP:
SYSTEM.PASCAL         121  15-Nov-81
SYSTEM.MISCINFO         1  15-Nov-81
SYSTEM.INTERP          26  15-Nov-81
SYSTEM.FILER           32  15-Nov-81
SYSTEM.EDITOR          49  15-Nov-81
SYSTEM.STARTUP         12  15-Nov-81
SYSTEM.SYNTAX          14  15-Nov-81
NAMEFILE                3  15-Nov-81
SCDEMO.CODE             3  15-Nov-81
COPYSCUNIT.CODE         4  15-Nov-81
UPDATE.CODE             4  15-Nov-81
COMPDEMO.TEXT           6  15-Nov-81
EDITDEMO.TEXT           4  15-Nov-81
UPDATE.TEXT             8  15-Nov-81
14/14 files<listed/in-dir>, 293 blocks used, 27 unused, 27 in largest
```

**Figure 5-1. Display of the L(ist directory command.**

If the directory is too long to be listed at once on the CRT screen, the Filer will stop after filling the screen. It will then prompt you to press the SPACE bar to continue, i.e. to display the next group of directory entries. If, at this point, you wish to terminate the L(ist command and leave displayed the partial directory already on the screen, press the ESC key instead.

Sometimes you may have too long a directory to be listed at once on the screen, but may wish to L(ist only selected file titles from the entire directory. For example, you might wish to display only the titles of

the .TEXT files on your disk. You can do this, when you respond to the L(ist command's prompt, by following the "wild card" naming conventions described in "Workfile commands" in this chapter as in:

**=.TEXT**

This will produce the display shown in Figure 5-2 based on the directory contents shown in Figure 5-1.

```
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate,? [C.11]
STARTUP:
COMPDEMO.TEXT          6  15-Nov-81
EDITDEMO.TEXT          4  15-Nov-81
UPDATE.TEXT            8  15-Nov-81
3/14 files<listed/in-dir>, 24 blocks used, 27 unused, 27 in largest
```

Figure 5-2. L(ist of =.TEXT files only.

If you want to L(ist the directory of a disk other than the one from which you bootloaded, then give its volume name. For example, to list all of the directory of a disk called OTHER, answer the L(ist command's prompt with:

**OTHER:**

followed by ENTER. If you want to L(ist only the file titles prefixed by SYSTEM on that disk, answer with:

**OTHER:SYSTEM=**

followed by ENTER.

Sometimes it is useful to have a copy of the directory for one of your disks printed out on paper. If you

have a teleprinter connected to the REMOUT: port of your computer, and wish to list out the directory of the disk called SNAP:, then answer the L(ist command's prompt with:

**OTHER:,REMOUT:**

followed by ENTER.

# V(olumes

The V(olumes command will display a list of the identifiers of I/O volumes currently available to programs running on your machine. Figure 5-3 shows an example:

```
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate,? [C.8]
Vols on-line:
  1      CONSOLE:
  2      SYSTERM:
  4 #    STARTUP:
  5 #    PASCAL:
  6      PRINTER:
  7      REMIN:
  8      REMOUT:
Root vol is - STARTUP:
Prefix is   - STARTUP:
```

Figure 5-3. Example of display by V(olumes command.

The numbers shown on the left of this list are the logical numbers of the I/O units. You can refer to any unit by substituting for the volume identifier with an entry like this:

**#4:**

which refers to the disk in unit 4. Normally, the volume names of your floppy disk will be found in units 4 and 5 in this display. The UCSD p-System provides space for additional floppy disk drives starting at unit 9. I strongly suggest that you avoid using the unit number designation for referring to disks with the Filer, since doing that gives you no protection if you happen to have a different disk than you thought in the drive.

# E(xtended directory list

The E(xtended directory command is similar to the L(ist command but provides more information in its display, as shown in Figure 5-4.

```
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate,? [C.11]
STARTUP:
SYSTEM.PASCAL     121  15-Nov-81    6  512  Datafile
SYSTEM.MISCINFO     1  15-Nov-81  127  512  Datafile
SYSTEM.INTERP      26  15-Nov-81  128  512  Datafile
SYSTEM.FILER       32  15-Nov-81  154  512  Codefile
SYSTEM.EDITOR      49  15-Nov-81  186  512  Codefile
SYSTEM.STARTUP     12  15-Nov-81  235  512  Codefile
SYSTEM.SYNTAX      14  15-Nov-81  247  512  Datafile
NAMEFILE            3  15-Nov-81  261  512  Datafile
SCDEMO.CODE         3  15-Nov-81  264  512  Codefile
COPYSCUNIT.CODE     4  15-Nov-81  267  512  Codefile
UPDATE.CODE         4  15-Nov-81  271  512  Codefile
COMPDEMO.TEXT       6  15-Nov-81  275  512  Textfile
EDITDEMO.TEXT       4  15-Nov-81  281  512  Textfile
< UNUSED >          2             285
UPDATE.TEXT         8  15-Nov-81  287  512  Textfile
< UNUSED >         25             295
14/14 files<listed/in-dir>, 293 blocks used, 27 unused, 25 in largest
```

Figure 5-4. E(xtended directory display corresponding to Figure 5-1.

# E(xtended directory list

This display includes information showing where on the disk each file begins (block number) and on the kind of information stored in each file. These items are primarily of use to advanced programmers.

One other item shown by the E(xtended directory will often be of assistance to readers of this book. Note that Figure 5-4 shows entries marked as *unused* along with their sizes and starting block locations on the disk. These are areas on the disk where there are no files currently allocated. They are areas that potentially could be used for additional files not yet stored on the disk. You will find that successive editing and compiling operations will eventually leave your disk with many small unused areas separating the useful files stored on the disk. When a substantial fraction of the disk is occupied by useful files, it may happen that no one of the unused areas is large enough to provide space for a new file that needs space. If the total area contained in the several unused areas is large enough to accommodate the new file, it may be time to use the K(runch command to "push" all the useful files together, leaving all the unused space at the end of the directory. The E(xtended directory command can be used to judge when it may be useful to use the K(runch command.

# D(ate

The D(ate command is used to display the date information currently stored on the disk with which you bootloaded. It can also be used to change the date. Figure 5-5 shows an example in which we are preparing to change the date. Our answer to the prompt will be completed when we press the ENTER key.

**Date set: <1..32>-<Jan..Dec>-<00..99>**
**Today is 1-Jan-82**
**New date ? 2**

Figure 5-5. D(ate command just before new date is completed.

If you find that the date displayed by the D(ate command is correct, terminate the command simply by pressing the ENTER key (without typing in a new date). If you do supply a new date, the D(ate command will verify its understanding of the date you have typed in.

Note that the format of the date you supply to the command must be:

**day-month-year**

where "day" is a one or two digit number, "year" is a two digit number, and "month" is a three letter abbreviation. The date command is unforgiving

about this format mainly because the program needed to accept other commonly used date formats unambiguously would occupy scarce space unnecessarily in the microcomputer's memory.

If you need to change only the day, leaving the month and year information unchanged, simply type in the one or two digits for the new day, followed by ENTER. Otherwise it is necessary to enter all three items, separated by dashes.

# P(refix

The P(refix command is used to display and/or change the default volume name prefix automatically applied by the System to file names given to it without any explicit mention of a volume name. The prompt message displayed by the P(refix command is:

**Prefix titles by what vol ?**

If you wish simply to display the name of the current default volume, press the colon (":") key followed by ENTER. For example, if you bootloaded from a volume called "KB99", and have not used the

P(refix command to change the default, then the P(refix command will display:

**Prefix is KB99:**

If you wish to change the default volume name to make it refer to a different disk, say NEWVOL, then you should answer the prompt by typing in:

**NEWVOL:**

followed by ENTER. As a result a future program reference to a disk file called "ANYFILE.TEXT" would, by lacking any explicit reference to a volume name, have the effect of referencing the full file title:

**NEWVOL:ANYFILE.TEXT**

# T(ransferring files from one place to another

The T(ransfer command is used to copy one or more files from a source device to a destination device. Most often, both devices are likely to be disks. When you press the "T" key, the Filer prompts with:

**Transfer what file ?**

If you respond with:

**SRCNAME.TEXT**

followed by ENTER, the Filer will again prompt with:

```
To where ?
```

to which you might answer:

```
NEWVOL:SRCNAME.TEXT
```

The Filer will then copy the file SRCNAME.TEXT from your default volume over to the disk whose volume name is NEWVOL. It will confirm the completion of each file transfer with a message similar to this:

```
OLDVOL:SRCNAME.TEXT -->
                    NEWVOL:SRCNAME.TEXT
```

The T(ransfer command is used in a wide variety of commonly occurring situations, some of which are described in the following subsections:

## Shorthand entry of the destination file name

Following the initial prompt "Transfer what file ?", you can type in both the source file name and the destination file name separated by a comma (","), as in:

```
SRCNAME.TEXT,NEWVOL:SRCNAME.TEXT
```

The effect of this is the same as the example given above, in which we waited for the second prompt message "To where ?".

# T(ransferring files from one place to another

We can carry this a step further by using the filename duplicator character ("$") as in:

**SRCNAME.TEXT,NEWVOL:$**

in which the character dollar sign ("$") will be interpreted as equal to "SRCNAME.TEXT".

> CAUTION: If the T(ransfer command responds to your typing in the source and destination names with a message such as:
>
> **Destroy NEWVOL:?**
>
> or
>
> **Transfer 320 blocks? (Y/N)**
>
> then you probably have forgotten to type in the dollar sign as an instruction of what file name to use. You should respond by pressing "N" for N(o, or the ENTER key. See the next subsection regarding T(ransfers in which only the volume names are used.

## Disk to Disk bulk T(ransfer

You can T(ransfer the entire contents of one disk to another by using disk volume names alone for the source and destination. For example, answer the T(ransfer command's prompt with:

**SRCVOL:,DESTVOL:**

to which the Filer should respond with the prompt message:

**Destroy DESTVOL:?**

# T(ransferring files from one place to another

If you wish the T(ransfer to proceed, answer with Y(es. The result will be that the disk whose volume name is currently "DESTVOL" will become an exact copy of the contents of the disk "SRCVOL" including even the volume identification. Any answer other than Y(es will terminate the T(ransfer command without any copy action taking place.

Since the result of a full volume to volume T(ransfer leaves two volumes with the same name on the System at the same time, it is important to resolve the volume name ambiguity immediately after the T(ransfer is completed. If you intend to keep both disks on-line at the same time, it would be best to C(hange the volume name of the destination disk. See "C(hange" in this chapter regarding the C(hange command.

## T(ransferring only selected files

There are two commonly used ways to T(ransfer only a selected group of files from the source disk to the destination.

You can T(ransfer all the files whose titles begin with the characters "SYS" from a disk called "SOURCE" to a disk called "DEST" by responding to the T(ransfer command's prompt as follows:

**SOURCE:SYS=,DEST:SYS=**

followed by ENTER. Along the same lines, you could transfer all of the .TEXT files from your default disk to the disk called "DEST" as follows:

**=.TEXT,DEST:=.TEXT**

# T(ransferring files from one place to another

Note that the equals character ("=") specifies all of the files on a disk if it is not "qualified" with additional characters as in these examples.

If you want a simplified way to review all or part of the directory of your source disk, indicating separately whether each file is to be T(ransferred, substitute the question mark character ("?") for the equals character ("=") in these examples. This will cause the Filer to pause after displaying each file name. If you respond with Y(es, the T(ransfer of the indicated file will be carried out. Otherwise, the Filer will pass over the indicated file and go on to the next.

## Rearranging the files on one disk

The Filer provides limited facilities to allow you to reorganize the order in which files are stored on a single disk by using the T(ransfer command. Of course you can respond to the T(ransfer command's prompt as follows:

**TESTFILE.TEXT,COPYFILE.TEXT**

which will leave the old file TESTFILE.TEXT intact, but create a new file COPYFILE.TEXT on the same disk containing the same information.

If you want to move the old file, use the same title for both source and destination, as in:

**TESTFILE.TEXT,TESTFILE.TEXT**

or the equivalent

**TESTFILE.TEXT,$**

# T(ransferring files from one place to another

This will cause a new copy of the old file to be made, and given the same directory name as the old file, and the directory entry pointing to the old copy of the file will be removed.

One common situation occurs when you have an unused area near the beginning of the directory (shown by E(xtended directory list at the top of the screen), and a frequently used file near the end of the directory. Use the L(ist command to find out how many blocks this file occupies, as shown by the number displayed by L(ist in the column just to the right of the file titles. For example, let us assume that the file TESTFILE.TEXT is 23 blocks long, and that the unused area is at least 23 blocks long. You can then cause that file to be moved to the beginning of the unused area by responding to the T(ransfer command's prompt as follows:

**TESTFILE.TEXT,TESTFILE.TEXT[23]**

By placing the length of the file, in blocks, at the end of the destination title within square brackets, you tell the Filer to place the destination file as near the beginning of the destination disk as possible without overwriting any other file.

The use of T(ransfer in this manner, together with the M(ake command which is described in "M(ake" in this chapter, can sometimes be used to force the Filer to place a copy of a file at some exact starting block number. You are not likely to need to do this unless your disk becomes damaged in some area, which then needs to be avoided. Normally, you can use the K(runch command to rearrange your disk when there are too many small unused areas, and dummy files of type .BAD will prevent the use of damaged areas. See "Checking for disk errors and repairing them" for additional information on handling damaged disks.

# T(ransferring files from one place to another

## Directory Maintenance Commands

Grouped within this section are several commands used primarily to make changes in the directory which describes the files on a disk, rather than in the files themselves. You can R(emove a file by deleting its directory entry and marking it unused. You can C(hange the directory title of a file. You can M(ake a new file by creating its directory entry and giving a title. You can "push" all of the files on a disk together in the lowest numbered group of blocks using the K(runch command, thus shifting all of the unused space into one area occupying the high numbered blocks. Finally, you can mark a disk to show its directory empty and all blocks unused with the Z(ero command.

## R(emove

---

R(emove is used to eliminate one or more entries from a disk directory, leaving the space formerly occupied by the file marked unused. R(emove only changes the directory, and all information stored within the file itself is left untouched by the R(emove command. The prompt message displayed by this command is:

**Remove what file ?**

You can respond with a single file title, or you can designate that several files are to be removed

selectively. To remove the file WORK.TEXT, answer the prompt by typing in:

**WORK.TEXT**

followed by ENTER. Note that it is not sufficient to give just the simplified name of a Workfile. Thus, if you have files WORK.TEXT and WORK.CODE on your disk, the R(emove command will respond with an error message if you answer the prompt by typing in only "WORK". The R(emove command does not recognize the simplified Workfile name because it often happens that you may wish to remove either the .TEXT or the .CODE portion of a Workfile without losing the other portion.

You can R(emove several files with one use of the command by listing their titles separated by comma characters (","). For example, to R(emove the files ALPHA, BETA, GAMMA and DELTA, answer the prompt with:

**ALPHA,BETA,GAMMA,DELTA**

followed at the end by ENTER. The Filer will respond with acknowledgement of its action with each file actually removed. If you misspell a file title, and the resulting title does not also exist in your directory, then the Filer will display an error message noting that the indicated file is not in your directory.

A more effective way to R(emove groups of files in one operation is to use either of the wild card naming options. For example, to remove both the .TEXT and .CODE portions of the Workfile WORK mentioned earlier, respond to the R(emove command's prompt with:

**WORK.=**

# R(emove

Since it is common for a user to forget that other files on the disk may have the same prefix, the Filer will display the titles of all files to be removed, and then it will prompt with:

**Update directory ?**

You should review the list of titles actually displayed before responding with Y(es, since it may be virtually impossible to reverse a mistake in this process.

If you wish to R(emove various files from your directory, but cannot find a common prefix or suffix, use the questionmark ("?") wild card reference. The Filer will display one file title at a time, and will wait for you to answer with Y(es or N(o. For example, respond to the R(emove command's prompt with:

**:?**

to indicate that you want a list of all files on your default disk. To R(emove only files whose titles end in ".TEXT" from a disk volume called "AUDIT", respond to R(emove's prompt with:

**AUDIT:?.TEXT**

As with the use of the equal sign wild card, the Filer will prompt at the end of the sequence to ask whether you really want to update the directory by making the indicated changes.

If at some point you wish to terminate the R(emove command without making any directory changes, press the ESC key to get back to the Filer's main command world.

# C(hange

The C(hange command is used to alter the directory titles of selected files. It can also be used to alter the name of a disk volume. The command prompts with:

**Change what file ?**

It waits until you type in the name of the file to be C(hanged, followed by ENTER, and then prompts for the name to which the name is to be C(hanged. If you answer the first prompt with

**ABC.TEXT**

followed by ENTER, and answer the second prompt with:

**XYZ.TEXT**

followed by ENTER, the Filer should respond with:

**VOL3:ABC.TEXT -> XYZ.TEXT**

assuming that the file is on a volume called "VOL3".

You could call for the same C(hange without waiting for the second prompt by answering the first prompt with the following:

**ABC.TEXT,XYZ.TEXT**

i.e. by listing both the original title and the desired new title separated by a comma (",").

To C(hange the name of a disk volume, use only the volume identifiers followed by colon (":")

characters, making no reference at all to any file in the directory of that volume. For example, if you want to C(hange a volume name "KB99" to "NEWID", answer the C(hange command's prompt with:

**KB99:,NEWID:**

followed by ENTER.

The wild card file naming options can be used with the C(hange command. The portion of all original file titles represented by the equal sign ("=") will be duplicated in place of the equal sign in the desired new titles. Additional title string characters may be used before or after the equal sign, or both, in either the original or desired new titles. For example, you might have a set of files WORK.TEXT and WORK.CODE and wish to change them to read OLDWORK.TEXT and OLDWORK.CODE in order to reuse the Workfile name "WORK". To do this, answer the C(hange command's prompt with:

**WORK=,OLDWORK=**

The M(ake command is used to create a new directory entry. The command prompts with:

**Make what file ?**

and expects a file name as a response. Normally, you should append the number of blocks the file is to occupy on the disk, within square brackets ("[" and "]"), as an extension to the file title. The occasion to use M(ake sometimes arises when you wish to prevent temporarily the assignment of files to an unused area of the disk which you intend to occupy with another file at a later time. Thus you might create a new temporary file called "DUMMY" which you want to fill a 20-block unused area near the beginning of the disk directory. To do this, you answer the M(ake command's prompt with:

**DUMMY[20]**

followed by ENTER. This causes the Filer to M(ake a new directory entry for a file called DUMMY occupying 20 blocks within the first (closest to beginning) unused area in the directory that is at least 20 blocks long. An unused area that is located closer to the beginning of the directory will be ignored if it is 19 or fewer blocks long.

If you leave out the file length specification in brackets, the M(ake command will create a file which completely fills the largest unused area in the directory when the M(ake command is entered. If you use an asterisk, i.e. ("*"), in place of a number in

the file length specification, the M(ake command will place the new file either in one-half of the largest unused area, or in all of the second largest unused area, whichever is larger.

The M(ake command is sometimes used to change the directory information on the number of 512-byte blocks occupied by a file. The need to do this can arise if you run a program which creates a new file on the disk, but fails to reduce the space occupied by that file to the minimum number of blocks needed before terminating. Let us assume that you have some independent way available to determine how many blocks the file should occupy. If the file title is DATA, and it originally occupies 97 blocks, but you want it to occupy only 43, then use the following sequence:

a) M(ake dummy files to fill all unused areas on your disk which occur closer to the beginning of the directory than the entry for the file DATA.

b) R(emove DATA. The result will be that the space occupied by DATA becomes part of the first unused area on the disk.

c) M(ake "DATA[43]".

d) R(emove the dummy files created in step (a).

# K(runch

The K(runch command moves files toward the beginning of the disk directory in such a way as to shift all unused areas into a single large unused area at the end of the directory. The command prompts with the following message:

**Crunch what vol ?**

You should answer by typing in the name of the volume you want K(runched, followed by a colon (":") and then ENTER. For example, to K(runch the volume "KB99", type:

**KB99:**

followed by ENTER. The Filer will then respond by displaying the message:

**From end of disk, block 320? (Y/N)**

If you respond with Y(es, the command will then move the files as commanded. The Filer will display a message confirming the name of each file actually moved on the disk by K(runch.

> CAUTION: Because the K(runch command is required to change the disk directory, and because it does not get around to doing this until after finishing a fairly lengthy sequence of operations, it is a dangerous command to use. If K(runch is interrupted in the midst of doing its work (power failure, damaged area on the disk, disk drive opened during K(runching operation, ...) your disk directory will no longer correctly describe the contents of your disk! It is generally very desirable to use the B(ad-blocks command before using K(runch. See Section 7 regarding strategies to use if your disk does in fact have bad blocks.

# Z(ero

The Z(ero command creates a new empty directory on the indicated disk volume. The previous directory on that volume will be destroyed as a result of this operation. Z(ero prompts with:

**Zero dir of what vol ?**

to which you respond with a volume identifier such as:

**OLDVOL:**

followed by ENTER. If the disk contains no directory, as would be the case with a new disk that has not previously been used with the UCSD p-System, then use the explicit unit designation in place of the volume identifier. For example, if the disk to be Z(eroed is in your spare disk drive (the one you do not use for bootloading), then respond with

**#5:**

followed by ENTER.

If the disk to be Z(eroed already contains an old directory, the Filer will prompt with:

**Destroy OLDVOL: ?**

If you answer with Y(es, the Filer will ask whether you want a duplicate directory to be created on the disk, with:

**Duplicate dir ?**

If you respond with Y(es, the System will maintain a duplicate copy of your disk directory for possible future use in recovering from an error associated with the main directory on your disk. Any of several conditions might cause such an error, as discussed in "Checking for disk errors and repairing them" in this chapter. In most cases an error in the main directory will not be reflected also in the duplicate copy of the directory. A utility program COPYDUPDIR is supplied with the System for copying the duplicate directory into the main directory's area of the disk, thus allowing recovery from the error.

The Filer will then ask how many blocks are to be available for files to be stored on the disk being Z(eroed. If the disk already has a directory, you will be asked to confirm with a Y(es response that the same number of blocks is again to be used:

**Are there 320 blks on the disk? (Y/N)**

If you respond with N(o, or if the disk previously contained no directory, the Filer will prompt with:

**# of blocks ?**

You should respond to this message with a number indicating the maximum capacity of the disk since the Filer has no way of knowing that capacity by itself. For the IBM Personal Computer, you should respond with 320. Note that the number of blocks given here includes a provision for the blocks occupied by the bootstrap loader, the main directory, and the optional duplicate directory, on your disk.

The Filer will then ask for the volume name you want to use with the disk being Z(eroed, with the message:

**New vol name ?**

# Z(ero

You should answer with an identifier (first character a letter, other characters may be letters or digits) up to 7 characters long. The Filer will prompt by asking you to verify the new volume name (since incorrectly typed volume names can lead to problems later in using the disk). If you respond with Y(es, the Filer will then write a new directory on the disk, and the Z(eroing process will be completed.

You can terminate the Z(ero command following any of the prompt messages by pressing the ESC key, and no new directory will be written on the disk as a result.

# Checking for disk errors and repairing them

One of the annoying facts of life in computing work is that secondary storage media such as floppy disks can often transmit imperfect copies of stored information when they are reread. If the information transmitted is not a perfect reproduction of the information that was originally sent to the disk, it is said to contain "errors". If a .CODE file contains no more than one 8-bit byte that is in error, that .CODE file may be effectively useless.

There are many potential causes for errors associated with disk files. Errors can be caused by a malfunctioning disk drive, by incorrect operation of the electronic connections between computer and disk drive, by a flawed area on the recording surface of the disk, by dust or grime that has found its way into the disk's protective cover, and so on.

Some errors are marginal in nature, with the result that correct information will be transmitted on some attempts and erroneous information will be transmitted on others. A standard part of most software systems, including the UCSD p-System, is an arrangement whereby the data read from the disk is checked for errors. If errors are detected, the System will automatically reread the data several times in an effort to complete a read operation without any indicated errors. If rereading in this manner fails to produce an error-free copy of the data, the error is said to be "unrecoverable". It is not unusual for a block of data to contain unrecoverable errors, when read on one disk drive, but to be readable without any errors at all when read on a different disk drive of the same type.

With careful handling of the disks and disk drives, you will usually need to contend with very few disk-related data errors indeed. However, anyone who makes much use of a computer learns to cope with occasional errors. This section deals with two commands provided with the Filer to assist in controlling disk errors when using the UCSD p-System.

# B(ad blocks scan

The B(ad-blocks command prompts with the message:

**Bad blocks scan of what vol?**

You should respond with the name of the disk volume which is to be checked for bad blocks. The disk volume must be in a disk drive and ready for use when the B(ad-blocks command is invoked.

This command reads each block on the disk, checking for unrecoverable errors. If there are no errors at all, you should note a regular clicking noise in the disk drive as the read/write head is moved from track to track. If the clicking noise comes at irregular intervals, there may be marginal errors in reading from the disk. If a block cannot be read without errors, after many tries, the Filer will display the number of the bad block on the screen, and the B(ad-blocks command will continue scanning for additional errors.

You should take note of the block numbers where errors are found. If you have access to a second disk drive, it would be best to try the B(ad-blocks scan again using that drive. If the list of bad blocks displayed with the second drive is identical to the first list, then the signs are not good for the blocks listed. If the list differs from the first drive to the second, then the errors are likely to be marginal and quite possibly recoverable.

# eX(amine command

The eX(amine command is provided as a tool to be used in repairing some types of marginal floppy disk recording errors. A common type of error arises when a disk drive uses too weak or too strong a recording signal in storing information on the disk. The problem may be the fault of either a maladjusted disk drive or of a disk with a slightly damaged recording surface. Either way, it is sometimes possible to rerecord the information (usually with a different disk drive) in such a way that it can be read without errors thereafter.

Errors on reading from a disk are usually detected through use of a check sum which is stored with each sector of information when it is recorded on the disk. The check sum is generally computed as the result of a "Cyclic Redundancy Check" (CRC). A two-byte CRC check sum stored on the disk with the useful data is compared with a CRC check sum computed from the useful data when it is read from the disk. If the two are not equal (the recorded and recomputed CRC check sums) then a read error is detected. The System usually tries to read the same block of data, which may contain several sectors containing their own individual CRC check sums, at least 10 times in an effort to complete one read operation without a detected CRC error. Only if no error free read can be completed will an unrecoverable error be detected.

The eX(amine command tries to read the blocks you select without unrecoverable errors. If it succeeds, it then rewrites the information thus read back to the same block on the disk. It then rereads that block, and cross-compares the information read first with that obtained after the rewrite operation. If the two

are the same, the Filer will inform you that the indicated block "may be ok" as a result of the operation. The eX(amine command first prompts with the message:

**Examine blocks on what volume ?**

After you respond with the volume name, followed by colon (":") and ENTER, the Filer will prompt with:

**Block-range?**

You should respond with a list of block numbers separated by commas (","), or by giving a starting block number and a stopping block number separated by a dash character ("-"). For example:

**234-240**

followed by ENTER. The Filer will then prompt by displaying the names of all files found in the directory to include blocks within this range. (It is possible that these files could be damaged by the use of Examine.) It will then prompt with:

**Try to fix them ?**

If you respond with Y(es, the Filer will attempt to read/write operation described above on each block in the indicated group (in the example blocks 234 through 240 inclusive). If, during this operation, the Filer finds any blocks which cannot be read correctly

after many tries, it will display a message stating which blocks are bad. If it prompts with:

**Fix them?**

and you respond with Y(es, the directory will be marked showing the damaged area of the disk to be in a file with a .BAD suffix. Subsequent K(runch operations will not attempt to move any files with the .BAD suffix.

> CAUTION: Even if the eX(amine operation terminates, showing that all indicated areas of the disk "may be ok", it is possible that your original information has been lost. This is possible because the error checking logic is not perfect, and the information read initially and rewritten by the eX(amine command may in fact be in error. It would be best to check the contents of your disk, with the Editor, by trying to eX(ecute a .CODE file, by checking the contents of a data file with an associated program, or by other means after using the eX(amine command.

# NOTES

# CHAPTER 6. PASCAL COMPILER-SYNTAX ERRORS

## Contents


Goals for this Chapter ................ 6-3
Preliminaries .......................... 6-4
Comments and Compiler Directives ... 6-5
Include Directive ...................... 6-7
The Compiler's CRT Display and the List Directive ................... 6-8
Miscellaneous Compiler "Switch" Directives ........................... 6-13
I/O Check Switch ..................... 6-13
Quiet Compilation Switch ............ 6-14
Syntax Errors ......................... 6-14
Unmatched BEGIN ... END pairs ..... 6-17
Comment Not Completed with a Closing "*)" Symbol ................. 6-19
Nested IF Statements ................. 6-20
Execution or Run-Time Errors ........ 6-27

# NOTES

# Goals for this Chapter

The Pascal Compiler is used to translate Pascal programs from their human readable .TEXT form, saved on the disk by the Editor, into their directly executable .CODE form, which the Compiler itself saves on the disk. The Compiler is designed to translate the entire contents of a .TEXT file in one continuous operation. Unlike the Editor and the Filer, the compiler has hardly any interactive commands. However, it is possible to change certain controls which govern the way in which the Compiler does its work. This is done using Compiler "Directives", which are written in the form of Pascal language comments that start with the dollar sign character ("$"). One of the main purposes of this chapter is to present those Compiler Directives available in the UCSD p-System which are of use to beginning users of the p-System.

Also included in this chapter is a discussion of strategies for coping with program errors. If your program contains statements which fail to conform with the syntax rules of the Pascal language, the Compiler will halt at each point where it finds an error. You then have the option of returning immediately to the Editor to correct the syntax errors, or of continuing with the Compilation to see if there are any additional errors.

Once the Compiler can go through the entire program file without finding any syntax errors, execution of the program may halt abruptly with the display of an exectuion error message (also called a "run-time" error message, since it occurs while the program is running). The run-time message contains coded information which can be used to find the place in the text of the Pascal program where the execution error occurred. Illustrations of both syntax and execution errors are given in this chapter, along with various suggestions on how to go about resolving the errors.

Following is a list of specific learning goals for beginners.

a) Use the Include-file option to compile a program from Pascal procedures located in two or more separate .TEXT files.

b) Use the Compiler's List option directing its output to the CONSOLE: display screen of your computer. Use the procedure number and byte offset values shown in the Listed output to find where a run-time error has occurred.

c) Place several types of syntax errors in a test program intentionally, and note how they are identified by the Compiler.

# Preliminaries

The Compiler performs its translation tasks by breaking the source program into "tokens", i.e. into logically separate items. Examples of tokens include identifiers (see the *PASCAL Reference for the UCSD p-System* for a definition), individual special characters like the semicolon (";"), comma (","), or equal sign ("="), and integer or real constants (numbers). In a few special cases, two characters together comprise a token (".."), ":=", "<>", "<=", and ">="). An entire quoted string such as "this is a string" comprises one token.

If the Compiler finds a place in the source program which fails to conform with the syntax of the Pascal language, it halts and causes a brief message to be displayed explaining the nature of the error. As it proceeds through the text of the source program, the Compiler maintains a pointer showing where the next token to be scanned begins. Thus, when an error is detected, the pointer indicates the beginning of the token immediately following the token found to be in error. Note that SPACE characters in a

Pascal program simply separate adjacent tokens. A SPACE character is permitted between any pair of adjacent tokens. Unless both tokens are identifiers, the SPACE is not required. From the point of view of the syntax, any number of adjacent SPACE characters are considered to be the equivalent of just one separator. Also the end of one line is considered to be adjacent to the beginning of the next, and thus is considered to be equivalent to one SPACE character in the program text. (However, remember that it is not permitted to break any single token into two or more parts located on separate lines.)

# Comments and Compiler Directives

A Comment may be placed in a Pascal program at any point where a SPACE character would be permitted. In the UCSD p-System, a comment may begin with the character pair "(*" and end with the matching pair "*)", or it may begin with a left curly bracket ("{") and end with a right curly bracket ("}"). As with the five two-character tokens discussed earlier, no SPACE is allowed between the asterisk ("*") and either the left parenthesis ("(") or right parenthesis (")"). Thus, the Compiler will not recognize

```
( * illegal comment * )
```

as a comment. However the following would be recognized as a comment:

```
(* this is a legal comment *)
```

Of course, the main reason why comments are permitted in Pascal program is to encourage programmers to include notes which explain what each portion of a program is intended to do. Though

a well written Pascal program should be relatively easy to read and understand without comments, judiciously placed comments can greatly improve the reader's chances of understanding a program quickly and thoroughly.

While a comment embedded in a Pascal program is not considered to be part of the program, and thus not a token to be translated into executable object code, it is possible for the Compiler to extract information from the characters contained within a comment. The UCSD Pascal Compiler recognizes any comment that begins with a dollar sign character ("$") as a "Directive" to the compiler itself. Note that the dollar sign must be the first character following "(*" or the left curly bracket ("{"). There can be no intervening SPACE characters.

Compiler Directives are instructions to the Compiler which cause it to change selected "switches" controlling the way it operates. For example, the Compiler is capable of sending a specially formatted copy of the source program text to a printer, or to a disk file for later printing, or even directly to the computer's console CRT. This formatted output adds substantially to the time taken by the Compiler to complete its program translation tasks. Consequently the formatted output is normally not activated. However, the source program can contain a Compiler Directive (the "List" directive) instructing the Compiler to begin generating the formatted output. If the slower formatted output is not needed throughout the entire source program, another compiler Directive can be included at the appropriate point in the source program to deactivate the formatted output thereafter. Details on the List Directive will be given in a later Section of this chapter.

# Include Directive

Sometimes it is convenient to keep portions of a Pascal program in separately Edited .TEXT files. The Include Directive tells the Compiler to regard the entire text contained in a named .TEXT file as if it were part of the source program text at the point where the Include Directive occurs. For example, in the following small piece of a program:

```
   PROGRAM TEST;
   VAR X,Y,Z;
   BEGIN
(*$I PREAMBLE.TEXT*)
    IF X>=100 THEN
    . . .
```

would instruct the Compiler to treat all of the program statements contained in a file called PREAMBLE.TEXT as if they had been included within the text of the program at the point immediately following "BEGIN".

One situation in which one might use the Include Directive occurs when one wants to develop several programs, all of which are to have an identical section of program statements. Of course, if the Included file is changed, then all of the programs which use the Include Directive referring to that file will have to be recompiled in order to take advantage of the changes.

Occasionally, one wants to Include a file which contains CONST, TYPE, VAR, PROCEDURE, and FUNCTION declarations. If the program file containing the Include Directive must also have its own set of declarations, it is implied that there must be a relaxation of the Pascal syntax requirement that CONST declarations occur before TYPE declarations, TYPE must occur before VAR, and so on. The UCSD Pascal Compiler allows relaxation of this strict sequence in the special case in which the Include Directive occurs between the last variable

declared in a VAR list, and the first PROCEDURE or FUNCTION heading declared in the main program.

Include directives may be placed in files which are themselves include files. These directives may only be nested to three levels, however.

# The Compiler's CRT Display and the List Directive

As the Compiler works its way through the text of the source program, it is capable of generating two kinds of displayed or printed output designed to assist a user to keep track of its progress. The principal uses of this output are associated with program debugging, and will be discussed further in later sections of this chapter.

Normally, the Compiler displays only a very terse summary of its progress as it goes through the source program. Figure 6-1 provides an example.

```
Compiling...

Pascal compiler - release level IV.0 b20-2
<    0>..........
REPEAT1
<   10>.......................
REVERSE
<   33>..............
REPEAT2
<   47>...................
BLOWUP
<   67>.....
      A[I]:=I*I;
      WRITELN(I,':   ',A[I] ; <---
Illegal symbol (terminator expected)
Line 73
Type <sp> to continue, <esc> to terminate, or 'e' to edit
```

Figure 6-1. Example of the Compiler's Display Showing a Syntax Error.

In this display, a line with the following appearance:

```
<     10>............
```

shows how many lines of Pascal source text have been compiled so far. One dot character is displayed for each line compiled, as the line is being compiled. The number within broken brackets at the left margin is the number of lines already compiled at the time when this line starts being displayed.

A line with the following appearance:

```
REPEAT1
```

shows the name of the procedure or function body (executable statement part) which the Compiler is just beginning to translate.

The output generated by the List Directive is illustrated in Figure 6-2.

```
 1  0   0:d   1  (*$L console: *)
 2  2   1:d   1  PROGRAM EDITDEMO;
 3  2   1:d   1  VAR G1,
 4  2   1:d   1    G2,
 5  2   1:d   1    G3,G4:INTEGER;
 6  2   1:d   5    B1,B2,B3:BOOLEAN;
 7  2   1:d   8
 8  2   1:d   8  PROCEDURE REPEAT1;
 9  2   2:d   1  VAR S,SG:STRING;
10  2   2:d  83      L,N:INTEGER;
11  2   2:0   0  BEGIN
12  2   2:1   0   WRITELN(
13  2   2:1   0    'TYPE ANY STRING FOLLOWED BY <e
14  2   2:1   3        );
15  2   2:1  20   READLN(S);
16  2   2:1  38   N:=1;
17  2   2:1  41   L:=LENGTH(S);
18  2   2:1  47   REPEAT
19  2   2:2  47    SG:=COPY(S,1,N);
20  2   2:2  63    WRITELN(SG);
21  2   2:2  80    N:=N+1;
22  2   2:1  85   UNTIL N>L
23  2   1:0   0  END (*REPEAT1*);
24  2   1:0   0  —
```

Figure 6-2. Illustration of Output using List Directive.

In addition to showing the text of the source program as it is being compiled, on a line by line basis, this display also includes formatted information of potential use in program debugging. The List option is activated by the Directive:

```
(*$L CONSOLE:*)
```

which may be seen in the top line of Figure 6-2. "CONSOLE:" is the volume identifier of the CRT display on output operations (It also is used for input from the keyboard.). In its place, you could put any desired disk file title, making sure to use the suffix

".TEXT". The resulting file will contain the formatted listing generated by the Compiler in a form that can be read using the Editor. You may substitute either "PRINTER:" or "REMOUT:" in place of "CONSOLE:" to have the listing sent to an external printer or other remote terminal device.

The List option can be deactivated by the Directive:

**(*$L-*)**

In Figure 6-2, the number displayed to the right of the dot character is the number of the program text line being Compiled. Next to the right is the digit "2" on each line. This is the number of the program "segment". Separately compiled program segments provide a means of controlling "overlays" in the UCSD p-System, i.e. a means to conserve on memory space when working with large programs. The rules on preparing programs containing separately compiled segments are beyond the scope of this book, but they are described in the *User's Guide for the UCSD p-System.*

To the right of the segment number, there is a number immediately followed by a colon character (":"). This number is assigned by the Compiler as a unique identification of each program block (procedure or function) within a segment. The main program itself is always block number 1. The block numbers are assigned in the order of appearance of the PROCEDURE and FUNCTION headings. The order of appearance of the procedure and function identifiers in the Compiler's normal display corresponds to the appearance of the executable parts of each block, and thus may not be the same as the order of block number assignments.

Immediately to the right of the colon character (":") is the character "D", in lines that pertain to the declarations, or a number, in lines containing executable program statements. This number is the level of "nesting" of Pascal statements, and it may be useful in finding unmatched "BEGIN" ... "END" pairs in a program.

The final column of numbers, located just to the left of the Pascal program statements proper, is to be interpreted differently depending upon whether the associated lines are declarations or executable statements. On a declaration line, the number tells how many two-byte memory cells intervene between the base address of the block and the first declared identifier in a list such as G1, G2, G3, G4. These location numbers are valuable when using the built in p-System Debugger (see the *User's Guide for the UCSD p-System*).

On a line containing executable statements, the number in the last column tells how many bytes of compiled code were generated before the first code bytes of the current line started being generated. These numbers can be of considerable assistance to a beginner who is searching for the source of a run-time error in a program. I will explore that topic in some detail in a later section.

# Miscellaneous Compiler "Switch" Directives

All of the miscellaneous Compiler "Switch" Directives tell the compiler to start or stop doing something as it goes through the source program. In each case, the Directive is selected using a single character followed by either a plus character ("+") to turn the switch "On", or a minus character ("-") to turn the switch "Off". For example:

```
(*$Q+*)
```

turns on the Compiler's Quiet mode. On the other hand:

```
(*$Q-*)
```

turns it off.

## I/O Check Switch

The UCSD p-System terminates a program abnormally in the event of an error encountered during an Input/Output operation. The Compiler can be instructed not to generate the code which checks on the result of an I/O operation using the option

```
(*$I-*)
```

Means are available then for the programmer to provide program checks to determine how to cope with an I/O error. This subject is beyond the scope of a book for beginners. Details may be found in the *User's Guide for the UCSD p-System.* Unless you find it essential to do your own checking for I/O errors in a program, I strongly urge you to forget about the I/O Check Switch Directive! However, its use is discussed in this book in Chapter 7, "Programming to use Disk Files".

## Quiet Compilation Switch

There may be situations when, during compilation, you will want to suppress the Compiler's normal progress messages. For example, if you have redirected the standard CONSOLE: output to a hard copy device (see the Users Guide for information about Redirection), you may be concerned about the time required to output all of the Routine names and lines of dots during a large compile. The Compiler's "Quiet" switch directive suppresses normal progress messages if it is turned on:

```
(*$Q+*)
```

Conversely, if you want to turn off the suppression of progress messages, use the directive:

```
(*$Q-*)
```

# Syntax Errors

If the Compiler finds a section of program text which fails to conform with the syntax rules of Pascal, it halts and causes an error message to be displayed. An example of the display you should expect to see is shown in Figure 6-1, which refers to a sample program called COMPDEMO. This program is supplied as one of the files on the STARTUP disk.

In Figure 6-1, the right-parenthesis character (")") which should be placed just to the left of the semicolon (";") has been left out of the program. The Compiler's progress display contains copies of the line where the program error is found (up to the token where the Compiler notes the error) and the previous program line. The symbol "<<<<" amounts to a cursor pointing to the token found to be in error.

The next line shown in Figure 6-1 describes the error encountered. The line following that indicates the line number of the program where the error was found. The final line provides several options. If you press the "E" key, to invoke the E(dit option, the result will be to return to the Editor. In this case, a message briefly describing the nature of the syntax error will be redisplayed at the top of your CRT screen. This is shown in Figure 6-3 which replicates the Editor's display resulting from this operation.

```
Illegal symbol (terminator expected). type <sp>

FUNCTION BLOWUP (X,Y:INTEGER):BOOLEAN;
VAR
  I,LB,UB:INTEGER;
  CH:CHAR;
  A:ARRAY[1..10] OF INTEGER;
BEGIN
  LB:=X;
  UB:=Y;
  FOR I:=LB TO UB DO
   BEGIN
    A[I]:=I*I;
    WRITELN(I,':  ',A[I] ;_
   END;
  BLOWUP:=UB > 10;
END (*BLOWUP*);

BEGIN (*MAIN PROGRAM*)
  WRITELN('START EDITDEMO');
  WRITELN;
 REPEAT1;
  WRITELN;
  REPEAT2;
  WRITELN;
```

Figure 6-3. Editor display after Syntax Error return from Compiler.

The Editor's cursor is left pointing at the same position where the symbol "<---" pointed when the Compiler halted. You can continue at this point to use the Editor by pressing the space bar.

The two other command options made available by the Compiler, as in Figure 6-1, are intended for use in working with large programs. If you press the SPACE bar key when the Compiler has halted, the Compiler will continue attempting to compile the rest of the program. Because of the nature of the Compiler itself, this may or may not be a sensible thing to do. Some Syntax error conditions leave the Compiler confused, and all it can do is to produce an unending sequence of error messages at the same program location. Other error conditions are not as drastic and the Compiler can sometimes continue all the way to the end of the program with no problem. If you suspect that the Compiler has become confused after a sequence of syntax error conditions, you can terminate further compilation without automatically invoking the Editor by pressing the ESC key while the Compiler is halted.

If the file SYSTEM.SYNTAX is not present on the boot disk, the error messages shown in the figures will not be displayed. A complete list of the numbered syntax error messages used in UCSD Pascal may be found in Appendix E of this book for reference purposes.

If you are working with a large Pascal program, it may be most efficient to use a printed listing of the program as an aid during compilation. If none of the errors are fatal (i.e. cause the compiler to terminate) then during the compiler's second pass all of the syntax errors will be printed in the listing. Then the Editor may be entered once and all errors may be corrected. This method saves the time that would otherwise be taken up in multiple switching back and forth from Compiler to Editor to Compiler.... It

also simplifies the process that you should go through after noting each error - i.e. the visual search for errors similar to the one just flagged by the Compiler.

The following subsections provide suggestions on how to find some of the more troublesome syntax errors that often arise in use of the UCSD p-System.

## Unmatched BEGIN ... END pairs

One of the more common errors in writing Pascal programs is the failure to match each BEGIN in a source program with a corresponding END. The problem is exaggerated when one uses a CRT for most program editing work, since then it is often the case that both BEGIN and END of a pair cannot be displayed on the screen at the same time. While the compiler has no trouble discovering that each BEGIN has not been matched with an END (or vice versa), it may point to the problem at a point far removed from the place in the source program where the error is actually caused. Figure 6-4 illustrates the problem.

```
Illegal symbol (terminator expected), type <sp>
  BLOWUP:=UB > 10;
(*END*) (*BLOWUP*);

BEGIN (*MAIN PROGRAM*)
  WRITELN('START EDITDEMO');
  WRITELN;
  REPEAT1;
  WRITELN;
  REPEAT2;
  WRITELN;
  IF BLOWUP(5,15) THEN
   WRITE('Upper Bound too large');
END._
```

Figure 6-4. Syntax Error caused by BEGIN not matched by END.

But the error is not caught until the "END." at the end of the program and the message is not helpful.

The section of program that is shown here is the same as shown in Figure 6-3, but the missing right parenthesis character (")") has been correctly restored. The error occurs at the line: (*END*) (*BLOWUP*). The "END" should not be commented out. The commenting prevents the Compiler from regarding the "END" as part of the program. The Compiler thus goes on translating, and regards the "END." which terminates the program as an inappropriate end to the function Blowup. The Compiler assumed that the lines which followed the commented "END" were still part of the BLOWUP function.

The failure to match END's with their corresponding BEGIN's is often a difficult error to trace to its cause when working with a large program. The block number and bytes-generated columns of the Compiler's List option provide a mechanism which should help materially to find these errors.

## Comment Not Completed with a Closing "*)" Symbol

In a similar vein, it is all too easy to forget to finish a comment with the necessary closing "*)" or "}" symbol. Figure 6-5 provides an illustration.

```
Semicolon expected. type <sp>
BEGIN
  LB:=X;
  UB:=Y;
  FOR I:=LB TO UB DO
    BEGIN
      A[I]:=I*I;
      WRITELN(I,':  ',A[I]);
    END;
  BLOWUP:=UB > 10;
END (*BLOWUP );

BEGIN (*MAIN PROGRAM*)
  WRITELN('START EDITDEMO');
  WRITELN;
  REPEAT1;
  WRITELN;
  REPEAT2;
  WRITELN;
  IF BLOWUP(5,15) THEN
    WRITE('Upper Bound too large');
END.
```

Figure 6-5. Error caused by improper closing delimiter in a comment.

In this case, the actual error occurs at the line "END (*BLOWUP );", where an asterisk character ("*") has been left out. The Compiler does not detect an error until a couple of lines later. The error message given is:

```
; expected (possibly on line above)
```

Since correct syntax clearly does not require a semicolon on the indicated line, we must look closer to notice that an open comment caused the compiler to disregard some of the Pascal source.

## Nested IF Statements

Nested IF statements are an invitation to make syntax errors, some of which the Compiler is unable to detect. Figure 6-6 provides an example of a correct small program for use in seeing how some of the errors arise.

```
>Edit: A(djst C(py D(let F(ind I(nsrt J(mp K(ol R(plc Q(uit X(ch Z(ap [E.7h]
Pascal Compiler IV.0 b20-2

 1  0  0:d    1  (*$L IB1.TEXT*)
 2  2  1:d    1  PROGRAM IFBOMB;
 3  2  1:d    1  VAR W,X,Y,Z:INTEGER;
 4  2  1:0    0  BEGIN
 5  2  1:1    0   WRITE('Enter value of W:'); READLN(W);
 6  2  1:1   29   WRITE('X:'); READLN(X);
 7  2  1:1   58   WRITE('Y:'); READLN(Y);
 8  2  1:1   87   Z:=0;
 9  2  1:1   90   IF W > X THEN
10  2  1:2   95    IF W > Y THEN
11  2  1:3  100     Z := W
12  2  1:2  100    ELSE
13  2  1:3  105     BEGIN
14  2  1:4  105      IF W = Y THEN
15  2  1:5  109       Z := Y
16  2  1:3  109     END
17  2  1:1  112   ELSE
18  2  1:2  114    Z := X;
19  2  1:1  117   WRITELN('Z=',Z);
20  2   :0    0  END.

End of Compilation.
```

Figure 6-6. Program containing nested IF statements - no errors.

Compiled listings are used in the examples in this section. In this case, one of the best clues to checking correct program construction is the column of numbers representing depth of nesting (the numbers immediately to the right of the column of colon (":") characters). Notice that the nesting depth is increased by 1 each time a statement controlled by another is entered. It is reduced by 1 when the same controlled statement terminates. For example, the IF statement in line 9 controls the IF statement starting in line 10. Line 9 is at level 1, while line 10 (the controlled statement) is at level 2. The ELSE in line 12 refers back to the IF ... THEN in line 10, and hence is shown at level 2. The compound statement (BEGIN ... END) starts in line 13 at level 3, being controlled by the IF ... THEN ... ELSE ... at level 2, and it ends in line 16, again at level 3.

Now consider Figure 6-7, in which we have placed an additional BEGIN ... END pair to make the program logic a little more obvious. This program is still correct, and carries out the same steps shown in Figure 6-6. However, the additional compound statement brings an additional level of nesting. Thus the new BEGIN in line 10 of Figure 6-7 is at level 2, while the IF ... THEN in line 11 is at level 3. This same IF ... THEN had been at level 2 in Figure 6-6.

```
>Edit: A(djst C(py D(let F(ind I(nsrt J(mp K(ol R(plc Q(uit X(ch Z(ap [E.7h]
Pascal Compiler IV.0 b20-2
     1  0   0:d    1   (*$L IB2.TEXT*)
     2  2   1:d    1   PROGRAM IFBOMB;
     3  2   1:d    1   VAR W,X,Y,Z:INTEGER;
     4  2   1:0    0   BEGIN
     5  2   1:1    0     WRITE('Enter value of W:'); READLN(W);
     6  2   1:1   29     WRITE('X:'); READLN(X);
     7  2   1:1   58     WRITE('Y:'); READLN(Y);
     8  2   1:1   87     Z:=0;
     9  2   1:1   90     IF W > X THEN
    10  2   1:2   95       BEGIN
    11  2   1:3   95         IF W > Y THEN
    12  2   1:4  100           Z := W
    13  2   1:3  100         ELSE
    14  2   1:4  105           BEGIN
    15  2   1:5  105             IF W = Y THEN
    16  2   1:6  109               Z := Y
    17  2   1:4  109           END
    18  2   1:2  112       END
    19  2   1:1  112     ELSE
    20  2   1:2  114       Z := X;
    21  2   1:1  117     WRITELN('Z=',Z);
    22  2    :0    0   END.
End of Compilation.
```

Figure 6-7. Nested IF program with extra BEGIN ... END pair.

The addition of extra BEGIN ... END pairs is often useful when working with a complicated set of nested IF statements as a way to force the program logic to go as one plans. If the extra compound statement is redundant, as in Figure 6-7, no harm is done since the compiler generates no corresponding code. However, the extra compound statement makes it unnecessary for the programmer to trace back through the nested IF's to make sure that the ELSE in line 17 of Figure 6-6 (line 19 of Figure 6-7) belongs to the IF ... THEN in line 9 of both figures.

Unfortunately, one sometimes decides to clarify a set of nested IF statements by using extra compound statements after getting a large part of the nested structure into the program via the Editor. Thus, a common error is to add the END but forget the corresponding BEGIN that should be placed earlier in the program. Figure 6-8 provides an illustration.

```
>Edit: A(djst C(py D(let F(ind I(nsrt J(mp K(ol R(plc Q(uit X(ch Z(ap [E.7h]
Pascal Compiler IV.0 b20-2
    1  0   0:d    1  (*$L IB2.TEXT*)
    2  2   1:d    1  PROGRAM IFBOMB;
    3  2   1:d    1  VAR W,X,Y,Z:INTEGER;
    4  2   1:0    0  BEGIN
    5  2   1:1    0   WRITE('Enter value of W:'); READLN(W);
    6  2   1:1   29   WRITE('X:'); READLN(X);
    7  2   1:1   58   WRITE('Y:'); READLN(Y);
    8  2   1:1   87   Z:=0;
    9  2   1:1   90   IF W > X THEN
   10  2   1:2   94     IF W > Y THEN
   11  2   1:3   98      Z := W
   12  2   1:2   98     ELSE
   13  2   1:3  101      BEGIN
   14  2   1:4  101       IF W = Y THEN
   15  2   1:5  104        Z := Y
   16  2   1:3  104      END
   17  2   1:0  107     END

---->  Syntax Error #  6
   18  2    :0    0   ELSE
   19  2    :0    0    Z := X;
   20  2    :0    0   WRITELN('Z=',Z);
   21  2    :0    0  END.
```

Figure 6-8. Nested IF statements with unmatched extra END.

The extra END appears on line 17. It should be matched by a BEGIN between lines 9 and 10. The END on line 17 is indented two columns less than line 16, a natural step to take when increasing the indentation by two for each additional statement level, and decreasing the indentation correspondingly for each statement level terminated. This time the Compiler again generates the ubiguitous "invalid symbol" message (error 6) which is virtually equivalent to "something is wrong but I don't see what".

The clue to look for in this situation is the level numbers on lines 15, 16, and 17. Since the level is shown as 0 in line 17, the Compiler considers this END to be the match for the BEGIN in line 4, i.e. the opening BEGIN of the block. But visual inspection of the program (if a reasonable effort at logical indentation has been made in writing the program) quickly shows that it had not been intended that the END in line 17 would be the closing END of the block. Otherwise that END would have been placed in the program with zero indentation. At this point we trace back, and discover that the level 3 statements are properly balanced, but that there is no BEGIN at level 2 to match the END in line 17. Since the nested IF structure began at level 1 in line 9, the END should necessarily have matched a BEGIN at level 2 somewhere after line 9. Thus the problem is narrowed quickly to the point where the infraction effectively took place.

Next, let us see what happens if one neglects to put in both the BEGIN and the END in a situation where the program logic is changed as a result. This is illustrated in Figure 6-9.

```
>Edit: A(djst C(py D(let F(ind I(nsrt J(mp K(ol R(plc Q(uit X(ch Z(ap [E.7h]

Pascal Compiler IV.0 b20-2

  1  0   0:d    1   (*$L IB4.TEXT*)
  2  2   1:d    1   PROGRAM IFBOMB;
  3  2   1:d    1   VAR W,X,Y,Z:INTEGER;
  4  2   1:0    0   BEGIN
  5  2   1:1    0    WRITE('Enter value of W:'); READLN(W);
  6  2   1:1   29    WRITE ('X:'); READLN(X);
  7  2   1:1   58    WRITE('Y:'); READLN(Y);
  8  2   1:1   87    Z:=0;
  9  2   1:1   90    IF W > X THEN
 10  2   1:2   95     IF W > Y THEN
 11  2   1:3  100      Z := W
 12  2   1:2  100     ELSE
 13  2   1:3  105      IF W = Y THEN
 14  2   1:4  109       Z := Y
 15  2   1:3  109    ELSE
 16  2   1:4  114     Z := X;
 17  2   1:1  117    WRITELN('Z=',Z);
 18  2    :0    0   END.

End of Compilation.
```

Figure 6-9. Nested IF's with BEGIN ... END missing.

In this illustration, the ELSE in line 15 has been left indented as if it belongs still with the IF ... THEN in line 9. However, if that were true, then the level of line 15 would be 1, as associated with the same ELSE in Figures 6-6 and 6-7. Since the level in line 15 is actually 3, it is clear that the ELSE associated back to the IF ... THEN in line 13, thus having quite a different effect than it did in the preceding figures. Here, there has been no error of syntax detectable by the Compiler, but there may well have been an error of program logic detectable because the level entries are not consistent with the indentation used when editing the program.

# Execution or Run-Time Errors

An execution error occurs at "run-time", i.e. while a program is running, if the program attempts an invalid action. A list of the execution errors detectable by the UCSD p-System is given in Appendix F of this book. The most likely error in most programs is a "Value range error", indicating that the program tried to assign a value outside the declared range of an array index or subrange variable. Other common errors are "stack overflow" (you ran out of working memory space), "integer overflow" (attempt to assign an integer value larger than can be expressed within a 16-bit memory word), "divide by zero", and "string too long".

When a run-time error occurs, the System halts and displays a three-line error message on the CRT. The top line is one of the messages tabulated in Appendix F. The second line contains an entry such as:

**Segment EDITDEMO, Proc #5, Offset #22**

meaning that the program halted within Segment EDITDEMO, Procedure (block) number 5, at a code offset of 22 bytes from the beginning of the block. The procedure and offset numbers correspond to the numbers in the third, and fifth columns of the Compiler's List option output.

As a concrete example, consider Figures 6-10 and 6-6-11. Figure 6-10 shows a section of the program file COMPDEMO, which is supplied with the files on your STARTUP: diskette. Not shown is the statement which calls the function BLOWUP, in which X is given the value 5, and Y the value 15. Figure 6-11 shows the displayed output of this program. The top few lines in Figure 6-11 result from the parts of COMPDEMO which have simply been copied from the EDITDEMO program used in Chapter 4.

```
58  2  3:3   70     WRITELN('TYPE ANOTHER STRING');
59  2  3:3   90     READLN(S);
60  2  3:2  107    END;
61  2  1:0    0   END (*REPEAT2*);
62  2  1:0    0
63  2  1:d    1   FUNCTION BLOWUP (X,Y:INTEGER):BOOLEAN;
64  2  5:d    1   VAR
65  2  5:d    1    I,LB,UB:INTEGER;
66  2  5:d    4    CH:CHAR;
67  2  5:d    5    A:ARRAY[1..10] OF INTEGER;
68  2  5:0    0   BEGIN
69  2  5:1    0    LB:=X;
70  2  5:1    3    UB:=Y;
71  2  5:1    5    FOR I:=LB TO UB DO
72  2  5:2   15     BEGIN
73  2  5:3   15        A[I]:=I*I;
74  2  5:3   27      WRITELN(I,':  ',A[I]);
75  2  5:2   73     END;
76  2  5:1   78    BLOWUP:=UB > 10;
77  2  1:0    0   END (*BLOWUP*);
78  2  1:0    0
79  2  1:0    0   BEGIN (*MAIN PROGRAM*)
80  2  1:1    0    WRITELN('START EDITDEMO');
81  2  1:1
```

Figure 6-10. Display of Function BLOWUP.

```
Running...
START EDITDEMO

TYPE ANY STRING FOLLOWED BY <RET>
ANY
A
AN
ANY

TYPE ANY STRING FOLLOWED BY <RET>

5:   25
6:   36
7:   49
8:   64
9:   81
10:  100

Value range error
Segment EDITDEMO Proc# 5      Offset# 22
type <space> to continue
```

Figure 6-11. COMPDEMO output with run-time error.

At the bottom of Figure 6-11, we see that the program "blew up" in Segment 1, Block 5, and at a point in the code 24 bytes from the beginning of the block. Referring to Figure 6-10, we see that this offset occurs within line 73, which starts in byte 15 and ends in byte 26. (The dot which appears coincidentally on the same line is part of the output of the compiler's second pass.) Since the error was a "Value range error", we immediately suspect the index value I in the subscripted array variable A[I]. There are no other items in line 73 which would correspond to a Value range error. Now we trace back through the program to see where I might have taken on a value outside the range 1..10 which was declared in line 67. Since the value of UB is initialized to the value of Y when the function was

entered, and since the value of Y is 15, we see that the FOR statement will inevitably generate a value of 11. This is the first value outside the declared range, and hence is the value which will trigger the Value range error. We cross-check this conclusion with the displayed output of the program itself in Figure 6-11. The program ran long enough to display lines for values of I ranging from 5 through 10, but it failed to continue to display values from 11 through 15. Thus the conclusion is confirmed that the Value range error arose because of a value of I outside the allowed range.

As an exercise, try using a similar method to find the error in the REVERSE procedure of the same program. This can be found by running the program, and by responding to the second prompt message with a string which contains an even number of characters, for example "even".

Of course, not all execution errors are as easy to find as the error illustrated in this section. The error message allows you to find out which block contains the statement where the program finally blew up. It may then be necessary to insert extra WRITELN statements into the program to determine the values of essential variables at times immediately before the execution error occurs. These values may or may not make sense relative to the program logic, and it may be necessary to go back to earlier points in the program, again with extra WRITELN statements, to determine how the essential variables took on the offending values. For more complicated program debugging, the Debugger may be used. For a description of the Debugger see the *User's Guide for the UCSD p-System.*

# CHAPTER 7. PROGRAMMING TO USE DISK FILES

DISK FILES

## Contents


Goals for this Chapter ................ 7-3
Overview ............................ 7-4
Physical Description of UCSD Pascal
Disk Files ........................... 7-7
Sector Interleaving .................. 7-8
512-Byte Blocks - Universal Units of
Disk Transfer ....................... 7-9
Structured Logical Records ........... 7-11
Text Files ........................... 7-13
Working with Structured Data Files ... 7-16
File Declarations and the Buffer
(Window) Variable .................. 7-19
Sample Program - Sequential
File-to-File Copying .................. 7-32
Random Access Handling of Disk Files... 7-35
Sample Program UPDATE ............ 7-35
Indexed Access - Efficiency
Considerations ...................... 7-41
Text Files ........................... 7-44
READ and WRITE .................. 7-45
EOLN, READLN, WRITELN:
End-Of-Line ........................ 7-47
Efficiency Considerations ............. 7-51
Error Recovery ...................... 7-52

# NOTES

# Goals for this Chapter

Perhaps the single most important area of applications programming of concern to users of the UCSD p-System is the handling of disk files. Whether your interest is in business databases, word processing, experimental data collection, process control, or some other field, you are likely to need to work with disk files.

The main goal of this chapter is to provide an introduction to programming for disk files using the UCSD p-System. It is unfortunate that the present accepted standard definition of the Pascal language lacks facilities for several important aspects of disk file handling. Since UCSD Pascal extends the standard language to allow random access handling of disk files, readers are warned that some of the facilities described in this chapter will not be found in all Pascal systems, or will differ in those systems.

Specific learning goals for this chapter include the following:

a) Create a new disk file containing structured records.

b) Update selected records in the file created in step (a).

c) Using a Pascal program, create a new text file on disk. Read the contents of this file using the Editor.

d) Process the data contained in the file contained in step (c), changing selected data within that file. Read the contents of the altered file to check your results.

e) Write a program capable of running without abnormal termination even if certain disk processing Input/Output errors are encountered.

**Note:** This chapter does not provide a comprehensive review of all kinds of disk oriented Input/Output facilities that are available with the UCSD p-System. Readers interested in going further should refer to the USER'S GUIDE for the UCSD p-System.

# Overview

Disk files are commonly used for any and all of the following purposes:

1) Storage of information one wishes not to lose when the computer is turned off.

2) Storage of files of information too large to fit within the computer's main memory all at once.

3) Saving data representing the status of a partially completed long computing task. This permits restarting the tast without repeating the entire computation, should the task be interrupted for any reason.

4) Communication of files of data from one machine to another via physical transportation of the disks themselves.

In programming to use disk files one must be concerned about several levels of detailed information. On one level, the physical characteristics of the disk medium and the mechanical drive on which it runs are important. The relationship of these characteristics to the UCSD p-System, and the resulting file descriptions are the subject of "Physical Description of UCSD Pascal Disk Files" in this chapter.

In Pascal, a file is an ordered sequence or collection of data items all of which are of the same declared type. In this sense, a file is similar to an array. Unlike an array, a Pascal file may contain a variable number of data items. Moreover, the time required for a program to gain access to any one data item in a file may range from tens of milliseconds (i.e. hundredths of a second) to several tenths of a second. The time taken to access an item in an array is typically only a few tens of microseconds (i.e. tens of millionths of a second). Because of these differences, the means of handling the storage of data into Pascal files, and retrieval of data from those files, is very different from the handling of Pascal arrays.

The data items stored in Pascal files are often composed of structured data types, usually Pascal Records. Formally, a file may be composed of items declared to be of any type that can be declared in the language. One exception is that a file of items that are themselves files is generally not allowed. A special file type of considerable importance among Pascal users is the Text file, which consists of a stream of single character items broken into lines. Generally a text file is accessed sequentially rather than by random record selection.

Pascal language facilities for handling files take the form of built-in procedures and functions. The philosophy surrounding these procedures and functions in the accepted standard definition of Pascal is oriented toward the use of magnetic tape files. UCSD Pascal includes two additional built-in procedures (SEEK and CLOSE), and slightly alters definitions of those in the standard language, in order to provide random disk access following a philosophy very close to that of the standard language. These changes are considered controversial among language specialists in the Pascal community, and should be regarded as unique to UCSD Pascal. Other Pascal implementations use

their own approaches, each typically altering the standard language in subtle but different ways. For this reason, readers are strongly urged to isolate their uses of input/output references to disk files in a small number of easily modified procedures and/or functions. This will reduce the effort needed to modify a program developed in UCSD Pascal for use in another system.

"Working with Structured Data Files" in this chapter presents the built-in facilities of Pascal for working with disk files composed of structured data. Wherever practical, without detracting from the readability of the presentation, differences between standard Pascal and UCSD Pascal are pointed out.

"Random Access Handling of Disk Files" in this chapter applies the built-in facilities to random access handling of disk files.

"Text Files" in this chapter discusses text files with particular attention to their storage on disks. Since text files are byte-stream oriented, they may also provide the best means of handling the Input/Output connected with a wide variety of peripheral devices including those interfaced to the UCSD p-System by users themselves.

"Error Recovery" in this chapter discusses error recovery, a troublesome but extremely important topic. Disks and tapes provide imperfect media for the storage of data, and it is generally necessary to provide means for coping with errors. Errors can be made in the process of recording data on a disk, in reading the data back from the disk, or even in passive storage intervening between recording and reading.

# Physical Description of UCSD Pascal Disk Files

Data is recorded on magnetic disks for digital computers in a manner reminiscent of recording on home phonograph records. In both cases, the information is contained in a large number of (nearly) circular tracks. The tracks of a phonograph record actually form one long spiral track. On a computer disk, the tracks are separate concentric circles which are not connected with each other.

Digital information is stored on a computer disk within a thin magnetic recording surface very similar to the surface of a magnetic recording tape. The important difference between a computer disk and a cassette tape intended for playing back music is the manner in which the information is expressed electronically. On a computer disk, the manner of recording is designed to store binary digital information with a very low probability that errors will be made on playback.

Within one of the tracks on a computer disk, the data is stored as a stream of binary bits. Usually, the stream of bits is a multiple of 8-bits long, and logically considered to be a stream of 8-bit bytes. The full capacity of one track on a floppy disk is about 4000 bytes. The disk is made to rotate continuously, because the time delay to start the disk drive spinning fast enough for data to be read can be at least several seconds. The rate of rotation results in the transfer of data between disk and computer so fast that it cannot be processed while the transfer is under way.

## Sector Interleaving

The disk interface hardware is relatively simple and leaves much of the logic to be carried out by the computer's central processor. This makes it impossible to read or write two or more adjacent sectors on the disk during a single rotation. If sectors in adjacent locations going around one track are given numbers in sequential 1, 2, 3, 4, ... order, the result can be to force a full rotation of the disk in between read/write transfers involving sectors with adjacent numbers. Transfers of groups of sectors with adjacent numbers are so common that the numbering is often arranged to provide a physical separation between sectors with adjacent numbers. Thus the (logical) sector number sequence on one track might be 1, 14, 2, 15, 3, 16, ... In reading the sequential sectors 1, 2, 3, ... there is a time delay between finishing the read of one sector, and starting the next, because of the rotation time associated with the intervening sector not found in that sequence. For example, after finishing the read of sector 2, there is a time delay for sector 15 to be passed over before reading of sector 3 can begin. This time delay is used by the computer's central processor to catch up with its work associated with the read operation. In that way, it becomes possible to read a second sector after wasting only a very small portion of one disk revolution, rather than having to wait for more than one complete revolution if the sectors were in adjacent physical locations.

## 512-Byte Blocks - Universal Units of Disk Transfer

The UCSD p-System regards all disk files as if they were composed of "blocks" 512-bytes long. In this respect, a block can be thought of as if it were a logical sector. The System interacts with the hardware through a set of low level driver routines known as the "Basic I/O Subsystem" or BIOS. The BIOS accepts a request for transfer of a numbered block, and takes care of collecting together the actual sectors on the disk which (in combination) make up the block. From the point of view of the operating system (the control software part of the UCSD p-System), all disk Input/Output transfers take place via 512-byte units called blocks. Details of how the BIOS copes with the actual physical sectors on the disk are of no direct concern to the software, nor to most programmers.

The blocks on a disk are given successive integer numbers starting at 0 and counting upwards, i.e. 0, 1, 2, 3, 4, ... Since the number of sectors on one track often does not work out to provide capacity exactly equal to an integer multiple of 512-bytes, some blocks overlap two tracks. The BIOS is expected to accept a block number, and to handle all the details of making the equivalent of that block out of sectors actually stored on the disk. The operating system retains an area of memory called a "buffer" for each file which is in use. The buffer has capacity to store one complete block. The upper portion of Figure 7-1 illustrates this part of the process.

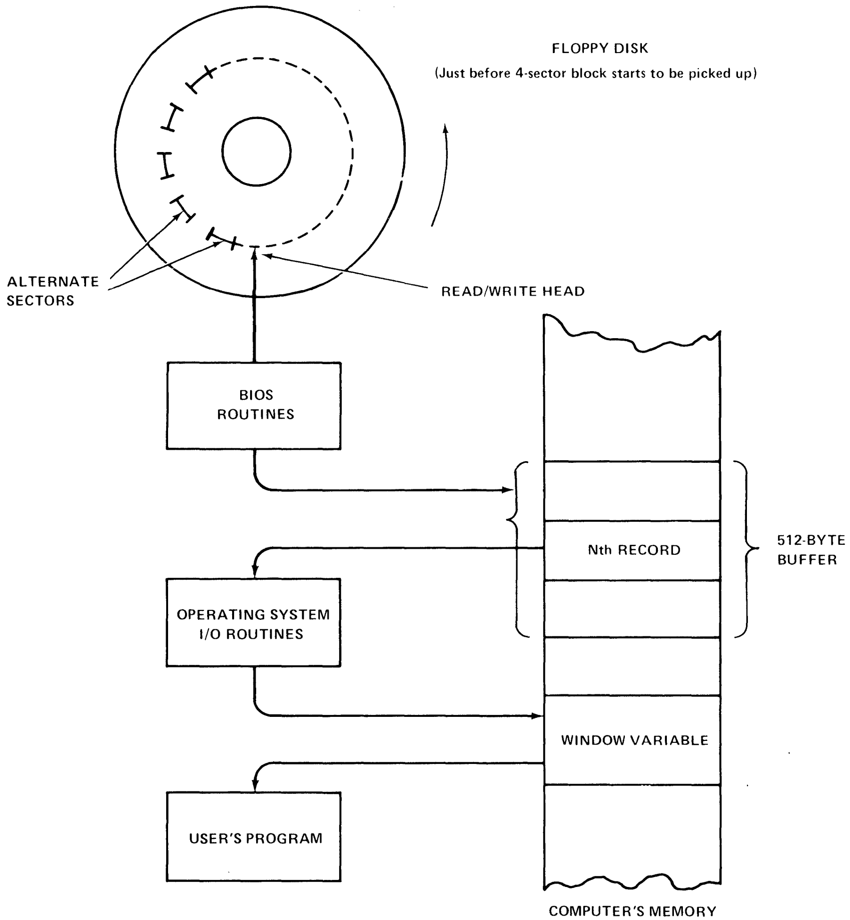



**Figure 7-1. Illustration of steps invoked on read of N'th record.**

## Structured Logical Records

Of course it is recognized that very few programmers will find it convenient to declare a Record type for their disk files that just happens to be exactly 512 bytes long. It is much more usual for the length of a Record to be less than 512 bytes and not evenly divisible into 512 bytes. Also, some Records are longer than 512 bytes, but not an integer multiple of 512.

To provide maximum flexibility for the Pascal programmer, the UCSD p-System takes care of packing logical typed Records into the 512-byte blocks, when writing to the disk, and unpacking the Records when reading from the disk. All of the necessary bookkeeping is done so that the programmer does not need to be directly concerned with the calculation of which block(s) any logical Record will occupy. Moreover, a logical Record may overlap from one block to another, allowing full use to be made of the storage capacity of each block in the file (except for the last one which usually is only partially occupied).

As a result, the user's Pascal program needs only to request access to a specific logical record in a file by using its number. For reading the N'th logical record from the file, Figure 7-1 shows how the operating system and BIOS routines team up to transfer just the requested Record into the "window variable" associated with the file. A similar process takes place in the reverse direction when writing to the disk. For a discussion of programming fine points associated with the file and its window variable, see "Working with Structured Data Files" in this chapter.

Sometimes the logic of a program will make it desirable to include Records, which are declared to be laid out quite differently, mixed together within the same file. Pascal allows you to declare that the

last field of a Record type has several different "variants". For example, we might want to mix together Records on people and on inventory items within the same file. We might also want to slip an occasional note into the file in the form of a long packed array of characters not broken into independent fields. The declarations associated with these records might appear as follows:

```
TYPE RECTYPE=(PERSON,INVENTORY,MEMO);
PERSREC=
  RECORD
   NAME,COMPANY:STRING[32];
   STREET:STRING[20];
   CITYSTATE:STRING[30];
   TEL:PACKED ARRAY[0..9] OF CHAR;
   BALANCE:INTEGER[8]
  END;
 INVREC=
  RECORD
   ITEMNAME:STRING[40];
   PLANT:INTEGER;
   LOCATION:PACKED ARRAY[0..3] OF
              CHAR;
   VALUE:INTEGER[6];
   DATE_ACQUIRED:PACKED ARRAY[0..5] OF
                    CHAR
  END;
 NOTEREC=PACKED ARRAY[0..131] OF CHAR;
 RECDEF=
  RECORD CASE RECTYPE OF
   PERSON:(PERS:PERSREC);
   INVENTORY:(INV:INVREC);
   MEMO:(NOTE:NOTEREC)
  END;
VAR
 RD:RECDEF;
```

I will defer until "Working with Structured Data Files" in this chapter any Pascal programming consideration of how to associate these Record types with a file. Notice that the type PERSREC occupies a total of 136 bytes (Strings include a length field. The total length of a string variable must fill an even

number of bytes), INVREC occupies 58 bytes, and NOTEREC 132 bytes. Since the UCSD p-System has no way to enforce which of the three types will occupy the variable RD at any instant, it is necessary for RD to occupy the maximum record size regardless of which type occupies RD. A similar consideration makes it necessary for a file constructed on disk from variant type records to use the size of the largest variant as the size of all records stored on the disk. In our example, this would mean that a substantial amount of disk space would be wasted if a large part of the file actually consisted of records of the INVREC type.

If you have in mind using disk storage for large numbers of records using several different Record types which differ markedly in size, it probably would be best to create separate files in order to save space. The largest record size is used if records vary in size. Variable length records are not arranged to occupy only their individual sizes on a disk. If they were, either access to these records would have to be sequential (i.e. not random), or a relatively complex indexing scheme would have to be used. The only simple method available in the UCSD p-System for storing variable length records on disk files uses Text files, as discussed in the following subsection.

## Text Files

The conceptual view of a Text file in Pascal is that it consists of an indefinite number of lines, each line being composed of an indefinite number of characters followed by an end-of-line marker. In this section I present a brief discussion of how this concept is implemented in the UCSD p-System. Programming details are deferred until "Text File" in this chapter. Most beginners who follow the Pascal rules on Text files will have very little reason to be concerned with the file characteristics

described in the rest of this section. They are presented here for those curious enough to get into trouble if they do not understand these details. The details will also be helpful to readers who wish to write programs which transfer text data between the UCSD p-System and another system which uses simpler text file conventions.

On the IBM Personal Computer, which uses the fixed length block scheme for disk storage, the storage of text information within the blocks is very simple. The characters in each line are written into a block sized buffer area in memory until the block fills up. The block is then transferred to the disk, and the buffer cleared for additional characters. The remaining characters on the line are then written into the buffer, and the end-of-line marker is also written into the buffer.

The design of Text files in UCSD Pascal has been strongly influenced by the requirements of the screen-oriented Editor. The objective was to make the Editor as fast and user-responsive as possible. Several of the design decisions have made UCSD Pascal Text files less similar to textfiles in other popular microcomuter systems than is probably necessary.

The principal differences between UCSD Pascal Text files, and the simpler format found on most small systems, are as follows:

a) The end-of-line character is a single ASCII CR character.

b) Blank characters at the beginning (left side) of a line are compressed into an "indentation code" which consists of an ASCII DLE character (decimal value is 16) followed by a character representing the number of blanks. The decimal equivalent value of this second character is 32 plus the number of blanks represented by the code. The indentation code-pair is missing if there are no blanks at the beginning of a line.

c) The text is written into the disk file in two-block logical records called "pages" (1024 bytes long).

d) No line of text is split between the end of one page and the beginning of the next. Instead, the empty space at the end of a page, which is too short to accommodate the line that would otherwise start there, is filled with ASCII NUL characters. The binary or decimal value of a NUL character is zero.

e) Page number zero of a Text file is reserved for control information used by the screen Editor. Text is stored in a Text file starting at the beginning of page number one. Unless you use special I/O facilities intended for advanced users of the UCSD p-System, a Pascal program using a Textfile will not be able to refer to the contents of page zero.

Note that the page-oriented layout, and the page zero requirement, are the reasons why the minimum size of a Text file in the UCSD p-System is 4 blocks.

# Working with Structured Data Files

The general philosophy associated with Input/Output operations in Pascal is conceptually similar to the model of a magnetic tape file. In this concept, the (usually imaginary) tape is seen as resting in a position such that one record of the type associated with the file will be transferred upon the next request for an input or output operation. The record is transferred from the tape to a special buffer variable associated with the file as a result of executing a GET statement. Similarly, the content of the buffer variable is transferred to the tape as a result of executing a PUT statement. The pointer indicating where the tape is positioned is advanced by the length of one record when GET or PUT is executed. In UCSD Pascal, as in standard Pascal, repeated execution of GET results in successive transfers of records in the sequence in which they are recorded on a disk. In standard Pascal, PUT may only be executed when the position pointer is at the current end-of-file position. One reason for this rule is that computer tape drives are not generally built to allow overwriting a previously written record within a file. Since UCSD Pascal is designed for work with disk files, which do allow overwriting records within a file, repeated execution of PUT is allowed as long as the file's position pointer indicates a record location within the range allocated to the file. UCSD Pascal provides a built-in SEEK statement for assigning a new value to the file's position pointer. Standard Pascal provides no equivalent of the SEEK facility.

All Pascal input and output data transfers take place via the buffer variable. Hence the buffer variable is sometimes referred to as a "window" through which the file may be viewed. The window, i.e. buffer, is treated as if it were an ordinary variable for purposes of assigning value to or from that variable within the Pascal program. However the buffer variable

behaves unlike ordinary variables in the sense that certain Input/Output operations leave the content of the buffer variable undefined, even when a value has previously been assigned by the Pascal program.

Pascal READ and WRITE statements are really composite statements constructed using GET and PUT respectively, in addition to implicit assignment statements involving the file's buffer variable.

In addition to dealing with the actual transfer of data to or from a disk file, the programmer also has to be concerned with instructions to the operating system on how the file should be handled. Until a file is "opened", using a Pascal RESET or REWRITE statement, the UCSD Pascal operating system does not allocate space for the 512-byte buffer illustrated in Figure 7-1. Moreover, it is necessary for the program to inform the operating system what directory name is to be associated with a file. The file declaration specifies the program's internal identifier and record type. However, the same file in the same program can be made to refer to many different disk files, each having a different directory name. In UCSD Pascal, the RESET and REWRITE statements of standard Pascal have been extended to provide the means for the program to communicate the file's directory name to the operating system. This allows one program to compute the directory names of several different files, each being associated with the same internal file identifier in the program at separate times. Only one of the actual disk files can be "open" at one time for input and/or output transfers.

Since there are several possible dispositions of a file after it has been used in a UCSD Pascal program, the program must generally inform the operating system how to "close" the file. This is done using the CLOSE statement, a UCSD extension to the language which does not exist in standard Pascal.

CLOSE with the LOCK option requests the operating system to retain the directory entry for a new file for future use. If LOCK is not used, a file newly created by the program will be regarded as temporary, and the space it occupies during the program execution will be marked as unused when the CLOSE action takes place. In addition to controlling the disposition of a file's directory entry, the CLOSE statement also informs the operating system that it may release the buffer space associated with the file for other uses. A CLOSE without LOCK is automatically invoked if no CLOSE statement has been explicitly executed before termination of the block (procedure or function) in which the file is declared.

Because the number of data items stored in a file may vary, it is necessary to provide a facility whereby a Pascal program repetition statement involving input or output may be told when to stop. For structured data files, the only facility provided for this purpose in standard Pascal is the built-in Boolean function EOF (End-Of-File). "EOF" in this chapter describes the handling of EOF in some detail because of its importance in controlling the other disk related statements.

As a final introductory note, RESET and CLOSE are automatically executed by UCSD Pascal for the predeclared files INPUT, OUTPUT, and KEYBOARD. Moreover, the definition of INPUT and OUTPUT files in UCSD Pascal differs from Standard Pascal because of fundamental differences in handling single-character transfers involving interactive terminals.

## File Declarations and the Buffer (Window) Variable

If T is a predeclared or user-declared data type in a Pascal program, one declares a file identifier, for example FID, along with other variable declarations as in:

```
TYPE
  T=RECORD
     IFIELD:INTEGER;
     B:BOOLEAN;
     S:STRING;
    END;
VAR
  FID:FILE OF T;
  X,Y:INTEGER;
  A,B:T;
      .
      .
      .
```

The buffer variable associated with the file FID is referred to as FID . The up arrow or carat character should not be confused with the up arrow cursor positioning key on your keyboard. Assuming that RESET or REWRITE has previously been executed for this file, it is then possible to assign the value of the file's buffer variable, for example:

```
A := FID^
```

or, if B has previously been assigned a set of values, then:

```
FID^:= B
```

Since the file is associated with a record type, it is also possible to assign individual fields from within the record as with any other record type variable. For example:

```
X := FID^.IFIELD
```

or

```
FID^.IFIELD := A.IFIELD
```

# GET, PUT, READ, and WRITE

Assuming that the file FID has been declared as in "File Declarations and the Buffer (Window) Variable", that the file is open, and that the position pointer of the indicated file points to a record actually stored on the disk, then:

**GET(FID)**

transfers one record from the disk file into the buffer variable FID. The record transferred is the one referred to by the position pointer. After the record is transferred, the position pointer associated with the file is advanced by one record position. Here again UCSD Pascal differs from Standard Pascal in which GET advances the position pointer before transferring the record. The order is reversed in UCSD Pascal in order to make SEEK operate in a straightforward manner.

If the position pointer associated with the file points to a position not occupied by a record when GET(FID) is executed, then the End-Of-File flag associated with the file becomes TRUE (see "EOF" in this chapter), and the content of FID^ is left undefined. Experimentation with UCSD Pascal will show you that execution of GET(FID) with EOF already set to true, or when the position pointer points outside the range of record numbers contained in the file, will leave the content of FID^ unchanged. This is not a behavior you should depend upon, since Standard Pascal specifies that the contents of FID^ are undefined under these conditions.

# GET, PUT, READ, and WRITE



Assuming the declarations shown in "File Declarations and the Buffer (Window) Variable", READ(FID,A) is equivalent to the following compound statement in Standard Pascal:

```
BEGIN
  A := FID^;
  GET(FID)
END
```

UCSD Pascal does not support the use of READ for structured data files, however.

Again assuming that the file FID is open, and that the position pointer points to a legal record position, then the statement:

```
PUT(FID)
```

transfers the current contents of the buffer variable FID^ to that record position in the disk file. It then advances the position pointer to the next position. Once again, the transfer of data takes place in Standard Pascal after the position pointer is advanced.

In Standard Pascal, the only legal record position prior to execution of a PUT statement is the position just before the first unoccupied record position at the end of the file. In UCSD Pascal, because of the need to update any record in a disk file, the legal record positions include all positions starting with record zero, and extending to the highest numbered unused position following the current end of the file. Therefore, the condition of EOF(FID) before PUT(FID) is executed has no effect on the PUT operation. If PUT transfers data to a record position located beyond the last currently occupied record in the file, i.e. in an unused area immediately following

the file and adjacent to it, then the disk directory will be updated to show that the file occupies all positions up to and including the position to which the transfer took place.

In UCSD Pascal, execution of PUT(FID) has no effect on the EOF(FID) flag unless it attempts to transfer data to an illegal record location, for example a location within a file which follows the open file to which PUT refers. In that case, EOF(FID) is set TRUE, and no data transfer takes place.

Analogous to READ, the draft description of Standard Pascal defines WRITE(FID,A) to mean:

```
BEGIN
  FID^:=A;
  PUT(FID)
END
```

UCSD Pascal does not, at this time, allow the use of WRITE with structured data files.

# RESET, REWRITE, and CLOSE



As described in the introduction to "Working with Structured Data Files" in this chapter, a program must inform the operating system when to allocate memory space to the block buffer for a disk file, and which file in the disk directory to equate with the internal file identifier. In the examples shown here, we will continue to use the internal file identifier FID, as declared in the example of "File Declarations and the Buffer (Window) Variable" in this chapter, but of course any declared file identifier could be used. The program passes the necessary information to the operating system by executing a RESET or a REWRITE statement. After one of these statements is executed successfully, i.e. without reporting an error, the file is then regarded as "open" and thus available to the program for input and output operations.

Unlike Standard Pascal, UCSD Pascal allows mixed input and output operations following either RESET or REWRITE. The draft description of Standard Pascal states that if PUT(FID) is not separated from a previous GET(FID) or RESET(FID) by an intervening execution of REWRITE(FID), then the results are implementation dependent. Since UCSD Pascal is designed to allow random access updating of disk files, it differs in several detailed respects from Standard Pascal regarding the opening and closing of disk files, and regarding the detection of End-Of-File status (see "EOF" in this chapter).

# RESET

To open a disk file which already exists, i.e. a file with a directory entry already on the disk, use:

**RESET(FID, <title-string> )**

where the title string may be either a quoted string or a string variable. Assuming that S is a variable of type STRING, either:

**RESET(FID, 'DATAFILE')**

or:

**S := 'DATAFILE';**
**RESET(FID,S)**

would open a file with the directory title DATAFILE for use associated with the internal identifier FID. Notice that this arrangement allows the value of the string variable S to be assigned while the program is running, perhaps through use of a READ(S) statement which calls for the user of the program to type in the name of the file.

The string parameter which gives the file's directory title in a RESET statement is a nonstandard extension unique to UCSD Pascal. Both Standard Pascal and UCSD Pascal provide the form:

**RESET(FID)**

In Standard Pascal, this is used to open the file, but the means of associating the file title with the internal identifier are left to be defined by the implementor (i.e. the person or organization that arranges to install Pascal within a software system).

# RESET

Once the file FID is open, the RESET intrinsic uses the form RESET(FID) to move the file's position pointer back to the beginning of the file. This also causes the contents of the first record stored in the file to be assigned to the file's buffer variable. Although UCSD Pascal is oriented to disk files, this operation is like performing a rewind operation on a magnetic tape, and then executing a single (Hidden) GET(FID) statement.

Either form of the RESET statement sets EOF(FID) = FALSE, assuming that there is no error indication. An error will be indicated if RESET(FID, filetitle ) cannot be completed because the requested filetitle cannot be found in the disk directory. The same form of RESET will also cause an error indication if FID is already open, since no indication will be available to the operating system at that point on what to do with the disk file that is already open. Repeated execution of RESET(FID) does not produce any error indication, since the only effect is to cause GET and PUT operations to start again at the beginning of the file. An error indication of either type mentioned here will cause your program to terminate abnormally unless you use a Compiler Directive which suppresses I/O error terminations. This option is provided to allow the programmer to arrange for program recovery in the case of Input or Output errors. See "Error Recovery" in this chapter for details.

# REWRITE

To open a new disk file, i.e. one with a title not matched by an existing directory entry, use:

**REWRITE(FID, <title-string> )**

where the title string is required. Unlike RESET, REWRITE in UCSD Pascal has no optional form without a title string parameter. REWRITE in Standard Pascal uses no title string, and leaves optional the question of how the directory title will be established.

This statement requests the operating system to establish a new temporary directory entry for the file, and to allocate the block buffer area in memory needed for input and output operations. The directory entry will be made permanent as a result of executing a CLOSE(FID,LOCK) statement. Any file already on the disk with a directory title which matches the title string will be removed in order to make way for the new file. Vestiges of a directory entry for the new file will also remain "permanently" on the disk, after the REWRITE is executed, if you open the door of the floppy disk drive and/or take the disk out of the machine. This is not an action that I advise, since the directory entry left on the disk in that way will probably not reflect accurately how many records have actually been PUT into the file. In fact, the directory will probably show that the file occupies the entire unused area to which it was allocated when the REWRITE was executed!

The form of the title string used with REWRITE will determine which unused area on the disk the

operating system will use in allocating space to the file. If you use a simple title string such as:

**NEWFILE**

then the file will be allocated starting at the beginning of the largest unused area currently in the directory. It will then be legal to execute PUT(FID) operations referring to any record position throughout that (initially) unused area. The ultimate size of the file, as shown in the directory after the file is closed, will be the number of blocks starting at the beginning of the unused area, and extending through the highest numbered block to which a PUT(FID) operation is directed.

If you know in advance that the new file will not have to occupy more than a certain number of blocks, then an alternate form of the title string may be useful. For example, if the number of blocks desired is no more than 15, then use the form:

**NEWFILE[15]**

Unfortunately, it is not possible to substitute a variable identifier for the desired number of blocks within this title string. However the value actually passed to the REWRITE(FID,S) statement can be computed by the program, and the value of a STRING variable S can then be composed using the STRING operations provided in UCSD Pascal. The operating system will respond to

**REWRITE(FID, 'NEWFILE[27]' )**

by allocating the file to the first unused area in the directory which contains at least 27 blocks. If no unused area at least 27 blocks long can be found, then an I/O error indication will result. In case of an

error, the program will terminate abnormally unless the methods described in "Error Recovery" in this chapter are used.

Another alternative is to use an asterisk within the square brackets after the file name:

```
REWRITE(FID, 'NEWFILEW[*]' )
```

This will cause the file to be created to fill either the second largest area on disk, or half of the largest area on disk, whichever is largest.

If you want to create two or more independent new files that are to be open simultaneously, you should use the square brackets in one of these two ways.

# CLOSE

After the completion of a program's work in a disk file, it may be necessary to request the operating system to deallocate the block buffer area assigned to the file in memory. In the case of a new file opened with REWRITE, a permanent directory entry must be completed, if the file is to be retained. In the case of an established file opened with RESET, the directory may have to be updated to reflect the use of record positions that had previously been in the unused area adjacent to the file. In UCSD Pascal, these operations are accomplished in response to execution of a CLOSE statement, for which there are

several forms. Standard Pascal provides no equivalent operation.

If the file has been opened with RESET, or if a new file opened with REWRITE is not to be retained, then use:

**CLOSE(FID)**

If the file is new, having been opened using REWRITE, and you wish to retain the file with a permanent directory entry, then use:

**CLOSE(FID, LOCK)**

If you wish to use a program to remove a disk directory entry, with an effect equivalent to the R(emove command of the Filer, then open the file using:

**RESET(FID, <title-string> )**

followed by:

**CLOSE(FID, PURGE)**

All forms of CLOSE will mark the file FID no longer open. Further attempts to use GET, PUT, SEEK, EOF, READ or WRITE referring to the file FID will result in an I/O execution error indication. Unless the method described in "Error Recovery" in this chapter is used, the program will then terminate abnormally.

If a program terminates normally, without ever executing CLOSE for any file that is open at termination time, then the first illustrated form of CLOSE is automatically executed for each open file.

# SEEK

To change the position pointer associated with an open file FID, use:

```
SEEK(FID, <record-number>)
```

where the record number is an integer valued arithmetic expression. For example:

```
SEEK(FID, 57);
SEEK(FID, INTVAR);
SEEK(FID, LASTREC - 2*I)
```

If the value of the record number is non-negative, then the next GET(FID) or PUT(FID) to be executed will refer to the disk record indicated by that value. If the value of the record number is negative, then the result of the SEEK is undefined.

SEEK with a non-negative record number always sets EOF(FID) to FALSE, regardless of whether the value of the record number is within the areas where GET or PUT operations would be legal. It is necessary to execute either GET or PUT to discover whether EOF(FID) will remain FALSE thus signifying successful completion of the GET or PUT. There is no equivalent of SEEK in Standard Pascal.

# EOF

The built-in Boolean "End-Of-File" function:

**EOF(FID)**

is used to determine the result of an Input/Output operation. Because of a desire to keep UCSD Pascal extensions beyond Standard Pascal to a minimum, EOF works somewhat differently in UCSD Pascal than in Standard Pascal when dealing with a disk file. This makes it unnecessary to extend the language with another special function to handle virtually the same purpose for disk files alone.

If the disk file FID is already open, RESET(FID) will leave EOF(FID) set to FALSE. If the disk file referred to by the title string is present in the disk directory, then RESET(FID, *title-string*) will leave EOF(FID) set to FALSE. Otherwise an I/O execution error will result. If there is enough room to allocate space for the requested new file, REWRITE(FID, *title-string* ) will leave EOF(FID) set to FALSE. Otherwise, an I/O execution error indication will result. If the record position pointer associated with an open file FID points to any position starting with position zero, and ending with the last position containing a valid data record, then GET(FID) will leave EOF(FID) set to FALSE. If the position pointer points to a location beyond the last valid data record, then GET(FID) will leave EOF(FID) set to TRUE.

If the record position pointer associated with an open file FID points to any position already established within the file, or to any position within the unused area following the file, the PUT(FID) will

leavel EOF(FID) set to FALSE. Otherwise PUT(FID) will leave EOF(FID) set to TRUE. SEEK(FID, *record-number*) will leave EOF(FID) set to FALSE if the value of the record number is non-negative. Otherwise the value of EOF(FID) will not be changed. If EOF(FID) is executed when the file FID is not open, it will return the value TRUE.

# Sample Program - Sequential File-to-File Copying

In this section we provide a simple concrete example of the use of the facilities just described for handling structured data files in UCSD Pascal. In this example, we copy the contents of one file into a new file on the disk. Both files are then left on the disk. Additional examples showing random access use of structured data disk files, are shown in "Random Access Handling of Disk Files" in this chapter.

```
PROGRAM FILECOPY;
CONST RECSIZE=199;
TYPE
 STRUCTURE=
  PACKED ARRAY[0..RECSIZE] OF CHAR;
VAR
 RECNUM:INTEGER;
 FIN,FOUT:FILE OF STRUCTURE;
BEGIN
 RESET(FIN,'OLDFILE');
 REWRITE(FOUT,'NEWFILE');
 RECNUM:=0;
 WRITE('Copying');
 WHILE NOT EOF(FIN) DO
  BEGIN
   FOUT^:=FIN^;
   PUT(FOUT);
   RECNUM:=RECNUM+1;
   WRITE('.');
   GET(FIN);
  END;
 WRITELN;
 WRITELN(RECNUM, 'records copied');
 CLOSE(FOUT,LOCK);
END.
```

Listing 7-1. Program which copies from OLDFILE to NEWFILE.

In this simple example, we ignore the internal layout of the fixed length records of type STRUCTURE. All we are concerned about is their total size, which is one more than the constant RECSIZE, or 200 bytes in this case. There is no information in the directory entry for a file indicating the structure of the records contained in the file. However, the directory does contain an integer value representing the number of the last record stored in the file. This number, multiplied by the size of the structured records originally PUT into the file, controls the value returned by EOF(FIN) following each use of GET(FIN). (Thus it is possible to refer to an old file

by associating the input file type with a structure whose size is not the same as that used in creating the old file. However doing this will yield records not matched to those originally written into the file, and the EOF function will return TRUE for a GET when the position pointer does not point at the true end of the file.)

In this sample program, we assume that the file with the directory title "OLDFILE" exists on the disk before the program is run. A file called "NEWFILE" might also exist before the program is run, but that previous file will be removed as a result of the REWRITE statement in this program. If you want to avoid inadvertent loss of an old file in this way, it would be best to try to RESET the old file (i.e. the existing file with the title "NEWFILE") as a first step. Your program can then inform you if the RESET(FIN, *oldfilename*) succeeds. If it does not, you will have to use the error recovery approach described in "Error Recovery" in this chapter to avoid abnormal termination of your program. The program FILECOPY leaves its new copy in the file "NEWFILE".

This program displays two lines on the computer's console device, namely:

```
Copying...........................
27 records copied
```

as direct verification for the user that the program is actually doing its work. The first line displays one dot after the PUT of the associated record is completed. The second line provides a simple summary. In general, you will probably find it useful to provide some visual indication of activity in any program that spends much time in disk I/O or other time consuming computations.

Notice that the GET takes place after the PUT within the main WHILE loop of the program. This is

because the first GET effectively takes place as a part of the RESET statement referring to the input file FIN. The last executed GET statement switches the EOF(FIN) flag to TRUE, and this information is immediately used to terminate the WHILE statement.



# Random Access Handling of Disk Files

In this section, we will start with a disk file containing name, address, and telephone number information on some imaginary people. We will then illustrate how to go about updating selectively chosen records already in the file, and also appending additional records to the file.

In the example given here, we will assume that it makes sense to determine which record to select from the file by simply making use of its record number. In practical applications, this is obviously not a suitable procedure, and some means of indexing the records in the file must be used. The last subsection provides a brief discussion of indexing strategies, but no solid sample program example because of space limitations.

## Sample Program UPDATE

Listing 7-2 shows a sample program which illustrates the creation and updating of a simple file. Display 7-1 shows a portion of the screen display associated with this program. All of the disk file handling is accomplished in the main program of Listing 7-2. In a larger program containing indexed access to the stored records, management of available record positions, and other amenities the disk file handling statements should be isolated in procedures which can be readily altered without changing the whole

program. This strategy reduces the amount of effort that may be necessary to change the program when moving from one machine to another with differing characteristics.

Handling of input from the keyboard, and display on the CRT screen, is very simple in this sample program. In this program, the user is prompted to type in the number of the record wanted. The current contents of the record are then displayed in a meaningful format. The user is then prompted to type in new contents for each field in the record separately. If the user wishes to leave a field unchanged, ENTER skips to the next field. ESC(ape followed by ENTER jumps out of the field updating cycle without any further change of a field. If the requested record position is in the unused area following the previous end of the file, the program prompts immediately for new contents.

```
PROGRAM UPDATE;
TYPE
 STRUCTURE=
  RECORD
   NAME,COMPANY:STRING[32];
   STREET:STRING[20];
   CITYSTATE:STRING[30];
   TEL:STRING[10]
  END;
VAR
 RECNUM:INTEGER;
 BUF:STRUCTURE;
 TITLE:STRING;
 FID:FILE OF STRUCTURE;
```

```
PROCEDURE ZEROREC
  (VAR REC:STRUCTURE);
BEGIN
 WITH REC DO
  BEGIN
   NAME:='';
   COMPANY:='';
   STREET:='';
   CITYSTATE:='';
   TEL:='';
  END;
END (*ZEROREC*);

PROCEDURE SHOWREC(REC:STRUCTURE);
BEGIN
 WRITELN;
 WITH REC DO
  BEGIN
   WRITELN('NAME:       ',NAME);
   WRITELN('COMPANY:  ',COMPANY);
   WRITELN('STREET:     '.STREET);
   WRITELN('CITY&STATE: ',CITYSTATE);
   WRITELN('TELEPHONE:',TEL);
  END;
END (*SHOWREC*);

PROCEDURE GETREC(VAR REC:STRUCTURE);
LABEL 1;
VAR S:STRING;

FUNCTION READIT(VAR T:STRING)
  :BOOLEAN;
 BEGIN
  READLN(S);
  READIT:=FALSE;
  IF LENGTH(S)>0 THEN
   IF S[LENGTH(S)]=CHR(27(*ESC*))
    THEN READIT:=TRUE
   ELSE
    T:=S;
  END (*READIT*);
```

```
BEGIN
 WRITE
  ('RETURN skips item with no change;');
 WRITELN
  (' ESC+RETURN skips whole Record');
 WRITELN;
 WITH REC DO
  BEGIN
   WRITE('NAME:        ');
   IF READIT(NAME) THEN GOTO 1;
   WRITE('COMPANY:    ');
    IF READIT(COMPANY) THEN GOTO 1;
   WRITE('STREET:     ');
   IF READIT(STREET) THEN GOTO 1;
   WRITE('CITY&STATE:');
   IF READIT(CITYSTATE) THEN GOTO 1;
WRITE('TELEPHONE: ');
   IF READIT(TEL) THEN GOTO 1;
  END;
1:
END (*GETREC*);

BEGIN (*main program*)
 WRITE('File title:');
 READLN(TITLE);
(*$I-*)    (*turn off I/O error checking*)
  RESET(FID,TITLE);
  IF IORESULT<>0 THEN REWRITE(FID,TITLE);
(*$I+*)    (*turn on I/O checking again*)
  RECNUM:=0;
  WHILE RECNUM>=0 DO
   BEGIN
    WRITELN;
    WRITE('Record number:');
    READLN(RECNUM);
    IF RECNUM>=0 THEN
```

```
      BEGIN
       SEEK(FID,RECNUM);
       GET(FID);
       IF EOF(FID) THEN
        BEGIN
         WRITELN('Enter new Record:');
         ZEROREC(FID^);
        END
       ELSE
        BEGIN
         WRITELN('Old Record:');
         SHOWREC(FID^);
         WRITELN;
         WRITELN('Enter Changes:');
        END;
       GETREC(FID^);
       SEEK(FID,RECNUM);
       PUT(FID);
      END;
   END (*WHILE*);
  CLOSE(FID,LOCK);
END.
```

Listing 7-2. Sample Program UPDATE.
(Program is contained on STARTUP: diskette.)

```
Execute what file? UPDATE
File title:NAMEFILE

Record number:3
Old Record:

NAME:              Bull, Terry
COMPANY:           Ramona Stock Farm
STREET:            Box 48 RFD #2
CITY&STATE:        Ramona, CA 92065
TELEPHONE:         789-1573

Enter Changes:
RETURN skips item with no change; ESC+RETURN skips whole Record

NAME:
COMPANY:
STREET:
CITY&STATE: Anytown, U.S.A._
```

Figure 7-2. Illustration of CRT display with UPDATE Program.

The Compiler Directive (*$I-*) turns off the IOCHECK option so that the program itself can cope with the problem that may arise if the file whose directory name is read into TITLE happens not to be on the disk. In this simple program, the response to this is to create a new file using the REWRITE statement. The use of the IOCHECK option in connection with recovery from I/O errors is discussed in "Error Recovery" in this chapter. If the IOCHECK option were not turned off, and the requested file not in the directory, then the RESET statement would cause the program to terminate abnormally.

The program cycles, each time requesting a new record number until a negative record number is typed in. It then terminates after closing the file with LOCK. If the file was already on the disk when the

program started, the LOCK option would be ignored in executing the CLOSE statement.

No provision is made for the problem which would occur if the requested record position were outside the existing file or unused area immediately following the file. In that case this program will terminate abnormally.

No provision is made for the case in which one simply wishes to read a record, and thus make no change at all in the record stored on the disk. In that situation, the PUT(FID) statement is not needed.

## Indexed Access - Efficiency Considerations

While the design of databases for use in the UCSD p-System is beyond the scope of this book, a few comments on efficiency and the construction of indexes may prove helpful.

In the terms relevant to this chapter, an index is a logical device which allows rapid determination of the position number of a record within a data file. The sequential index, which is one of the simplest index designs, is also one of the most useful in the interactive environment for which the UCSD p-System is designed. A sequential index is basically a table in which each entry is a record containing two fields:

a) A copy of one field of a record from the main data file, for example the name of a person. This field is called the "Key" referred to by the index.

b) The position number of the data record in the main data file which is associated with the value of the Key in field (a).

The "table" might be stored, during processing, in an array of records containing these two fields. It is more likely to be stored on two levels, one called the "coarse index" in a small array, and the other called the "fine index" stored in a file. Because of space limitations, only one "page" of this file will be brought into the computer's memory at any one time (See below for a definition of "page").

In any event, the records in the index table are sorted according to the value of the Key, usually in ascending order. This makes it possible to use an efficient searching algorithm such as a binary search to find any entry having a specific Key value in the index. In the interactive situation, one often knows only an approximate value for a desired Key. One may not even know whether a given Key is stored in the file and index. Either way, a binary search yields the location in the index where the Key would be located if it were present. It is then possible to display on the CRT screen a listing of a few index entries both before and after the one desired. A visual scan of this list will allow using a simple process to indicate whether the full data record associated with any particular index Key value should be retrieved from the main data file. The record position number associated with the Key in the index table entry is then used to make the desired random access to the data file. Many people refer to the use of a sequential index, to make random access to a data file, as the "Indexed Sequential Access Method", or simply "ISAM", because of the widespread use of that term on the large IBM computers.

In a floppy disk based interactive system, as in any other, there are questions about the design of the index, the order in which the data records of the main file should be stored, and many others. Typically the sequential index file is broken up into groups of several dozen to several hundred index table records, each group being stored in a separate

large record in the index file. This makes it possible to bring a whole group of index table records into main memory in one GET operation. Because it takes time to complete a GET operation, each group or "page" of index table entries should contain a reasonably large number of index entry records. If the number gets larger than can be accommodated in roughly half of one floppy disk track, then the time taken just to transfer the index page into memory becomes an important consideration.

If the size of the index is large enough to occupy several pages, a small "coarse" index should be maintained just to allow fast computation of the number of the index page in which a desired Key value will be found. The time taken to perform a binary search within one index page will usually be far smaller than the time taken to access, i.e. GET, just one index page from the disk. The entries in the coarse index usually contain copies of the last key value found in each page of the main index, which is now called the "fine" index. In floppy disk based systems, the size of the coarse index will almost always be quite small. It can therefore be loaded into memory when the program is initialized, and maintained there without disk accesses until the files are closed.

If the sorted sequential index table is arranged to fill every available index record position, then there will be serious problems in providing rapid interactive response to a user who is updating a file. Each update will probably require adding a new index record somewhere in the middle of the index, or deleting an old record from the middle. The time taken to resort a sequential index is likely to be prohibitive. With floppy disks, a good strategy is the following. A portion of each index page is kept unused and available for expansion of the contents of the page. It then is only necessary to PUT the updated page containing the Key when the update operation terminates.

Maintenance of the main data file will probably require occasional sorting at times when it is desired to conduct a batch (non-interactive) or bulk update operation involving a large fraction of all the data records in the file. Batch operations usually proceed sequentially from beginning to end of a file, rather than using randomly ordered accesses. The time needed to sort the main data file in a floppy disk system will probably run into many minutes, or even hours. However, the time saved in making it unnecessary to use an index in the batch update will often more than compensate for the time taken in the sort.

The data records are usually not moved during interactive updating activities. New records are appended to the end of the file, or they replace records that have been marked as "empty" during previous update transactions. Once it becomes necessary to sort the main data file, any indexes referring to that file must also be updated. The simplest and most efficient procedure will probably involve rebuilding the indexes after the sort is completed.

# Text Files

Much of the Input/Output environment of Standard Pascal is designed for working with Text files which can be thought of as stored on magnetic tape. UCSD Pascal provides two similar kinds of files for handling text streams of characters. A general description of Text files in UCSD Pascal was given in "Physical Description of UCSD Pascal Disk Files" in this chapter. One kind, associated with the predeclared type TEXT, works essentially the same as type TEXT in Standard Pascal. The other, associated with type INTERACTIVE, is intended primarily for use with interactive terminal devices. Both types can be

used with disk files however. File variable declarations have the following appearance.

```
FT:TEXT;
FI:INTERACTIVE
```

Differences between these two forms are explained in the following subsections.

## READ and WRITE

The READ statement is used to obtain characters from the input device, and to assign a value based on those characters to a variable within the program. If the variable is of type INTEGER or REAL, the value represented by the input stream of characters is converted into the internal binary form used by the program.

If CH is a variable of type CHAR, then:

```
READ(FT,CH)
```

is equivalent to:

```
BEGIN
 CH := FT^;
 GET(FT)
END
```

while:

```
READ(FI,CH)
```

is equivalent to

```
BEGIN
 GET(FI);
 CH := FI^
END
```

In effect, a READ involving a variable of any other type causes repeated use of this form. If X is a variable of type INTEGER or of type REAL, then:

```
READ(FT,X)
```

and:

```
READ(FI,X)
```

carry out format conversion of the input character stream, and the internal binary form of the number is assigned to X. In either case, termination of the READ operation occurs upon detection of the first character which is not legally part of a constant of INTEGER or REAL type, as the case may be. In either case, the value of the window variable is left equal to the first non-numeric character following the number scanned by the READ statement. This is equivalent to including the first implicit GET(FI) operation of the next READ(FI,...) at the end of READ(FI,X). Consequently, the next READ(FI,...) statement omits execution of the first implicit GET(FI). In this way, a sequence of READ(FI,...) or READ(FT,...) operations will produce the same values when reading the same disk file.

Note that RESET(FI, *title-string*) does not execute an implied GET(FI), whereas RESET(FT, *title-string*) executes GET(FT) automatically.

If S is a variable of type STRING, then:

```
READ(FT,S)
```

and:

```
READ(FI,S)
```

both assign all characters from the input stream to S up to the next end-of-line character, or up to the

maximum capacity of S. The end-of-line character is not moved into the string, and the file's character pointer is left pointing at the end-of-line character.

Output using:

```
WRITE(FT,CH)
```

or:

```
WRITE(FI,CH)
```

both produce the equivalent of:

```
BEGIN
  FT^ := CH;
  PUT(FT)
END
```

In other words, output using WRITE gives the same results regardless of whether the file is declared to be of type INTERACTIVE or TEXT. If WRITE refers to a variable of type INTEGER or REAL, then format conversion takes place from internal binary form to an external stream of characters.

## EOLN, READLN, WRITELN: End-Of-Line

Text files in Pascal are subdivided into lines, each consisting of a sequence of characters terminated by an End-Of-Line marker is a single ASCII CR control character (Carriage Return) which has a decimal value of 13. In Standard Pascal, the End-Of-Line marker is not regarded as a character, and it cannot be handled in the way normal characters are handled. If you use conventional Pascal I/O operations with Text files in UCSD Pascal, you will have no occasion to work directly with the CR control character.

The built in Boolean function EOLN(*file-identifier*) returns the value TRUE when the position pointer of

a Text file points at an End-Of-Line marker. This occurs at the termination of a READ statement which finishes its work because it encounters an End-of-line marker. Consider the program fragment shown in Listing 7-3:

```
WRITELN;
RESET(FT);
WHILE NOT EOF(FT) DO
 BEGIN
  WHILE NOT EOLN(FT) DO
   BEGIN
    READ(FT,CH);
    WRITE(CH);
   END;
  READLN(FT);
  WRITELN;
 END;
```

**Listing 7-3. Program fragment showing use of EOLN(FT).**

The first WRITELN statement moves the CRT display cursor to the left margin of a new line. (Remember that READ or READLN without an explicit reference to a file identifier refer by default to the predeclared file INPUT, which in UCSD Pascal obtains characters from the keyboard. Similarly WRITE and WRITELN refer by default to the file OUTPUT.)

RESET(FT) leaves EOLN(FT) and EOF(FT) set to FALSE. Individual characters from the first line in the file are then assigned to the character variable CH, and then written to the computer's console CRT (or other display device). READ(FT,CH) for the last text character on a line assigns that character to CH, and then executes the GET(FT) which picks

DISK FILES

up the End-of-line character. This leaves EOLN(FT) set TRUE, and the WHILE loop terminates.

READLN(FT) is equivalent to:

```
BEGIN
  WHILE NOT EOLN(FT) DO GET(FT);
  GET(FT);
END
```

which causes the file's position pointer to skip to the beginning of the next line of text. At the end of the WHILE statement in this fragment, the position pointer points to the End-of-line marker, the value returned by EOLN(FT) is TRUE, and the content of the window variable FT^ is a SPACE character. The single GET(FT) then advances the pointer and picks up the first character on the next line. The matching WRITELN does the equivalent on the display in preparation for the next line of text.

To produce precisely the same result using a disk file FI declared to be of type INTERACTIVE, the program fragment in Listing 7-4 should be used:

```
WRITELN;
RESET(FI);
WHILE NOT EOF(FI) DO
 BEGIN
  WHILE NOT EOLN(FI) DO
   BEGIN
    READ(FI,CH);
    IF NOT EOLN(FI) THEN
     WRITE(CH);
   END;
  READLN(FI);
  WRITELN;
 END;
```

**Listing 7-4. Program fragment showing use of EOLN(FI).**

The extra IF NOT EOLN(FI) ... within the inner compound statement is needed to suppress writing the blank character assigned to CH by the last READ(FI,CH) on a line. If the program is only to be used for display purposes, there may be no reason to include the extra IF statement, since the display of this implied blank will usually not be noticed. If the program is to be used for copying one disk file into another, there may be no reason to use files of type INTERACTIVE, since WRITE statements function in the same manner for files of both type TEXT and type INTERACTIVE. Thus the principal reason for using the program fragment shown in Listing 7-4 would be a desire to use precisely the same program taking its input character stream either from a disk file or from the keyboard (by using the title string "CONSOLE:").

In order to account for the difference between handling of files of type INTERACTIVE, from those of type TEXT, READLN(FI) is equivalent to the fragment:

```
WHILE NOT EOLN(FI) DO GET(FI)
```

The trailing GET(FI) is not needed, as in the case of READLN(FT) since the next READ(FI,...) statement will implicitly perform a GET(FI) as its first action.

WRITELN(FT) is equivalent to WRITELN(FI), and both have the effect of appending an End-of-line marker to the output file.

## Efficiency Considerations



For practical reasons associated with the way in which Text files have been implemented in the UCSD p-System, it will generally cost much less processing time to READ into variables of type INTEGER, REAL, or STRING than to carry out the equivalent steps using repeated READ(CH) with associated program logic. The same general observation also applies to WRITE.

Note that it is often convenient in UCSD Pascal to fill a sequence of character positions with blank SPACE characters using a statement like this:

```
WRITE(<output-file-id>,' ': <field-width> )
```

where field width is an integer valued expression. This is compiled to be roughly the equivalent of:

```
  FOR I:=1 TO <field-width> DO
WRITE(<output-file-identifier>,' ')
```

and thus is much slower than:

```
WRITE(<output-file-id>, S: <field-width> )
```

where S is a variable of type STRING which has been preassigned a string of SPACE characters.

If you plan to work with large Text files in UCSD Pascal, you will probably find it useful to become acquainted with several built-in procedures and functions provided with the UCSD p-System expressly for working with packed arrays of characters (of which strings are a special case). These include MOVERIGHT, MOVELEFT, SCAN, and FILLCHAR. These are implemented so as to run about as fast as possible in assembly language.

# Error Recovery

Before you do much work with disk files, you will learn that I/O related processing errors do occur, and that it would be best to write programs capable of recovering from those errors without terminating abnormally. The potential sources of error are many. They include the following, as well as others not mentioned here:

- Marginal recording or playback error due to a flawed surface on the disk itself, or due to improper adjustment of the disk drive. This often will cause just a single isolated bit to be recovered by a program incorrectly. The IBM Personal Computer provides hardware intended to check for errors of this type. The operating system then rereads the data on input, and attempts to complete a GET operation without an error being signalled. The data thus obtained will usually be correct, but may contain an error in one or more bytes.

- Failure of a recording or playback operation with the result that a complete 512-byte block of data is unrecoverable (i.e. for all practical purposes destroyed). This can happen as a result of having an intermittent eletronic failure, or as a result of a power failure at the time when a PUT operation is in progress.

- Failure of a READ statement due to encountering data of the wrong format. For example, a READ into an INTEGER variable will expect to find a blank SPACE character, "+", "-", or a numeric digit. If the first character is a letter or special punctuation character, the READ statement will fail on a format I/O error.

- Loss of a complete disk for some reason. An example might be excessive temperature in the

room where the disk is stored. Another might be failure of the disk drive mechanism.

- An attempt to PUT a record outside the disk area allocated to a file.

- An attempt to open a disk file that is not currently available on a disk drive (or volume) connected to the machine.

- An attempt to create a new disk file with a title that matches the title of a file already in the disk directory.

The UCSD p-System is programmed to terminate abnormally when an Input/Output related error is detected in a user program, unless the programmer has suppressed error termination logic using the Compiler Directive:

**(*$I-*)**

If this option is in use, then the programmer can determine whether each I/O statement has completed its work properly by checking the value of the built-in function IORESULT which returns an integer value. If the value of IORESULT is zero, then the most recent I/O related statement terminated normally, i.e. with no error. Otherwise, the value of IORESULT is determined by the nature of the error, and it can be used to control whatever recovery action the programmer may wish to take. The values of IORESULT correspond to the I/O error messages given in Appendix G of this book.

Just how your program should proceed to cope with an error once it has been discovered is a large topic that I cannot discuss in more than a cursory way in this book. As a brief example, let us assume that you want to create a new disk file which is to occupy an area of 100 blocks. Since the disk may already contain other files, it is possible that there will be no

unused area large enough to hold the file. A suitable recovery procedure for the user of your program might be to mount an alternative disk, which has previously been initialized with a UCSD Pascal file directory, but which is known to have enough space for the file. The program fragment shown in Listing 7-5 shows how this might be handled:

```
(*$I-*)
REPEAT
 REWRITE(FID,'VOLID:NEWFILE[100]');
 RSLT:=IORESULT;
 IF RSLT<>0 THEN
 IF RSLT=8 THEN
   WRITELN
('No room for file; Mount another disk')
  ELSE
   IF RSLT=9 THEN
    WRITELN
('Requested volume is not on-line')
   ELSE
    WRITELN
('Unable to open new file! Check disk drive');
UNTIL RSLT=0;
(*$I+*)
```

Listing 7-5. Program fragment illustrating use of IORESULT.

This program fragment is designed to loop until the REWRITE statement terminates normally. The user is given suggestions about how to cope with the most likely errors. The integer variable RSLT is needed to provide temporary storage for the value of IORESULT at termination of the REWRITE statement. Otherwise the WRITELN statement containing an error message will reset the value of IORESULT to zero, and the loop will terminate immediately whether there is an error or not!

In general, the IOCHECK option should be enabled again after any section of a program which requires it to be turned off in order to cope with specific errors. Otherwise the program may encounter an I/O error from which it cannot recover, yet it might continue to run causing further damage. To re-enable the IOCHECK option use the Compiler Directive (*$I+*) as shown in Listing 7-5.

As a general strategy for recovering from errors in working with disk files, you should generally arrange to save "backup" copies of master disk files periodically. How long the period is will depend upon how much work you are willing to do in recovering from an error which completely destroys the current working version of a file containing important data. To assist in backing up to an earlier version of your working file, it may be useful to retain a Text file containing a copy of all input from the keyboard which resulted in updates to that file. It then should be possible to rerun the update program using the copies "audit trail" of input text from that file, and thus to recreate the state of the main data file as it was just before the fatal error took place.

# NOTES

# CHAPTER 8. USING LIBRARIES OF SPECIALIZED ROUTINES (UNITS)



## Contents

Goals for this Chapter ................ 8-3
The Reason for Having Preprogrammed Units ................................ 8-3
Overview of Units .................... 8-5
A Sample Unit and its Use ............ 8-8
The Librarian ....................... 8-13

# NOTES

# Goals for this Chapter

Once you have become an experienced beginner in the use of Pascal, you are likely to realize that it would be possible to extend the Pascal language to simplify the writing of programs in whatever field of applications you happen to prefer. Rather than extending the language itself, it turns out to be better to provide sets of preprogrammed routines which perform frequently needed computations in various fields of applications. In the UCSD p-System, a set of preprogrammed routines can be grouped together in a separate "Unit" in such a way that any of the routines (procedures and/or functions) may be used as if they had been declared within the using Pascal program. Several Units may be grouped together into a disk file called a "Library".

LIBRARIES

The main goal of this chapter is to provide an introduction showing how preprogrammed Units and Libraries of Units can be used by Pascal programmers. Instructions on how to prepare a Unit to be used in this manner are beyond the scope of this book, but may be found in the *User's Guide for the UCSD p-System*. The introduction provided here is left rather general of necessity.

# The Reason for Having Preprogrammed Units

A principal reason for the growing popularity of Pascal is the fact that the language is powerful yet very concise. By "powerful" I mean that Pascal can be used with a minimum of effort to write programs in almost any field of application, not to mention its use in writing system software. By "concise" I mean that the translation of Pascal programs into executable form requires a relatively small, and

relatively simple, compiler compared with the compilers needed for COBOL, FORTRAN, PL/I, or similar general purpose programming languages. Another principal strength of Pascal is that programs written in Pascal tend to be relatively "clean", i.e. free of logical errors, compared to programs written in the other popular languages to perform the same actions.

A growing community of programmers have come to realize that Professor Wirth's original design for Pascal provides a remarkable balance between conciseness and power. Programmers who have started to use Pascal for creating large and complex programs often have realized that the language lacks various specialized facilities that they know are built into COBOL, FORTRAN, PL/I or other high level languages. They have then tried to bring about agreement with other programmers who want to use Pascal for similar purposes on how Pascal should be "extended" to provide the missing facilities. It has turned out to be virtually impossible to obtain any such agreement, because very few programmers agree on details of how the extensions should be designed. The only point on which agreement has been growing is that an international standard on Pascal is needed, and that it should be based almost entirely on Wirth's original definition of Pascal (with a few minor errors or misconceptions corrected).

The preprogrammed Units facility, provides a way to extend the utility of the language through the use of a very small set of simple extensions to the standard base language. This facility makes it possible to reduce the number of extensions contained in the UCSD Pascal language, while allowing the utility of UCSD Pascal to be greatly expanded.

# Overview of Units

In UCSD Pascal, a Unit is a collection of procedures and/or functions which can be used as if they were declared within the using program. A Unit may also contain CONST, TYPE, and VAR declarations, and these may be used as if they were declared in the using program. A Unit is similar to an "Include file" (see "Include Directive" in Chapter 6 of this book) in that the content of the Unit is prepared separately from the text of the program in which it will be used. Unlike an Include file, a Unit is usually compiled separately from the using program. This makes it unnecessary to spend the time needed to compile the Unit each time it is used.

An Include file is just an ordinary text file, the contents of which are substituted by the Compiler for the directive:

**(*$I Include-file-name*)**

Thus the Compiler must treat the entire contents of the Include file as if they were contained in the main source program file. If the Include file is long, then a large amount of compile time may be needed each time one compiles the program file containing the Include directive, even though the Include file may never be changed.

A Unit is prepared in two main sections, the INTERFACE section and the IMPLEMENTATION section. The INTERFACE section contains CONST, TYPE, and VAR declarations, as well as PRODCEDURE and FUNCTION heading declarations. All of these declarations look just as they would if the same declarations were placed directly in the program which uses the Unit. All of the declarations in the INTERFACE section are intended to be treated as if they were actually present within the declarations at the global level of

the program which uses the Unit. All of the declarations in the INTERFACE section are intended to be treated as if they were actually present within the declarations at the global level of the program which uses the Unit. The IMPLEMENTATION section contains any LABEL declaration, and additional CONST, TYPE, VAR, PROCEDURE and FUNCTION declarations, along with the local declarations and executable parts of all the PROCEDUREs and FUNCTIONs. The INTERFACE section may also contain an Initialization Section for the Unit itself. (For more information about these see the *User's Guide*.

The using program may refer only to the items contained in the INTERFACE section. All of the contents of the IMPLEMENTATION section are considered to be "private" to the Unit, and not available directly to the using program. The contents of the INTERFACE section are considered to be "public" and thus available directly to the using program.

During compilation of the using program, the contents of the INTERFACE section of a Unit are treated as if they were in an Include file referred to by an Include directive at the beginning of the using program. This allows the compiler to treat the public parts of a Unit just as if they had been included in the using program. Since the IMPLEMENTATION section of the Unit is precompiled, it does not need to be compiled again. In other words, the executable code part of the Unit needs to be generated by the compiler only once - not each time the Unit is used. However the routines (PROCEDUREs and FUNCTIONs) whose headings appear in the INTERFACE section of the Unit may be called by the using program just as if their entire contents had been compiled along with the program.

The advantage of this approach is that a programmer can now be given a large library of routines designed to carry out most of the "primitive" operations commonly needed to write Pascal programs for almost any field of applications. For example, anyone who writes a program designed to display data on a CRT screen, and/or to collect input data by filling in the blanks in a "form" displayed on the screen, needs to perform certain simple operations over and over again. These operations may include placing the cursor at a particular location on the screen, clearing all parts of a line to the right of the cursor, clearing the entire screen from the cursor location to the end, underlining a specified "field" of columns on one line and accepting only certain data values within that field, and so on. The Screen Control Unit, which provides routines for these and other purposes is available as part of the UCSD p-System.

# A Sample Unit and its Use

Listing 8-1 shows the INTERFACE section of a simplified screen control Unit. Listing 8-2 shows a test program SCDEMO which uses this Unit. Both the test program and the Unit with the files are available on the STARTUP disk. You should be able to experiment with the program to verify your understanding of how Units work.

```
UNIT SCDEMO;
INTERFACE
TYPE
  SCCHSET = SET OF CHAR;
  SCKEYCOMMAND =

(BACKSPACEKEY,ETXKEY,UPKEY,DOWNKEY,
    LEFTKEY,RIGHTKEY,NOTLEGAL);

VAR
  SCCH:CHAR;

PROCEDURE SCINITIALIZE;
PROCEDURE SCLEFT;
PROCEDURE SCRIGHT;
PROCEDURE SCUP;
PROCEDURE SCDOWN;
PROCEDURE SCGETCCH(VAR CH:CHAR;
        RETURNONMATCH:SCCHSET);
FUNCTION  SCMAPCRTCOMMAND
  (KCH: CHAR): SCKEYCOMMAND;

IMPLEMENTATION
(*The Implementation section begins here*)
```

Listing 8-1. Sample of a Simplified Screen Control Unit.
(Only the INTERFACE section is shown.)

```
PROGRAM TESTSCUNIT;

USES (*$U SCDEMO.CODE*) SCDEMO;

VAR DONE:BOOLEAN;CH:CHAR;
 CHOK:SCCHSET;

PROCEDURE SQUAWK;
BEGIN
 WRITE(CHR(7(*BEL*)));
END;

PROCEDURE CONTROL
              (CMD:SCKEYCOMMAND);
BEGIN
 IF CMD IN [BACKSPACEKEY, UPKEY,
   DOWNKEY,LEFTKEY,RIGHTKEY,
   EXTKEY] THEN
  CASE CMD OF
    BACKSPACEKEY:
     BEGIN
      SCLEFT;
      WRITE(' ');
      SCLEFT;
     END;
    LEFTKEY: SCLEFT;
    RIGHTKEY: SCRIGHT;
    UPKEY: SCUP;
    DOWNKEY: SCDOWN;
    ETXKEY: DONE:=TRUE
  END (*CASES*)
 ELSE
  SQUAWK;
END (*CONTROL*);
```

LIBRARIES

```
BEGIN (*MAIN PROGRAM*)
 SCINITIALIZE;
 CHOK:=[CHR(0)..CHR(31), 'A'..'Z'];
 WRITE
 ('Arrow keys move cursor <CTRL-C> <BS>');
 DONE:=FALSE;
 REPEAT
  SCGETCCH(CH,CHOK);
  IF CH IN [CHR(0)..CHR(31)] THEN
   CONTROL(SCMAPCRTCOMMAND(CH))
  ELSE
   WRITE(CH);
 UNTIL DONE;
END.
```

Listing 8-2. Sample Program TESTSCUNIT which Uses the SCDEMO Unit.

The program TESTSCUNIT makes use of the procedures and functions contained in the SCDEMO Unit. TESTSCUNIT also makes use of the scalar type SCKEYCOMMAND, and the variable SCCH, both of which are declared in the INTERFACE section of the Unit.

For example, the first statement in the main section of the program TESTSCUNIT is a call to the procedure SCINITIALIZE which is contained in the Unit SCDEMO. Only the heading of the procedure appears in the INTERFACE section. In effect, the heading of the procedure here is like a FORWARD procedure declaration. The body of the procedure is declared in the IMPLEMENTATION section of the Unit, and its detailed contents are of no concern to us in working on the program. Of course one needs to know what each procedure in the Unit does in order to write the program sensibly. The SCINITIALIZE procedure is used to load initial values into tables used by the other procedures and function in the Unit.

The program TESTSCUNIT provides a means of moving the cursor about on the screen, and for typing upper case letters wherever the cursor may be located. The BACKSPACE key may be used to back over and erase a displayed character. Pressing CTRL-C causes the program to terminate. Cursor movement is controlled by the procedures SCUP, SCDOWN, SCRIGHT, and SCLEFT, all of which are contained in the Unit.

The procedure SCGETCCH is used to read one character (a Command CHaracter) from the keyboard, returning the value of that character in the variable parameter CH. The procedure fails to return if a character typed on the keyboard is not in the set RETURNONMATCH. Instead, additional characters are read from the keyboard until a character falling in that set is pressed. In the main program of TESTSCUNIT, the variable CHOK is initialized to the set of characters considered "OK" when SCGETCCH is called within the REPEAT loop.

If the character returned by SCGETCCH falls in the group of control characters, which in ASCII have decimal equivalent values ranging from 0 to 31, then the procedure CONTROL is called. The single parameter of CONTROL is of type SCKEYCOMMAND, which is the scalar type declared in the Unit. But the values returned by SCGETCCH correspond to the codes assigned to the keys on your keyboard. Since there are no industry standards on which character codes should be associated with the cursor positioning arrows (Up, Down, Right, Left), it is necessary to arrange for the arrow keys to cause the corresponding display procedures to be called. This is accomplished with the help of the function SCMAPCRTCOMMAND which accepts a character value as its input parameter, and returns a value of type SCKEYCOMMAND. This function makes use of a

table hidden in the Unit which relates each ASCII control code to one value of type SCKEYCOMMAND. The values in the table are initialized by the procedure SCINITIALIZE, often by reading information stored in the miscellaneous information file supplied with the UCSD p-System.

You might ask why we do not simply arrange to have the arrow keys move the cursor on the screen without having to handle the problem explicitly in a Pascal program. The reason is that many programs are written to control the response to cursor movement key commands in different ways depending upon circumstances. For example, in the Editor's I(nsert command, use of the arrow keys could cause a mess on the screen if they were not trapped out and translated into the question-mark character "?". However, outside the I(nsert command, the Editor makes use of the arrow keys to move the cursor in the familiar way.

Notice the two lines immediately following the program's heading line, i.e.

```
PROGRAM TESTSCUNIT;
USES (*$U SCDEMO.CODE*) SCDEMO;
```

The Unit is made available to the program by the USES statement, which must appear immediately following the PROGRAM heading, before any of the program's own declarations. The comment line contains an optional compiler directive which informs the Compiler which disk file to reference for any subsequent Units referred to by the USES statement. If all the Units you wish to employ are in the file SYSTEM.LIBRARY then there is no need to employ the directive:

```
(*$U library-filename*)
```

since the Compiler assumes that all Units referred to in the USES list are to be found in

SYSTEM.LIBRARY unless told otherwise. If you want to use Units called UNITA and UNITB, both located in the SYSTEM.LIBRARY, and also the Unit SCDEMO as above, then the USES statement should read as follows:

```
USES UNITA,UNITB,
(*$U SCDEMO.CODE*)
      SCDEMO;
```

If Units from several different files are to be used, then place the appropriate compiler directive referring to each file before the list of Units contained in that file in the USES statement. The program may contain only one USES statement.

LIBRARIES

## The Librarian

A utility "Library" program file called LIBRARY.CODE is supplied with the UCSD p-System for the purpose of binding together various separately compiled Units into a single "library" file or to bind units into a host file. This utility is described in the *User's Guide for the UCSD p-System.*

# NOTES

# APPENDIXES

## Contents


Appendix A. Special Keys ............. A-1
Appendix B. Screen Editor Commands... B-1
Appendix C. Filer Commands .......... C-1
Appendix D. Operating System Commands .......................... D-1
Appendix E. Compiler Syntax Error Messages ............................ E-1
Appendix F. Execution Error Messages... F-1
Appendix G. I/O Error Messages ...... G-1

# NOTES

# APPENDIX A. SPECIAL KEYS ON THE IBM PERSONAL COMPUTER KEYBOARD

## UCSD p-System Related Keys

The following table lists the special keys that are used by the UCSD p-System on the IBM Personal Computer:

| FUNCTION | KEY |
|---|---|
| ESC | ESC |
| DEL | Ctrl-BACKSPACE |
| RETURN | The bent left arrow key. |
| EOF | Ctrl-C |
| BACKSPACE | The left arrow at the upper right of the keyboard. |
| ETX | Ctrl-C |
| TAB | The left/right arrow key at the upper left of the keyboard. |
| BREAK | Ctrl-BREAK - This key is a "hard" break which forces an immediate halt in program execution, followed by system re-initialization. |

| | |
|---|---|
| BREAK | Ctrl-@ - This key is a "soft" break which causes a halt in program execution at the next I/O operation, followed by System re-initialization. |
| STOP | Ctrl-S - Stops execution at the next I/O operation until pressed again. |
| DC1 | Ctrl-Q - In Editor's I(nsert mode, pressing this twice jumps to left margin. |
| LF | Ctrl-RETURN |
| FLUSH | Ctrl-F - Discards output waiting to be displayed. |
| INS | INS - Inserts a blank character in the Editors eX(change mode. |
| DEL | DEL - Deletes a character in the Editors eX(change mode. |

# CRT Related Keys

The PRINT key will send whatever is currently displayed on the CRT to the Printer.

When the screen is in 40 character mode Ctrl-Right Arrow will shift the window 20 characters to the right. Ctrl-Left Arrow will perform the same shift 20 characters to the left. These commands will wrap around if the right-most (or left-most) portion of the screen is already displayed. Also, in 40 character mode, the usual cursor moving keys (Left Arrow, Space Bar, etc.) will perform the same shift of the display window whenever the cursor moves off the screen.

For color monitors, ALT-C is a toggle which turns color on and off.

# APPENDIX B. SCREEN EDITOR COMMANDS

*repeat factor* is a number typed before any of the following commands. If not typed at all, value of *repeat factor* is assumed to be 1. '/' in place of a number causes repetition until end of file is reached.

*direction* is either forward or backward. Current *direction* is indicated by the broken bracket in 1st character position of the prompt line. '>' signifies forward. '<' signifies backward. Press '>' key to force direction to be forward, or '<' to force backward.

*down-arrow* moves *repeat-factor* lines down.

*up-arrow* moves *repeat-factor* lines up.

*right-arrow* moves *repeat-factor* spaces right.

*left-arrow* moves *repeat-factor* spaces left.

*space* moves *repeat-factor* spaces in *direction*

*back-space* moves *repeat factor* spaces left.

*tab* moves *repeat-factor* tab positions in *direction.*

*return* moves to the beginning of line *repeat-factor* lines in indicated *direction.*

"<" "," "-" changes indicated *direction* to *backward.*

">" "." "+" changes indicated *direction* to forward.

"=" moves to the beginning of what was just found, replaced, inserted, or exchanged.

A(djust: Adjusts the indentation of the line that the cursor is on. Use the arrow keys to move. Moving up (down) adjusts line above (below) by same amount of adjustment as on the line you were on. *repeat-factor* is valid. *ETX* terminates.

C(opy B(uffer - Copies what was last inserted, deleted, or zapped into the file at the position of the cursor.

C(opy F(ile - Copies from a portion or all of a text file that exists in the directory. Partial files are identified by use of file markers.

D(elete: Treats the starting position of the cursor as the anchor. Use any moving commands to move the cursor. *ETX* deletes everything between the cursor and the anchor. *ESC* cancels the deletion.

F(ind: Operates in L(iteral or T(oken mode. Finds the *target* string. Use any special character to delimit the *target. repeat-factor* is valid. *direction* is applied. 'S' may be substituted for the *target* previously used.

I(nsert: Inserts text. Use *Backspace* to erase one inserted character, *DEL* erases last inserted whole line. *ETX* accepts inserted text. *ESC* cancels the insertion.

J(ump: Jumps to the B(eginning, E(nd or previously set marker.

M(argin: Adjusts anything between two blank or command lines to the margins which have been S(et in the E(nvironment. Command lines are identified by ^ in 1st column. Invalidates the copy buffer.

P(age: Moves the cursor one page in indicated *direction. repeat-factor* is valid. *direction* is applied.

Q(uit: Leaves the editor. You may U(pdate, E(xit, W(rite, or R(eturn.

R(eplace: Extension of the F(ind command. Operates in L(iteral or T(oken mode. Replaces the *target* string with the *substitute* string. V(erify option asks you to indicate whether each occurrence of the *target* is to be replaced or skipped. S may substitute for either *target* or *substitute* and means that previous *target* or *subst* is to be used. *repeat-factor* applies. *direction* is valid. *ESC* aborts R(eplace before specifications are complete.

S(et: M(arkers by assigning a string name to them.

S(et: E(nvironment for A(uto-indent, F(illing, margins, T(oken, and C(ommand characters.

V(erify: Redisplays the screen with the cursor in center of screen.

eX(change: Exchanges the current text for the text typed while in this mode. Each line must be done separately. *backspace* causes the original character to reappear. *ETX* completes the exchange.

Z(ap: Treats the starting position of the last thing found, replaced, or inserted as an anchor and deletes everything between the anchor and the current cursor position.

# NOTES

# APPENDIX C. FILE MANAGER (FILER) COMMANDS

*wild* indicates wildcards can be used. = substitutes for all or part of each file name, and causes automatic reference to all files matching the resulting pattern. ? is similar to = but requests the user to indicate whether each individual file should be affected by the command.

B(ad blocks: Scans the disk and detects bad blocks.

C(hange: Changes a file or volume name. *wild*

D(ate: Lists the current system-disk date, and enables the user to change all or part of that date.

E(xtended list: Lists the specified directory as in L(dir but in more detail. *wild*

G(et: Loads the designated file into the workfile.

K(runch: Moves the files on the specified volume so that all the unused blocks are moved to the end of the disk.

L(dir: List a specified disk's directory or any subset thereof to the volume and file specified (CONSOLE: is default). *wild*

M(ake: Creates a directory entry with the specified name and size.

N(ew: Clears the workfile (workspace).

P(refix: Changes the current default volume identifier to the volume specified.

Q(uit: Returns user to the Command: world.

R(emove: Removes file entries from the specified directory. *wild*

S(ave: Saves the workfile under the name specified by the user.

T(ransfer: Copies the specified file(s) to the given destination. *wild*

W(hat: Identifies the name and state (saved or not saved) of the workfile.

V(olumes: Lists volumes currently on-line, along with their unit numbers.

eX(amine: Attempts to physically repair suspected bad blocks.

Z(ero: Creates a blank directory on the specified volume. The previous directory is no longer retrievable. Creates a directory on previously uninitialized disks (but does not {format} a previously unformatted disk).

# APPENDIX D. OPERATING SYSTEM COMMANDS

A(ssem: Executes the Assembler (SYSTEM.ASSEMBLER). The Assembler expects its input in SYSTEM.WRK.TEXT and generates its output in SYSTEM.WRK.CODE.

C(omp: Executes the Pascal Compiler (SYSTEM.COMPILER).

D(ebug: Invokes the Pascal Debugger which is a unit within the Operating System.

E(dit: Executes the Screen Editor (SYSTEM.EDITOR).

F(ile: Executes the File Manager (SYSTEM.FILER).

H(alt: Stops execution of the Pascal P-machine Interpreter. The System must be bootloaded to restart.

I(nit: Re-initializes the System.

L(ink: Executes the System's Linker program, used for linking assembly language routines to a "host" routine.

U(ser: Re-starts the program which most recently was executed.

X(ecute: Prompts for the name of a CODE file for a program to be executed. If that name (leaving off the ".CODE") is typed in, and terminated with RETURN, and the name is found in the disk directory, then the named program will be executed.

# NOTES

# APPENDIX E. COMPILER SYNTAX ERROR MESSAGES



| | |
|---|---|
| 1 | Error in simple type |
| 2 | Identifier expected |
| 3 | Unimplemented error |
| 4 | ) expected |
| 5 | : expected |
| 6 | Invalid symbol (terminator expected) |
| 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

54 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 of type SEMAPHORE

61 Must be of type PROCESSID

62 Process not allowed at this nesting level

63 Only main task may start processes

| | |
|---|---|
| 101 | Identifier declared twice |
| 102 | Low bound exceeds high bound |
| 103 | Identifier is not of the appropriate class |
| 104 | Undeclared identifier |
| 105 | Sign not allowed |
| 106 | Number expected |
| 107 | Incompatible subrange types |
| 108 | File not allowed here |
| 109 | Type must not be real |
| 110 | <tagfield> type must be scalar or subrange |
| 111 | Incompatible with <tagfield> part |
| 112 | Index type must not be real |
| 113 | Index type must be a scalar or a subrange |
| 114 | Base type must not be real |
| 115 | Base type must be a scalar or a subrange |
| 116 | Error in type of standard procedure parameter |
| 117 | Unsatisfied forward reference |
| 118 | Forward reference type identifier in variable declaration |
| 119 | Re-specified params not OK for a forward declared procedure |

| | |
|---|---|
| 120 | Function result type must be scalar, subrange or pointer |
| 121 | File value parameter not allowed |
| 122 | A forward declared function's 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 | Invalid 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 | Invalid 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 Invalid parameter solution

143 Invalid type of loop control variable

144 Invalid type of expression

145 Type conflict

146 Assignment of files not allowed

147 Label type incompatible with selecting expression

148 Subrange bounds must be scalar

149 Index type must be integer

150 Assignment to standard function is not allowed

151 Assignment to formal function is not allowed

152 No such field in this record

153 Type error in read

154 Actual parameter must be a variable

155 Control variable cannot be formal or non-local

156 Multidefined case label

157 Too many cases in case statement

158 No such variant in this record

159 Real or string tagfields not allowed

160 Previous declaration was not forward

161 Again forward declared

162 Parameter size must be constant

163 Missing variant in declaration

164 Substitution of standard proc/func not allowed

165 Multidefined label

166 Multideclared label

167 Undeclared label

168 Undefined label

169 Error in base set

170 Value parameter expected

171 Standard file was re-declared

172 Undeclared external file

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 INTERFACE section |
| 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 | Forward declaration was not segment |
| 192 | Forward declaration was segment |
| 193 | Not enough room for this operation |
| 194 | Flag must be declared at top of program |
| 195 | Unit not importable |
| 201 | Error in real number - digit expected |
| 202 | String constant must not exceed source line |
| 203 | Integer constant exceeds range |
| 204 | 8 or 9 in octal number |
| 250 | Too many scopes of nested identifiers |
| 251 | Too many nested procedures or functions |

| | |
|---|---|
| 252 | Too many forward references of procedure entries |
| 253 | Procedure too long |
| 254 | Too many long constants in this procedure |
| 256 | Too many external references |
| 257 | Too many externals |
| 258 | Too many local files |
| 259 | Expression too complicated |
| 300 | Division by zero |
| 301 | No case provided for this value |
| 302 | Index expression out of bounds |
| 303 | Value to be assigned is out of bounds |
| 304 | Element expression out of range |
| 398 | Implementation restriction |
| 399 | Implementation restriction |
| 400 | Invalid 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 | PROGRAM or UNIT expected |
| 406 | Include file not legal |
| 407 | Include file nesting limit exceeded |
| 408 | INTERFACE section not contained in one file |
| 409 | Unit name reserved for system |
| 410 | Disk error |
| 500 | Assembler error |

# NOTES

# APPENDIX F. EXECUTION ERROR MESSAGES

EXE MESSAGES

| | |
|---|---|
| 0 | System error (fatal) |
| 1 | Invalid index, value out of range |
| 2 | No segment, bad code file |
| 3 | Procedure not present at exit time |
| 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 (fatal) |
| 10 | User I/O error |
| 11 | Unimplemented instruction |
| 12 | Floating point math error |
| 13 | String too long |
| 14 | Halt, Breakpoint |
| 15 | Bad Block |

(fatal) indicates a fatal error. All fatal errors either cause the system to rebootstrap, or if the error was totally lethal to the system, the user will have to reboot. All errors cause the system to re-initialize itself.

# NOTES

# APPENDIX G. INPUT/OUTPUT ERROR MESSAGES

| | |
|---|---|
| 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 |

# INDEX

**Note:** Items containing '(' as second character are commands or command options. Items with lower case first character are generic.

## A


A(djust - Editor 4-42
arrow command - Editor 4-5
arrow keys 2-10
A(uto indent - Editor 4-49


## B


BACKSPACE A-1
B(ad blocks - Filer 5-44
block - disk 7-9
block-range - Filer 5-46
bootload 2-5
B(uffer - Editor 4-39
buffer - I/O 7-9


## C


C(enter - Editor 4-42
C(hange - Filer 5-35
CLOSE 7-23
.CODE 5-8
CODE file 3-7
Comment - Compiler 6-5
COMPDEMO program 6-14
Compiler 6-3
CONSOLE: 6-10
Ctrl key 2-10
C(opy - Editor 4-39
crunch directory 5-39
cursor 2-6
cursor movement 4-4


## D


Data: example 2-19
D(ate - Filer 5-24
D(elete - Editor 3-15
directory - disk 5-5
direction flag - Editor 4-5
diskette 2-5
disk file 5-5
dollar sign - Compiler 6-3
down arrow 4-9


## E


Editor 4-3
End-line character 7-14
End of Line 7-14
ENTER 2-10
E(nvironment - Editor 4-18
EOF 7-31
EOLN 7-47
Equal sign command - Editor 4-28
errors - disk 5-42

error recovery - I/O 7-52
ESC A-1
ESC - Editor D(elete 4-36
ESC - Editor I(nsert 4-30
ETX A-1
eX(amine - Filer 5-45
eX(change - Editor 4-43
E(xecute 2-12
execution error 6-27
E(xit - Editor 3-16
E(xtended directory - Filer 5-22

## F

file 5-3
F(illing - Editor 4-49
F(ind - Editor 4-21
floppy disk 7-7
F(rom file - Editor 4-39

## G

GET(FID) 7-21
G(et Filer 5-13

## I

IMPLEMENTATION section 8-5
Include - Compiler directive 6-7
indentation - Editor 3-12
I(nsert - Editor 4-30
INTERFACE section 8-5
I/O check - Compiler directive 6-13

## J

J(ump - Editor 4-17

## K

K(runch - Filer 5-39

## L

left arrow key 4-9
Librarian 8-13
List - Compiler directive 6-10
L(ist - Filer 5-17
L(it - Editor 4-21
L(iteral - Editor 4-21
LOCK (CLOSE) 7-29
logical record 7-11

## M

M(ake - Filer 5-37
M(argin - Editor 4-50
M(arker - Editor 4-17
Maze: exercise 2-6

## N

N(ew - Filer 5-15

## P

P(age - Editor 4-16
page - text file 7-15
P(refix - Filer 5-25

program concept 2-14
PURGE (CLOSE) 7-29
PUT(FID) 7-21

## Q

Quiet - Compiler
directive 6-14
Q(uit - Editor 3-17

## R

READ 7-20
R(emove - Filer 5-32
repeated commands -
Editor 4-9
R(eplace - Editor 4-37
RESET 7-23
REWRITE 7-23
right arrow 4-8
R(ight margin 4-42
R(un - a program 3-7
run-time error 6-27

## S

S(ame - Editor 4-37
S(ave - Filer 3-21, 5-15
scrolling - Editor 4-10
SEEK 7-30
S(et M(arker - Editor 4-18
S(et E(nvironment -
Editor 4-50
SPACE - Editor 4-14
<sub> Editor 4-37
switch - Compiler 6-13
syntax error 3-18, 6-14
SYSTEM.LIBRARY 8-13
SYSTEM.STARTUP 3-25
SYSTEM.WRK.CODE 5-11
SYSTEM.WRK.TEXT 5-11

## T

TAB - Editor 4-15
<target> - Editor 4-37
.TEXT 5-10
Text file 3-6, 7-15
title - file 5-6
title string 7-26
T(ransfer - Filer 5-26

## U

Unit - precompiled 8-5
<unused> - Filer 5-23
up arrow key 4-8
U(pdate - Editor 3-17, 4-46
USES (a Unit) 8-8

## V

V(erify - Editor 4-28
volume 5-5
V(olume - Filer 5-20

## W

W(hat - Filer 5-16
wildcard - Filer 5-8
workfile 3-6
WRITE 7-17
W(rite to file - Editor 4-47

# X

eX(amine - Filer 5-45
eX(change - Editor 4-43
eX(ecute - CODE file 2-12

# Z

Z(ap - Editor 4-44
Z(ero - Filer 5-40

= command - Editor 4-28
$ - Compiler directive 6-6

**Product Comment Form**

Beginner's Guide

6936583

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in anyway it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Comments:

If you wish a reply, provide your name and address in this space.

Name __________________________________

Address ________________________________

City __________________ State ____________

Zip Code ______________

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

# BUSINESS REPLY MAIL

**FIRST CLASS PERMIT NO. 123 BOCA RATON, FLORIDA 33432**

POSTAGE WILL BE PAID BY ADDRESSEE

IBM PERSONAL COMPUTER
SALES & SERVICE
P.O. BOX 1328-C
BOCA RATON, FLORIDA 33432

Fold here

Please do not staple

Tape

**Product Comment Form**

Beginner's Guide

6936583

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in anyway it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Comments:

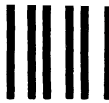
If you wish a reply, provide your name and address in this space.

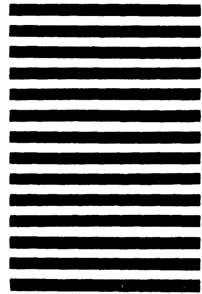
Name ____________________________________

Address __________________________________

City ____________________ State ______________

Zip Code _______________






# BUSINESS REPLY MAIL

**FIRST CLASS PERMIT NO. 123 BOCA RATON, FLORIDA 33432**

POSTAGE WILL BE PAID BY ADDRESSEE

IBM PERSONAL COMPUTER
SALES & SERVICE
P.O. BOX 1328-C
BOCA RATON, FLORIDA 33432

Fold here

Please do not staple

Tape

Continued from inside front cover

SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE.

IBM does not warrant that the functions contained in the program will meet your requirements or that the operation of the program will be uninterrupted or error free.

However, IBM warrants the diskette(s) or cassette(s) on which the program is furnished, to be free from defects in materials and workmanship under normal use for a period of ninety (90) days from the date of delivery to you as evidenced by a copy of your receipt.

**LIMITATIONS OF REMEDIES**

IBM's entire liability and your exclusive remedy shall be:

1. the replacement of any diskette(s) or cassette(s) not meeting IBM's "Limited Warranty" and which is returned to IBM or an authorized IBM PERSONAL COMPUTER dealer with a copy of your receipt, or
2. if IBM or the dealer is unable to deliver a replacement diskette(s) or cassette(s) which is free of defects in materials or workmanship, you may terminate this Agreement by returning the program and your money will be refunded.

IN NO EVENT WILL IBM BE LIABLE TO YOU FOR ANY DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE SUCH PROGRAM EVEN IF IBM OR AN AUTHORIZED IBM PERSONAL COMPUTER DEALER HAS BEEN ADVISED OF THE POSSIBLITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

SOME STATES DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU.

**GENERAL**

You may not sublicense, assign or transfer the license or the program except as expressly provided in this Agreement. Any attempt otherwise to sublicense, assign or transfer any of the rights, duties or obligations hereunder is void.

This Agreement will be governed by the laws of the State of Florida.

Should you have any questions concerning this Agreement, you may contact IBM by writing to IBM Personal Computer, Sales and Service, P.O. Box 1328–W, Boca Raton, Florida 33432.

YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT, UNDERSTAND IT AND AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS. YOU FURTHER AGREE THAT IT IS THE COMPLETE AND EXCLUSIVE STATEMENT OF THE AGREEMENT BETWEEN US WHICH SUPERSEDES ANY PROPOSAL OR PRIOR AGREEMENT, ORAL OR WRITTEN, AND ANY OTHER COMMUNICATIONS BETWEEN US RELATING TO THE SUBJECT MATTER OF THIS AGREEMENT.

International Business Machines Corporation

P.O. Box 1328-W
Boca Raton, Florida 33432

6936583

Printed in United States of America