Alan G. Porter and Martin G. Rezmer

BASIC Business Subroutines
for the
Apple II™ and IIe™
BASIC BUSINESS SUBROUTINES
FOR THE APPLE II AND Ile
Is there life after the *Applesoft Tutorial*? This book is a first step in the quest for the answer to this question. After reading the *Tutorial* cover to cover several times, you may still find it difficult to perform certain functions with your Apple computer. Even though you are armed with an understanding of how INPUT, PRINT, and FOR–NEXT work, you may still have difficulty putting them together in a meaningful order. You know what you want the computer to do, but you are not a professional programmer, so you do not know how to make the computer do it. In this book we present solutions to some of the most frequently encountered programming problems. We define what each problem is, show how to solve it, and give an exact solution (a program) in Applesoft BASIC. We also explain how you can modify the program for your own needs. The basis here is to learn efficient programming techniques and good style by example—a very powerful teaching method.
This book is intended for use by people with widely varying skill levels—from the enthusiastic novice to the advanced programmer. Each chapter will provide you with tools and building blocks to be used in programs you will create in the future.

The material is presented in modular fashion. That is, the materials from Chapter 2 (an input line editor) are expanded on in Chapter 3 to create a screen editor, and so on. The end result is a set of tools that can be used in every program you write. With these tools your programs will be more professional and easier to use, take less time to write, and be able to be modified easily when change becomes necessary.

Our special thanks to the following people: Carol Beal, Tom Bell, Zach Bovinette, Kathy Cukar, Takeshi Endo, Barb Odom, Vicki Porter, Jim Speir, Hal Tobin, Shelley Wright.
# CONTENTS

## PROGRAMMING FUNDAMENTALS

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>Subroutines: What They Are and How to Use Them</td>
<td>2</td>
</tr>
<tr>
<td>Programming Style</td>
<td>4</td>
</tr>
<tr>
<td>Writing Your Programs</td>
<td>8</td>
</tr>
<tr>
<td>Make the Apple Work for You</td>
<td>9</td>
</tr>
<tr>
<td>About the Structure of the Book</td>
<td>11</td>
</tr>
</tbody>
</table>

## AN INPUT LINE EDITOR FOR APPLESOFT BASIC

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>13</td>
</tr>
<tr>
<td>Line Editor Test Routine</td>
<td>17</td>
</tr>
<tr>
<td>Basic Line Editor Program</td>
<td>20</td>
</tr>
<tr>
<td>Displaying a Cursor</td>
<td>33</td>
</tr>
<tr>
<td>Processing a Key</td>
<td>35</td>
</tr>
<tr>
<td>Processing Control Keys: Editing Routines</td>
<td>39</td>
</tr>
<tr>
<td>CONTENTS</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td></td>
</tr>
<tr>
<td>User Instructions</td>
<td>49</td>
</tr>
<tr>
<td>Complete Line Editor Program</td>
<td>50</td>
</tr>
</tbody>
</table>

Chapter 3

<table>
<thead>
<tr>
<th>SCREEN TEXT EDITOR</th>
<th>61</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>61</td>
</tr>
<tr>
<td>Part 1: Text Editor Program</td>
<td>64</td>
</tr>
<tr>
<td>Screen Editor Control Character Commands</td>
<td>70</td>
</tr>
<tr>
<td>Summary of Part I</td>
<td>80</td>
</tr>
<tr>
<td>Part 2: Complete Text Editor Program</td>
<td>81</td>
</tr>
<tr>
<td>Command Display and Processor</td>
<td>82</td>
</tr>
<tr>
<td>Enhancements</td>
<td>96</td>
</tr>
<tr>
<td>Merging Programs by Using EXEC</td>
<td>96</td>
</tr>
<tr>
<td>User Instructions</td>
<td>97</td>
</tr>
<tr>
<td>Complete Screen Text Editor Program</td>
<td>101</td>
</tr>
</tbody>
</table>

Chapter 4

<table>
<thead>
<tr>
<th>ANSWERING USER HELP REQUESTS</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>111</td>
</tr>
<tr>
<td>Help Program</td>
<td>113</td>
</tr>
<tr>
<td>Pause Subroutine</td>
<td>118</td>
</tr>
<tr>
<td>Turning INVERSE On and Off</td>
<td>119</td>
</tr>
<tr>
<td>Help Test Routine</td>
<td>120</td>
</tr>
<tr>
<td>User Instructions</td>
<td>121</td>
</tr>
<tr>
<td>Complete Help Program</td>
<td>122</td>
</tr>
</tbody>
</table>

Chapter 5

<table>
<thead>
<tr>
<th>A DATA ENTRY SCREEN PROCESSOR</th>
<th>125</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>125</td>
</tr>
<tr>
<td>Creating a Data Entry Screen</td>
<td>128</td>
</tr>
<tr>
<td>Sample Variable-Exchange Routine</td>
<td>130</td>
</tr>
<tr>
<td>Data Entry Program</td>
<td>131</td>
</tr>
<tr>
<td>Changes to the Help Subroutine</td>
<td>135</td>
</tr>
<tr>
<td>Subroutine for Setting the Field Parameters</td>
<td>136</td>
</tr>
<tr>
<td>Displaying the Original Values</td>
<td>138</td>
</tr>
<tr>
<td>Editing Subroutine</td>
<td>139</td>
</tr>
<tr>
<td>Additional Option</td>
<td>142</td>
</tr>
<tr>
<td>User Instructions</td>
<td>143</td>
</tr>
<tr>
<td>Complete Data Entry Screen Program</td>
<td>145</td>
</tr>
</tbody>
</table>

Chapter 6

<table>
<thead>
<tr>
<th>A MENU SYSTEM</th>
<th>151</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>151</td>
</tr>
<tr>
<td>Menu Program</td>
<td>154</td>
</tr>
</tbody>
</table>
## CONTENTS

### CHAPTER 7

**REPORT GENERATION**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>165</td>
</tr>
<tr>
<td>Philosophical Considerations</td>
<td>166</td>
</tr>
<tr>
<td>Simple Report Generator</td>
<td>168</td>
</tr>
<tr>
<td>Program Features</td>
<td>170</td>
</tr>
<tr>
<td>Complete Report Generator Program</td>
<td>174</td>
</tr>
</tbody>
</table>

### CHAPTER 8

**PERSONAL CALENDAR: A SAMPLE PROGRAM**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>179</td>
</tr>
<tr>
<td>Basic Calendar Program</td>
<td>183</td>
</tr>
<tr>
<td>User Instructions</td>
<td>198</td>
</tr>
<tr>
<td>Complete Personal Calendar Program</td>
<td>200</td>
</tr>
</tbody>
</table>

**INDEX**

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>223</td>
</tr>
</tbody>
</table>
This book is divided into the following subject areas:

- The input of data, Chapter 2,
- Data storage and manipulation, Chapters 3–6,
- Outputting the data, Chapter 7,
- Putting it all together, a summary exercise, Chapter 8.

Subroutines are provided that show you how to input data into the computer, how to store and work with this data, and then how to output the data to the screen or printer.

The final chapter provides a stand-alone program that summarizes all of these techniques into a personal calendar program.
In this preliminary chapter we will discuss subroutines, programming style and technique, and efficient methods for writing and testing your own programs. We will also present the common structure used in the succeeding chapters in order to prepare you for getting the most out of this book.

**SUBROUTINES: WHAT THEY ARE AND HOW TO USE THEM**

Simply stated, a subroutine is a program that is used over and over again in one program or in many programs. A subroutine can be as small as two lines or as large as several thousand lines. In general, however, subroutines are kept small so that they will be understandable and manageable. Ideally, a subroutine will only perform one function, such as allowing alphanumeric input from the keyboard. By performing only one function, it will always behave as expected, and you will not be surprised by an unusual response. If a subroutine is to perform several functions, it can be made up of several single-function subroutines that are nested together.

A subroutine is distinguished from a "regular" program by two BASIC statements: GOSUB and RETURN. A GOSUB is used by the "calling" program (the main or originating program) to access the subroutine, and a RETURN is used by the subroutine, upon completing its task, to return to the calling program. Except for these two statements, a subroutine is a regular BASIC program.

**GOSUB–RETURN** When the program encounters GOSUB, it unconditionally branches to the referenced line number. Upon encountering a RETURN, the program branches back to the statement immediately following the most recently executed GOSUB.

**EXAMPLE**

```
5000  GOSUB 6000
5100  PRINT X*$3
5200  END
6000  INPUT
6100  RETURN
RUN
?100
300
```
In this GOSUB example, when line 5000 is encountered, the program execution sequence jumps to line 6000 and asks for the input of X\%. The number 100 is input from the keyboard (see the line following RUN). Then execution resumes at line 5100, and the result of X\%*3 (which is 300) is printed on the screen.

The reason for the existence of a subroutine is fairly straightforward: to make the computer do as much of your work as possible. If you have a function in your program that is required several times, you have the option of retyping the function in several places or typing it in once, adding RETURN as the last line and using a GOSUB when you want to use it. Why should you do all the work when the computer will gladly do it at the mere typing of the command GOSUB? Subroutines serve one additional purpose: They make the programs consistent. If the same subroutine is used throughout your programs, then this function will be performed exactly the same way each time. You will not have to remember the exact details of how it works everytime you wish to use it in the program.

To visualize a subroutine, we can think in terms of any function or action that is done repeatedly. By using a subroutine to perform this action, we are always assured that it will be performed exactly the same way each time we need it. Let’s use a common example to illustrate this point. How do we start our car? A modern automobile simply requires us to get in and turn the key. The manufacturers have created a “subroutine” (actually, a group of subroutines nested together) to perform the required tasks for us when we turn the key. The action of turning the key starts the subroutine chain that does the following tasks:

- Determines if the choke is needed,
- Turns on the fuel pump,
- Runs the electrical system checkout,
- Engages the starter.

And we are done. We get the same results everytime—unless the system has broken down on us!

The following chapters will present a series of subroutines that you may use in your own programs. These subroutines present only one of many possible ways to solve the problem and may be modified for your own requirements.
PROGRAMMING STYLE

We like to consider computer programming as an art form. As with all art forms, the creator has a "style." This style can determine whether the creation is a work of beauty or something else. Some of us were not born with enormous amounts of style; we have to study others and copy where we can. Style also evolves with time; most of us get better as we gain experience. We rarely, however, go back to an older creation and improve or update its style. It is therefore important to do as good a job as possible the first time through.

The subroutines in this volume reflect our style. Some people will like it, and others will not; but that is art. In the following subsections we summarize some of the elements of our programming style. The elements of style that are most important are those that lead to an increase in understanding and readability of the program. We try to adhere to them as much as possible, but, being human, we do slip from time to time. We hope that by studying our style, you will be able to add those characteristics that you like to your own style.

Meaningful Variable Names

It will come as no surprise to you that not all variable names are meaningful. Even a variable name that is meaningful to you may be totally confusing to another person reading your program. Part of this problem stems from Applesoft BASIC, which only recognizes the first two characters in a variable name. Although BASIC may consider only the first two characters, there is nothing that restricts the use of longer names if you keep the first two characters unique. Thus it is our convention to use as long a name as necessary to clearly define the variable being addressed. Sometimes, the first two characters are a little unusual, but we can still understand what the variable is to be used for.

It is important to remember that for programs in BASIC, variables are used by the entire program. Any variable may be assigned a value at any point in a program, and that value can be used at any other place in the program. This procedure is how information is passed to and from subroutines. Before calling a subroutine, we assign values to the vari-
ables used by that subroutine. After the subroutine has completed its task, those same variables are still available for use by the rest of the program, even though some of the values may have changed in the subroutine.

**Line Numbers and Subroutines**

We use lots of subroutines in our programs. Since we must use line numbers and not labels (names) to address subroutines, how do we keep them all straight in our mind? We do not renumber the subroutines. Once a subroutine is created and assigned a starting line number, we keep that line number intact. It may look nice to have an entire program evenly numbered, but even numbering is not worthwhile if the subroutines keep moving around. Therefore, we only renumber the main program sections if we must; we do not renumber the subroutine sections.

Our choice of line numbers was not random. There was a plan.

First, we wanted you to be able to add these routines to existing programs, and since most people tend to use smaller-value line numbers, we elected to use larger-value line numbers. This way our routines will not conflict with yours.

Second, by carefully selecting line numbers, we can make a program run as fast as possible. When BASIC looks for a line number (as in GOSUB 10000 or GOTO 11000), it first checks to see if the current line number is larger or smaller than the one being searched for. If the current line number is smaller, it begins the search for the desired line beginning at the current line. If the current line number is larger, it begins the search with the first line number in memory.

For example, consider the following program:

```
100
.
.
.
1000 PRINT "HELLO"
1100 GOSUB 2000
1200 GOTO 1000
2000 PRINT "RANDY"
2100 RETURN
```
When BASIC executes line 1100

```
GOSUB 2000
```

it begins the search for line 2000 at line 1200. But when it executes line 1200

```
GOTO 1000
```

it must begin searching at the first line in memory, which is line 100.

If this program were a large one, a lot of time would be wasted going from line 100 to line 1000, simply because there are a lot of line numbers to check. However, getting to line 2000 will be faster because there are fewer line numbers to check. Therefore, because of this characteristic of BASIC, we have tried to place subroutines at a line larger than the line calling the GOSUB or GOTO. Obviously, this technique is not always possible, but it is an easy constraint to live with.

**Remark Statements, or What’s This?**

Most programmers fail to use enough remark (REM) statements in their programs.

**REM**  REM statements are nonexecuting line statements. They are used in programs to provide notes and reminders to the original program author and to others who may subsequently need to go back into the program and figure out its purpose or method.

```
EXAMPLE

1090 REM
1095 REM
1100 REM
1105 X = 11.005 REM THIS IS A FIXED VALUE
1110 Y = 5.026 REM THIS IS A FIXED VALUE
1115 REM
1120 REM
1125 REM
```
In this example lines 1090-1100 and 1115-1125 are used to isolate what is found between them. This technique makes the program easier to read and calls attention to lines 1105 and 1110. The remarks after lines 1105 and 1110 indicate what the values in these lines are, where they came from, or what they are used for. In your remarks, use any description desired to remind yourself just what these lines are doing.

When the original author of a REMless program is gone, who will support and modify the work? Usually, no one; the REMless program will be thrown out and rewritten from scratch by another programmer. Therefore it is good practice to use remark statements as much as possible to help both yourself and subsequent users of a program.

In the programs in this book we have used remark lines to separate major sections of the program and to clearly explain, in detail, how it works and what it does. We use remarks wherever possible in the body of a routine to help clarify the processes it is going through. Even groups of blank REM lines add to the clarity of a program by being used to separate the text.

Since program branches, such as GOTO and GOSUB, use line numbers, we use a remark with each one to clarify where the program is going. We also often branch to a REM line that contains the meaning of the routine.

Regardless of who you are or what your position is, the debugging of a program is tedious. But the more remarks you have in a program, the sooner you can fix it and get on to another program.

**Multiple Statements on a Line**

Most versions of the BASIC language allow you to put several program statements on the same line, usually separated by a colon. Applesoft also allows this procedure. In general, this technique is a poor one, and we do not use it or recommend it, but it does enhance the execution speed of completed programs by eliminating the need to process the extra line numbers. Also, certain commands such as IF-THEN statements frequently require multiple commands on the same line, and they are acceptable there. However, if the line is very long, it should probably be made into a
subroutine and a GOSUB used. As far as we are concerned, only REM statements should be tagged onto a line. So please feel free to add on REM statements as often as you like.

WRITING YOUR PROGRAMS

Once you begin to write your own programs, your own personal programming style will develop, which may be very different from ours as we have described it so far. However, all of your programs should incorporate two important features: They should be user friendly, and they should be tested.

User-Friendly Programs

A user-friendly program is also a programmer-friendly program. A program is considered to be user friendly if it is understandable, predictable, and easy to use. If the program meets these requirements, then the user will be friendly to the programmer. If the program does not meet these requirements, then the user will be very unfriendly to the programmer. Therefore the easier it is to use your program, the happier everyone will be.

Testing Your Programs

All programs should be thoroughly tested before they are given to users. Testing is time-consuming; and the larger a program is, the more variables and conditions there are to test. However, you must remember that a running program is the most essential element of a program that is user friendly, and the only way to verify that a program works is to test it. We readily admit that we have delivered programs that users subsequently found errors in. Unless you spend years testing, you may never find all the errors in your programs, but you must try to be as thorough as possible. If you test a program in steps, as it is being developed, many problems can be discovered and corrected before they become serious. It is also beneficial to have another person test your work as you progress. A second opinion can be very valuable.
MAKE THE APPLE WORK FOR YOU

The whole purpose in writing a program is to have the computer do some of your work. When you design the program, think about the problems that may arise and how they can be solved by the computer automatically as they are encountered. Once again, user-friendly software is programmer friendly. This idea gets us back to the building block concept.

The building blocks we are providing in this book are intended to make the Apple work for both the programmer and the user. The programmer benefits by being able to use ready-made pieces over and over again, and the user benefits by having a consistent and professional program to work with.

We suggest that you purchase a software development system or tool kit. There are several different products available. One such product that is an invaluable aid to development is the Apple DOS Tool Kit. It contains many programs, including the Programmer's Aid. This program can renumber and merge programs, remove remark statements, and produce a variable cross-reference table. This program will come in very handy for those planning to write their own software. The new Apple IIe comes with a merge program and a renumber program on diskette, but it lacks the other helpful programs.

Other hints for making the Apple work for you are given in the following subsections.

Review the Reference Manual

Before you start reading the next chapter, we recommend that you review two sections of the Applesoft Basic Programming Reference Manual. They contain some helpful and informative suggestions. Our comments on some of these points follow.

First, read Appendix D, "Space Savers." The following hints are given there:

• Hint 1. "Use multiple statements per line." This technique is not a good one. The readability of the program by you or anyone else is greatly hindered by multiple statements on a line, making the program much more difficult to debug.
- Hint 2. "Delete all REM statements." This procedure is a good idea after the program is completely debugged.
- Hint 3. "Use integer instead of real arrays whenever possible." A very good idea.
- Hint 4. "Use variables instead of constants." Another very good idea.
- Hint 6. "Reuse the same variables." This procedure can be dangerous. If you must do it, then pick a certain combination of letters to represent your "garbage" variables and use the same ones throughout your programming.
- Hint 9. Using $X = \text{FRE}(0)$ to houseclean old strings is a good thing to remember and to do.

Second, read Appendix E, "Speeding Up Your Program." The following hints are given there:

- Hint 1. "Use variables instead of constants."
- Hint 2. "Place the most frequently used variables at the top of your program."
- Hint 4. "Frequently referenced line numbers should be located as early in the program as possible."

These three hints can't be emphasized enough—these are the small things that make a big difference.

*Note:* The Apple IIe owners will not find these same hints in their manuals. In this case newer is not better.

**Apple II Family Differences**

The introduction of the Apple IIe with the 80-column card necessitated some changes so that this book is useful for all Apple II users. All of our examples and test programs are formatted for 40-column screens while at the same time working with the IIe 80-column card activated. Satisfying both requirements at once meant that we had to make compromises in our programs. The FLASH command is an excellent example. This command gives your menus and screens a very commanding presence,
but it does not work when the 80-column card is activated. If you are working in 40 columns, you can implement FLASH in the same way as you implement INVERSE (shown in a following chapter).

The 80-column card can also be turned on and off, and its presence can be checked for under program control. Checking for card presence is done by PEEKing memory location C300 and comparing the first ten bytes found with the first ten bytes of the 80-column card ROM.

Perhaps the strongest single factor in favor of the 80-column card is the availability of uppercase and lowercase letters. They provide the most aesthetic screens and are highly recommended if available.

**ABOUT THE STRUCTURE OF THE BOOK**

The real learning experience in this book lies in the programs that are supplied. Although individual cases may vary, probably the easiest way to read this book is to go through the text of each chapter lightly to get a feel for what is to be done. Then study the programs carefully and study how each was created (recall the learning-by-example statement from the Preface). Diagraming the flow of the program can be very helpful. Read the text again in more detail, and then type in the program, verifying your progress at each test point. Next, start your debugging process—correcting the mistakes made in the entry of the program. By the time the debugging is done, you will have a good feel for what we have presented. Please enjoy yourself, and remember to back up your diskettes as you go.

The following chapters are divided into sections such as design, user features, and programmer features. In using these section topics, we are trying to structure your thinking to help you create programs more efficiently. Here are the points we are trying to make:

- **Design**
  - Define the basic program function.

- **User features**
  - Define specific user functions and the special conditions to be met.

- **Programmer features**
  - Define specific features needed to meet the design criteria.
As each new BASIC command is encountered in the text, we will provide a brief explanation and example. These examples are given to refresh your memory only, and we suggest that you review your Applesoft II reference manual for more detail, as needed.

The format of the program listings in the chapters that follow cannot be precisely duplicated on the Apple II. We have taken artistic license in the placement of the remark statements (through the use of our word processor) to make the listings easier to read.

Many of the chapters build on one another. For instance, the program presented in Chapter 2, the line editor, gets combined with the additional material in Chapter 3 to yield the screen editor. The following list describes which chapters are added together to yield the new one:

| Chapter 2 | Stand-alone |
| Chapter 3 | Chapters 2 + 3 |
| Chapter 4 | Stand-alone |
| Chapter 5 | Chapters 2 + 4 + 5 |
| Chapter 6 | Chapters 2 + 4 + 6 |
| Chapter 7 | Stand-alone |
| Chapter 8 | Chapters 2 + 4 + 5 + 6 + 7 + 8 |

For Chapters 2 and 3 the programs are entered by using the Apple's built-in editing capability, which is limited and cumbersome. Once you have a finished product from Chapter 3, you can use the resulting screen editor to enter and debug the programs in the remaining chapters—a real time-saving tool.
INTRODUCTION

One feature is common to almost every program: The program asks a question and the user types in an answer. In a BASIC program you normally get the user’s answer by using an INPUT statement.

**INPUT** The INPUT statement requests an input from the user at the keyboard, and the program will not proceed until the input is made.

**EXAMPLE**

```
5000   INPUT AGE%
5100   PRINT AGE%*2
RUN
?34
68
```
In this example line 5000 causes the user to be prompted, by a question mark on the screen, to supply a number for the variable AGE% (34). Once the number is typed in at the keyboard, line 5100 is executed, and the result (68) is printed on the screen.

The INPUT statement, however, accepts all user input, and the Apple itself only allows rudimentary editing. The programmer tests every entry for a valid response (i.e., a number in the proper range, or a name with only alphabet characters), and the user reenters the information if an error is detected. This process is repeated for every INPUT statement.

Why not have the Apple do some of the programmer's work and at the same time give the user some additional editing capabilities? This task can be done with a line editor. A line editor is a subroutine that accepts data entered on the keyboard and processes any special editing characters entered. These characters perform such functions as inserting a space or deleting a character. The line editor is also used to control the exact characters the user is allowed to enter. For example, you could restrict the user to entering numbers only, with no other characters allowed, or you could ask for a simple yes or no response. Of course, many more functions are available with a line editor.

The line editor program is one of the largest routines in this book. It is presented first because it is used as a building block for most of the routines in the following chapters. We have attempted to present this routine in an understandable manner, but don't be too concerned if you must reread a couple of sections before understanding it.

The development of the line editor program in this chapter will progress through a discussion of the major components needed, the methods for displaying the cursor and processing input keystrokes, some editing routines, and finally a set of user instructions. The sections that immediately follow will review the thought processes we want you to go through each time you consider a programming problem.

In this chapter, as in all the following chapters, you will encounter program modules that are to be entered into your computer and debugged. These modules will ultimately result in complete working programs, one for each chapter of the book.
Design

Before writing a computer program, you must define what problem is to be solved and exactly what you want the program to do—a task we call design. The problem we wish to begin solving in this chapter is an unfriendly-user interface to the computer. The INPUT statement does not allow editing of existing text or any program control over what the user enters. INPUT does not display the default or existing value; therefore the user cannot edit an existing value. The solution to these problems is a line editor.

What general characteristics should the line editor have? It should display the original text, if any, and allow the user to edit it. Periods should be used to show the user the maximum number of characters that can be entered. For example, to enter a field (a piece of information the user is entering or editing) 10 characters long, we could have the following displays:

```
........
Hello.....
110.27....
```

Our line editor includes these features.

The editor should also allow the user to edit and correct mistakes in the text. We want the computer to do some error checking for us, so the line editor should be able to select a type of data entered. It should be able to force numeric entries, or a yes or no response, or to accept any text. Also, since one of the most useful features of a program is an on-line help capability, the line editor should be able to notify the program of a user's help request. All these features are incorporated in our line editor program.

User Features

The specific features of the line editor can be divided into those directed toward the user and those directed toward the programmer. We will address the user features first.
It would be nice if the user could correct typing mistakes in addition to being able to move the cursor left and right. So we implement these editing functions:

- Move cursor left one position.
- Move cursor right one position.
- Skip to the previous word.
- Skip to the next word.
- Skip to end of line.
- Insert a character and slide the text to the right.
- Delete the current character and move the remaining text to the left.
- Delete text from the current character to the end of the line.

Another user feature of the line editor is that it will show one period per allowed character on the screen; the programmer will select the maximum allowable number of characters per field when setting up the screen. This feature is handy, for example, when the user is working with items such as zip code fields, where only five spaces are required, or names, where one space is needed for a middle initial.

As mentioned, an important part of any program is the on-line help system. The line editor supports the help system by allowing the user to request help by striking control Q. (The help system will be presented in detail in Chapter 4.)

**Programmer Features**

The programmer's features relate to how the programmer interfaces to the line editor. This interface should be as simple as possible, because we do not want it to add more work in creation than it saves in entry. Thus for the basic editor we require four pieces of information:

1. Screen row number where the input is to take place,
2. Screen column number where the input is to take place,
3. A "mask" to define the type and the length of the field,
4. The original text to be edited, if any.

A *mask* allows the programmer to specifically define what is going to be entered and how many characters are going to be entered. That is,
the program tests the input and rejects any that does not meet the mask requirement. For instance, suppose we want a zip code to be entered. Then we will have the program accept only numerical input up to five spaces. Nonnumeric input is rejected, and any input over five spaces is rejected. In this case we are using a numeric-only, five-space mask.

The line editor accepts four types of input data: help request, number, yes/no, and any text. The yes/no field only accepts the letters Y or N, and the any-text field accepts any printable ASCII character. A number field may contain the following symbols:

- . 0 1 2 3 4 5 6 7 8 9

A Word of Advice

Before you start to enter the first lines of the program given in the next section, we have several suggestions.

First, it is important to type in the program with all of the remark statements intact. The REMs will help you in the debugging of the program, and they are also used as entry points for most of the GOSUBs.

Second, Apple has kindly included some elementary editing capabilities in the Apple II computer family. Use of this built-in editing ability can save many hours of time and frustration in the entry of the line editor program presented in this chapter and the screen editor program in Chapter 3. The editing commands are not exactly the same for all members of the Apple II family, so please check the following references for your specific machine:

- Apple II and Apple II+: page 116 of the *Applesoft Tutorial*, middle of the page—the ESC I, ESC J, ESC K, or ESC M.
- Apple IIe: pages 71–86 of the new *Applesoft Tutorial* for all of the above plus expanded ESC commands.

LINE EDITOR TEST ROUTINE

The program segment that follows is the first program module you will enter in your computer. This module is the first of many you will enter in this chapter and succeeding chapters. The programs (one for each chap-
ter] are broken down into modules for two reasons. First, it is easy to understand and enter small modules. Second, it is far easier to debug small pieces of code than to debug large pieces.

This first module is a test program for the line editor modules presented in succeeding sections of the chapter. It allows you to define a field at any location on the screen and then use the line editor to input and edit.

The line editor test routine program is as follows:

```
10    REM
20    REM   LINE EDITOR TEST ROUTINE
30    REM   ASKS FOR INITIAL CONDITIONS THEN IT
40    REM   USES THE LINE EDITOR.
50    REM
60    HOME
70    INPUT  "ENTER MASK ";MASK$
80    INPUT  "ENTER TEXT ";ENTRY$
90    INPUT  "ENTER ROW ";ROW%
100   INPUT  "ENTER COL ";COL%
110   HOME
120   GOSUB  50000  : REM   THE LINE EDITOR
130   V TAB  20
140   PRINT
150   IF HELP% = 1 THEN PRINT "HELP REQUESTED"
160   PRINT
170   PRINT ">";ENTRY$;"<"
180   V TAB  24
190   INPUT  "ENTER (CR) TO CONTINUE OR END TO EXIT ";A$
200   IF A$ < > "" THEN END
210   GOTO  60
220   REM
230   END
50000  REM   BASIC LINE EDITOR
52130  REM   DISPLAY TEXT$
60000  RETURN  : REM   UNIVERSAL RETURN FOR TESTING
```

At the end of the test routine, note the many strange line numbers. Each line number represents a subroutine that may be called but isn't
written yet. By including the line number, we prevent our getting an error message later. The RETURN on line 60000 sends the program back to the calling program. As each routine is entered, we simply overwrite and thus erase a line number as it is used.

TEST POINT

Test points, encountered here for the first time, will appear regularly from now on to enable you to test each program module before adding it to the next module. We will point out the most pertinent conditions to test for, simplifying your debugging to small modules.

Perform the following steps after the line editor test program has been entered:

1. Save the program on your disk.
2. Enter

   RUN (CR)

   where (CR) means to strike the return key.

The program should clear the screen and ask you to ENTER MASK. Enter the desired mask and hit return. Then you will be asked for the text to fill the mask and also for the row/column location for the display on the screen. After you enter (CR), the process should repeat itself. When you wish to stop going through this loop, strike control C and reset, or enter END (CR) on the ENTER (CR) TO CONTINUE line.

It is always a good idea to provide your users with a consistent and clean way to exit your programs. For example, in the line editor test program you are allowed to enter the word END as a response. Terrible problems can crop up if your users get in the habit of striking control C or reset to exit a program. For example, if users strike reset when working with disk files, there may be some information in the disk buffer area that
has not been saved on the disk. It will be saved only if the file is closed properly. Therefore, striking the reset key will effectively ruin the file because it has not been updated completely.

BASIC LINE EDITOR PROGRAM

A flowchart for the basic line editor process is shown in Fig. 2.1. This flowchart gives a general overview of the program to be developed. Basically, the flowchart says that the programmer defines the characteristics of the field to be entered and the current contents of the field. The line editor first displays the field with periods, and then it allows the user to enter and/or edit the field. Finally, the field will be redisplayed and the periods erased.

The basic line editor program is as follows:

```
50000   REM  BASIC LINE EDITOR
50005   REM
50010   REM
50015   REM  THIS IS A BASIC LINE EDITOR
50020   REM
50025   REM  THE PROGRAMMER CALLS IT USING THE FOLLOWING VARIABLES
50030   REM
50035   REM  ROW%  => SCREEN LINE NUMBER
50040   REM  COL%  => SCREEN COLUMN NUMBER
50045   REM  ENTRY$  => TEXT TO BE EDITED
50050   REM  MASK$  => DATA TYPE TO BE ALLOWED
50055   REM  WHERE:
50060   REM    A  =  ALPHANUMERIC
50065   REM    #  =  NUMBER FIELD ONLY
50070   REM    Y  =  YES/NO FIELD
50075   REM    Q  =  HELP REQUEST OK, USE IN ANY CHARACTER
50080   REM  THE LENGTH OF MASK$ IS THE MAXIMUM LENGTH OF THE
50085   REM  INPUT STRING
50090   REM
50095   REM
50100   PLACE%  =  1                   : REM  SET THE STARTING POSITION
```
FIG. 2.1 Overview of line editor program

The explanations for various parts of this program and its GOSUBs are given in the following subsections.
Explanation of Variables

The basic line editor program consists mostly of REM statements (remarks). It is always good practice, at the beginning of a subroutine, to define who wrote it, when it was last changed, what it does, and the purpose of the major variables used, as shown in the remarks in lines 50000-50160.

The variables ROW% and COL% (lines 50035 and 50040) are used to position the text on the screen. ROW% is a number between 1 and 24. In 40-character mode COL% is a number between 1 and 40; while in 80-character mode it is a number between 1 and 80. HELP% will be set equal to 1 if the user requests help; otherwise, it will be set equal to 0.

The % symbol in the names means that these variables are integer variables. An integer is a whole number between -32,767 and +32,767. Throughout this book we will use integers whenever possible. We do so for several reasons. First, most programmers do not use integer variables; therefore by using integer variables, we can avoid variable name conflicts within your programs. For example, A, A%, and A$ are all treated as individual and unique variables, even though they have similar names. Second, integer variables require less memory space than floating-point (decimal-point) variables, and we want the subroutines to be as compact as possible.

FILL$ (line 50135) is the character used to illustrate the field’s maximum length. For example, if FILL$ equals a “.” and the field is to be six characters long, then the line editor will display

. . . . . .

six dots, to show the field’s maximum length.

ENTRY$ (line 50045) contains the text to be edited. If there is no text to be edited, then ENTRY$ is cleared by the line

ENTRY$"

MASK$ (line 50050) is used to define the type and the length of the field to be edited. In the programs in this book we will use one symbol to represent each character in the field. Thus an A will be used to indicate
FIG. 2.2 Field parameter examples

<table>
<thead>
<tr>
<th>MASK$</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAAAAAAAAAAAAAAAAAAAAAAA</td>
<td>Accept any 20 characters.</td>
</tr>
<tr>
<td>AAAAAAAAAAAAAAAAAAAAAAAA</td>
<td>Accept any 8 characters.</td>
</tr>
<tr>
<td>AAAAAAAAAAAAAAAAAAAAAAAA</td>
<td>Accept any 20 characters.</td>
</tr>
<tr>
<td>#######</td>
<td>Accept a 6-digit number.</td>
</tr>
<tr>
<td>#######</td>
<td>Accept a 10-digit number.</td>
</tr>
<tr>
<td>Y</td>
<td>Allow only Y or N entry.</td>
</tr>
<tr>
<td>AAA###</td>
<td>Accept a total of 6 characters; the first 3 may be any character and the last 3 must be numeric. (For example, this entry could be an inventory part number.)</td>
</tr>
</tbody>
</table>

that any text may be entered, that is, any printable ASCII character. For example,

```
MASK$ = "AAAAAAAAAA"
```

means "accept any character up to a maximum length of ten characters." A # is used for numeric fields; a Y for yes/no fields. Figure 2.2 contains several examples of how these various parameters are used. And, as always, help is available to the user anytime entry is requested, no matter what the mask definition.

Notice that we have not restricted the variable names to only two characters, but we have tried to use meaningful names. And not just a name meaningful today as we design the program, but one still meaningful twelve months from now when we return to make further enhancements.
TEST POINT

At this point the program is not yet functional, but it is a good idea to test what is entered to make sure it at least returns you to the beginning of the program. Execute the program by entering

```
RUN (CR)
```

If the program does not return you to the test routine, then verify that you typed in the above routine correctly.

**FIG. 2.3** Display subroutine
Explanation of the Display Subroutine

The display subroutine, called by line 50125 GOSUB 52130, is flowcharted in Fig. 2.3. This routine displays the text in the current ENTRY$ at the requested screen position and the requested length of the field, using the FILL$ character. If there is no text to display, then it will simply show one dot for each character allowed.

FILL$ must be set before this subroutine is called. FILL$ is the character used to show the maximum field length. Figure 2.1 shows that the first time this subroutine is used, it displays a period; the last time it is used, a blank or space character is used as FILL$. The space will remove the periods and clean up the screen display.

The display program corresponding to the flowchart in Fig. 2.3 is as follows:

52135 REM FILL$ IS THE FILL CHARACTER
52140 REM TXTSIZE% IS THE LENGTH OF ENTRY$
52145 REM MAXSIZE% IS THE MAXIMUM ALLOWED LENGTH
52150 REM
52155 REM
52160 TXTSIZE% = LEN (ENTRY$) : REM HOW LONG IS THE CURRENT FIELD?
52165 MAXSIZE% = LEN (MASK$) : REM WHAT IS THE MAX LENGTH ALLOWED?
52170 REM
52175 REM IS ENTRY$ TOO LONG?
52180 REM
52185 IF TXTSIZE% > MAXSIZE% THEN
52190 ENTRY$ = LEFT$ (ENTRY$ , MAXSIZE% ) : TXTSIZE% = MAXSIZE%
52195 REM
52199 REM POSITION THE CURSOR
52200 REM
52205 VTAB ROW% : REM ROW POSITION
52210 POKE 36 , COL% : REM COLUMN NUMBER - HTAB
52215 REM
52220 REM PRINT THE TEXT
52225 REM
52230 PRINT ENTRY$ ; : REM NO LINE FEED
52235 REM
52240 REM PRINT THE FILL CHARACTER
52245 REM
The display subroutine starts by using the LEN (LENgth) command (lines 52160 and 52165) to determine the length of ENTRY$ and MASK$.

**LEN** The LEN command returns the number of characters contained in the referenced string as an integer value between 0 and 255.

**EXAMPLE**

```
5000   A$ = "HAPPY DAYS"
5100   PRINT A$;LEN(A$)
RUN
HAPPY DAYS 10
```

As the example shows, the number of characters and spaces in A$, HAPPY DAYS, is equal to ten.

Remember that the length of MASK$ is the maximum length of the field.

Next, a test is made in the display program (line 52185) to see if ENTRY$ is larger than MASK$. This test can only occur the first time the line editor is called, because the editor will not allow the user to enter too many characters. If ENTRY$ is too long, then the editor will truncate the extra characters.

After ENTRY$ fits into the allotted space, the cursor is positioned and ENTRY$ is printed (lines 52205, 52210, and 52230). Finally, a FOR-
NEXT loop (lines 52250 through 52265) is used to fill the remainder of the field with the FILL$ character.

**FOR–NEXT** The FOR–NEXT looping command executes the statements after the FOR statement until the NEXT statement is encountered. Then the counting variable is incremented, and the process is repeated until the counting variable reaches the maximum value desired. Once the maximum value is reached, program execution proceeds to the statement following the NEXT. A STEP statement can also be used in a FOR–NEXT loop to increment the loop in values other than 1 (the default value).

**EXAMPLE**

```
5000 FOR I = 1 TO 5
5100 PRINT I
5200 NEXT
```

RUN

1
2
3
4
5

In this FOR–NEXT loop we simply count up from 1 to 5 by ones.

**EXAMPLE**

```
5000 FOR I = 1 TO 20 STEP 5
5100 PRINT I
5200 NEXT
```

RUN

1
6
11
16

This FOR–NEXT loop counts up in intervals of five.
EXAMPLE

5000 FOR I = 10 TO 1 STEP -1
5100 PRINT I
5200 NEXT
RUN
10
9
8
7
6
5
4
3
2
1

This example shows how to count down by using a negative step.

At this point in the display subroutine, we have positioned ENTRY$ on the screen and shown the field's maximum length.

Variables such as X, XX, and Y (see line 52255) are "garbage" variables. That is, they have only immediate meaning and may be changed by any routine. They are used to count loops and to temporarily hold values. Remember that garbage variables are not used by a subroutine that you call. Always use a unique name if you want to make sure that the variable remains unchanged by subroutines.

TEST POINT

The display program is the first subroutine that really does something; it displays any preexisting text and a dot to indicate the number of allowable characters. How do you select a series of tests to perform? Experienced programmers test the extremes of a subroutine. Programs tend to be stable in the midranges of their operation but may fail at the extremes of their operations. The best way to illustrate what is meant by this statement is to look at what tests should be performed on this subroutine.

Take a moment now and think about what tests you might perform. Although this routine is simple, there are a number of extremes that need to be tested. First, identify the important variables used in this subroutine. They are as follows:
CHAPTER 2  AN INPUT LINE EDITOR FOR APPLESOFT BASIC

ENTRY$  Text field
MASK$  Field type and length
ROW%  Screen row number
COL%  Screen column number

Next, identify the extreme values for each of these variables:

<table>
<thead>
<tr>
<th>VARIABLE</th>
<th>TEST EXTREME</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENTRY$</td>
<td>Length = 0 (i.e., null field)</td>
</tr>
<tr>
<td></td>
<td>Length = length of MASK$</td>
</tr>
<tr>
<td></td>
<td>Length &gt; length of MASK$</td>
</tr>
<tr>
<td>MASK$</td>
<td>Length = 1</td>
</tr>
<tr>
<td></td>
<td>Length = screen width</td>
</tr>
<tr>
<td>ROW%</td>
<td>Line 1</td>
</tr>
<tr>
<td></td>
<td>Line 24</td>
</tr>
<tr>
<td>COL%</td>
<td>First column</td>
</tr>
<tr>
<td></td>
<td>Last column</td>
</tr>
</tbody>
</table>

Since this subroutine is to be controlled by the programmer, certain extremes that will cause an error condition have been intentionally ignored. These errors occur if ROW% or COL% point to locations off the screen, as when one or both are equal to 0, or when ROW% > 24 or COL% > screen width. Also, if MASK$ is a null string, an error will occur. If you find yourself being forgetful, then you can add simple tests before line 50100 in the basic line editor program to verify that ROW%, COL%, and MASK$ have valid values. The meaning of "extremes" should become apparent to you as you read through the above list.

Make sure that the routine works properly for all ENTRY$ text fields. Test it with no text, too much text, and the maximum acceptable amount of text. Additionally, be sure to test those combinations of extremes that appear to be extreme extreme cases—for example, ROW% = 24, MASK$ length = screen width, and ENTRY$ > MASK$ length.

Note that the editor described in this chapter is not a "wraparound" editor—it is a line editor. If the input string goes beyond the last column of the screen, some commands, such as DELETE TO END OF LINE, will give strange results. Keep the limits of the program in mind during your testing and experiment with them.
Explanation of the Edit Subroutine

The flowchart shown in Fig. 2.4 describes the process for editing the field. This subroutine is called by line 50130, GOSUB 50165, in the basic line editor program. The flowchart details the following sequence of events: Position and display a cursor over the first character in the field. Next, accept a single character from the keyboard, process it, and then get another one. When a RETURN is entered, or if a help request is made, then return to the calling program.

Here is the program that edits the field:

```
50165 REM EDIT THE ENTRY$ FIELD
50170 REM
50175 REM POSITION THE CURSOR
50180 REM
50185 VTAB ROW% : REM VERTICAL POSITION
50190 GOSUB 52000 : REM PRINT THE CHARACTER IN INVERSE
50195 REM
50200 REM ACCEPT A KEY FROM THE KEYBOARD
50205 REM
50210 KEY% = PEEK (49152) : REM TEST FOR INPUT
50215 IF KEY% < 128 THEN GOTO 50210 : REM LOOP UNTIL ENTRY
50220 REM
50225 REM IF HERE THEN A KEY PUSHED
50230 REM
50235 XX = PEEK (49168) : REM CLEAR KEYBOARD
50240 KEY% = KEY% - 128 : REM STRIP OFF FLAG BIT
50245 REM
50250 REM PROCESS THE KEY
50255 REM
50260 GOSUB 50295 : REM KEY% PROCESSOR
50265 IF HELP% > 0 THEN RETURN : REM HELP REQUESTED BY USER
50270 IF CTRL% > 0 THEN GOSUB 52070: RETURN : REM CONTROL KEY EXIT
50275 GOTO 50210 : REM GET THE NEXT KEY
50280 REM
50285 REM ************************************
50290 REM
```

VTAB (line 50185) is used to position the cursor on the correct screen row.
VTAB  The VTAB command places the cursor on the specified line without changing the horizontal position.

**EXAMPLE**

VTAB 10

This command moves the cursor to row 10.

PEEK (see lines 50210 and 50235) is used to accept characters from the keyboard. PEEK returns the numeric value of a memory location; it can return a value between 0 and 255. You may recall that the Apple uses eight-bit memory, named bit 0 through 7. Bit 0 is the low bit, and bit 7
## FIG. 2.5 ASCII character codes

| Hex. No. | Binary No. | 0    | 1    | 2    | 3    | 4    | 5    | 6    | 7    | 8    | 9    | 10   | 11   | 12   | 13   | 14   | 15   | 16   | 17   | 18   | 19   | 20   | 21   | 22   | 23   | 24   | 25   | 26   | 27   | 28   | 29   | 30   |
|----------|------------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| 0        | 0000       | 0    | 1    | 2    | 3    | 4    | 5    | 6    | 7    | 8    | 9    | 10   | 11   | 12   | 13   | 14   | 15   | 16   | 17   | 18   | 19   | 20   | 21   | 22   | 23   | 24   | 25   | 26   | 27   | 28   | 29   | 30   | 31   |
| 1        | 0001       | 1    | 2    | 3    | 4    | 5    | 6    | 7    | 8    | 9    | 10   | 11   | 12   | 13   | 14   | 15   | 16   | 17   | 18   | 19   | 20   | 21   | 22   | 23   | 24   | 25   | 26   | 27   | 28   | 29   | 30   | 31   | 32   |
| 2        | 0010       | 1    | 2    | 3    | 4    | 5    | 6    | 7    | 8    | 9    | 10   | 11   | 12   | 13   | 14   | 15   | 16   | 17   | 18   | 19   | 20   | 21   | 22   | 23   | 24   | 25   | 26   | 27   | 28   | 29   | 30   | 31   | 32   |
| 3        | 0011       | 3    | 4    | 5    | 6    | 7    | 8    | 9    | 10   | 11   | 12   | 13   | 14   | 15   | 16   | 17   | 18   | 19   | 20   | 21   | 22   | 23   | 24   | 25   | 26   | 27   | 28   | 29   | 30   | 31   | 32   | 33   |
| 4        | 0100       | 4    | 5    | 6    | 7    | 8    | 9    | 10   | 11   | 12   | 13   | 14   | 15   | 16   | 17   | 18   | 19   | 20   | 21   | 22   | 23   | 24   | 25   | 26   | 27   | 28   | 29   | 30   | 31   | 32   | 33   | 34   |
| 5        | 0101       | 5    | 6    | 7    | 8    | 9    | 10   | 11   | 12   | 13   | 14   | 15   | 16   | 17   | 18   | 19   | 20   | 21   | 22   | 23   | 24   | 25   | 26   | 27   | 28   | 29   | 30   | 31   | 32   | 33   | 34   | 35   |
| 6        | 0110       | 6    | 7    | 8    | 9    | 10   | 11   | 12   | 13   | 14   | 15   | 16   | 17   | 18   | 19   | 20   | 21   | 22   | 23   | 24   | 25   | 26   | 27   | 28   | 29   | 30   | 31   | 32   | 33   | 34   | 35   | 36   |
| 7        | 0111       | 7    | 8    | 9    | 10   | 11   | 12   | 13   | 14   | 15   | 16   | 17   | 18   | 19   | 20   | 21   | 22   | 23   | 24   | 25   | 26   | 27   | 28   | 29   | 30   | 31   | 32   | 33   | 34   | 35   | 36   | 37   |
| 8        | 1000       | 8    | 9    | 10   | 11   | 12   | 13   | 14   | 15   | 16   | 17   | 18   | 19   | 20   | 21   | 22   | 23   | 24   | 25   | 26   | 27   | 28   | 29   | 30   | 31   | 32   | 33   | 34   | 35   | 36   | 37   | 38   |
| 9        | 1001       | 9    | 10   | 11   | 12   | 13   | 14   | 15   | 16   | 17   | 18   | 19   | 20   | 21   | 22   | 23   | 24   | 25   | 26   | 27   | 28   | 29   | 30   | 31   | 32   | 33   | 34   | 35   | 36   | 37   | 38   | 39   |
| A        | 1010       | 10   | 11   | 12   | 13   | 14   | 15   | 16   | 17   | 18   | 19   | 20   | 21   | 22   | 23   | 24   | 25   | 26   | 27   | 28   | 29   | 30   | 31   | 32   | 33   | 34   | 35   | 36   | 37   | 38   | 39   | 40   |
| B        | 1011       | 11   | 12   | 13   | 14   | 15   | 16   | 17   | 18   | 19   | 20   | 21   | 22   | 23   | 24   | 25   | 26   | 27   | 28   | 29   | 30   | 31   | 32   | 33   | 34   | 35   | 36   | 37   | 38   | 39   | 40   | 41   |
| C        | 1100       | 12   | 13   | 14   | 15   | 16   | 17   | 18   | 19   | 20   | 21   | 22   | 23   | 24   | 25   | 26   | 27   | 28   | 29   | 30   | 31   | 32   | 33   | 34   | 35   | 36   | 37   | 38   | 39   | 40   | 41   | 42   |
| D        | 1101       | 13   | 14   | 15   | 16   | 17   | 18   | 19   | 20   | 21   | 22   | 23   | 24   | 25   | 26   | 27   | 28   | 29   | 30   | 31   | 32   | 33   | 34   | 35   | 36   | 37   | 38   | 39   | 40   | 41   | 42   | 43   |
| E        | 1110       | 14   | 15   | 16   | 17   | 18   | 19   | 20   | 21   | 22   | 23   | 24   | 25   | 26   | 27   | 28   | 29   | 30   | 31   | 32   | 33   | 34   | 35   | 36   | 37   | 38   | 39   | 40   | 41   | 42   | 43   | 44   |
| F        | 1111       | 15   | 16   | 17   | 18   | 19   | 20   | 21   | 22   | 23   | 24   | 25   | 26   | 27   | 28   | 29   | 30   | 31   | 32   | 33   | 34   | 35   | 36   | 37   | 38   | 39   | 40   | 41   | 42   | 43   | 44   | 45   |
is the high bit. In the Apple, text information is stored in ASCII format (see Fig. 2.5). There are only 128 ASCII characters; therefore bit 7 is not used when describing an ASCII character.

The Apple keyboard is located at memory address location 49152. If a PEEK is made of this location, then the value of the last key entered will be returned. The Apple informs you that a key is entered by setting bit 7 equal to 1 and bits 0 through 6 equal to the ASCII value of the key entered. Whenever bit 7 is equal to 0, no key input has been received from the keyboard. In other words, if the PEEK returns a value greater than 128, then a key has been entered; if the value is less than 128, no key has been entered. Once a key is entered and accepted, Apple considers it the program’s responsibility to inform the Apple that the key has been received and that another key may be entered. This information is given by PEEKing address 49168; this procedure will set bit 7 back to 0 until another key is entered.

The following sample routine is provided to let you experiment with and to help you fully understand the keyboard PEEK routine just discussed. Type it in and experiment by running this routine by itself.

```
10  KEY% = PEEK (49152)
20  PRINT KEY%
30  IF KEY% < 128 THEN GOTO 10
40  XX = PEEK (49168)
50  KEY% = KEY% - 128
60  PRINT KEY%, CHR$(KEY%)
70  INPUT "ENTER CR";A$
80  GOTO 10
```

You might experiment and delete line 40 to see what effect this action will have.

---

**DISPLAYING A CURSOR**

In order for the user to know what character is being edited, the current cursor position must be displayed. In the field-editing program (line 50190, GOSUB 52000) the current character is displayed in inverse
video. When there is no character to display, then a space is used. A space is a bright box when displayed in inverse.

The following program will print a character in inverse:

```
52000 REM PRINT CHARACTER IN INVERSE
52005 REM THIS GIVES THE ILLUSION OF CURSOR MOVEMENT
52010 REM
52012 VTAB ROW% : REM POSITION CURSOR
52015 POKE 36, (COL% + PLACE% - 1) : REM HTAB
52020 INVERSE : REM REVERSE VIDEO
52025 XX$ = MID$ (ENTRY$,PLACE%,1) : REM MOVE FOR THE NEXT IF
52030 IF XX$ = "" THEN XX$ = " " : REM IF NULL MAKE IT A SPACE
52035 PRINT XX$; : REM PRINT THE INVERSE
52040 NORMAL : REM RESTORE TO NORMAL VIDEO
52045 POKE 36, (COL% + PLACE% - 1) : REM REPOSITION THE CURSOR - HTAB
52050 RETURN
52055 REM
52060 REM ******************************
52065 REM
```

PLACE% is used to keep track of the current cursor position. However, PLACE% could have been set by the calling program. For example, if we wanted to begin editing at the end of a field instead of the beginning, we would set PLACE% equal to the length of the field instead of 1.

After the cursor is positioned over the current character, using a POKE (line 52015), the INVERSE command is given (line 52020) and the character is printed. A NORMAL command is given (line 52040), and then the cursor is repositioned over the character (line 52045).

PRINT  The PRINT command causes a line feed (increments one line) when encountered without option parameters. With options, the values of the list following the PRINT command are evaluated and printed.

INVERSE  The INVERSE command causes all subsequent characters to be printed as black letters on a white background instead of the normal white on black.

NORMAL  The NORMAL command negates the preceding command and restores the video to the regular white-on-black mode.
TEST POINT

The print-in-inverse subroutine displays the cursor. After the line editor displays the text field to be edited, it positions the cursor over the first character. If there is a character in the first position, it will be shown in inverse; otherwise, a bright block will be shown.

PROCESSING A KEY

Subroutine GOSUB 50295 (line 50260 of the field-editing program) is used to display or process the key pressed by the user. A key can be either a control, a special character, or regular text.

Control characters are used for editing the text and moving the cursor. A control character is produced by holding down the CTRL (control) key while simultaneously pushing any letter key, A through Z. From Fig. 2.5 we see that the ASCII value of control A is 1, while the ASCII value of letter A is 65. Similarly, the ASCII value of control Z is 26, while the letter Z is represented by the value 90. The ESC (escape) key is a special key that has an ASCII value of 27.

The program that tests for a control key is as follows:

```
50295  REM TEST FOR CONTROL KEY
50300  REM
50305  IF KEY% < = 31 THEN GOSUB 51000: RETURN: REM PROCESS AND RETURN
50310  REM
50315  REM MUST BE AN ALPHANUMERIC
50320  REM
50325  REM TEST THE MASK TO DETERMINE DATA TYPE
50330  REM
50335  IF MID$(MASK$,PLACE%,1) = "A" THEN GOSUB 50900: RETURN
50340  IF MID$(MASK$,PLACE%,1) = "#" THEN GOSUB 50390: RETURN
50345  IF MID$(MASK$,PLACE%,1) = "Y" THEN GOSUB 50440: RETURN
50355  REM
50360  REM BAD MASK CHARACTER
50365  REM
50370  RETURN
50375  REM
50380  REM ************************************
50385  REM
```
If the user presses a control or special key, a branch is made to the control character subroutine, GOSUB 51000 (line 50305). If any other key is pressed, the program checks this character to see if it is acceptable. MID$ (lines 50335 through 50345) is used to determine the data type defined in MASK$. Then a branch is made to the appropriate character-checking subroutine.

**MID$(A$,N,X)** The MID$ command returns the character string starting at N in the string A$ for X characters. If X is not present, then the program continues until the end of the string is reached.

**EXAMPLE**

```
5000    A$ = "HAPPY DAYS"
5100    PRINT MID$(A$,7,3)
RUN
DAY
```

In this example of MID$ we count seven characters into A$ and extract the next three characters encountered. The result is that DAY is extracted from A$.

As an example, suppose that in the line editor program we are viewing the first character position, and MASK$ is defined as

```
MASK$ = "AAAAAAAAAA"
```

Then the branch is made to subroutine 50900, the universal-character-accept routine. In contrast, if MASK$ is

```
MASK$ = "#####"
```

then the branch is made to subroutine 50390, where a check verifies that a number has been entered. When a number is entered, it will be added to ENTRY$ and displayed; otherwise, it will be ignored. These tests occur in lines 50335–50345, but the subroutines called in these lines have not been presented yet.

In our version of the editor only five different data types are allowed. You can add other branches to this routine if more data types are desired. For example, Apple IIe users may wish to have a mask that converts all lowercase letters to uppercase.

Additional features of processing a key are illustrated in the following subsections.
Processing the Input Keys

After a key is pressed, the program must determine whether it is a control key (character) or an alphanumeric key. If it is a control character, then the routine must decide whether it is a valid character, and, if so, perform the required action. If it is not a control character, the mask

FIG. 2.6 Flowchart for accepting a character
character must be checked and the character-type test performed. If a valid character has been entered, then it is added to ENTRY$ and displayed on the screen. This process is flowcharted in Fig. 2.6.

**Character-Accept and Display Routines**

When a valid character is entered, it is added to ENTRY$ and displayed on the screen. Then the cursor is moved right one position.

There are several tests that must be performed in the character-accept routine. First, a check is made to see whether the insert mode is on (INSERT% = 1); if it is, a space is inserted before the character is added. Second, a check is made to determine how to add the character; there are different ways to add the character depending on where the character is to go. Finally, a check is made to see whether the new field is too long; if it is, it must be truncated to MAXSIZE%.

The test program to check for an alphanumeric field is as follows:

```
50900  REM PRINT KEY% AND ADD TO ENTRY$
50905  REM
50910  IF INSERT% = 1 THEN GOSUB 51315 : REM INSERT A SPACE
50915  TXTSIZE% = LEN (ENTRY$) : REM MAKE SURE WE HAVE CORRECT TXTSIZE%
50922  REM ADD TO END OF ENTRY
50925  IF PLACE% > TXTSIZE% THEN ENTRY$ = ENTRY$ + CHR$ (KEY%): GOTO 50945
50926  REM ADD AS FIRST CHARACTER
50930  IF PLACE% = 1 THEN ENTRY$ = CHR$(KEY%) + MID$(ENTRY$,PLACE% + 1)
50932  REM ADD AS LAST CHARACTER
50935  IF PLACE% = TXTSIZE% THEN
      ENTRY$ = LEFT$(ENTRY$, PLACE% - 1) + CHR$(KEY%): GOTO 50945
50936  REM ADD IN THE MIDDLE SOMEWHERE
50940  ENTRY$ = LEFT$(ENTRY$, PLACE% - 1) + CHR$(KEY%) + MID$(ENTRY$, PLACE% + 1)
50945  TXTSIZE% = LEN (ENTRY$)
50946  REM IF TOO BIG TRUNCATE IT
50950  IF TXTSIZE% > MAXSIZE% THEN ENTRY$ = LEFT$ (ENTRY$, MAXSIZE%)
50955  REM
50960  REM NEED TO MOVE RIGHT ONE PLACE
50965  REM
50970  GOSUB 51425 : REM RIGHT ARROW
50975  RETURN
50980  REM
50985  REM ********************************************
50990  REM
```
The programs for the other two data tests—yes/no field and number field—are listed next.

50390 REM ACCEPT A NUMBER
50395 REM
50400 REM TEST TO SEE IF IT IS A VALID NUMERIC TYPE OF CHARACTER
50405 REM
50410 IF (KEY% < 45) OR (KEY% > 57) THEN RETURN : REM BAD KEY
50415 IF KEY% = 47 THEN RETURN : REM BAD KEY ALSO
50420 GOSUB 50900 : REM GOOD KEY SO ACCEPT IT
50425 RETURN
50430 REM ************************************************************************
50435 REM TEST FOR YES OR NO
50440 IF (KEY% <> 89) AND (KEY% <> 78) THEN RETURN : REM BAD KEY
50445 GOSUB 50900 : REM ACCEPT IT
50450 RETURN
50460 REM ************************************************************************

For each data type, if KEY% is in the correct range, then a valid key has been entered. If a bad key is entered, then the routine RETURNs and the key is effectively ignored. If a good key is entered, then the routine is called that places the character into the field.

**PROCESSING CONTROL KEYS: EDITING Routines**

The line editor uses control characters to move the cursor and edit the text. These characters are summarized in Fig. 2.7. The letter selections are arbitrary. If you do not like them or if another program on your computer uses different characters, you may redefine them to be consistent.

The right and left arrows, control U and H, respectively, are used to move the cursor right and left. In our editor the right arrow key will not erase the characters as it passes over them. Control A is used to skip to the previous word. Control D is used to delete a character and compress the field. A character may be inserted into the middle of text by using a control F to turn the insert mode on. The insert mode is kept on until any control character is entered, including a control F. A control N is used to jump to the end of the line. Control W is used to skip to the next word, and control Y will delete everything to the right of the cursor.
FIG. 2.7 Control characters

Control A  
Skip to the previous word.
Control D  
Delete this character and compress the line.
Control F  
Insert a character into the line.
Control N  
Skip to the end of the line.
Control Q  
HELP request at any time.
Control W  
Skip to the next word.
Control Y  
Erase to the end of the line.
Right arrow  
Move right one character.
Left arrow  
Move left one character.

Subroutine 51000 in the program that follows uses conditional IF statements to branch to the editing routines. To add new editing features, you would simply add more conditionals and insert the code in the space before line 52000.

The following subroutine processes the control keys. The individual editing routines for the various control characters are described in the succeeding subsections.

```
51000  REM  PROCESS A CONTROL KEY
51005  REM
51010  REM  EXIT KEYS SUCH AS RETURN SET CTRL%
51015  REM
51020  REM
51025  REM  ^A = 1  > PREVIOUS WORD
51030  REM  ^D = 4  > DELETE THIS CHARACTER
51035  REM
51040  REM  ^F = 6  > FILL WITH A SPACE
51045  REM  ^H = 8  > LEFT ARROW
51050  REM  ^N = 14 > SKIP TO END
51055  REM  ^Q = 17 > HELP REQUEST
51060  REM  ^U = 21 > RIGHT ARROW
51065  REM  ^W = 23 > NEXT WORD
```
Moving the Cursor Right and Left

The right and left arrow subroutines, which are presented below, are very simple. First, the program redisplay the current cursor position in normal video. Next, it increments or decrements PLACE%, and then it displays the new cursor position in inverse video. Notice in the program that follows that both routines test the size of PLACE% and that it is changed only if it meets the boundary condition.
BASIC BUSINESS SUBROUTINES FOR THE APPLE II AND IIE

51380   REM  LEFT ARROW
51385   REM
51390   GOSUB 52070   : REM  DISPLAY NORMAL
51395   IF PLACE% > 1 THEN PLACE% = PLACE% - 1: REM  MOVE LEFT ONE
51400   GOSUB 52000   : REM  DISPLAY INVERSE
51405   RETURN
51410   REM
51415   REM  *********************************
51420   REM
51425   REM  RIGHT ARROW
51430   REM
51435   IF MID$ (ENTRY$,PLACE%,1) = "" THEN RETURN
51440   GOSUB 52070   : REM  DISPLAY AS NORMAL
51445   IF PLACE% < MAXSIZE% THEN PLACE% = PLACE% + 1
51450   GOSUB 52000   : REM  DISPLAY AS INVERSE
51455   RETURN
51460   REM
51465   REM  *********************************
51470   REM

TEST POINT

Enter some text, then enter the left arrow. The cursor should move left one position each time you strike the key. Continue striking the left arrow key until you have returned to the beginning of the field. Now strike the left arrow a couple of times just to make sure it works properly in the first character position. Test the right arrow by striking it until you have gone to the end of the field.

Jump to the Next Word

Skipping to the next word on the line moves the cursor to the first character of the next word, to the right of the current cursor position. To do this operation, the program looks at each character to the right of the cursor and stops at the first one after the next space or group of spaces. If the cursor is positioned over a word, then it must move right to the first space and then over the spaces to the first nonspace.

The following routine allows the user to skip to the next word:
CHAPTER 2  AN INPUT LINE EDITOR FOR APPLESOFT BASIC

51475  REM  SKIP TO NEXT WORD
51480  REM
51485  REM
51490  IF PLACE% = > TXTSIZE% THEN RETURN : REM  ALREADY AT END
51495  GOSUB 52070 : REM  REMOVE CURSOR
51500  PLACE% = PLACE% + 1 : REM  LOOK FOR FIRST SPACE
51505  IF PLACE% = TXTSIZE% THEN GOTO 51530
51510  IF MID$ (ENTRY$,PLACE%,1) <> " " THEN GOTO 51500: REM  IS IT A SPACE?
51515  PLACE% = PLACE% + 1 : REM  MOVE RIGHT ONE
51520  IF PLACE% = TXTSIZE% THEN GOTO 51530
51525  IF MID$ (ENTRY$,PLACE%,1) = " " THEN GOTO 51515: REM  SKIP OVER SPACES
51530  GOSUB 52000 : REM  DISPLAY CURSOR
51535  RETURN
51540  REM
51545  REM  ***************************************
51550  REM

TEST POINT

Enter the sentence:

"THIS IS A TEST."

Now use the left arrow to move to the front of the line. Enter a ^W and the
cursor should jump to the front of the next word. Try this exercise with
several sentences and vary the number of spaces between words.

Jump to the Previous Word

To skip to the previous word, we first want to force the cursor to move
over any spaces, in case we are at the front of a word, and then stop at
the character to the right of the next space. The following program
performs the skip to the previous word:

51555  REM  SKIP TO PREVIOUS WORD
51560  REM
51565  IF PLACE% = 1 THEN RETURN : REM  AT THE FRONT ALREADY
51570  GOSUB 52070 : REM  REMOVE CURSOR
BASIC BUSINESS SUBROUTINES FOR THE APPLE II AND IIe

51575 place% = place% - 1 : rem look for space
51580 if place% = 1 then goto 51610: rem force move at least one space
51585 if mid$(entry$,place%,1) = " " then goto 51575: rem skip over spaces
51590 place% = place% - 1
51595 if place% = 1 then goto 51610
51600 if mid$(entry$,place%,1) <> " " then goto 51590: rem is it a space?
51605 place% = place% + 1 : rem position over first letter
51610 gosub 52000 : rem display the cursor
51615 return
51620 rem
51625 rem ***************************

TEST POINT

Enter the sentence:

"PREVIOUS WORD TEST."

Then enter a ^A and the cursor should move left one word. Repeat the tests you just did for jumping to the next word.

Jump to End of Line

We can skip to the end of the line by testing the length of the field and moving the cursor there. The following program gives the routine:

51680 rem skip to end of line
51685 rem
51690 gosub 52070 : rem move the cursor
51695 place% = len (entry$) + 1
51700 if place% > maxsize% then place% = maxsize%: rem do not go past end
51705 gosub 52000 : rem show the cursor
51710 return
51715 rem
51720 rem ***************************

TEST POINT

After you enter some text, move the cursor left and enter ^N. The cursor should jump to the end of the line.
Deleting a Character

The delete subroutine uses MID$ and LEFT$ commands to compress the field, and then it displays the new text with the cursor. In the delete program that follows you will note that there are several conditional tests. These tests check for special circumstances at the boundaries of the subroutine. Many routines require special handling at the minimum or maximum points of the routine. In this case the special handling is needed when the cursor is at the very beginning or end of the field.

As you develop programs, you should carefully design and test the boundaries or extremes of the routine, because most errors will occur at the boundaries and not in the middle range of a routine. For example, filling a disk or initializing a file are common boundary conditions that can cause problems. While you may not always be able to think of every possible error condition, you can generally think of the limits of a routine so that they can be tested.

The delete subroutine is as follows:

```
51210 REM DELETE AND PACK
51215 REM
51220 TXTSIZE% = LEN (ENTRY$)
51225 IF TXTSIZE% = 0 THEN RETURN : REM NOTHING TO DELETE
51230 IF TXTSIZE% = 1 THEN ENTRY$ = ":PLACE% = 1:GOTO 51250: REM DELETE LINE
51235 IF PLACE% = 1 THEN ENTRY$ = MID$ (ENTRY$,2): GOTO 51250
51240 IF PLACE% >= TXTSIZE% THEN
      ENTRY$ = LEFT$(ENTRY$,TXTSIZE% - 1):PLACE% = PLACE% - 1:GOTO 51250
51245 ENTRY$ = LEFT$(ENTRY$,(PLACE% - 1)) + MID$(ENTRY$,(PLACE% +1)
51250 GOSUB 52130 : REM PRINT NEW STRING
51255 GOSUB 52000 : REM PRINT INVERSE
51260 RETURN
51265 REM
51270 REM *******************************
51275 REM
```

**TEST POINT**

At this point you should be able to enter text and move the cursor back and forth. To test the delete routine, enter some text and then delete a character at the following positions:
1. Cursor on the first character,
2. Cursor in middle of text,
3. Cursor on last character,
4. Cursor past end of text.

**Inserting Characters**

Characters are inserted by an insert mode toggle. In other words, we turn, or toggle, the insert mode on or off. The control F key is used to turn the mode on, and this or any other control key can be used to turn the mode off.

Once the insert mode is turned on, any regular characters typed will be inserted into the field, with any characters to the right of the cursor being shifted to the right one place for each character inserted. Any characters at the very end of the field will spill off into that never-never land of lost characters.

The actual insertion is done by first inserting a space into the field and printing the line. Next, the desired character is written over this space. Finally, the cursor is moved right one place. The **LEFT$** command is essential in this routine.

**LEFT$** The **LEFT$(A$,N)** command returns the character string starting at the left end of **A$** for **N** characters.

**EXAMPLE**

```
5000 A$ = "HAPPY DAYS"
5100 PRINT LEFT$(A$,4)
RUN
HAPP
```

In this example of **LEFT$** we take the character in **A$** starting at the leftmost position and count to the right four positions and stop. In this case the results are **HAPP**.

Any characters pushed off the end of the field are lost. A variation you could implement is to allow insertion only until the field is full and then stop. If users did not want the characters at the end, they would have to move the cursor there and erase them.
The program for inserting characters is as follows:

```basic
51280 REM TOGGLE THE INSERT MODE
51285 REM
51290 IF INSERT% = 1 THEN INSERT% = 0: RETURN : REM TURN IT OFF
51295 INSERT% = 1 : REM TURN IT ON
51300 RETURN
51305 REM ****************************************
51310 REM
51315 REM INSERT A SPACE
51320 REM
51325 REM
51330 REM IS IT THE FIRST CHARACTER?
51335 IF PLACE% = 1 THEN ENTRY$ = " " + ENTRY$: GOTO 51350
51340 REM INSERT IN THE MIDDLE
51345 ENTRY$ = LEFT$(ENTRY$,PLACE% - 1) + " " + MID$(ENTRY$,PLACE%)
51350 GOSUB 52130 : REM PRINT THE FIELD
51355 GOSUB 52070 : REM REPOSITION CURSOR
51360 RETURN
51365 REM
51370 REM ****************************************
51375 REM
```

**TEST POINT**

Begin by entering some text on the line and moving the cursor into the middle of the text. Now enter `^F` followed by some other letters. As each character is entered, the text should split, with the right side sliding right one place for each new character entered. The cursor should step right one place. Press another control key—left arrow, for example—and make sure that the insert mode is turned off. Next, toggle insert mode back on and verify that a second `^F` turns it off. Finally, move the cursor to the front of the text and check that the insert mode works there.

**Displaying a Character**

After a valid key has been pressed, the following display subroutine will display the character on the screen. If a character has been removed, then the FILL$ character will be displayed.
BASIC BUSINESS SUBROUTINES FOR THE APPLE II AND Ile

TEST POINT

The display subroutine is tested in conjunction with the character-accept and display routine. RUN the program and press a key. This key's character should be displayed on the screen in a NORMAL video, and the cursor should move one position to the right.

Erase to End of Line

This erase feature is very handy. You will frequently decide to change everything to the right of the cursor, and the erase routine allows you to do so in one keystroke. Erasing to the end of the line only requires a LEFT$ command and then a redisplay of the field.
CHAPTER 2 AN INPUT LINE EDITOR FOR APPLESOFT BASIC

TEST POINT

Enter some text and move the cursor left. Now enter ^Y and everything to the right of the cursor should disappear. Test this routine at the beginning, the middle, and the end of the line.

The Escape Key

Striking the ESC (escape) key will cause the CTRL% flag to be set and the line editor to return to the calling program. This feature is used extensively later in the book.

USER INSTRUCTIONS

Whenever it is appropriate, we will include a sample set of instructions that can be included in your user's (operator's) manual. We naturally are assuming that every program that you write includes a user's manual. Portions of the user's manual can be extracted and used as help files.

The line editor program allows you to enter and edit text. You can move the cursor left and right, insert and delete characters anywhere on the line, and request help from the computer.

The line editor will check every character that you enter and verify that it is acceptable. For example, if the program is requesting a number, then it will allow you to enter only numbers, not letters. When it wants a yes or no response, it will only allow you to enter a Y or an N. Periods (.) are displayed to illustrate the maximum length of the field. You cannot enter text that will exceed the space shown.

Whenever you have any doubts about how you should respond to an input, you can request help by pressing control Q (for question). If help is available, then a message will be displayed for you. After you have read the message, enter RETURN and the program will continue.

The following list summarizes the editing commands available. Recall that a control character is entered by holding down the CTRL key while entering the desired letter. For example, control A is activated by holding down the CTRL key while striking the A key.
BASIC BUSINESS SUBROUTINES FOR THE APPLE II AND IIe

Control A  Skip to the previous word.
Control D  Delete this character and compress the line.
Control F  Insert characters into the line.
Control N  Skip to the end of the line.
Control Q  HELP request at any time.
Control W  Skip to the next word.
Control Y  Erase to the end of the line.
Right arrow Move right one character.
Left arrow Move left one character.

When you are in the insert mode, you will keep inserting letters until you enter any control or arrow key.

COMPLETE LINE EDITOR PROGRAM

The complete program listing for the line editor follows. It may look like a formidable program, but if the remark statements were removed, the program would be about 160 lines long. Considering what this routine does, that length is not overly long.

```
10 REM
20 REM LINE EDITOR TEST ROUTINE
30 REM ASKS FOR INITIAL CONDITIONS THEN IT
40 REM USES THE LINE EDITOR.
50 REM
60 HOME
70 INPUT "ENTER MASK "; MASK$
80 INPUT "ENTER TEXT "; ENTRY$
90 INPUT "ENTER ROW "; ROW$
100 INPUT "ENTER COL "; COL$
110 HOME
120 GOSUB 50000 : REM THE LINE EDITOR
130 VTAB 20
140 PRINT
150 IF HELP% = 1 THEN PRINT "HELP REQUESTED"
```
160  PRINT
170  PRINT ">";ENTRY$;"<"
180  VTAB 24
190  INPUT "> ENTER (CR) TO CONTINUE OR END TO EXIT ";A$
200  IF A$ <> ">" THEN END
210  GOTO 60
220  REM
230  END

50000  REM BASIC LINE EDITOR
50005  REM
50010  REM
50015  REM THIS IS A BASIC LINE EDITOR
50020  REM
50025  REM THE PROGRAMMER CALLS IT USING THE FOLLOWING VARIABLES
50030  REM
50035  REM ROW% => SCREEN LINE NUMBER
50040  REM COL% => SCREEN COLUMN NUMBER
50045  REM ENTRY$ => TEXT TO BE EDITED
50050  REM MASK$ => DATA TYPE TO BE ALLOWED
50055  REM WHERE:
50060  REM     A = ALPHANUMERIC
50065  REM     # = NUMBER FIELD ONLY
50070  REM     Y = YES/NO FIELD
50075  REM     Q = HELP REQUEST OK, USE IN ANY CHARACTER
50080  REM THE LENGTH OF MASK$ IS THE MAXIMUM LENGTH OF THE
50085  REM INPUT STRING
50090  REM
50095  REM
50100  PLACE% = 1 : REM SET THE STARTING POSITION
50105  REM
50110  FILL$ = "." : REM DISPLAY DOTS
50115  HELP% = 0 : REM CLEAR THE HELP FLAG
50120  CTRL% = 0 : REM CLEAR THE EXIT FLAG
50125  GOSUB 52130 : REM DISPLAY ENTRY$
50130  GOSUB 50165 : REM EDIT THE STRING
50135  FILL$ = "." : REM CLEAR THE SCREEN
50140  GOSUB 52130 : REM DISPLAY ENTRY$
50145  RETURN : REM GO BACK TO CALLER
50150  REM
50155  REM ********************
50160  REM
50165  REM  EDIT THE ENTRY$ FIELD
50170  REM
50175  REM  POSITION THE CURSOR
50180  REM
50185  VTAB ROW%            : REM  VERTICAL POSITION
50190  GOSUB 52000          : REM  PRINT THE CHARACTER IN INVERSE
50195  REM
50200  REM  ACCEPT A KEY FROM THE KEYBOARD
50205  REM
50210  KEY% = PEEK (49152)  : REM  TEST FOR INPUT
50215  IF KEY% < 128 THEN GOTO 50210 : REM  LOOP UNTIL ENTRY
50220  REM
50225  REM  IF HERE THEN A KEY PUSHED
50230  REM
50235  XX = PEEK (49168)    : REM  CLEAR KEYBOARD
50240  KEY% = KEY% - 128   : REM  STRIP OFF FLAG BIT
50245  REM
50250  REM  PROCESS THE KEY
50255  REM
50260  GOSUB 50295          : REM  KEY% PROCESSOR
50265  IF HELP% > 0 THEN RETURN : REM  HELP REQUESTED BY USER
50270  IF CTRL% > 0 THEN GOSUB 52070: RETURN : REM  CONTROL KEY EXIT
50275  GOTO 50210          : REM  GET THE NEXT KEY
50280  REM
50285  REM  *******************************
50290  REM
50295  REM  TEST FOR CONTROL KEY
50300  REM
50305  IF KEY% <= 31 THEN GOSUB 51000: RETURN : REM  PROCESS AND RETURN
50310  REM
50315  REM  MUST BE AN ALPHANUMERIC
50320  REM
50325  REM  TEST THE MASK TO DETERMINE DATA TYPE
50330  REM
50335  IF MID$ (MASK$,PLACE%,1) = "A" THEN GOSUB 50900: RETURN
50340  IF MID$ (MASK$,PLACE%,1) = "#" THEN GOSUB 50390: RETURN
50345  IF MID$ (MASK$,PLACE%,1) = "Y" THEN GOSUB 50440: RETURN
50350  REM
50360  REM  BAD MASK CHARACTER
50365  REM
CHAPTER 2 AN INPUT LINE EDITOR FOR APPLESOFt BASIC

50370 RETURN
50375 REM
50380 REM ******************************
50385 REM
50390 REM ACCEPT A NUMBER
50395 REM
50400 REM TEST TO SEE IF IT IS A VALID NUMERIC TYPE OF CHARACTER
50405 REM
50410 IF (KEY% < 45) OR (KEY% > 57) THEN RETURN : REM BAD KEY
50415 IF KEY% = 47 THEN RETURN : REM BAD KEY ALSO
50420 GOSUB 50900 : REM GOOD KEY SO ACCEPT IT
50425 RETURN
50430 REM ******************************
50435 REM
50440 REM TEST FOR YES OR NO
50445 IF (KEY% < > 89) AND (KEY% < > 78) THEN RETURN : REM BAD KEY
50450 GOSUB 50900 : REM ACCEPT IT
50455 RETURN
50460 REM ******************************
50465 REM
50470 PRINT KEY% AND ADD TO ENTRY$
50475 REM
50480 IF INSERT% = 1 THEN GOSUB 51315 : REM INSERT A SPACE
50485 TXTSIZE% = LEN (ENTRY$) : REM MAKE SURE WE HAVE CORRECT TXTSIZE%
50490 REM ADD TO END OF ENTRY
50495 IF PLACE% > TXTSIZE% THEN ENTRY$ = ENTRY$ + CHR$(KEY%): GOTO 50945
50500 REM ADD AS FIRST CHARACTER
50505 IF PLACE% = 1 THEN ENTRY$ = CHR$(KEY%) + MID$(ENTRY$,PLACE% + 1):
50510 REM ADD AS LAST CHARACTER
50515 IF PLACE% = TXTSIZE% THEN
50520 ENTRY$ = LEFT$(ENTRY$,PLACE% - 1) + CHR$(KEY%): GOTO 50945
50525 REM ADD IN THE MIDDLE SOMEWHERE
50530 TXTSIZE% = LEN (ENTRY$)
50535 REM IF TOO BIG TRUNCATE IT
50540 IF TXTSIZE% > MAXSIZE% THEN ENTRY$ = LEFT$(ENTRY$,MAXSIZE%)
50545 REM
50550 REM NEED TO MOVE RIGHT ONE PLACE
50555 REM
50560 GOSUB 51425 : REM RIGHT ARROW
50565 RETURN
BASIC BUSINESS SUBROUTINES
FOR THE APPLE II AND IIe

50980 REM
50985 REM ********************************************
50990 REM
51000 REM PROCESS A CONTROL KEY
51005 REM
51010 REM EXIT KEYS SUCH AS RETURN SET CTRL%
51015 REM
51020 REM
51025 REM ^A = 1 > PREVIOUS WORD
51030 REM ^D = 4 > DELETE THIS CHARACTER
51035 REM
51040 REM ^F = 6 > FILL WITH A SPACE
51045 REM ^H = 8 > LEFT ARROW
51050 REM ^N = 14 > SKIP TO END
51055 REM ^Q = 17 > HELP REQUEST
51060 REM ^U = 21 > RIGHT ARROW
51065 REM ^W = 23 > NEXT WORD
51070 REM ^Y = 25 > ERASE TO END
51075 REM IGNORE ALL OTHER KEYS
51080 REM
51085 CTRL% = 0 : REM CLEAR EXIT FLAG
51090 IF KEY% = 6 THEN GOSUB 51280: RETURN : REM INSERT
51095 INSERT% = 0 : REM TURN INSERT OFF
51100 IF KEY% = 1 THEN GOSUB 51555: REM PREVIOUS INSERT
51105 REM
51110 IF KEY% = 4 THEN GOSUB 51210: REM DELETE
51115 REM
51120 IF KEY% = 8 THEN GOSUB 51380: REM LEFT ARROW
51125 REM
51130 IF KEY% = 13 THEN CTRL% = 1: REM RETURN KEY
51135 REM CHECK HELP REQUEST
51140 IF KEY% = 17 THEN HELP% = 1 : RETURN : REM HELP REQUEST
51145 REM
51150 IF KEY% = 14 THEN GOSUB 51680: REM GOTO END
51155 REM
51160 IF KEY% = 21 THEN GOSUB 51425: REM RIGHT ARROW
51165 REM
51170 IF KEY% = 23 THEN GOSUB 51475: REM NEXT WORD
51175 REM
51180 IF KEY% = 25 THEN GOSUB 51630: REM ERASE TO END
51184 IF KEY% = 27 THEN CTRL% = 27 : RETURN : REM ESC
CHAPTER 2  AN INPUT LINE EDITOR FOR APPLESOFT BASIC

51185  REM
51190  RETURN
51195  REM
51200  REM  *********************************
51205  REM
51210  REM  DELETE AND PACK
51215  REM
51220  TXTSIZE% = LEN (ENTRY$)
51225  IF TXTSIZE% = 0 THEN RETURN : REM  NOTHING TO DELETE
51230  IF TXTSIZE% = 1 THEN ENTRY$ = "":PLACE% = 1:GOTO 51250: REM  DELETE LINE
51235  IF PLACE% = 1 THEN ENTRY$ = MID$ (ENTRY$,2): GOTO 51250
51240  IF PLACE% < TXTSIZE% THEN
51245         ENTRY$ = LEFT$(ENTRY$,TXTSIZE% - 1):PLACE% = PLACE% - 1:GOTO 51250
51245  ENTRY$ = LEFT$(ENTRY$,(PLACE% - 1)) + MID$ (ENTRY$,PLACE% + 1)
51250  GOSUB 52130 : REM  PRINT NEW STRING
51255  GOSUB 52000 : REM  PRINT INVERSE
51260  RETURN
51265  REM
51270  REM  *********************************
51275  REM
51280  REM  TOGGLE THE INSERT MODE
51285  REM
51290  IF INSERT% = 1 THEN INSERT% = 0: RETURN : REM  TURN IT OFF
51295  INSERT% = 1 : REM  TURN IT ON
51300  RETURN
51305  REM  *********************************
51310  REM
51315  REM  INSERT A CHARACTER
51320  REM
51325  REM
51330  REM  IS IT THE FIRST CHARACTER?
51335  IF PLACE% = 1 THEN ENTRY$ = " " + ENTRY$: GOTO 51350
51340  REM  INSERT IN THE MIDDLE
51345  ENTRY$ = LEFT$(ENTRY$,PLACE% - 1) + " " + MID$ (ENTRY$,PLACE%)
51350  GOSUB 52130 : REM  PRINT THE FIELD
51355  GOSUB 52070 : REM  REPOSITION CURSOR
51360  RETURN
51365  REM
51370  REM  *********************************
51375  REM
BASIC BUSINESS SUBROUTINES FOR THE APPLE II AND IIe

51380      REM  LEFT ARROW
51385      REM
51390      GOSUB 52070  : REM  DISPLAY NORMAL
51395      IF PLACE% > 1 THEN PLACE% = PLACE% - 1: REM  MOVE LEFT ONE
51400      GOSUB 52000  : REM  DISPLAY INVERSE
51405      RETURN
51410      REM
51415      REM  ***********************************************************
51420      REM
51425      REM  RIGHT ARROW
51430      REM
51435      IF MID$(ENTRY$,PLACE%,1) = "" THEN RETURN
51440      GOSUB 52070  : REM  DISPLAY AS NORMAL
51445      IF PLACE% < MAXSIZE% THEN PLACE% = PLACE% + 1
51450      GOSUB 52000  : REM  DISPLAY AS INVERSE
51455      RETURN
51460      REM
51465      REM  ***********************************************************
51470      REM
51475      REM  SKIP TO NEXT WORD
51480      REM
51485      REM
51490      IF PLACE% = > TXTSIZE% THEN RETURN : REM  ALREADY AT END
51495      GOSUB 52070  : REM  REMOVE CURSOR
51500      PLACE% = PLACE% + 1: REM  LOOK FOR FIRST SPACE
51505      IF PLACE% = TXTSIZE% THEN GOTO 51530
51510      IF MID$(ENTRY$,PLACE%,1) < > " " THEN GOTO 51500: REM  IS IT A SPACE?
51515      PLACE% = PLACE% + 1: REM  MOVE RIGHT ONE
51520      IF PLACE% = TXTSIZE% THEN GOTO 51530
51525      IF MID$(ENTRY$,PLACE%,1) = " " THEN GOTO 51515: REM  SKIP OVER SPACES
51530      GOSUB 52000  : REM  DISPLAY CURSOR
51535      RETURN
51540      REM
51545      REM  ***********************************************************
51550      REM
51555      REM  SKIP TO PREVIOUS WORD
51560      REM
51565      IF PLACE% = 1 THEN RETURN : REM  AT THE FRONT ALREADY
51570      GOSUB 52070  : REM  REMOVE CURSOR
51575      PLACE% = PLACE% - 1: REM  LOOK FOR SPACE
51580      IF PLACE% = 1 THEN GOTO 51610: REM  FORCE MOVE AT LEAST ONE SPACE

56
CHAPTER 2  AN INPUT LINE EDITOR FOR APPLESOFT BASIC

51585  IF MID$(ENTRY$,PLACE%,1) = " " THEN GOTO 51575: REM SKIP OVER SPACES
51590  PLACE% = PLACE% - 1
51595  IF PLACE% = 1 THEN GOTO 51610
51600  IF MID$(ENTRY$,PLACE%,1) <> " " THEN GOTO 51590: REM IS IT A SPACE?
51605  PLACE% = PLACE% + 1 : REM POSITION OVER FIRST LETTER
51610  GOSUB 52000 : REM DISPLAY THE CURSOR
51615  RETURN
51620  REM
51625  REM *************************************************************
51630  REM ERASE TO END OF LINE
51635  REM
51640  IF PLACE% = 1 THEN ENTRY$ = "": GOTO 51650: REM ERASE WHOLE LINE
51645  ENTRY$ = LEFT$(ENTRY$,PLACE% - 1)
51650  GOSUB 52130 : REM PRINT THE FIELD
51655  GOSUB 52000 : REM DISPLAY THE CURSOR
51660  RETURN
51665  REM
51670  REM *************************************************************
51675  REM
51680  REM SKIP TO END OF LINE
51685  REM
51690  GOSUB 52070 : REM MOVE THE CURSOR
51695  PLACE% = LEN (ENTRY$) + 1
51700  IF PLACE% > MAXSIZE% THEN PLACE% = MAXSIZE%: REM DO NOT GO PAST END
51705  GOSUB 52000 : REM SHOW THE CURSOR
51710  RETURN
51715  REM
51720  REM *************************************************************
51725  REM
52000  REM PRINT CHARACTER IN INVERSE
52005  REM THIS GIVES THE ILLUSION OF CURSOR MOVEMENT
52010  REM
52012  VTAB ROW% : REM POSITION CURSOR
52015  POKE 36, (COL% + PLACE% - 1) : REM HTAB
52020  INVERSE : REM REVERSE VIDEO
52025  XX$ = MID$(ENTRY$,PLACE%,1): REM MOVE FOR THE NEXT IF
52030  IF XX$ = " " THEN XX$ = " " : REM IF NULL MAKE IT A SPACE
52035  PRINT XX$; : REM PRINT THE INVERSE
52040  NORMAL : REM RESTORE TO NORMAL VIDEO
52045  POKE 36, (COL% + PLACE% - 1) : REM REPOSITION THE CURSOR - HTAB
52050  RETURN

57
52055 REM
52060 REM ****************************
52065 REM
52070 REM POSITION AND DISPLAY NORMAL
52075 REM
52077 VTAB ROW% : REM POSITION CURSOR
52080 POKE 36, (COL% + PLACE% - 1) : REM HTAB
52085 XX$ = MID$ (ENTRY$,PLACE%,1) : REM PRINT ONE LETTER
52090 IF XX$ = "" THEN XX$ = FILL$ : REM IF NULL THEN MAKE IT A SPACE
52095 PRINT XX$;
52100 REM
52105 POKE 36, (COL% + PLACE% - 1) : REM REPOSITION THE CURSOR
52110 RETURN
52115 REM
52120 REM ****************************
52125 REM
52130 REM DISPLAY TEXT$
52135 REM FILL$ IS THE FILL CHARACTER
52140 REM TXTSIZE% IS THE LENGTH OF ENTRY$
52145 REM MAXSIZE% IS THE MAXIMUM ALLOWED LENGTH
52150 REM
52155 REM
52160 TXTSIZE% = LEN (ENTRY$) : REM HOW LONG IS THE CURRENT FIELD?
52165 MAXSIZE% = LEN (MASK$) : REM WHAT IS MAX LENGTH ALLOWED?
52170 REM
52175 REM IS ENTRY$ TOO LONG?
52180 REM
52185 IF TXTSIZE% > MAXSIZE% THEN
52190 ENTRY$ = LEFT$ (ENTRY$,MAXSIZE%):TXTSIZE% = MAXSIZE%
52195 REM
52200 REM
52205 VTAB ROW% : REM ROW POSITION
52210 POKE 36, COL% : REM COLUMN NUMBER - HTAB
52215 REM
52220 REM PRINT THE TEXT
52225 REM
52230 PRINT ENTRY$; : REM NO LINE FEED
52235 REM
52240 REM PRINT THE FILL CHARACTER
52245 REM
52250 IF TXTSIZE% = MAXSIZE% THEN RETURN : REM NO FILL$ TO PRINT
52255 FOR XX = TXTSIZE% TO MAXSIZE% - 1
52260 PRINT FILL$;
52265 NEXT XX
52270 RETURN : REM ALL DONE
52275 REM
52280 REM ****************************
52285 REM
60000 RETURN : REM UNIVERSAL RETURN FOR TESTING
INTRODUCTION

In the previous chapter we created a line editor capable of editing a single line of text. Often, however, it is necessary to enter and edit a paragraph or more of text. In this chapter we will develop a screen text editor that will enable us to enter and edit many pages of text. This program will be developed in two parts. The first part is the actual text editor that can be added to your own programs. The second part provides disk input/output capability. By combining both parts, you will have a stand-alone, text-editing and program development system.

Before we present part 1 of the program, we will discuss the design and the features of the text editor and the complete text editor.
Design

This program is going to be a multipurpose text editor. Not only do we want to be able to edit text as part of a program, but we also want to develop a stand-alone text editor that can be used for writing letters or computer programs. Thus our program supports both purposes.

Another design feature of the program is the output to the printer portion of the program, which may require special commands to be passed depending on the type of printer used. One of the most popular makes, the Epson, requires the code CONTROL I 80 N to be sent to the printer to initialize it before printing more than 40 columns (many other makes also have this requirement). The “80” can be any number between 1 and 255 or the limit of the printer, whichever is smaller. The most common value used is 80. Many special features such as different character fonts and bold printing are possible. See your printer manual for the special codes it requires.

In part 2 we will add to the text editor, giving the complete editor a command area. In this area the user is given the option to print, load, save, auto number, or quit.

Building the Program

The heart of the text editor is the line editor developed in Chapter 2. To build the screen text editor, start with a copy of the line editor program from Chapter 2 and add the additional program lines presented in this chapter. Be sure to make backups at each stage of entry and each time you successfully get through a test point.

User Features

To edit several pages of text, the user must be able to move the cursor up and down a line or a page of text at a time and insert or delete lines. These features are incorporated in the text editor.

Also, you may have noticed that your Apple II screen is only 40 characters wide (this limitation also applies to the Apple IIe with the 80-column card deactivated or off), but most printers are 80 or more
characters wide. Since it would be very nice to be able to print a document wider than 40 characters, we include in our program a line continuation (wraparound) feature that allows us to combine several lines into one. However, in entering text, the user will have to enter a CR at the end of each line; the editor will not do wraparound on the screen.

For a stand-alone text editor we also need to be able to send text to the printer, to load and save text files to the disk, or to quit and forget everything entered. These features are included in our program and are presented in detail in the part 2 discussion. The following list gives the commands, and their explanations, for these additional features. The commands consist of a complete word followed, in some commands, by a disk file name or number. The command words are as follows:

- **LOAD FN**: Load disk file called FN.
- **CATALOG**: Display directory.
- **SAVE FN**: Save text to disk file called FN.
- **DONE FN**: Save text and then QUIT.
- **PACK FN**: Concatenate and save the text file and then QUIT.
- **AUTO #**: Turn auto line numbering on or off.
- **AUTO**: Turn auto line numbering off.
- **EDIT #**: Edit the text starting at line number #.
- **EDIT** or **ESC key**: Return to current page in text editor.
- **PRINT**: Print text file.
- **FORMAT**: Pack and print text file.
- **QUIT**: Return to BASIC and clear the array.

For example, to load and then edit a text file called SALES LETTER, the user would enter

```
LOAD SALES LETTER
```

After the file is loaded, the user could enter either

```
EDIT or EDIT 1 or ESC
```

to begin editing on line 1.
Programmer Features

So that the text editor can be used as a program development tool, the only additional feature needed is auto line numbering, which we have included.

From a programming standpoint all text is kept in a string array. The calling program must tell the editor how large the array is, what line to begin editing on, and the number of the last array element that contains a valid text field.

For part 2, the complete text editor, the programmer must decide how many lines the LINE$ array should contain. In a 48K system 1000 lines are a good choice. This choice will allow the average line to be about 24 characters long. The same choice applies to the 64K Apple //e, since the additional memory is not available for use by a BASIC program.

PART 1: TEXT EDITOR PROGRAM

The flowchart for the basic text editor is shown in Fig. 3.1. This flowchart says that the calling program defines the starting conditions; then the text editor displays a screen of text and edits the first line of text. After the line is edited, it tests to see if the user is done. If the user is not done, the text editor will process the last command. To leave the line editor, the user has to enter either a control character (a RETURN is control M), or an ESCape. Depending on the key entered, the text editor will either move up/down a line/page of text or insert/delete a line.

The program corresponding to the flowchart in Fig. 3.1 (page 66) is as follows:

```basic
41000 REM TEXT EDITOR
41005 REM
41010 REM VARIABLE DEFINITION
41015 REM LROW% STARTING ROW NUMBER
41020 REM LCOL% STARTING COL NUMBER
41025 REM LINE$( ) TEXT ARRAY
41030 REM LAST% DIMENSIONS OF TEXT ARRAY
41035 REM MLINE% LARGEST LINE USED IN ARRAY
41040 REM LINE% CURRENT LINE BEING EDITED
41045 REM FIRST% LINE AT TOP OF SCREEN
```
The following subsections describe various aspects of the text editor program.

**TEST POINT**

Enter the following program lines. They will be used later to test the screen editor.

```plaintext
100  REM
110  REM  SCREEN EDITOR TEST ROUTINE
120  REM
130  DIM LINE$(50) : REM  DEFINE SCREEN ARRAY
140  LAST% = 50 : REM  NUMBER OF TEXT LINES
150  MLINE% = 0 : REM  INITIALIZE VARIABLES
160  ANUM% = 0
170  LROW% = 1
180  LCOL% = 1
190  LINE% = 1
200  FIRST% = 1
500  REM
```
510 REM LEAVE A GAP HERE
520 REM
530 GOSUB 41000 : REM CALL THE EDITOR
590 END

Explanation of Variables

The text editor program requires only a few more variables than the line editor. LROW% (line 41060) and LCOL% (line 41065) are used to set the starting position on the screen. The first time the text editor is called, these variables should be set to 1.
The text to be edited is contained in array LINE$( ) (see line 41080). The calling program must either clear (equate to nulls) or fill in the array with text before it calls the text editor. LAST% is the dimension of the array LINE$. For example, LINE$ is dimensioned by the statement DIM LINE$(LAST%), where LAST% is the number of lines in the array.

**DIM**  The DIMension statement is used to allocate space for an array.

**EXAMPLE:**

```plaintext
DIM A(12)
```

This command provides for elements in the array A from position 0 through 12.

MLINE% is the number of the last used array element. For instance, if the array is dimensioned to 100, but only the first 23 lines contain information, then MLINE% is set equal to 23 and LAST% is equal to 100. If the array is cleared, then MLINE% is set equal to 1.

LINE% is the number of the line currently being edited. The calling program must set it equal to 1 if there is nothing in the array or if editing is to begin on line 1, the first line. If editing is not to begin on the first line, then LINE% is set equal to the line number to be used. For example, if you wish to begin editing on line 25, you set LINE% equal to 25.

FIRST% is the number of the line to appear at the top of the screen. Normally, FIRST% will start with the same value as LINE%.

ANUM% (line 160 of the test routine) will be used to contain the current auto numbering incremental value. If it is equal to 0, then auto numbering will be turned off.

**Explanation of the Program**

The text editor routine displays the current screen, keeps track of the cursor’s position, and processes the exiting control characters. The text editor enters the line editor routine at line 50170 (see line 41100 of the
text editor program). This GOSUB is used because the text editor does not need to have the line redisplayed before or after editing.

PLACE% (lines 41085 and 41090) is used by the line editor to mark the current character being edited. In the text editor PLACE% is set equal to 1 for a RETURN but not for a line feed or a down arrow. Setting PLACE% equal to 1 will cause the cursor to start at the beginning of the line. If the user desires to use the up and down arrows, it looks nicer and is usually more convenient if the cursor stays in the current column position as the cursor is moved up and down through the lines of text, rather than return to the beginning of the line. If PLACE% is larger than the length of the line the cursor is moving to, PLACE% is set equal to the length of the new line so that editing may begin at the end of the line.

**Explanation of Screen Display Subroutine**

The screen display subroutine (line 41055, GOSUB 41800) clears the screen by using the HOME command and then uses a FOR-NEXT loop to display the text array, starting with text line FIRST%.

**HOME** The HOME command clears the text area and moves the cursor to the upper left corner of the screen.

The program for the screen display routine is as follows:

```
41800 REM DISPLAY THE CURRENT SCREEN
41805 REM
41810 HOME : REM CLEAR THE SCREEN
41815 FOR X = 1 TO 24
41820 Z = FIRST% + X - 1
41825 POKE 36, 1 : REM POSITION THE CURSOR - HTAB
41830 VTAB X
41835 PRINT LINE$(Z);
41840 NEXT X
41845 RETURN
41850 REM *****************************************
41855 REM
```
**TEST POINT**

Before testing the screen display routine, you will need to add a temporary program line:

```
41057   END
```

Since we do not want to proceed past this subroutine, this temporary line will cause execution to stop after the screen is displayed. If you now enter

```
RUN (CR)
```

the screen should clear and the program will stop with the cursor on the last line.

If this routine worked so far, add some more temporary lines:

```
400   FOR X = 1 TO 50
410   LINE$(X) = "THIS IS LINE NUMBER " + STR$(X)
420   NEXT X
```

These lines fill the array with text.

Enter

```
RUN (CR)
```

and you should now see the text for lines 2 through 24 displayed. The text for line 1 scrolled off the top of the screen because of the END statement on line 41057.

After the screen display section is tested, line 41057 is no longer needed, so delete it and add

```
41107   END
```

You can now test the line editor as part of the screen editor. Enter

```
RUN (CR)
```
and the screen should clear, the text in the first 24 lines should be displayed, and the cursor should be left on line 1 over the L. You should now be able to edit this line. The program will stop after you enter (CR).

After you are satisfied that everything is working, delete line 41107.

SCREEN EDITOR CONTROL CHARACTER COMMANDS

The screen text editor requires several new control character commands to be added to the line editor routine. These characters will set the CTRL% variable. Recall that if CTRL% is not 0, the line editor will return to the caller. After the text editor saves the edited line, it will process the exit character.

The following new characters and commands are implemented:

- ^B (control B) Insert a blank line into array.
- ^E (control E) Jump to last page.
- ^I (control I) Tab over 8 spaces or add spaces to a line.
- ^J (control J) Down arrow, move down one line.
- ^K (control K) Up arrow, move up one line.
- ^M (control M) RETURN, move down one line.
- ^O (control O) Jump to first page of text.
- ^P (control P) Concatenate two lines of text.
- ^R (control R) Scroll up one full page of text.
- ^T (control T) Scroll down one full page of text.
- ^Z (control Z) Delete a line and compress array.
- ESC (ESCAPE) All done. Exit the text editor.

The tests for these commands are inserted directly into the line editor program from Chapter 2 (see the program lines that follow). They must be inserted manually. After the line editor exits, the text editor interprets the CTRL% character and performs the requested action.

The program lines for inserting the new control characters are as follows:
In the following subsections we will enter and test the new commands.

**Inserting a Blank Line**

Control B (\textasciicircum B) is used to insert a blank line. To insert a blank line, we split the array at the current line number and move all the text on higher line numbers down one line. After the array is moved, the current line becomes the blank line, MLINE\% is incremented, and the screen is redisplayed.

The following routine is used to insert a blank line:

```plaintext
41560 REM INSERT A BLANK LINE
41565 REM
41570 IF MLINE\% < LAST\% THEN MLINE\% = MLINE\% + 1
41575 Y = LINE\%
41580 IF LINE\% = 1 THEN Y = 2
41585 FOR X = MLINE\% TO Y STEP - 1
41590 LINE$(X) = LINE$(X - 1): REM MOVE TEXT DOWN A LINE
41595 NEXT X
41600 LINE$(LINE\%) = "": REM CLEAR THE OLD LINE
41605 GOSUB 41800
41610 RETURN
```
TEST POINT

There are two special boundary conditions that must be tested for at this point. These occur when inserting on either the first or the last line of the array.

Enter RUN (CR) and the cursor will appear on line 1. You can test the first boundary condition by entering ^B. The screen should clear and all the lines be redisplayed, but shifted down one line. The first line should be blank.

The testing of the second boundary condition will be done later, after the cursor can be moved to the bottom of the text.

Jump to the Last Page

Control E (^E) is used to jump to the last page. When you are editing a document, it is very convenient to be able to jump directly to the end of the document. You must test to verify that there is more than one page of text; otherwise, you simply adjust the pointers for the last page.

The following subroutine allows you to jump to the last page of text:

41435 REM JUMP TO LAST PAGE
41440 REM
41445 FIRST% = MLINE% - 23 : REM FIND THE LINE AT THE TOP OF THE SCREEN
41450 IF FIRST% < 1 THEN FIRST% = 1: REM CANNOT HAVE LINE LESS THAN 1
41455 LINE% = FIRST%
41460 ROW% = 1
41465 COL% = 1
41470 GOSUB 41800 : REM DISPLAY THE SCREEN
41475 RETURN
41480 REM ************************************
TEST POINT

Execute the program and enter ^E. The last page of the text should be displayed and the cursor should be on the top line of the screen.

Tab Stops

A control I is the traditional TAB key on a computer terminal, and the default tabs are set at eight spaces each.

Tab stops are a convenient feature to include in the text editor. Their use in program development provides increased readability. In a letter they provide the ability to easily indent text.

The TAB key will move the cursor right to the next tab stop, every eight characters (this is the default value), if there is already text on the line. However, if we are moving the cursor past the end of the text on the line, we must insert spaces into the text line as we TAB.

As an option, this feature could be added directly to the line editor. If you choose to do so, you must modify the line feed routine for the continuation character since it currently uses the TAB routine to indent the next line.

The TAB routine is as follows:

```
41320 REM TAB
41325 REM
41330 PLACE% = ( INT (PLACE% / 8) + 1 ) * 8: REM SLIDE THE CURSOR RIGHT
41332 IF PLACE% > MAXSIZE% THEN PLACE% = MAXSIZE%
41335 IF PLACE% <= LEN (LINE$(LINE%)) THEN
41338 RETURN : REM WITHIN CURRENT FIELD
41340 FOR X = LEN (LINE$(LINE%)) TO PLACE% - 1
41345 LINE$(LINE%) = LINE$(LINE%) + " ": REM ADD SPACES TO THE END
41350 NEXT X
41355 RETURN
41360 REM ********************************************
41365 REM
```
**TEST POINT**

At this point you will do two tests. First, enter ^I on line 1, and the cursor should move over to the eighth column. Since the line contains text, no blanks will be inserted. Next, move the cursor to the front of the line and then delete all of the text by using the delete-to-end-of-line command. Now enter ^I, and eight spaces should be inserted and the cursor positioned on column 8.

**Moving Down a Line**

Two different keys can be used to move down a line: ^J (line feed) or RETURN. The line feed command will keep the cursor in the same column on the screen. The RETURN acts like a carriage return on a typewriter and will move the cursor to the first character of the next line.

To move down a line, we increment both LINE% and ROW% by one. However, we must also test a number of boundary conditions. LINE% cannot be larger than LAST%, since LAST% is the array dimension. If LINE% becomes larger than MLINE% (the bottom of the array), then MLINE% must be incremented. Next, we test the line we are leaving for continuation characters. If there is a continuation character and the next line is blank, editing will begin at the first tab stop on the next line. This feature is especially convenient for the user if a BASIC program is being written. Indenting the line will make it stand out from the rest of the text. We do not indent if the line is not blank because a nonblank line means that the user is editing existing text. Finally, we test the line we are leaving to see whether it is the bottom line. If we want to go down one more line, the screen must be moved up. The screen is moved up by calling the page scroll subroutine.

The following routine moves the text down a line:

```
41135 REM CARRIAGE RETURN
41140 REM
41145 PLACE% = 1
41150 GOSUB 41170 : REM LINE FEED
41155 RETURN
41160 REM *******************************************************
41165 REM
```
CHAPTER 3 SCREEN TEXT EDITOR

41170 REM LINE FEED
41175 REM
41180 IF LINE% = LAST% THEN RETURN : REM MAX NO MORE LINES LEFT
41185 LINE% = LINE% + 1
41190 IF LINE% > MLINE% THEN MLINE% = LINE%: REM INC THE LARGEST LINE COUNTER
41195 ROW% = ROW% + 1
41200 IF ROW% > 24 THEN ROW% = 24:X = 1:LINE% = LINE% - 1: GOSUB 41755
41205 IF (RIGHT$(ENTRY$,1) = "&") AND (LEN(LINE$(LINE%)) = 0) THEN
   PLACE% = 0: GOSUB 41320: REM TAB IN ON NEXT LINE
41210 RETURN
41215 REM ******************************************************
41220 REM

TEST POINT

After the program for moving down a line has been typed in, enter RUN (CR). Once the new screen is displayed, enter another (CR) and the cursor should move to line 2. Now edit line 2 and enter ^J. If there is no text on line 3, the cursor drops down to line 3 and moves to the first column. If there is text on the following lines, then the cursor drops down to the next line, staying in the same column. The page-scrolling test will have to be done after you have entered that routine (which is presented in a later subsection).

Moving Up a Line

The control K (^K) is used to move up one line. To move up a line, we decrement both LINE% and ROW% by one. As in moving down a line, we must also test a number of boundary conditions. LINE% must be greater than 0. If ROW% becomes equal to 0, the screen must be rolled down a line. Rolling down a line is done by telling the page scroll subroutine to move up one line.

The following routine moves the text up a line:

41225 REM UP ARROW
41230 REM
41235 IF LINE% = 1 THEN RETURN : REM AT TOP ALREADY
41240 LINE% = LINE% - 1
Instead of scrolling just one line up or down, you may wish to scroll half a page or 12 lines. You will have to adjust both \texttt{ROW\%} and \texttt{LINE\%} to do so. Some users may be annoyed that the screen refreshes every time it scrolls one line. Others may be annoyed by the cursor jumping from either the top or the bottom row to the middle of the screen. Your implementation is a matter of personal preference.

**TEST POINT**

Execute the program we have presented so far, and after the screen has displayed, enter a couple of carriage returns to move the cursor down the screen. Now enter \^K and move back up one line. Edit this line and enter another \^K. Next, enter a (CR) and return to the line you just edited. It should contain the text you edited. Before proceeding to the next section, check to make sure everything is working correctly. Move the cursor up and down several times and edit several lines. Also, try some of the previous commands, such as line insert and jump to the last page.

**Jump to the Home Page**

To jump to the original or home page, you strike control 0 (^O). This feature comes in handy, for example, when you are proofreading a document. Jumping to the first page is done by setting \texttt{LINE\%}, \texttt{ROW\%}, \texttt{COL\%}, and \texttt{FIRST\%} equal to 1 and then displaying the text.

The program for returning to the home page is as follows:

```
41245  \texttt{ROW\% = ROW\% - 1}
41250  \texttt{IF} \texttt{ROW\% < 1 \texttt{THEN} ROW\% = 1:LINE\% = LINE\% + 1:X = 1: GOSUB 41695}
41255  \texttt{RETURN}
41260  \texttt{REM}
41265  \texttt{REM} \texttt{*****************************}
```

Instead of scrolling just one line up or down, you may wish to scroll half a page or 12 lines. You will have to adjust both \texttt{ROW\%} and \texttt{LINE\%} to do so. Some users may be annoyed that the screen refreshes every time it scrolls one line. Others may be annoyed by the cursor jumping from either the top or the bottom row to the middle of the screen. Your implementation is a matter of personal preference.

**TEST POINT**

Execute the program we have presented so far, and after the screen has displayed, enter a couple of carriage returns to move the cursor down the screen. Now enter \^K and move back up one line. Edit this line and enter another \^K. Next, enter a (CR) and return to the line you just edited. It should contain the text you edited. Before proceeding to the next section, check to make sure everything is working correctly. Move the cursor up and down several times and edit several lines. Also, try some of the previous commands, such as line insert and jump to the last page.

**Jump to the Home Page**

To jump to the original or home page, you strike control 0 (^O). This feature comes in handy, for example, when you are proofreading a document. Jumping to the first page is done by setting \texttt{LINE\%}, \texttt{ROW\%}, \texttt{COL\%}, and \texttt{FIRST\%} equal to 1 and then displaying the text.

The program for returning to the home page is as follows:

```
41635  \texttt{REM} \texttt{GOTO} \texttt{THE} \texttt{HOME} \texttt{PAGE}
41640  \texttt{REM}
41645  \texttt{GOSUB} 41860 \texttt{: REM RESET} \texttt{THE} \texttt{POINTERS}
41650  \texttt{GOSUB} 41800 \texttt{: REM SHOW} \texttt{THE} \texttt{SCREEN}
41655  \texttt{RETURN} \texttt{: REM A OK}
```
TEST POINT

To test the routine for jumping to the home page enter ^E to jump to the last page. Enter ^O to jump back to the first page, and test some of the previous commands to be sure they are still working correctly.

Concatenate Two Lines

We use control P (^P) to concatenate (combine) two lines of text into one. The routine that follows will combine the two lines and then delete the second line from the array.

If we are already on the last line, there is nothing to concatenate—this condition is the only special one. If the new line is too long, it will be truncated by the line editor.

The following program combines two lines:

```plaintext
41370  REM  PACK TWO LINES
41375  REM
41380  IF LINE% = MLINE% THEN RETURN : REM  AT THE END
41385  LINE$(LINE%) = LINE$(LINE%) + LINE$(LINE% + 1): REM  PACK THE LINES
41390  IF LINE% = MLINE% THEN GOTO 41410: REM  LAST LINE
41395  FOR X = LINE% + 1 TO MLINE% - 1
41400  LINE$(X) = LINE$(X + 1) : REM  MOVE LINE UP ONE
41405  NEXT X
41410  LINE$(MLINE%) = "" : REM  CLEAR THE LAST LINE
41415  MLINE% = MLINE% - 1 : REM  REDUCE MAX LINE BY ONE
41420  GOSUB 41800 : REM  DISPLAY SCREEN
41425  RETURN
41430  REM  ****************************************
```
original line 2. The rest of the text lines will move up one line. If you jump to the last line, you can test the boundary condition of concatenating a blank line.

**Scroll Up a Page**

Control R (\^R) is used to move up one page of text at a time. In the routine that follows, FIRST% and LINE% are decremented, and the text window (the space occupied by text) is moved up. For this routine we need to test for the top of the document. If the user is already on the top page of text, we will move them to the top row on the screen.

The scrolling-up routine is as follows:

```
41675 REM SCROLL UP A PAGE
41680 REM
41685 X = 24 : REM JUMP A FULL PAGE
41690 REM ENTRY POINT FOR ROLL UP
41695 IF FIRST% <= X THEN GOSUB 41860 :
  ROW% = 1: GOSUB 41800: RETURN : REM JUMP TO TOP OF FIRST PAGE
41700 FIRST% = FIRST% - X : REM MOVE THE TOP LINE
41705 LINE% = LINE% - X : REM CHANGE THE ARRAY POINTER
41710 GOSUB 41800 : REM DISPLAY THE SCREEN
41715 RETURN
41720 REM
41725 REM ***********************************************
41730 REM
```

**TEST POINT**

Enter \^E and jump to the last page. Then use \^R to scroll up one page. Test all of the functions as you did for moving the cursor up one line. Verify that when \^K is entered on the top line of the screen, the text is scrolled down one line.

**Scroll Down a Page**

Control T (\^T) is used to move down a page of text. Scrolling down is just the opposite of scrolling up. Scrolling down is used whenever a document is longer than 24 lines, since a display longer than 24 lines cannot be
shown on one screen page. So in large documents it becomes necessary to be able to jump up and down through the text 24 lines at a time.

The scroll-down subroutine must increment FIRST% and LINE%, then redisplay the screen. The boundary conditions occur at the bottom line and the last page. The variable X is used to set the number of lines to be scrolled. For a full page, scroll X is set equal to 24. The single-line scroll sets X equal to 1, and a half-page scroll sets X equal to 12. If the cursor is already on the last page of text, then it will move to the last line; otherwise, it will remain on the same row on the screen and the text will move one page.

The scrolling-down routine is as follows:

```
41735 REM SCROLL DOWN A PAGE
41740 REM
41745 X = 24 : REM JUMP A FULL PAGE
41750 REM ENTRY POINT FOR ROLL DOWN
41755 IF MLINE% <= 24 THEN LINE% = MLINE% :
     ROW% = MLINE%: RETURN : REM ON FIRST PAGE
41760 IF FIRST% + X > MLINE% THEN FIRST% = MLINE% - 23 :
     LINE% = MLINE%:ROW% = 24: GOTO 41775: REM BOTTOM
41765 FIRST% = FIRST% + X
41770 LINE% = LINE% + X
41775 GOSUB 41800 : REM DISPLAY THE SCREEN
41780 RETURN
41785 REM
41790 REM **********************************************
41795 REM
```

**TEST POINT**

Use ^T to scroll down a page. Then perform the same tests you performed for scrolling up a page.

**Deleting a Line**

The control Z (^Z) command is used to delete a line of text. To delete a line, we split the array at the current line number, and all the text on the higher line numbers is moved up one line. MLINE% is decremented, and the last line is cleared.
The program for deleting a line is as follows:

```
41490 REM DELETE A LINE
41495 REM
41500 IF MLINE% = 1 THEN LINE$(1) = "": GOTO 41535: REM ONLY ONE LINE
41505 IF LINE% = MLINE% THEN
   ROW% = ROW% - 1: LINE% = LINE% - 1: GOTO 41525: REM LAST LINE IN TEXT
41510 FOR X = LINE% TO MLINE% - 1
41515 LINE$(X) = LINE$(X + 1): REM MOVE THE LINES UP
41520 NEXT X
41525 LINE$(MLINE%) = "" : REM CLEAR BOTTOM LINE
41530 IF MLINE% > 1 THEN MLINE% = MLINE% - 1
41535 GOSUB 41800 : REM DISPLAY SCREEN
41540 RETURN
41545 REM
41550 REM *****************************************
41555 REM
```

**TEST POINT**

Special boundary conditions occur on the first and last lines of the program, so special tests are performed by the subroutine for deleting a line to detect these conditions.

Enter RUN (CR) and the cursor will appear on the first line. Enter ^Z and the screen should clear and show what appears to be lines 2 through 25 but are now really lines 1 through 24. Now enter ^E and jump to the bottom of the text. Enter ^Z and the last line should disappear.

**The Escape Key**

The ESC (escape) key is used to exit the text editor subroutine and return the user to the calling program (this will be the command area of part 2).

**SUMMARY OF PART 1**

We have now created a simple text editor. This subroutine can be used as part of a larger program. There are countless applications where such a text editor can be used.
In part 2 of this chapter we will add the routine needed to make this program a full stand-alone screen text editor.

**PART 2: COMPLETE TEXT EDITOR PROGRAM**

The previous subroutines will edit text. It is up to the program that calls the text-editing subroutines to load or save the information edited by the user. If we add these routines to the text editor, we will have a stand-alone screen text editor. We will call this program the complete text editor.

The flowchart for the complete text editor is shown in Fig. 3.2. When a valid command is entered, the editor will execute the command and return to the command screen to await another command.

**FIG. 3.2** Flowchart for the complete text editor
The rather lengthy program listing that follows is the command display and processor routine. The display gives information on the current status of the text being edited and a HELP screen illustrating the available commands.

Before the display is shown, a FRE( ) command is executed. If there is a lot of text in the editor, this command can take several seconds to execute. The FRE( ) command is valuable for two reasons: First, it tells the user how much memory is available to accept additional text; second, it does some housekeeping of the memory for you.

FRE(0) The FRE command returns the amount of memory available to the user in bytes. FRE(0) performs the housekeeping task of clearing unused values from memory, freeing that space for current use.

The number of lines used and the auto line numbering values, if any, are shown in the command area of the program. Additionally, an explanation of the command is displayed.

The line editor is used to accept a command from the user. LEFT$ is used in the command routine to determine if a valid command has been entered. We have chosen to use full words and not abbreviations for the commands.

After a command is executed, the command screen is redisplayed and another command is accepted. Notice in the listing that follows that some of the commands return to line 40045 and some to line 40040. If the command affects the amount of free memory, then another FRE( ) command is executed; otherwise, no FRE( ) command is needed. As mentioned above, the FRE( ) command can take some time to execute if there is a lot of text, so we do not want to use it more often than necessary.

The complete text editor line numbering begins with number 40000. For this complete editor, change the value on lines 130 and 140 in the screen editor test routine from 50 to 1000. Also, delete lines 500 through 540 from the test routine.

The program listing for the command display and processor is as follows:
CHAPTER 3 SCREEN TEXT EDITOR

40000 REM EDITOR * COMPLETE TEXT EDITOR WITH COMMAND AREA
40005 REM
40010 REM USES THE LINE EDITOR WITH A FEW ADDITIONAL EXIT KEYS
40015 REM
40020 REM COMMAND AREA ROUTINE
40025 REM
40030 MLINE% = 1 : REM SET THE MAX LINE COUNTER
40035 GOSUB 41860 : REM CLEAR VARIABLES
40040 FX = FRE (0) : REM CLEAR THE MEMORY
40045 HOME
40050 PRINT "EDITOR COMMAND AREA FREE ";FX: REM TITLE AND FREE MEMORY
40055 PRINT "LINES USED ";MLINE%;" ON LINE ";LINE%;
40060 IF ANUM% > 0 THEN PRINT " AUTO ";ANUM%;
40065 PRINT : PRINT
40070 PRINT ";
40075 PRINT
40080 PRINT
40085 PRINT "LOAD NAME LOAD A TEXT FILE"
40090 PRINT "CATALOG DISPLAY DIRECTORY"
40095 PRINT
40100 PRINT "EDIT ## EDIT LINE NUMBER"
40105 PRINT "ESC EDIT CURRENT LINE NUMBER"
40110 PRINT "EDIT EDIT CURRENT LINE NUMBER"
40115 PRINT
40120 PRINT "AUTO ## AUTO LINE NUMBER"
40125 PRINT "AUTO TURN OFF LINE NUMBERING"
40130 PRINT
40135 PRINT "SAVE NAME SAVE THE FILE"
40140 PRINT "DONE NAME SAVE AND EXIT TO BASIC"
40145 PRINT "PACK NAME PACK, SAVE AND EXIT"
40150 PRINT
40155 PRINT "PRINT PRINT TEXT FILE"
40160 PRINT "FORMAT PACK AND PRINT TEXT FILE"
40165 PRINT
40170 PRINT "QUIT EXIT TO BASIC"
40175 MASK$ = "NAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
40180 ENTRY$ = " " : REM CLEAR IT
40185 ROW% = 4 : REM COMMAND INPUT LINE
40190 COL% = 3
40195 PLACE% = 1 : REM EDITOR SETS PLACE%
40200 FILL$ = " "
The following routine is used to initialize several variables:

```
41860 REM CLEAR EVERYTHING
41865 REM
41870 LINE% = 1 : REM CURRENT LINE NUMBER
41875 FIRST% = 1 : REM TOP LINE ON THE SCREEN
41880 LROW% = 1 : REM START ON FIRST LINE
41885 LCOL% = 1
41890 ROW% = 1
41895 COL% = 1
41900 RETURN
41905 REM ****************************************
```

Various aspects of the command display routine are described in the subsections that follow.
command. The only command that can be executed at this time is QUIT. Try it to verify that at least this command works. As you can see from line 40265, QUIT clears the screen and stops execution.

The Edit Command

The EDIT command, or the ESC key, is used to toggle from the command area to the text editor. The ESC key is used to be consistent with using the ESC key to exit the text editor. The EDIT command has two modes: with or without a line number. If the user knows which line number is to be edited, then EDIT ## is used, where ## is the line to be edited. If the current text window is desired, the user does not enter a line number.

The program detects which method is used by checking the length of the command entered. If it is four characters long, then only EDIT has been entered. If it is longer than four characters, a number must be extracted from ENTRY$. This number is extracted by using a VAL( ) command. If the value (line number) does not fall within the range of line numbers available, then the user is returned to the command area and gets to try again.

VAL( ) The VAL( ) command searches the assigned string for a numerical value and returns that value. The search stops when the first nonnumerical value is encountered.

**EXAMPLE**

```
5000 HAZEL$ = "AGE 12"
5100 CAT$ = VAL(MID$(HAZEL$,4))
5200 PRINT CAT$
RUN
12
```

In this example we search the string HAZEL with VAL( ), starting at character position 4 (use of MID$), to avoid the search from stopping when it encounters the first nonnumerical value.
The edit routine is as follows:

40335  REM EDIT THE TEXT
40340  REM
40345  IF LEN (ENTRY$) = 4 THEN GOSUB 41000: RETURN : REM START AT CURRENT LINE
40350  FIRST% = VAL (MID$ (ENTRY$,5)): REM LINE NUMBER USER WANTS TO EDIT
40355  IF (FIRST% < 1) THEN FIRST% = 1
40360  LINE% = FIRST%
40365  LROW% = 1 : REM START AT TOP LINE
40370  LCOL% = 1 : REM START IN FIRST COLUMN
40375  GOSUB 41000 : REM TEXT EDITOR
40380  RETURN
40385  REM ***********************************************
40390  REM

**TEST POINT**

After the edit routine is entered, you will be able to move back and forth between the command area and the editor. Either enter EDIT (CR) or hit the ESC key. The screen should clear and you should be able to enter and edit text. Use the ESC key when you wish to exit the editor and return to the command area. Once in the command area, hit the ESC key again to be sure that the editor returns the correct display. You are rapidly approaching an operational text-editing system.

**Text-Saving Subroutine**

Another important capability in editing is saving the text that has been entered or edited. Text is saved on disk in a sequential text file. But before a sequential text file is saved to disk, any old file with the same name must be erased. If the old text file is not erased, the new text file will be written over the old text file. If the new file is not as long as the old file, part of the old file will remain at the end of the new file.

The DELETE command is used to delete or erase a file. However, a two-line command sequence is needed and not simply a DELETE command. The sequence that must be used is

```
PRINT DSK$;"OPEN SALES LETTER"
PRINT DSK$;"DELETE SALES LETTER"
```
An attempt to delete a file that does not exist will generate an error message, and the program will abort. If the file is opened before a DELETE is issued, then you have ensured that there is a file to delete. In other words, if the file was there, it was OPENed; but if the file was not there, then the OPEN command created a new file. In either case we have a file that can be deleted, and we do not have to be concerned about getting a FILE NOT FOUND error.

**DELETE** The DELETE command removes a file name from the disk directory, and you will no longer have access to the file. The DELETE command must be preceded by a CHR$(4) in the program.

**EXAMPLE**

5000 PRINT CHR$(4) "DELETE LETTER"

or

5000 A$ = "FRED"
5100 PRINT CHR$(4); "DELETE "; A$

These two examples simply show how you would delete files from within a program. The first example deletes a specific file named LETTER. The second example deletes whatever file has been assigned to A$.

**OPEN** The OPEN command is used in conjunction with the READ and WRITE commands to create and retrieve sequential text files. It allocates a buffer in memory for the text file, and it allows the system to read or write from the beginning of the file.

As mentioned above, if a file with the name you selected does not exist, then the OPEN command will automatically add it to the disk catalog. Once we are sure that a file is deleted, we OPEN it and begin saving the text. If this process seems complex, it is. But it is part of the Apple operating system, and we must work within it.

Text is sent to the disk by using a FOR–NEXT loop and PRINTing the text after a WRITE command has been issued, as follows:

PRINT DSK$; "WRITE SALES LETTER"
**WRITE**  The WRITE command causes all subsequent PRINT statements to print to the disk. WRITE is in effect until an error, an INPUT statement, or a CHR$(4) occurs. A WRITE command must be preceded by a CHR$(4).

**EXAMPLE**

```
5000 PRINT CHR$(4);"WRITE MAY SALES"
```

or

```
5000 A$ = "JUNE WHEAT"
5100 PRINT CHR$(4);"WRITE ";A$
```

The first example shows how a specific file can be written to disk (SALES). The second example shows how any file can be assigned the value of A$.

The text-saving routine that follows allows text to be saved in two different formats. In the first format the text is saved exactly as it looks on the screen. In the second format the continuation symbols are removed and the text is saved in packed or concatenated form. For the second format the array is processed looking for & symbols as the last character of a line. When an & is found, the current line and the next line are concatenated and MLINE% is decremented. After the entire array is packed, it is written to disk by using the same save routine that is used by the first save format.

The & symbol is used as the continuation line symbol within the text editor. When you are typing a lengthy piece of text within the editor, use an & as the last character on a line. The PACK command later removes the & symbols and saves the text.

For the reading of a sequential file without error, the number of lines in the file must be known. Therefore the number of lines (MLINE%) is printed as the first line in the file.

When two lines are concatenated, all leading spaces in the second line are removed. The leading spaces are there to indent the text and make it more legible. If you want spaces at this point in the line, they must be added in front of the & symbol.
Once all the text is written to the disk, it is necessary to CLOSE the file. Closing the file must be done to ensure that the text has actually been written to the disk. That is, as a line is PRINTed, the Apple does not immediately put it on the disk. The text first goes to a temporary storage location in memory (commonly called a BUFFER). This buffer is 256 bytes (characters) long; it is the same size as one sector on the disk. Rather than write the text out at every PRINT command, the Apple waits until it has a full 256-byte sector before it writes. This feature is included for efficiency. It is done automatically as data is sent to or retrieved from the disk.

What happens if the buffer is not completely filled and nothing else is to be sent to the disk? Well, the computer will wait for you to finish filling the buffer, not realizing that you are done. If you turn off the computer, that last partial buffer will never make it to the disk. Hence we have the CLOSE command. It tells the computer that you are done using that disk file and to write the buffer to disk.

**CLOSE**  The CLOSE command deallocates the buffer and, in the WRITE mode, forces the remaining bytes in the buffer to disk. CLOSE must follow a CHR$(4). There are two modes: with and without a file name. With a file name, only that file is closed. Without a file name all files are closed.

**EXAMPLE**

```
5000 PRINT CHR$(4);"CLOSE LETTER"
```

or

```
5000 A$ = "MAY SALES"
5100 PRINT CHR$(4);"CLOSE ";A$
```

or

```
5000 PRINT CHR$(4);"CLOSE"
```

In these three examples we see that the CLOSE command must always be preceded by a CHR$(4), the disk command, and that we do not have to be file-specific, although we may find it convenient to be.
The following subroutine saves text on a disk:

```
40510 REM SAVE TEXT AND END
40515 REM
40520 GOSUB 40595 : REM SAVE THE TEXT
40525 HOME
40530 END
40535 REM **************************************************
40540 REM
40545 REM CONCATENATE AND SAVE FILE
40550 REM
40555 Y = 0
40560 FOR X = 1 TO MLINE%
40565 Y = Y + 1
40570 LINE$(Y) = LINE$(X) : REM PACK THE ARRAY
40575 IF RIGHT$(LINE$(X),1) = ";" THEN GOSUB 40700: GOTO 40580
40580 NEXT X
40585 MLINE% = Y : REM NEW MAX LINE COUNT
40590 REM
40595 REM SAVE THE TEXT
40600 REM
40605 DSK$ = CHR$(4) : REM SET FOR DISK IO
40610 PRINT
40615 ENTRY$ = MID$(ENTRY$,5) : REM GET THE FILE NAME
40620 IF PRT% > 0 THEN GOTO 40650 : REM PRINT IT DON'T WRITE TO DISK
40625 PRINT DSK$;"OPEN ";ENTRY$
40630 PRINT DSK$;"DELETE ";ENTRY$: REM ERASE IT
40635 PRINT DSK$;"OPEN ";ENTRY$ : REM OPEN A NEW CLEAN FILE
40640 PRINT DSK$;"WRITE ";ENTRY$
40645 PRINT MLINE% : REM NUMBER OF LINES
40650 FOR X = 1 TO MLINE%
40655 IF PRT% > 0 THEN
        PRINT LINE$(X): REM SEND IT TO THE PRINTER NO QUOTES
40660 IF PRT% = 0 THEN PRINT CHR$(34);LINE$(X);CHR$(34): REM SEND IT TO THE DISK
40665 NEXT X
40670 IF PRT% > 0 THEN PRT% = 0: PRINT CHR$(12);PR#0: RETURN : REM SET CRT AND FORM FEED
40675 PRINT DSK$;"CLOSE ";ENTRY$
40680 PRINT
40685 RETURN
```
CHAPTER 3 SCREEN TEXT EDITOR

40690   REM ***********************
40695   REM
40700   REM CONCATENATE IT
40705   IF MID$ (LINE$(X + 1),1,1) = " " THEN
          LINE$(X + 1) = MID$ (LINE$(X + 1),2): GOTO 40705: REM STRIP SPACES
40710   XX = LEN (LINE$(Y)) - 1
40715   LINE$(Y) = LEFT$ (LINE$(Y),XX) + LINE$(X + 1): REM STRIP ' & ' AND PACK
40720   X = X + 1
40725   IF RIGHT$ (LINE$(Y),1) = " & " THEN GOTO 40700: REM CONTINUE PACKING
40730   RETURN
40735   REM
40740   REM ***********************

TEST POINT

This test is the first part of a two-part test. (The second part occurs after
the next subsection.) Edit some text, and then save it by entering SAVE
TEST TEXT (CR). Next, execute the save subroutine to see if TEST TEXT
was saved.

Text-Loading Subroutine

Text is saved as a sequential text file by using a PRINT statement. In con­
trast, an INPUT command is used to load the text into the editor.

The Apple II uses the same commands for reading and writing to the
disk and the screen. For the computer to know that you really want the
data to go to or come from the disk, you must use CHR$(4), control D, in
front of the disk command. For convenience, instead of typing CHR$(4)
every time we have a disk command in our programs, we set DSK$ equal
to CHR$(4).

To load a text file, we use a READ command, such as

PRINT DSK$; "READ SALES LETTER"

While the READ command is in effect, all INPUT statements will receive
their data from the disk and not the keyboard. The READ command is in
effect until another CHR$(4) is encountered.
READ The READ command is used to allow the INPUT command or the GET command to retrieve data from a sequential text file one field at a time.

The following routine loads text into the editor:

```basic
40395 REM LOAD A TEXT FILE
40400 REM
40405 DSK$ = CHR$ (4) : REM SET FOR DISK IO
40410 PRINT
40415 ENTRY$ = MID$ (ENTRY$, 5)
40420 IF LEN (ENTRY$) = 0 THEN RETURN : REM THE USER FORGOT THE NAME
40425 PRINT DSK$; "OPEN "; ENTRY$
40430 PRINT DSK$; "READ "; ENTRY$
40435 INPUT X : REM NUMBER OF LINES
40440 REM
40445 FOR Y = 1 TO X
40450 Z = LINE% + Y - 1: REM START INSERT AT THIS LINE
40455 INPUT LINE$(Z): REM READ A LINE
40460 NEXT Y
40465 PRINT : REM TO CLEAR EVERYTHING
40470 PRINT DSK$; "CLOSE "; ENTRY$
40475 IF MLINE% < Z THEN MLINE% = Z : REM SET THE UPPER LIMIT
40480 RETURN
40485 REM
40490 REM
40495 REM *****************************
40500 REM
40505 REM
```

TEST POINT

The test here is to reload the file just saved in the previous test point. To be sure that all the variables are cleared, enter QUIT, RUN the program, and then enter LOAD TEST TEXT (CR). Depress the ESC key, and the screen should display the text you previously saved. If it does not, determine whether the error occurred in the load or the save routine. Check the save routine first.
Auto Line Numbering

Auto line numbering is a convenience feature for editing BASIC programs with the editor. The routine consists of two sections: the switch section, which turns the line-numbering feature on and off, and the actual line-numbering routine.

When the AUTO command is entered without a number or with the number 0, then line numbering will be turned off. Otherwise, line numbering will begin with the line number entered.

When line numbering is turned on, the editor calls the numbering routine with every key (see line 50347 in the program for inserting control characters). The numbering routine tests to see if a space was entered (ASCII value 32) as the first character. If a space was not entered or if numbering is turned off, the key will be accepted as normal and the routine will return to the caller. If a space is entered and line numbering is on, a line is created by using the current line number. Next, the line number is incremented. We prefer to increment by five, but you can select any value. It is not advisable to increment by one unless you write error-free programs.

A new MASK$ character, N, has been added to support auto numbering. The screen editor’s MASK$ is defined only once (on line 40175 in the command display routine). There are a number of different ways to implement this MASK$ feature, but we believe that the method we have used is the most understandable and straightforward.

The routine for auto line numbering is as follows:

```
40285 REM SET THE AUTO NUMBER SWITCH
40290 REM FLIP ITS VALUE
40295 REM
40300 TXTSIZE% = LEN (ENTRY$)
40305 IF TXTSIZE% = 4 THEN ANUM% = 0: RETURN : REM TURN OFF AUTO NUM
40310 ENTRY$ = MID$ (ENTRY$,5) : REM GET THE VALUE
40315 ANUM% = VAL (ENTRY$)
40320 RETURN
40325 REM ******************************************
40330 REM
41270 REM NUMBER
41275 IF KEY% <> 32 THEN GOSUB 50900: RETURN
```
41280 IF ANUM% = 0 THEN GOSUB 50900: RETURN
41285 ENTRY$ = STR$ (ANUM%) + " 
41290 ANUM% = ANUM% + 5
41295 PLACE% = LEN (ENTRY$)
41300 GOSUB 52130 : REM PRINT ENTRY$
41305 GOSUB 52000 : REM CURSOR DISPLAY
41310 RETURN
41315 REM ************************************************************

**TEST POINT**

In the command area, turn auto line numbering on by using the command

```
AUTO 100 (CR)
```

Hit the ESC key to move to the editor. Once in the editor, hit the space bar. The number 100 should appear and the cursor should be positioned at the first tab stop. Enter some text and a (CR). On the second line, hit the space bar again. The number 105 should appear with the cursor at the first tab stop.

To turn auto line numbering off, return to the command area and enter

```
AUTO (CR)
```

Return to the editor and verify that it has turned off.

**Catalog**

There is nothing more annoying than being in a program and discovering that you have forgotten the name of a file or the specific diskette that is in the disk drive. The CATALOG command saves you time and frustration because it eliminates the need to exit the program to do a CATALOG. This feature is user friendly. Since in our editor we use everything the user enters in ENTRY$ as a disk command, this feature can be used to catalog any diskette. For example, CATALOG D2 entered as ENTRY$ will print a catalog of disk 2 to the screen.
**CATALOG** The CATALOG command displays the directory of the files on diskette. When used in a program, it must be preceded by a CHR$(4).

**EXAMPLE**

5000 PRINT CHR$(4); "CATALOG"

or

5000 PRINT CHR$(4); "CATALOG D2"

These examples illustrate the use of the CATALOG command from a program. As shown, the command must be preceded by a CHR$(4), the disk command.

The INPUT command at the end of the routine that follows is provided to allow the user time to read the last group of file names.

The catalog routine is as follows:

```plaintext
40745 REM CATALOG DISK
40750 REM
40755 HOME : REM CLEAR SCREEN
40760 DSK$ = CHR$(4)
40765 PRINT
40770 PRINT DSK$; ENTRY$: REM ENTRY$ CONTAINS FULL REFERENCE
40775 PRINT
40780 PRINT
40785 INPUT "ENTER RETURN TO CONTINUE"; ENTRY$: REM PAUSE AT BOTTOM
40790 RETURN
40795 REM ****************************
40800 REM
```

**TEST POINT**

Type in catalog (CR) from the command area. Drive one should activate, and the directory should appear on the screen. The catalog routine is the end of the text editor program. To be sure that nothing has been acciden-
tally changed during entry, go back and test every feature and verify that the program works properly.

If everything has tested satisfactorily so far, delete lines 100–590 and replace them with the following lines:

```
100  LAST% = 1000: REM NUMBER OF LINES OF TEXT
110  DIM LINE$(LAST%): REM THE TEXT ARRAY
120  REM
```

These lines are necessary for executable code in the future.

**ENHANCEMENTS**

The screen text editor is very useful for program development and for creating help files and letters. It should be used to enter the programs for the subsequent chapters. Enhancements can be made either by implementing more commands in the command mode or by using the remaining control characters in the text editor itself. By looking at some of the word processors on the market, you may find additional features to add. Remember, however, that as more features are added, the amount of memory available for text becomes smaller, so the size of your largest possible document shrinks.

**MERGING PROGRAMS BY USING EXEC**

Working your way through the rest of the book will become much easier. Now you can use the editor, which you just typed in and debugged, for the entry of all the subsequent programs.

Keep in mind that the files you type in with the editor are saved as text files. To load a program saved as a text file, use the command EXEC (file name). EXEC loads a text file and executes it as a program. During an EXEC the Apple treats each line of the text file as if it were being typed in from the keyboard. In this manner the text is converted to program format (it can be saved as an Applesoft file now).
If you used continuation symbols in your text, then you must save the file by using the PACK command to strip out the continuation symbols. When using PACK, remember to use a file name different from the one you used with SAVE; otherwise, you will erase the original unpacked text file. Next, this packed text file can be EXECed; the DOS SAVE command is used to save the file as an Applesoft file.

Two text files can be merged together by EXECing first one and then the other into memory (they are loaded sequentially) and then saving them to disk under a new name—thus giving you a new, contiguous program. These merging techniques will see a lot of use in the coming chapters and in your own programming as you write programs and add pieces of others to new programs.

As a text file is being EXECed, the Apple prints a ] symbol for each line accepted. Occasionally, a SYNTAX ERROR message will be printed on the screen. This error means that the Apple has encountered an illegal command and that this line was not accepted. You will have to correct the line by comparing a listing of the original text file with the accepted program listing. Remember, the Apple treats every line of the EXECed text file as though it were typed directly from the keyboard. When you enter a bad program line from the keyboard, the Apple gives you a SYNTAX ERROR; therefore it will give the same message when a bad line is EXECed.

Be aware that EXECing a file into memory will cause it to merge with anything that is already present in memory—like your HELLO program. It is a good idea to type NEW before EXECing a file into memory.

USER INSTRUCTIONS

The user instructions presented here are included for two reasons. First, they will help you understand the text editor program and its capabilities. Second, they should be part of the documentation you prepare for any program you write that incorporates the text editor.

The following sections cover entry and editing of text, using the command area, creating new documents, and saving, loading, merging, and printing documents.
Entering and Editing Text in the Text Editor

With the text editor you are able to enter and edit text. You may move the cursor up or down from line to line or page to page. You can insert blank lines or delete lines anywhere in the text, and you can enter lines longer than the screen width.

The text editor uses all the commands of the line editor plus several additional control keys. The newly added keys and their functions are as follows:

- `^B` (control B) Insert a blank line into array.
- `^E` (control E) Jump to last page.
- `^I` (control I) Tab over 8 spaces or add spaces to line.
- `^J` (control J) Down arrow, move down one line.
- `^K` (control K) Up arrow, move up one line.
- `^M` (control M) RETURN, move down one line.
- `^O` (control O) Jump to first page of text.
- `^P` (control P) Concatenate two lines of text.
- `^R` (control R) Scroll up one full page of text.
- `^T` (control T) Scroll down one full page of text.
- `^Z` (control Z) Delete a line and compress array.
- ESC (ESCAPE) All done. Exit the text editor.

Command Area

In addition to being able to enter and edit text, the screen editor is capable of loading and saving text to the disk or sending it to the printer. The commands consist of a complete word followed, in some commands, by a disk file name or number. The command words are as follows:

- LOAD FN Load the disk file called FN.
- CATALOG Display the disk directory.
- AUTO # Turn auto line numbering on or off.
Creating a New Document

To enter a new document, you simply enter either

EDIT 1 or EDIT or ESC

while in the command mode. When the cursor reaches the bottom of the screen, the text will scroll up a line so that you may continue editing without interruption. Also, if you are on the top row of the screen and going up, the text scrolls down a line until you reach the first line of text.

When you finish editing text, enter ESC to return to the command area.

Saving a Document

There are three different ways to save text. First, you can use the SAVE FN command, where FN is the disk file name of your choice. This command will save the current text file and return you to the command area. Second, DONE FN will save the text file and return you to BASIC. Finally, PACK FN will concatenate all the continuation lines and save the concatenated file. Once completed, PACK will return you to BASIC.

Here are some examples of how to use these commands:
SAVE LETTER TO FRED
SAVE MAILING LIST

DONE EDITOR PROGRAM
DONE PRICES

PACK LETTER TO FRED PACKED
PACK PACKED EDITOR PROGRAM

All the files saved to disk are called TEXT FILES. They are identified by a T in front of them when you do a catalog.

**Loading an Existing Document**

An existing document is loaded by using the LOAD FN command. For example, to load and then edit an existing text file called PARTS LIST, you enter

```
LOAD PARTS LIST
```

After the file is loaded, enter either

```
EDIT or EDIT 1 or ESC
```

to begin editing the text on line 1.

If you wish to begin editing the document on some line other than line 1, enter

```
EDIT #
```

where # is the number of the line you wish to edit. For example, to begin editing on line 34, enter EDIT 34.

**Merging Documents**

A text file on the disk can be merged with text already in memory. This task is accomplished by placing the cursor one line lower than the last desired line of the text in memory; the line marked by the cursor and all fol-
lowing lines will be lost. Then ESC to the command mode. Load the new
text from disk (you must load all of it) by using the LOAD FN command,
and the text will be merged. Save this new document under a new name
or you will lose the original.

**Printing Documents**

Any loaded document can be printed by using either the PRINT FN or the
FORMAT FN command. The PRINT command will print the document ex-
actly as seen on the screen. The FORMAT command will concatenate the
continuation lines before printing.

**COMPLETE SCREEN TEXT EDITOR PROGRAM**

Here is the complete screen text editor listing:

```plaintext
100     LAST% = 1000   : REM NUMBER OF LINES OF TEXT
110     DIM LINE$(LAST%)  : REM THE TEXT ARRAY
120     REM
40000   REM EDITOR * COMPLETE TEXT EDITOR WITH COMMAND AREA
40005   REM
40010   REM USES THE LINE EDITOR WITH A FEW ADDITIONAL EXIT KEYS
40015   REM
40020   REM COMMAND AREA routine
40025   REM
40030   MLINE% = 1       : REM SET THE MAX LINE COUNTER
40035   GOSUB 41860      : REM CLEAR VARIABLES
40040   FX = FRE (0)     : REM CLEAR THE MEMORY
40045   HOME
40050   PRINT "EDITOR COMMAND AREA   FREE ";FX: REM TITLE AND FREE MEMORY
40055   PRINT "LINES USED ";MLINE%;" ON LINE ";LINE%;
40060   IF ANUM% > 0 THEN PRINT " AUTO ";ANUM%;
40065   PRINT : PRINT
40070   PRINT ">";
40075   PRINT
40080   PRINT
```

101
PRINT "LOAD NAME LOAD A TEXT FILE"
PRINT "CATALOG DISPLAY DIRECTORY"
PRINT "EDIT ## EDIT LINE NUMBER"
PRINT "ESC EDIT CURRENT LINE NUMBER"
PRINT "EDIT EDIT CURRENT LINE NUMBER"
PRINT "AUTO ## AUTO LINE NUMBER"
PRINT "AUTO TURN OFF LINE NUMBERING"
PRINT "SAVE NAME SAVE THE FILE"
PRINT "DONE NAME SAVE AND EXIT TO BASIC"
PRINT "PACK NAME PACK, SAVE AND EXIT"
PRINT "PRINT PRINT TEXT FILE"
PRINT "FORMAT PACK AND PRINT TEXT FILE"
PRINT "QUIT EXIT TO BASIC"

MASK$ = "NAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
ENTRY$ = ""
REM CLEAR IT
ROW% = 4
REM COMMAND INPUT LINE
COL% = 3
PLACE% = 1
REM EDITOR SETS PLACE%
FILL$ = " 
GOSUB 50120
REM ENTER LINE EDITOR AT HELP% = 0 LINE

REM IF E THEN CALL THE EDITOR
IF KEY% = 27 THEN ENTRY$ = "EDIT": REM ESC MEANS EDIT TEXT
IF LEFT$ (ENTRY$,4) = "AUTO" THEN GOSUB 40285: REM AUTO LINE NUMBER
IF LEFT$ (ENTRY$,4) = "PACK" THEN GOSUB 40545: HOME : END : REM PACK
IF LEFT$ (ENTRY$,4) = "EDIT" THEN GOSUB 40335: GOTO 40040: REM EDIT
IF LEFT$ (ENTRY$,4) = "DONE" THEN GOTO 40510: REM SAVE AND END
IF LEFT$ (ENTRY$,4) = "LOAD" THEN GOSUB 40395: GOTO 40040: REM LOAD
IF LEFT$ (ENTRY$,7) = "CATALOG" THEN GOSUB 40745: REM DISK CATALOG
IF LEFT$ (ENTRY$,4) = "SAVE" THEN GOSUB 40595: REM SAVE THE TEXT
IF LEFT$ (ENTRY$,5) = "PRINT" THEN PR# 1:PRT% = 1: GOSUB 40595: REM PRINT
IF LEFT$ (ENTRY$,6) = "FORMAT" THEN
PR# 1:PRT% = 1: GOSUB 40545: HOME : END : REM PACK PRINT
IF LEFT$ (ENTRY$,4) = "QUIT" THEN HOME : END : REM CLEAR AND QUIT
GOTO 40045: REM TRY AGAIN
REM ******************************************************
40285  REM  SET THE AUTO NUMBER SWITCH
40290  REM  FLIP ITS VALUE
40295  REM
40300  TXTSIZE% = LEN (ENTRY$)
40305  IF TXTSIZE% = 4 THEN ANUM% = 0: RETURN : REM  TURN OFF AUTO NUM
40310  ENTRY$ = MID$ (ENTRY$,5) : REM  GET THE VALUE
40315  ANUM% = VAL (ENTRY$)
40320  RETURN
40325  REM  ************************************************************
40330  REM
40335  REM  EDIT THE TEXT
40340  REM
40345  IF LEN (ENTRY$) = 4 THEN GOSUB 41000: RETURN : REM  START AT CURRENT LINE
40350  FIRST% = VAL (MID$ (ENTRY$,5)): REM  LINE NUMBER USER WANTS TO EDIT
40355  IF (FIRST% < 1) THEN FIRST% = 1
40360  LINE% = FIRST%
40365  LROW% = 1 : REM  START AT TOP LINE
40370  LCOL% = 1 : REM  START IN FIRST COLUMN
40375  GOSUB 41000 : REM  TEXT EDITOR
40380  RETURN
40385  REM  ************************************************************
40390  REM
40395  REM  LOAD A TEXT FILE
40400  REM
40405  DSK$ = CHR$ (4) : REM  SET FOR DISK IO
40410  PRINT
40415  ENTRY$ = MID$ (ENTRY$,5)
40420  IF LEN (ENTRY$) = 0 THEN RETURN : REM  THE USER FORGOT THE NAME
40425  PRINT DSK$;"OPEN ";ENTRY$
40430  PRINT DSK$;"READ ";ENTRY$
40435  INPUT X : REM  NUMBER OF LINES
40440  REM
40445  FOR Y = 1 TO X
40450  Z = LINE% + Y - 1: REM  START INSERT AT THIS LINE
40455  INPUT LINE$(Z): REM  READ A LINE
40460  NEXT Y
40465  PRINT : REM  TO CLEAR EVERYTHING
40470  PRINT DSK$;"CLOSE ";ENTRY$
40475  IF MLINE% < Z THEN MLINE% = Z: REM  SET THE UPPER LIMIT
40480  RETURN
40485  REM
REM *******************
REM
REM SAVE TEXT AND END
REM
GOSUB 40595 : REM SAVE THE TEXT
HOME
END
REM *******************
REM
REM CONCATENATE AND SAVE FILE
REM
Y = 0
FOR X = 1 TO MLINE%
Y = Y + 1
LINE$(Y) = LINE$(X)
NEXT X
MLINE% = Y
REM 
REM SAVE THE TEXT
DSK$ = CHR$(4)
PRINT DSK$; "&" THEN GOSUB 40700: GOTO 40580
: REM NEW MAX LINE COUNT
ENTRY$ = MID$(ENTRY$,5) : REM GET THE FILE NAME
IF PRT% > 0 THEN GOTO 40650: REM PRINT IT DON'T WRITE TO DISK
PRINT DSK$; "OPEN ";ENTRY$: REM 
PRINT DSK$; "DELETE ";ENTRY$: REM ERASE IT
PRINT DSK$; "OPEN ";ENTRY$: REM OPEN A NEW CLEAN FILE
PRINT DSK$; "WRITE ";ENTRY$
PRINT MLINE% : REM NUMBER OF LINES
FOR X = 1 TO MLINE%
IF PRT% > 0 THEN PRINT LINE$(X): REM SEND IT TO THE PRINTER NO QUOTES
IF PRT% = 0 THEN PRINT CHR$(34);LINE$(X);CHR$(34): REM SEND IT TO THE DISK
NEXT X
IF PRT% > 0 THEN PRT% = 0: PRINT CHR$(12): PR#O: RETURN : REM SET CRT AND FORM FEED
PRINT DSK$; "CLOSE ";ENTRY$
CHAPTER 3  SCREEN TEXT EDITOR

40680  PRINT
40685  RETURN
40690  REM  *****************************************************
40695  REM
40700  REM CONCATENATE IT
40705  IF MID$(LINE$(X + 1),1,1) = " " THEN
        40705  LINE$(X + 1) = MID$(LINE$(X + 1),2): GOTO 40705: REM STRIP SPACES
40710  XX = LEN(LINE$(Y)) - 1
40715  LINE$(Y) = LEFT$(LINE$(Y),XX) + LINE$(X + 1): REM STRIP "&" AND PACK
40720  X = X + 1
40725  IF RIGHT$(LINE$(Y),1) = "&" THEN GOTO 40700: REM CONTINUE PACKING
40730  RETURN
40735  REM
40740  REM  *****************************************************
40745  REM CATALOG DISK
40750  REM
40755  HOME : REM CLEAR SCREEN
40760  DSK$ = CHR$(4)
40765  PRINT
40770  PRINT DSK$;ENTRY$ : REM ENTRY$ CONTAINS FULL REFERENCE
40775  PRINT
40780  PRINT
40785  INPUT "ENTER RETURN TO CONTINUE";ENTRY$: REM PAUSE AT BOTTOM
40790  RETURN
40795  REM  *****************************************************
40800  REM
41000  REM TEXT EDITOR
41005  REM
41010  REM VARIABLE DEFINITION
41015  REM  LROW%  STARTING ROW NUMBER
41020  REM  LCOL%  STARTING COL NUMBER
41025  REM  LINE$( ) TEXT ARRAY
41030  REM  LAST%  DIMENSIONS OF TEXT ARRAY
41035  REM  MLINE%  LARGEST LINE USED IN ARRAY
41040  REM  LINE%  CURRENT LINE BEING EDITED
41045  REM  FIRST%  LINE AT TOP OF SCREEN
41050  FILL$ = " " : REM DEFINE THE FILL CHARACTER
41055  GOSUB 41800 : REM DISPLAY THE SCREEN
41060  ROW% = LROW% : REM START AT LAST ROW
41065  COL% = LCOL% : REM START AT LAST COL
41070  REM
41075 REM TOP OF EDIT LOOP
41080 CTRL% = 0; REM CLEAR THE EXIT FLAG
41085 ENTRY$ = LINE$(LINE%) : REM PUT CURRENT LINE INTO LINE EDITOR
41090 IF PLACE% > LEN (ENTRY$) THEN PLACE% = LEN (ENTRY$) + 1: REM
ASSIGN PLACE% HERE
41095 IF PLACE% = 0 THEN PLACE% = 1: REM NULL LINE
41100 GOSUB 50165: REM EDIT THE TEXT BUT DO NOT REDISPLAY ENTRY$
41105 LINE$(LINE%) = ENTRY$ : REM SAVE THE EDITED LINE
41110 IF KEY% = 27 THEN LROW% = ROW%; LCOL% = COL%: RETURN : REM BACK TO CALLER
41115 ON CTRL% GOSUB 41130,41165,41215,41635,41675,
41120 41735,41490,41560,41435,41320,41370 : REM PROCESS KEY
41125 GOTO 41075
41130 REM *********************************************
41135 REM CARRIAGE RETURN
41140 REM
41145 PLACE% = 1
41150 GOSUB 41170 : REM LINE FEED
41155 RETURN
41160 REM *********************************************
41165 REM
41170 REM LINE FEED
41175 REM
41180 IF LINE% = LAST% THEN RETURN : REM MAX NO MORE LINES LEFT
41185 LINE% = LINE% + 1
41190 IF LINE% > MLINE% THEN MLINE% = LINE%: REM INC THE LARGEST LINE COUNTER
41195 ROW% = ROW% + 1
41200 IF ROW% > 24 THEN ROW% = 24:X = 1:LINE% = LINE% - 1: GOSUB 41755
41205 IF (RIGHT$(ENTRY$,1) = "&") AND (LEN (LINE$(LINE%)) = 0) THEN
PLACE% = 0: GOSUB 41320: REM TAB IN ON NEXT LINE
41210 RETURN
41215 REM *********************************************
41220 REM
41225 REM UP ARROW
41230 REM
41235 IF LINE% = 1 THEN RETURN : REM AT TOP ALREADY
41240 LINE% = LINE% - 1
41245 ROW% = ROW% - 1
41250 IF ROW% < 1 THEN ROW% = 1:LINE% = LINE% + 1:X = 1: GOSUB 41695
41255 RETURN
41260 REM
41265  REM ********************************************************************
41270  REM NUMBER
41275  IF KEY% < > 32 THEN GOSUB 50900: RETURN
41280  IF ANUM% = 0 THEN GOSUB 50900: RETURN
41285  ENTRY$ = STR$ (ANUM%) + " "
41290  ANUM% = ANUM% + 5
41295  PLACE% = LEN (ENTRY$)
41300  GOSUB 52130 : REM PRINT ENTRY$
41305  GOSUB 52000 : REM CURSOR DISPLAY
41310  RETURN
41315  REM ********************************************************************
41320  REM TAB
41325  REM
41330  PLACE% = ( INT (PLACE% / 8) + 1) * 8: REM SLIDE THE CURSOR RIGHT
41332  IF PLACE% > MAXSIZE% THEN PLACE% = MAXSIZE%
41335  IF PLACE% <= LEN (LINE$(LINE%)) THEN
41340     RETURN : REM WITHIN CURRENT FIELD
41345  FOR X = LEN (LINE$(LINE%)) TO PLACE% - 1
41350     LINE$(LINE%) = LINE$(LINE%) + " ": REM ADD SPACES TO THE END
41355  NEXT X
41360  RETURN
41365  REM ********************************************************************
41370  REM PACK TWO LINES
41375  REM
41380  IF LINE% = MLINE% THEN RETURN : REM AT THE END
41385  LINE$(LINE%) = LINE$(LINE%) + LINE$(LINE% + 1): REM PACK THE LINES
41390  IF LINE% = MLINE% THEN GOTO 41410: REM LAST LINE
41395  FOR X = LINE% + 1 TO MLINE% - 1
41400     LINE$(X) = LINE$(X + 1) : REM MOVE LINE UP ONE
41405  NEXT X
41410  LINE$(MLINE%) = " " : REM CLEAR THE LAST LINE
41415  MLINE% = MLINE% - 1 : REM REDUCE MAX LINE BY ONE
41420  GOSUB 41800 : REM DISPLAY SCREEN
41425  RETURN
41430  REM ********************************************************************
41435  REM JUMP TO LAST PAGE
41440  REM
41445  FIRST% = MLINE% - 23 : REM FIND THE LINE AT THE TOP OF THE SCREEN
41450  IF FIRST% < 1 THEN FIRST% = 1: REM CANNOT HAVE LINE LESS THAN 1
41455  LINE% = FIRST%
ROW% = 1
COL% = 1
GOSUB 41800 : REM DISPLAY THE SCREEN
RETURN
REM **********************************************************
REM DELETE A LINE
REM IF MLINE% = 1 THEN LINE$(1) = "": GOTO 41535: REM ONLY ONE LINE
IF LINE% = MLINE% THEN
   ROW% = ROW% - 1:LINE% = LINE% - 1: GOTO 41525: REM LAST LINE IN TEXT
FOR X = LINE% TO MLINE% - 1
   LINE$(X) = LINE$(X + 1): REM MOVE THE LINES UP
NEXT X
LINE$(MLINE%) = "" : REM CLEAR BOTTOM LINE
IF MLINE% > 1 THEN MLINE% = MLINE% - 1
GOSUB 41800 : REM DISPLAY DSCREEN
RETURN
REM **********************************************************
REM INSERT A BLANK LINE
REM IF MLINE% < LAST% THEN MLINE% = MLINE% + 1
Y = LINE%
FOR X = MLINE% TO Y STEP - 1
   LINE$(X) = LINE$(X - 1) REM MOVE TEXT DOWN A LINE
NEXT X
LINE$(LINE%) = "" : REM CLEAR THE OLD LINE
GOSUB 41800 : REM DISPLAY SCREEN
RETURN
REM **********************************************************
REM GOTO THE HOME PAGE
GOSUB 41860 : REM RESET THE POINTERS
GOSUB 41800 : REM SHOW THE SCREEN
RETURN : REM A OK
CHAPTER 3 SCREEN TEXT EDITOR

41660 REM
41665 REM ******************************************************
41670 REM
41675 REM SCROLL UP A PAGE
41680 REM
41685 X = 24 : REM JUMP A FULL PAGE
41690 REM ENTRY POINT FOR ROLL UP
41695 IF FIRST% <= X THEN GOSUB 41860 :
41700 FIRST% = FIRST% - X : REM MOVE THE TOP LINE
41705 LINE% = LINE% - X : REM CHANGE THE ARRAY POINTER
41710 GOSUB 41800 : REM DISPLAY THE SCREEN
41715 RETURN
41720 REM
41725 REM ******************************************************
41730 REM
41735 REM SCROLL DOWN A PAGE
41740 REM
41745 X = 24 : REM JUMP A FULL PAGE
41750 REM ENTRY POINT FOR ROLL DOWN
41755 IF MLINE% <= 24 THEN LINE% = MLINE% :
41760 IF FIRST% + X > MLINE% THEN FIRST% = MLINE% - 23 :
41765 FIRST% = FIRST% + X
41770 LINE% = LINE% + X
41775 GOSUB 41800 : REM DISPLAY THE SCREEN
41780 RETURN
41785 REM
41790 REM ******************************************************
41795 REM
41800 REM DISPLAY THE CURRENT SCREEN
41805 REM
41810 HOME : REM CLEAR THE SCREEN
41815 FOR X = 1 TO 24
41820 Z = FIRST% + X - 1
41825 POKE 36, 1 : REM POSITION THE CURSOR - HTAB
41830 VTAB X
41835 PRINT LINE$(Z);
41840 NEXT X
41845 RETURN
41850  REM  **********************************************************************
41855  REM
41860  REM  CLEAR EVERYTHING
41865  REM
41870  LINE% = 1  : REM  CURRENT LINE NUMBER
41875  FIRST% = 1  : REM  TOP LINE ON THE SCREEN
41880  LROW% = 1  : REM  START ON FIRST LINE
41885  LCOL% = 1
41890  ROW% = 1
41895  COL% = 1
41900  RETURN
41905  REM  **********************************************
50347  IF MID$ (MASK$,PLACE%,1) = "N" THEN
51102  GOSUB 41270: RETURN : REM  50347 AUTO NUMBER
51112  IF KEY% = 2 THEN CTRL% = 8 : REM  51112 ^B
51116  IF KEY% = 5 THEN CTRL% = 9 : REM  51116 ^E
51122  IF KEY% = 9 THEN CTRL% = 10 : REM  51122 ^I TAB
51124  IF KEY% = 10 THEN CTRL% = 2 : REM  51124 ^J
51126  IF KEY% = 11 THEN CTRL% = 3 : REM  51126 ^K
51132  IF KEY% = 15 THEN CTRL% = 4 : REM  51132 ^O
51134  IF KEY% = 16 THEN CTRL% = 11: REM  51134 ^P
51142  IF KEY% = 18 THEN CTRL% = 5 : REM  51142 ^R
51157  IF KEY% = 20 THEN CTRL% = 6 : REM  51157 ^T
51182  IF KEY% = 26 THEN CTRL% = 7 : REM  51182 ^Z
An on-line help system is one of the most user-friendly features a program can have. A good help system will save everyone involved with the computer both time and frustration. If the operators are not familiar with computers, they often feel afraid of making mistakes or of appearing stupid when they are confused about what the computer wants them to do. With an on-line help system they can ask the computer for assistance as often as needed. They do not have to bother you, and you will not have to answer the same question a dozen times. Once again, user-friendly software is programmer friendly.

In this chapter we develop a subroutine to display user-help screens. These screens are kept on disk and can be called automatically by the help routine whenever the user has questions. This subroutine uses the line editor subroutine from Chapter 2 as one of its building blocks.
A help system is easy to implement with the use of our line editor. We add new features to the line editor that set a variable called HELP% whenever the user requests help by entering control Q ("Q). In addition to the help subroutine, the help system consists of a series of sequential text files. A separate text file is used for each help request to be supported.

In your program you must test HELP% after every call to the line editor. If the user has requested help, then the help subroutine is called. It will clear the screen and display the appropriate help file and pause at the bottom of the screen with a request for a carriage return. After users have read the help message, they enter RETURN, and the program redisplays the original screen and continues processing.

The help text files can be entered by using the text editor developed in Chapter 3. For a help screen to be useful, it should give a manual page reference (you are going to document your program, aren't you?) and as much information as possible on what the program wants the user to do at that point. Be sure to incorporate these features in your program.

In the following sections we develop the user help program. First, we describe the design and features of the program. Then we create and test each of its component subroutines.

Note that Chapter 4 is a stand-alone chapter, i.e., it does not merge with any other chapter.

**Programmer Features**

For the help program presented in this chapter, the programmer only needs to pass a file name to the help subroutine and restore the original screen after the help subroutine is finished. The help file contains information about what to display in normal or inverse text and when to pause.

**Design and User Features**

We want the help subroutine to display a text file with highlights and pause periodically to allow the user to read the screen. Thus our help system supports INVERSE video and pauses every 22 lines (a full screen) or whenever directed by the text file.
The user should be able to terminate the help display and return to the program and should be able to control the pace at which the screen is read. These features are included in our help program.

Building the Program

To build the help program, start with a copy of the line editor program from Chapter 2 and add the program lines from this chapter. The easiest way to accomplish this task is to proceed as follows:

1. Start with a fresh, initialized disk for the program in this chapter.
2. Transfer a copy of the line editor from Chapter 2 to the new disk (it should be an A, or Applesoft, file), using DOS.
3. Using the screen editor from Chapter 3, type in the program lines presented in this chapter and save them (these lines will be T, or text, files), call them HELP.T.
4. Using DOS, load the copy of the line editor from Chapter 2 into memory, and EXEC the file from this chapter (HELP.T), or as much as you have entered so far, into memory. Now the two files are merged and can be saved again on the disk as Applesoft files, ready to run the next time you load them.
5. Repeat steps 3 and 4 each time you enter more of the program from this chapter.

HELP PROGRAM

Figure 4.1 shows the flowchart for the help program. The processing of the screen is straightforward. After the text file is opened and the total number of lines contained in the file (LAST%) is determined, then each line is read and processed, one character at a time. The subroutine is looking for the characters used to toggle inverse video or the pause symbol. If one of these characters is found, then a GOSUB is made to the appropriate supporting subroutine. If a regular text character is found,
then it is printed. Finally, after all the characters on a line have been processed, a carriage return is printed and a line counter is incremented.

After 22 lines have been shown, the subroutine pauses and the user is asked,

DO YOU WISH MORE?
This message gives the user time to read the screen and an opportunity to stop the help display process if the question has already been answered. This procedure is repeated until all the lines contained in the help file have been shown or the user requests an exit.

The following program corresponds to the flowchart in Fig. 4.1:

```
21000 REM SHOW HELP * SCREEN DISPLAY ROUTINE
21005 REM
21010 REM
21015 REM
21020 REM DISPLAYS SCREEN AND USES INVERSE
21025 REM WILL PAUSE WHEN THE SCREEN IS FULL
21030 REM PROVIDES USER TIME TO READ
21035 REM
21040 REM
21045 REM IMPORTANT VARIABLES USED:
21050 REM LINE% NUMBER OF LINES TO DISPLAY
21060 REM Y INVERSE FLAG
21065 REM ROW% TOTAL NUMBER OF LINES DISPLAYED
21070 REM XX NUMBER OF LINES IN CURRENT SCREEN
21075 REM NOPAUSE% 1 = DO NOT PAUSE AT END OF PAGE. 0 = PAUSE
21080 REM HELP$ NAME OF HELP TEXT FILE
21085 REM
21090 REM LOAD AND DISPLAY THE SCREEN
21095 REM
21100 REM SCREEN READ ONE CHARACTER AT A TIME
21110 REM INVERSE TOGGLED ON ^ character
21115 REM
21120 REM
21125 REM
21130 REM
21135 ROW% = 0 : REM CLEAR COUNTER
21140 XX = 0
21145 DSK$ = CHR$(4) : REM DISK ^D
21150 PRINT : REM CLEAR ANY DSK COMMANDS
21155 PRINT DSK$; "OPEN ";HELP$
21160 PRINT DSK$; "READ ";HELP$
21165 REM
21170 REM READ THE NUMBER OF LINES ON THE SCREEN
21175 REM
21180 INPUT LINE%
21185 REM
```

115
REM CLEAR THE COUNTERS
ROW% = 0
REM
Y = 0 : REM INVERSE FLAG
REM
REM INPUT THE SCREEN
REM
HOME : REM CLEAR SCREEN
REM
ENTRY$ = MID$(A$,Z,1)
REM GET A TEXT LINE
FOR Z = 1 TO LEN(A$)
ENTRY$ = MID$(A$,2,1)
REM CLEAR SCREEN
REM GET A TEXT LINE
IF ENTRY$ = "@" THEN GOSUB 21430: REM PAUSE WANTED
IF ENTRY$ = "^" THEN GOSUB 21380: REM TOGGLE INVERSE VIDEO
PRINT ENTRY$;
NEXT Z
PRINT
ROW% = ROW% + 1 : REM INCREMENT LINE COUNTER
XX = XX + 1 : REM INCREMENT THIS PAGE LINE COUNTER
IF (ROW% = LINE%) OR (XX = 22) THEN GOSUB 21430: REM DO I PAUSE?
IF ROW% > = LINE% THEN PRINT DSK$;"CLOSE ";HELP$: RETURN :
   REM RETURN TO THE CALLER
REM
GOTO 21240 : REM GET NEXT CHR
REM
REM ************************************************************
REM
The following subsections explain various aspects of the help routine.

**Explanation of Variables**

Most of the variables used in the help program are the garbage variables Y, XX, and YY. Check to be sure that the calling routine is not also using these variables if you are adding HELP to one of your programs.

There are two variables that must be set by the calling routine: HELP$ and NOPAUSE%. HELP$ is to contain the name of the text file to
be shown. NOPAUSE% is tested to determine whether the pause line is to be printed. It is normally set to 0 so that the pause is performed, but occasionally a pause may not be wanted. For example, this feature is used in Chapter 5.

**Explanation of Main Help Routine**

The help routine is fairly simple, but we would like to discuss a few specific lines and explain why they were used.

Line 21250 sets

\[
\text{ENTRY$} = \text{MID$(A$, Z, 1)}
\]

This statement sets one character from the input line and puts it in ENTRY$.

On lines 21255 and 21265 ENTRY$ is compared with the special help characters. We could have written these tests as

\[
\text{IF MID$(A$, Z, 1) } = \text{"®" THEN GOSUB 21430}
\]

and so on. However, MID$ is a slow command, and the routine runs much faster if MID$ is used only once.

Two separate counters are maintained in the routine: one for the total number of lines to be displayed and the other for the number of lines shown on this screen. When the screen counter reaches 22, or when the end of the text file is reached, then the pause subroutine is called. After all the lines have been shown, the screen is cleared and the subroutine returns to the calling program.

**TEST POINT**

Before you can start testing the help routine, you need to enter a sample help screen. We suggest that you use the editor to enter the CHAPS MENU HELP text on pages 193–194.

The following program will test the help routine. It will not pause or set inverse, but it will display the text and then end.
PAUSE SUBROUTINE

The pause subroutine is called when an @ symbol is encountered in the help text and at the end of the text file.

When the help text is created, the writer can make the routine pause by placing @ symbols in the body of the text. Pauses improve the readability of the help text by separating topics in the text. For example, the help text may first give a brief explanation of what the user is to do, then give several screens of detailed explanation. If the brief explanation only uses ten lines, it would look awkward and confusing to mix it with the beginning of the detailed explanation that follows. So a pause is used to separate the sections. The pause subroutine also gives the user the chance to exit the help system and return to the original screen.

The pause routine looks like this:

21430 REM PAUSE AND ASK FOR MORE?
21435 REM
21440 REM
21445 IF NOPAUSE% > 0 THEN RETURN : REM PROGRAMMER DOES NOT WANT PAUSE
21450 PRINT : REM CLEAR GET COMMAND
21455 PRINT DSK$: REM TURN READ OFF
21460 VTAB 23: REM PAUSE LINE
21465 ENTRY$ = " ": REM MAKE SURE NOTHING HERE
21470 INPUT "DO YOU WISH MORE? ";ENTRY$
21475 HOME : REM CLEAR THE SCREEN
21480 IF ENTRY$ = "N" THEN ROW%= LINE%: RETURN : REM THEY WANT OUT
21485 PRINT DSK$;"READ ";HELP$: REM TURN DISK INPUT BACK ON
If NOPAUSE% is greater than 1, then this whole routine is ignored and a pause is not allowed. If the user responds N to the question

DO YOU WISH MORE?

then the program sets the line counter YY to LINE%. This statement tells the line-processing routine that the complete file has been read, and it will CLOSE the help text file and return to the calling routine.

**TEST POINT**

The pause routine can be tested by using the previous test routine, but first delete line 21430 from the pause routine. Execute the program and it should pause at the end of the help screen.

**TURNING INVERSE ON AND OFF**

The Apple BASIC command INVERSE can be used to highlight important sections of the help text. This feature enhances the overall quality and appearance of the help screens.

The ^ symbol is used to mark the beginning and the end of the text to be shown in INVERSE. Whenever this symbol is encountered, the current state of INVERSE is reversed. In other words, if the display is in the NORMAL condition and a ^ is encountered, then the INVERSE command is issued and a flag is set to remind the user that INVERSE is on. When the next ^ is encountered, a NORMAL command is issued and the flag is reset to 0. This technique allows an individual letter or word or the entire text to be highlighted.
EXAMPLE:

THIS IS A ^TEST^  

This statement results in the word TEST being shown in inverse video.

The program listing for turning INVERSE on and off is as follows:

```
21375 REM
21380 REM TOGGLE INVERSE ON/OFF
21385 REM
21390 ENTRY$ = ""
21395 IF Y > 0 THEN Y = 0: NORMAL : RETURN : REM CLEAR INVERSE
21400 INVERSE : REM TURN INVERSE ON
21405 Y = 1 : REM SET FLAG
21410 RETURN
21415 REM
21420 REM ************************************
21425 REM
```

TEST POINT

The inverse routine can also be tested by using the previous test routine, but first delete line 21375 from the inverse routine. Once executed, the program should show highlighted areas and pause at the bottom of the screen.

HELP TEST ROUTINE

This help test routine can be used to test any help file. It asks for the name of the help file and displays it. A sample help file is provided on the disk called CHAP8 MENU HELP.

The test routine is as follows:

```
100 REM TEST ROUTINE FOR HELP DISPLAY
110 REM
```
CHAPTER 4  ANSWERING USER HELP REQUESTS

Note that the test routine presented above is a general-use testing routine. It is not for use in this chapter if the previously presented testing routine is used. They will not work together.

USER INSTRUCTIONS

The following list contains the user instructions for the help system just created. Please add these instructions to your user's manual.

• If you have any questions about what information is to be entered or how to respond to a particular request from the computer, you can request help by striking a control Q (^Q). If the computer can help you in this section, it will clear the screen and display a help message.

• Periodically, and at the end of the message, the computer will pause and ask,

DO YOU WISH MORE?

If you enter either a Y and a RETURN or simply a RETURN, then more text will be displayed if it is available. If you enter an N, or if the end of the help text has been reached, then the computer will re-display the original screen, and you may continue processing.

• If the help message does not answer your questions, refer to the manual or contact the system operator.
COMPLETE HELP PROGRAM

Here is the complete listing of the help program:

100 REM TEST ROUTINE FOR HELP DISPLAY
110 REM
120 HOME
130 PRINT "SAMPLE IS CALLED: CHAP8 MENU HELP"
140 PRINT
150 INPUT "ENTER HELP SCREEN NAME ";HELP$
160 GOSUB 21000 : REM DISPLAY SCREEN
170 GOTO 100
180 REM
190 REM ************************************************************************
200 REM
21000 REM SHOW HELP * SCREEN DISPLAY ROUTINE
21005 REM
21010 REM
21015 REM
21020 REM DISPLAYS SCREEN AND USES INVERSE
21025 REM WILL PAUSE WHEN THE SCREEN IS FULL
21030 REM PROVIDES USER TIME TO READ
21035 REM
21040 REM
21045 REM IMPORTANT VARIABLES USED:
21050 REM LINE% NUMBER OF LINES TO DISPLAY
21060 REM Y INVERSE FLAG
21065 REM ROW% TOTAL NUMBER OF LINES DISPLAYED
21070 REM XX NUMBER OF LINES IN CURRENT SCREEN
21075 REM NOPAUSE% 1 = DO NOT PAUSE AT END OF PAGE. 0 = PAUSE
21080 REM HELP$ NAME OF HELP TEXT FILE
21085 REM
21090 REM LOAD AND DISPLAY THE SCREEN
21095 REM
21100 REM SCREEN READ ONE CHARACTER AT A TIME
21110 REM INVERSE TOGGLED ON '//' CHARACTER
21115 REM
21120 REM
21125 REM
REM ROW% = 0
XX = 0
DSK$ = CHR$ (4)
PRINT DSK$; "OPEN ";HELP$
PRINT DSK$; "READ ";HELP$
REM
REM READ THE NUMBER OF LINES ON THE SCREEN
REM
INPUT LINE%
REM
REM CLEAR THE COUNTERS
REM
ROW% = 0
REM
Y = 0
REM
REM INPUT THE SCREEN
REM
HOME
REM CLEAR SCREEN
INPUT A$
REM GET A TEXT LINE
FOR Z = 1 TO LEN (A$)
ENTRY$ = MID$ (A$,Z,1)
REM CLEAR SCREEN
NEXT Z
PRINT
REM INCREMENT LINE COUNTER
XX = XX + 1
REM INCREMENT THIS PAGE LINE COUNTER
IF (ROW% = LINE%) OR (XX = 22) THEN GOSUB 21430: REM DO I PAUSE?
IF ROW% > = LINE% THEN PRINT DSK$; "CLOSE ";HELP$: RETURN :
REM RETURN TO THE CALLER
REM
GOTO 21240
REM GET NEXT CHR
REM
REM ***********************
REM
REM TOGGLE INVERSE ON/OFF
REM
ENTRY$ = ""
IF Y > 0 THEN Y = 0: NORMAL : RETURN : REM CLEAR INVERSE
INVERSE
Y = 1
RETURN
REM
REM **************************************
REM PAUSE AND ASK FOR MORE?
REM
REM
IF NOPAUSE% > 0 THEN RETURN : REM PROGRAMMER DOES NOT WANT PAUSE
PRINT
PRINT DSK$; REM TURN READ OFF
VTAB 23: REM PAUSE LINE
ENTRY$ = ": REM MAKE SURE NOTHING HERE
INPUT "DO YOU WISH MORE? ";ENTRY$
HOME : REM CLEAR THE SCREEN
IF ENTRY$ = "N" THEN ROW% = LINE%: RETURN : REM THEY WANT OUT
PRINT DSK$; "READ ";HELP$: REM TURN DISK INPUT BACK ON
XX = 0: REM RESET PAGE LINE COUNTER
ENTRY$ = ": REM REMOVE Y
RETURN : REM GET NEXT LINE
REM
REM **************************************
REM
REM
REM
REM
REM
REM
REM
REM
REM
REM
REM
REM
REM
REM
REM
REM
REM
REM
REM
REM
REM
REM
REM
REM
REM
REM
REM
REM
REM
REM
REM
REM
REM
REM
REM
REM
REM
REM
Two of the most time-consuming problems encountered when you are developing a program are data entry screens and printed reports. The layout, verification, and modification of these items takes up a significant portion of your time. These two functions, however, really represent the finished product. They are what the user actually sees and interacts with.

Users can appreciate this interface with the computer, and it is this interface—data entry screens and printed reports—that forms their image of the computer, the program, and you. No matter how much energy you put into creating a solution to a problem, the user only sees as far as the input/output. They do not care how flexible you made the program, how easy it is to maintain, or how much thought you put into it.
They only care about how readable, presentable, and understandable the input/output is.

As a programmer, you have two choices: You can accept this fact and give the users whatever they want, or you can give them what you think they need and deal with their complaints. We naturally believe in following the path of least resistance, so this chapter is about a user-friendly and very programmer-friendly data entry screen processor. Chapter 7 will deal with the problem of putting the output information on paper.

The data entry screen is what the user sees and interfaces with. Many users consider the screen a barrier between them and their getting a job done. The screens and their logical flow can make the use of the program an enjoyable, productive task—or an unpleasant chore. The data entry screen is where most of the input errors take place. Thus it is particularly important that this part of your program be understandable, predictable, and forgiving.

In this chapter we design and develop a data entry screen program. This program incorporates both the line editor from Chapter 2 and the help system from Chapter 4. This program is a good example of how we build programs from pieces previously created.

In the following subsections we describe the design and features of the data entry screen program. Then in the remaining sections of the chapter we develop, test, and document this program.

**Design**

We want the data entry system to be easy for the user to work with. We also want it to be flexible and easily modifiable, with any modifications not seriously impacting the existing program.

The data entry program builds on the routines that have been developed in previous chapters. The routines of this chapter will be combined with those of Chapters 2 and 4 by the EXEC technique (described in Chapter 3 in the section “Merging Programs by Using EXEC”) to give you the full data entry screen processor.

**Building the Program**

Here is the method to use to build the data entry system:
1. Start with a fresh, initialized disk for the program of this chapter.

2. Transfer copies of the programs developed in Chapters 2 and 4 to your new disk (A, or Applesoft, copies), using DOS. Merge them and save the results under the name DES.A.

3. Using the screen editor from Chapter 3, type in the program lines presented in this chapter and save them (these lines will be T, or text, files) under the name DES.T.

4. Using DOS, load DES.A into memory and EXEC the file of this chapter (DES.T), or as much of it as you have entered so far, into memory. Now the two programs are merged and can be saved again on disk as Applesoft files, ready to run the next time you load them.

5. Repeat steps 3 and 4 each time you enter more of the program from this chapter.

User Features

The line editor is used in the data entry system so that all those wonderful editing features, particularly the ability to edit existing data, are available to the user. The entry system allows the user to move up and down through the fields, editing and making corrections. Before exiting the screen, the user is given the chance to verify and correct entries. All of the editing is done without excessive keystrokes or the implication that the user does not know how to operate the computer. (It is never good practice to have the computer program talk down to the user.)

One very useful feature of the data entry system is the use of original, or default, values. You may recall that the line editor can either accept new information or be started with an initial value. This feature is used to allow default values for every field in the data entry screen. For example, if the user is entering data not used previously, all the data fields are blank. However, if the user wishes to edit data already entered, the user can actually go to the field of interest and modify it by using all the editing features created in Chapter 2. In contrast, the Apple INPUT statement does not allow the display of a default value or allow the user to edit it.
**Programmer Features**

For the programmer we want to minimize the amount of work that has to be done to create and modify screens. It is rather pointless and very boring to keep writing the same type of program over and over again (this application is one that most program generators do fairly well). Therefore we want to write one routine, give it the data, and have it process and return the user input without having to worry about the display or the editing. Our data entry system includes these features.

For our data entry system the programmer must enter two routines plus the data entry screen. The data screen processor edits data stored as a string array. For each routine the programmer must write one routine to load the data into the string array and another routine to retrieve the edited data from the array.

The data entry screen is made up of text describing what information is desired plus data fields. (Recall that a field is what we call each piece of information that the user will be inputting.) A field's data type, screen position, and length, along with all the general screen text, is stored in a sequential text file (created by using the screen text editor of Chapter 3).

The data entry program also loads and displays the screen. As it is displaying the screen, it extracts each field's characteristics. After it has displayed the text portion of the screen, it displays the original, or default, values of each of the fields and then begins editing in the first field. The program performs these steps for every screen in the program—hence there is no duplicated effort and every screen is of a consistent, high quality.

**CREATING A DATA ENTRY SCREEN**

The data entry screen processor uses both the line editor and the help subroutines. As mentioned previously, the line editor is more flexible than the INPUT statement. The help subroutine, with a few additional program lines, is used to display the screen text and process the input mask information. By using the help subroutine, you can have highlighted areas, and you do not duplicate the program lines necessary to display
the screen text. Thus you are in the position of being able to implement help with all of your screens without having to do anything special with the programs.

When using the help subroutine, you can utilize \( ^\wedge \) (inverse) symbols to highlight areas of special interest. For example, displaying the title of the screen in inverse by enclosing it in \( ^\wedge \) symbols is a nice touch. Be careful not to overuse this feature though; overuse can make the screens appear cluttered and confusing.

In addition to containing the text the user is to see, the data screen contains information defining where variables are to be accepted and what their masks will be. For this feature the data mask is enclosed between \(<\) and \(>\) symbols (less than and greater than). For example, if a ten-character alphabet field is desired, it is represented as

\(<\text{AAAAAAAAAAAAAA}>\)

Text entry begins after the \(<\) symbol (the \(<\) and \(>\) symbols will not appear on the screen that the user fills out). As an example, Fig. 5.1 illustrates a sample screen for a mailing list program.

**FIG. 5.1** Sample data entry screen (saved on disk as CHAP5 SCREEN)

```
^NAME AND ADDRESS^

1. NAME <AAAAAAAAAAAAAAAAAAAAAAAAAA>
2. TITLE <AAAAAAAAAAAAAAAAAAAAAAAAAA>
3. ADDRESS <AAAAAAAAAAAAAAAAAAAAAAAAAA>
4. CITY <AAAAAAAAAAAAAAAAAAAAAAAAAA>
5. STATE <AA>
6. ZIP CODE <############>
7. TELEPHONE <AAAAAAAAAAAAAAAAAAAAAAAAAA>
```
A data entry screen can be created by using the text editor presented in Chapter 3. By using the text editor, you can lay out the screen exactly the way you wish it to appear. This method is easier and faster than the trial-and-error approach necessary when using POKEs, HTABs, and VTABs. If the appearance of a screen is to change and the variables that are used in the screen are not changed, then there will be no changes required in the BASIC program. You merely edit the screen, and the next time the screen is used, it will be the new screen. This technique also allows you to easily print copies of the screens so that they can be used as part of the specifications for the project or as part of the user's manual. Even if you have not created as many data entry screens as we have, you can appreciate the ease and flexibility this procedure gives you in creating screens. Having them automatically processed is an added bonus.

SAMPLE VARIABLE-EXCHANGE ROUTINE

The variables used for the example in Fig. 5.1 would have names like NAMES$, ADDRESS$, ZIP, and so on. A general-purpose screen editor cannot use these exact names but must work with a string array. So we use the array LINE$( ) for this purpose.

In order for the data screen processor to use actual data, the actual variables must be exchanged with the string array before the data screen is called. After the data is edited, the array is exchanged with the actual variables. The exchange is done by two subroutines for each screen. One moves the data from the variables into the string array, and the other moves the edited array values back into the actual variables. For example, the following routine moves the data for Fig. 5.1 from the variables into the string array:

```
26000 REM DEFINE CHAP5 SCREEN
26010 REM
26020 REM FILL SCREEN ARRAY WITH VALUES
26030 REM
26040 LINE$(1) = NAME$
26050 LINE$(2) = TITLE$
26060 LINE$(3) = ADDRESS$
```
After the data has been edited and the user exits the data entry screen, the following routine is used to exchange the string array with the actual variables:

```plaintext
26500   REM DEFINE CHAP5 SCREEN
26510   REM
26520   REM FILL VALUES FROM SCREEN ARRAY
26530   REM
26540   NAME$ = LINE$(1)
26550   TITLE$ = LINE$(2)
26560   ADDRESS$ = LINE$(3)
26570   CITY$ = LINE$(4)
26580   ST$ = LINE$(5)
26590   ZIP = VAL(LINE$(6))
26600   TELE$ = LINE$(7)
26610   RETURN
26620   REM ******************************
26630   REM
26640   REM
```

Other than a few minor changes, which are noted in the next section, these routines are all that you have to write to process the screen of Fig. 5.1. A pair of subroutines like these are required for every screen created.

### DATA ENTRY PROGRAM

The flowchart for the data entry screen processor is shown in Fig. 5.2. The basic steps are as follows: display the data entry screen, using the help screen processor; load the string edit array with the starting, or de-
fault, values and show the default values; edit the data. Next, save the edited values and test to see if help has been requested. If help was requested, then show the HELP screen and redisplay the entire screen. If help was not requested, then return to the calling program.
The following program listing corresponds to the flowchart in Fig. 5.2:

```plaintext
20000 REM DATA ENTRY * SCREEN PROCESSOR
20005 REM
20010 REM
20015 REM
20020 REM DISPLAYS SCREEN, LOADS MASK DATA
20025 REM DISPLAYS DEFAULT VALUES
20030 REM EDITS AND SAVES VALUES
20035 REM
20040 REM VARIABLES USED:
20045 REM LINE$( ) HOLDS EDIT DATA
20050 REM ITEM% NUMBER TO EDIT
20055 REM PAGE% PAGE TO EDIT
20060 REM SROW$( ) FIELD ROW NUMBER
20065 REM SCOL$( ) FIELD COL NUMBER
20070 REM SMASK$( ) FIELD MASK$.
20075 REM SCREEN$( ) NAME OF SCREEN
20080 REM LINE% CURRENT LINE BEING EDITED
20085 REM
20090 REM
20095 REM
20100 REM
20105 REM EDIT A DATA SCREEN
20110 REM
20115 REM SHOW THE DATA SCREEN
20120 REM
20125 ITEM% = 0 : REM CLEAR NUMBER OF ITEMS
20130 NOPAUSE% = 1 : REM INFORM HELP SCREEN NOT TO PAUSE
20135 COL% = 1 : REM RESET POSITION COUNTER
20140 HELP$ = SCREEN$(PAGE%) : REM NAME OF SCREEN TO PROCESS
20145 GOSUB 21000 : REM SHOW SCREEN
20150 NOPAUSE% = 0 : REM RESET TO PAUSE
20155 ON PAGE% GOSUB 25000, 26000
20160 GOSUB 20295 : REM SHOW DEFAULT VALUES
20165 GOSUB 20350 : REM EDIT DATA
20170 ON PAGE% GOSUB 25500, 26500 : REM SAVE THE EDITED DATA
20175 IF HELP% > 0 THEN HELP$ = HELP$ + " HELP": GOSUB 21000 :
GOTO 20000 : REM SHOW HELP SCREEN AND START OVER
```
Various aspects of the data screen entry processor are discussed in the following subsections.

**Explanation of the Program**

Lines 20155 and 20170 are GOSUBs to the program sections that load and retrieve the data from the screen processor. This program allows multiple screens to be processed by the subroutine. The screen you wish to process is selected by setting PAGE% to the appropriate value. For example, to use the screen you entered for Fig. 5.1, starting on line 26000, you would set PAGE% equal to 2. (Note: We have not created a screen for lines 25000; this task is left as an exercise for you.) If you wish to use more than two screens, then simply add these line numbers to lines 20155 and 20170. If only one screen is being processed, replace the ON GOSUB commands with a GOSUB command.

If the user requests help, line 20175 will create a help text file name and call the help routine to display the text. We are adopting the convention of adding HELP to the end of the screen name to create a name of the screen’s help text. For example, if the data entry screen has the name MAIL LIST, then the help text will have the name MAIL LIST HELP. If you choose to use a different convention, then modify this line.

**Explanation of Variables**

NOPAUSE% (line 20130) and HELP$ (line 20140) are used by the help subroutine while displaying the screen. If NOPAUSE% is not equal to 0, then the help routine will not ask DO YOU WISH MORE? at the end of the file. HELP$ is the name of the data entry screen.

PAGE% (line 20140) and SCREEN$( ) are used in multiple-screen processing to define which data screen is being processed. If there is only
one screen, you will not need these variables. Just assign the screen name
directly to HELP$.

This data entry program uses several arrays that must be dimen­sioned before they are used. In the test routine (presented in the next sec­tion) they are dimensioned on lines 1030 through 1080. But in your pro­gram you may need to put them elsewhere. The arrays are as follows:

```
1030   LAST% = 10                   : REM (OR WHATEVER SIZE IS REQUIRED)
1040   DIM LINE$(LAST%)            
1050   DIM SROW%(LAST%)            
1060   DIM SCOL%(LAST%)            
1070   DIM SMASK$(LAST%)           
1080   DIM SCREEN$(LAST%)          
```

LAST% is the maximum number of fields allowed. LINE$( ) holds
the values of the fields being edited. SROW%( ) and SCOL%( ) contain
the screen coordinates of each field. SMASK$( ) is the data entry mask,
and SCREEN$( ) has the actual name of each screen.

### CHANGES TO THE HELP SUBROUTINE

Four lines must be added to the help subroutine so that it can process the
field masks or definitions. These lines are as follows:

```
21262  IF ENTRY$ = "<" THEN GOSUB 20205 : REM ASSIGN FIELD CHARACTERISTICS
21263  IF ENTRY$ < > CHR$ (13) THEN COL% = COL% + 1 : REM INC COLUMN
       : REM POSITION COUNTER
21286  COL% = 1                     : REM RESET COLUMN COUNTER
21457  IF NOPAUSE% < > 0 THEN PRINT DSK$; "CLOSE "; SCREEN$: RETURN : REM
```

These lines recognize the < symbol and increment or set the screen col­umn position counter, COL%. Also, a line is added to close the file when
NOPAUSE% does not equal 0.

A < symbol is used to mark the beginning of a field definition. When
it is encountered, a branch is made to a subroutine that sets the field pa­rameters, GOSUB 20205.
The field parameter routine sets the screen row and column coordinates and the edit mask for the field. It continues stepping across the text line one character at a time searching for the > symbol that marks the end of the mask. All the characters in between the symbols are moved into the field mask.

The program looks like this:

```
20205 REM SET FIELD PARAMETERS
20210 REM
20215 ITEMS% = ITEMS% + 1 : REM INC FIELD COUNTER
20220 SMASK$(ITEMS%) = "" : REM CLEAR IT
20225 SROW%(ITEMS%) = ROW% + 1 : REM CURRENT ROW NUMBER
20230 SCOL%(ITEMS%) = COL% : REM CURRENT COLUMN NUMBER
20235 PRINT " "; : REM PRINT SPACE
20240 Z = Z + 1
20245 ENTRY$ = MID$(A$,2 , 1) REM GET ONE CHARACTER
20250 IF ENTRY$ = " >" THEN GOTO 20270 : REM ARE WE AT END OF MASK?
20255 SMASK$(ITEM% ) = SMASK$(ITEMS% ) + ENTRY$ : REM ADD TO FIELD MASK
20260 COL% = COL% + 1 : REM INC CNT
20265 GOTO 20235 : REM GET NEXT CHAR
20270 ENTRY$ = " ": REM CLEAR ENTRY$
20275 RETURN
20280 REM
20285 REM ***************
20290 REM
```

In the program above three arrays plus one counter are used to contain the field parameters. In lines 20225 and 20230 the integer arrays SROW%( ) and SCOL%( ) contain the starting screen coordinates for the fields. These arrays will be assigned to ROW% and COL% when the line editor routine is called. SMASK$( ) in line 20200 is a string array used to contain the mask for the fields. MASK$ will be set from this array. Finally, ITEMS% (line 20215) is used to count the number of fields on the screen. ITEMS% is incremented by one as each field is processed. You may have noticed that ITEMS% is reset to 0 before the screen display subroutine is called.
TEST POINT

At this point you should have entered all the programs presented thus far. Now enter the following program and the data screen shown in Fig. 5.1 (saved as CHAPS SCREEN). After you enter RUN (CR), this screen should be displayed. The program will stop on line 20160 because the default display routine has not been entered yet.

The following test routine for the data entry screen program is designed for only 10 fields and will display the results of the editing. The first time the program is run, the data screen will be blank; the second and subsequent times it will display the values of the previous edit sessions as defaults.

The test routine is as follows:

```plaintext
1000 REM TEST PROGRAM FOR DATA SCREEN PROCESSOR
1010 REM
1020 REM
1030 LAST% = 0
1040 DIM LINE$(LAST%)
1050 DIM SROW%(LAST%)
1060 DIM SCOL%(LAST%)
1070 DIM SMASK$(LAST%)
1080 DIM SCREEN$(LAST%) : REM SCREEN NAMES
1090 REM
1100 REM
1110 REM
1120 PAGE% = 2 : REM ENTER CHAP5 SCREEN
1130 SCREEN$(2) = "CHAP5 SCREEN"
1140 GOSUB 20000 : REM EDIT THE DATA
1150 HOME
1160 PRINT "NAME",NAME$
1170 PRINT "TITLE",TITLE$
1180 PRINT "ADDRESS",ADDRESS$
1190 PRINT "CITY",CITY$
1200 PRINT "STATE",ST$
1210 PRINT "ZIP CODE",ZIP
1220 PRINT "TELEPHONE",TELE$
1230 VTAB 23
1240 POKE 36, 1 : REM HTAB
```
After the text is displayed and LINE$( ) has been loaded with the original or default values, subroutine GOSUB 20295 is called to display these values. By displaying the values, we are presenting the user with a complete picture of what data the computer currently contains. This technique is better than the method of serially showing and editing one field at a time. By being shown all the information at once, the user has a better understanding of what is being requested.

The display routine uses a FOR-NEXT loop to position the cursor from the values in SROW%( ) and SCOL%( ) and then prints the corresponding value from LINE$( ) . The program listing is as follows:

20295 REM SHOW DEFAULT VALUES
20300 REM
20305 FOR X = 1 TO ITEMS%
20310 VTAB SROW%(X)
20315 POKE 36, SCOL%(X)
20320 PRINT LINE$(X);
20325 NEXT X
20330 RETURN
20335 REM
20340 REM ********************************************
20345 REM

TEST POINT

After the display subroutine is entered, the program should display the screen and stop at line 20165. Since values have not been assigned to LINE$( ) , no default will be shown. As an exercise, you might temporarily
add the following lines and RUN the program again just to make sure everything is correct.

1131 LINE$(1) = "NAME"
1132 LINE$(2) = "TITLE"
1133 LINE$(3) = "ADDRESS"
1134 LINE$(4) = "CITY"
1135 LINE$(5) = "STATE"
1136 LINE$(6) = "ZIP CODE"
1137 LINE$(7) = "PHONE NUMBER"

To prevent confusion later, delete these lines after a successful test.

EDITING SUBROUTINE

Once the screen text and the values are displayed, all that remains is to edit the individual fields. The line editor is used, because it allows editing in an existing field and offers cursor controls not available with INPUT. So that the up arrow, down arrow, and ESC can be used, the following program lines must be added to the standard line editor routine:

51124 IF KEY% = 10 THEN CTRL% = 2 : REM ^J LINE FEED 51124
51126 IF KEY% = 11 THEN CTRL% = 3 : REM ^K UP ARROW EXIT 51126

Figure 5.3 (page 141) is a flowchart of the field-editing subroutine, which is as follows:

20350 REM EDIT THE DATA FIELDS
20355 REM
20360 LINE% = 1 : REM START IN DATA FIELD
20365 ENTRY$ = LINE$(LINE%) : REM FIELD
20370 ROW% = SROW%(LINE%) : REM ROW
20375 COL% = SCOL%(LINE%) : REM COL
20380 MASK$ = SMASK$(LINE%) : REM MASK
20385 GOSUB 50000 : REM EDIT FIELD
20390 LINE$(LINE%) = ENTRY$ : REM SAVE THE EDITED DATA FIELD
Editing begins at the first field, and LINE% (line 20360) is used to mark the field being edited. Next, the variables required for the line editor and ENTRY$ are set. After editing, the contents of ENTRY$ are returned to LINE$( ) . Then a check is made to see if the user entered an up arrow or an ESC or requested help. If none of these requests were made, then the program steps to the next field.

After the last field has been edited, or an ESC has been entered, the user is asked,

CHANGE WHICH ITEM?

If either 0 (CR) or (CR) is entered, then the editing is completed and the program returns to the main routine. Otherwise, the cursor is moved to the field requested.
FIG. 5.3  Flowchart for field-editing routine

Start editing on field one

Move field from LINE$() to ENTRY$

Edit the field

Move field from ENTRY$ to LINE$()

Was an up arrow entered?

Yes

Was an ESC entered?

Yes

Is a field to be changed?

Yes

Set field number

No

Ask, CHANGE WHICH ITEM?

Yes

RETURN

No

A

RETURN

Was help requested?

Yes

No

Is this the last field?

Yes

RETURN

No

Step to next field

Step to previous field

Is field one being edited?

Yes

RETURN

No

A

Already at top, so ignore it

RETURN

Yes
The up arrow and ESC are user-friendly, laborsaving features. The user does not have to enter a RETURN for every field on the screen just to get to the CHANGEs line, and it is not necessary to move to the bottom of the screen in order to move up a variable. You may have noticed that the user can enter an up arrow on the CHANGEs line in order to move to the last field. It is often easier to enter several up arrows than to find and enter a field number.

TEST POINT

The data entry system is now complete. You will now want to test the up arrow (‘K) and ESC keys. Also, verify that a field number can be entered on the CHANGEs line and that the field can be edited.

After you enter (CR) on the CHANGEs line, you will be asked by the test routine to ENTER RETURN TO CONTINUE. After you enter this (CR), you will be looped back to the beginning of the data entry screen, and the values you just entered should be displayed as the new defaults.

Before you test the help feature, you will have to create a text file named PAGE 2 HELP. This file can be created by using the text editor from Chapter 3 and can contain any of the help features developed in Chapter 4. After the help screen is displayed and you wish to return to the data entry screen, the screen should be cleaned and the current screen redisplayed.

ADDITIONAL OPTION

In our editing subroutine we chose to begin editing with the first field. You may wish to begin editing with the CHANGEs line instead. When you are editing a blank screen with no defaults, then you will want to begin editing at field one. When you are editing an existing item, then you will wish to begin editing at the CHANGEs line.

If you allow for both alternatives, you have the option of using the editing screen to review data. In addition, one user keystroke, ESC, is removed. This option can be implemented by changing line 20165 to branch
to either 20350 or 20425 depending on a variable set by the routine that called the screen editor. For example, you could use

\[ 20165 \text{ ON EDITSTART}\% \text{ GOSUB 20350, 20425} \]

where EDITSTART\% equals 1 to start in field one and 2 to start on the CHANGEs line.

**USER INSTRUCTIONS**

The following subsections contain the user instructions for the data entry system. Please add these instructions to your user’s manual.

**Entering and Editing Data**

All data entry screens function in the same manner. They all have features that make it easier for you to edit and change data. The data entry screen consists of three parts: (1) the text used to describe the screen; (2) the data fields, i.e., the actual data you can edit; and (3) a line at the bottom of the screen that asks,

CHANGE WHICH ITEM?

We refer to this line as the CHANGEs line.

Portions of the text may be highlighted and appear as black letters on a white background.

In addition to all the editing capabilities described in the line editor chapter, you can move up and down through the fields or jump directly to the bottom of the screen to the CHANGEs line.

**Editing a Field**

Whenever a field is being edited, its current value is shown. If you do not wish to change the field, then enter a RETURN or a line feed (control J, \(^J\)) and you will move to the next field. If you do wish to change a field, then
just change it by using the various editing commands. Remember that the
field is accepted exactly as you see it, so do not leave out or forget any­
thing.

Moving Between Fields

If you are in a field and wish to go to the next field, enter either a
RETURN or a line feed (\'). If you wish to go to the previous field, then
enter an up arrow (^K). You may step up and down through the fields as
often as necessary to correct the entries.

How to Exit the Data Entry Screen

If you are editing fields and wish to stop editing and go to the CHANGEs
line, you may either step down through the fields by entering RETURNS or
jump across all the fields directly to the CHANGEs line by pushing the
ESC key.

CHANGE WHICH ITEM?

The CHANGE WHICH ITEM? question is asked at the bottom of every
data entry screen. If you wish to change a field, you enter its number fol­
lowed by a RETURN. The program will then jump to the field you re­
quested.

An up arrow (^K) can be used to step up from the CHANGEs line, one
field at a time, to the field you wish to change.

If you do not wish to make any corrections, then enter a RETURN and
the program will proceed.

Help Requests

All data entry screens have help text to explain what information the
screen wants and what each of the fields means. To request help, go to
the CHANGEs line and press control Q (^Q). After the help text has been
displayed, the original screen will be displayed and you may edit the
data.
The complete listing of the data entry program is as follows:

```
1000  REM TEST PROGRAM FOR DATA SCREEN PROCESSOR
1010  REM
1020  REM
1030  LAST% = 10
1040  DIM LINE$(LAST%)
1050  DIM SROW%(LAST%)
1060  DIM SCOL%(LAST%)
1070  DIM SMASK$(LAST%)
1080  DIM SCREEN$(LAST%) : REM SCREEN NAMES
1090  REM
1100  REM
1110  REM
1120  PAGE% = 2 : REM ENTER CHAP5 SCREEN
1130  SCREEN$(2) = "CHAP5 SCREEN"
1140  GOSUB 20000 : REM EDIT THE DATA
1150  HOME
1160  PRINT "NAME", NAME$
1170  PRINT "TITLE", TITLE$
1180  PRINT "ADDRESS", ADDRESS$
1190  PRINT "CITY", CITY$
1200  PRINT "STATE", ST$
1210  PRINT "ZIP CODE", ZIP
1220  PRINT "TELEPHONE", TELE$
1230  VTAB 23
1240  POKE 36, 1 : REM HTAB
1250  INPUT "ENTER RETURN TO CONTINUE "; A$
1260  GOTO 1120 / : REM LET'S DO IT AGAIN
1270  REM ****************************
1280  REM
1290  REM
26000  REM DEFINE CHAP5 SCREEN
26010  REM
26020  REM FILL SCREEN ARRAY WITH VALUES
26030  REM
26040  LINE$(1) = NAME$
26050  LINE$(2) = TITLE$
```
BASIC BUSINESS SUBROUTINES FOR THE APPLE II AND IIe

26060  LINE$(3) = ADDRESS$
26070  LINE$(4) = CITY$
26080  LINE$(5) = ST$
26090  LINE$(6) = STR$ (ZIP)
26100  LINE$(7) = TELE$
26110  RETURN
26120  REM *************************************
26500  REM DEFINE CHAP5 SCREEN
26510  REM
26520  REM FILL VALUES FROM SCREEN ARRAY
26530  REM
26540  NAME$ = LINE$(1)
26550  TITLE$ = LINE$(2)
26560  ADDRESS$ = LINE$(3)
26570  CITY$ = LINE$(4)
26580  ST$ = LINE$(5)
26590  ZIP = VAL (LINE$(6))
26600  TELE$ = LINE$(7)
26610  RETURN
26620  REM *************************************
26630  REM
26640  REM

20000  REM DATA ENTRY * SCREEN PROCESSOR
20005  REM
20010  REM
20015  REM
20020  REM DISPLAYS SCREEN, LOADS MASK DATA
20025  REM DISPLAYS DEFAULT VALUES
20030  REM EDITS AND SAVES VALUES
20035  REM
20040  REM VARIABLES USED:
20045  REM  LINE$()  HOLDS EDIT DATA
20050  REM  ITEM%  NUMBER TO EDIT
20055  REM  PAGE%  PAGE TO EDIT
20060  REM  SROW%()  FIELD ROW NUMBER
20065  REM  SCOL%()  FIELD COL NUMBER
20070  REM  SMASK$()  FIELD MASK$
20075  REM  SCREEN$()  NAME OF SCREEN
20080  REM  LINE%  CURRENT LINE BEING EDITED
20085  REM
REM ITEM% = 0 : REM CLEAR NUMBER OF ITEMS
30130 NOPAUSE% = 1 : REM INFORM HELP SCREEN NOT TO PAUSE
30135 COL% = 1 : REM RESET POSITION COUNTER
30140 HELP$ = SCREEN$(PAGE%) : REM NAME OF SCREEN TO PROCESS
30145 GOSUB 21000 : REM SHOW SCREEN
30150 NOPAUSE% = 0 : REM RESET TO PAUSE
30155 ON PAGE% GOSUB 25000,26000
30160 GOSUB 20295 : REM CLEAR NUMBER OF ITEMS
30165 GOSUB 20350 : REM SHOW DEFAULT VALUES
30170 ON PAGE% GOSUB 25500,26500: REM SHOW THE EDITED DATA
30175 IF HELP% > 0 THEN HELP$ = HELP$ + " HELP": GOSUB 21000 :
30180 RETURN
30190 REM ****************************
30195 REM
30200 REM
30205 REM SET FIELD PARAMETERS
30210 REM
30215 ITEMS% = ITEMS% + 1 : REM INC FIELD COUNTER
30220 SMASK$(ITEMS%) = "": REM CLEAR IT
30225 SROW%(ITEMS%) = ROW% + 1 : REM CURRENT ROW NUMBER
30230 SCOL%(ITEMS%) = COL% : REM CURRENT COLUMN NUMBER
30235 PRINT " ";
30240 Z = Z + 1
30245 ENTRY$ = MID$(A$,Z,1) : REM GET ONE CHARACTER
30250 IF ENTRY$ = ">" THEN GOTO 20270 : REM ARE WE AT END OF MASK?
30255 SMASK$(ITEM%) = SMASK$(ITEM%) + ENTRY$: REM ADD TO FIELD MASK
30260 COL% = COL% + 1 : REM INC COL CNT
30265 GOTO 20235 : REM GET NEXT CHAR
30270 ENTRY$ = " ": REM CLEAR ENTRY$
30275 RETURN
30280 REM
30285 REM ****************************
REM SHOW DEFAULT VALUES
FOR X = 1 TO ITEMS%
VTAB SROW%(X) : REM LINE
POKE 36, SCOL%(X) : REM COLUMN - HTAB
PRINT LINE$(X); : REM DATA
NEXT X
RETURN
REM **************
REM EDIT THE DATA FIELDS
REM START IN DATA FIELD
ENTRY$ = LINE$(LINE%) : REM FIELD
ROW% = SROW%(LINE%) : REM ROW
COL% = SCOL%(LINE%) : REM COL
MASK$ = SMASK$(LINE%) : REM MASK
GOSUB 50000 : REM EDIT FIELD
LINE$(LINE%) = ENTRY$ : REM SAVE THE EDITED DATA FIELD
IF (CTRL% = 3) AND (LINE% > 1) THEN LINE% = LINE% - 1 :
    GOTO 20365 : REM UP ARROW
IF CTRL% = 3 THEN GOTO 20365 : REM UP ARROW BUT ALREADY AT TOP
IF CTRL% = 27 THEN GOTO 20425 : REM ESC SO GO TO BOTTOM
IF LINE% < ITEMS% THEN LINE% = LINE% + 1: GOTO 20365: REM MOVE DOWN A LINE
REM VERIFY ENTRIES
REM GOTO BOTTOM
VTAB 24
HTAB 10
PRINT "CHANGE WHICH ITEM?";
MASK$ = "##" : REM ALLOW HELP AND UP TO 99 FIELDS
ENTRY$ = "0" : REM DEFAULT
ROW% = 24
COL% = 30
GOSUB 50000 : REM EDIT DATA
IF CTRL% = 3 THEN LINE% = ITEMS%: GOTO 20365: REM UP ARROW
LINE% = VAL (ENTRY$) : REM LINE TO EDIT
20485 IF LINE% = 0 THEN RETURN : REM ALL DONE WITH THIS SCREEN
20490 IF (LINE% <= ITEMS%) AND (LINE% > 0) THEN GOTO 20365 : REM EDIT
 THE REQUESTED FIELD
20495 GOTO 20425 : REM BAD ENTRY
20500 REM
20505 REM **************************************************************
20510 REM
21262 IF ENTRY$ = "<" THEN GOSUB 20205 : REM ASSIGN FIELD CHARACTERISTICS
21263 IF ENTRY$ < > CHR$ (13) THEN COL% = COL% + 1 : REM INC COLUMN
   POSITION COUNTER
21286 COL% = 1 : REM RESET COLUMN COUNTER
21457 IF NOPAUSE% < > 0 THEN PRINT DSK$; "CLOSE ";SCREEN$ : RETURN : REM
 RETURN WITHOUT A PAUSE
51124 IF KEY% = 10 THEN CTRL% = 2 : REM ^J LINE FEED 51124
51126 IF KEY% = 11 THEN CTRL% = 3 : REM ^K UP ARROW EXIT 51126
In this chapter we develop a menu display program. A menu is a list of actions the computer can perform. The user selects one action, and then either a particular portion of the current program is executed or an entirely new program is loaded and executed. Almost all programs have at least one menu. From a programming standpoint a menu is similar to a data entry screen: It is a simple (and boring) program to write and time-consuming to adjust and modify.

The menu system presented in this chapter is based on the concepts and techniques developed in the previous chapters. The text editor is used to create a menu screen. Imbedded within the text of this screen is information about what programs are to be executed when a particular
option is selected. As for the data entry screen, the help subroutine is used to display the menu. This routine allows highlighting of important information and provides consistency for all of the screens.

In the following subsections we describe the program design and its features.

Design

Depending on how we wish to implement the menu driver, we should be able to either chain to and run another program or branch to a subroutine within the current program. The menu program we present allows for these options. In addition, the program builds on the routines that have been developed in the previous chapters. That is, the routines of this chapter are combined with those of Chapters 2 and 4 by the EXEC technique described in Chapter 3 (see the "Merging Programs by Using EXEC" section) to give you the complete menu system.

Building the Program

Here is the method to use to build the menu program:

1. Start with a fresh, initialized disk for the program of this chapter.
2. Transfer copies of the programs developed in Chapters 2 and 4 to your new disk (A, or Applesoft, copies), using DOS. Merge them and save the results under the name MENU.A.
3. Using the screen editor from Chapter 3, type in the program lines presented in this chapter and save them (these lines will be T, or text, files) under the name MENU.T.
4. Using DOS, load MENU.A into memory and EXEC the file of this chapter (MENU.T), or as much of it as you have entered so far, into memory. Now the two programs are merged and can be saved again on disk as Applesoft files, ready to run the next time you load them.
5. Repeat steps 3 and 4 each time you enter more of the program from this chapter.
User Features

The menu is the user's road map. Without a menu to remind them about what options are available, most users would be hopelessly lost. The menu text must clearly and unambiguously describe what options are available to the user and what the consequences of a particular selection are. A help file, such as we have provided, is therefore absolutely essential with every menu.

In a menu system it is important that consistency be maintained. In our menus we always use the ESC key to step the user back to the previous menu. By knowing this convention, the users will always be able to retrace their steps and return to the beginning of the program. Another technique we always include is an option on every menu that will return the user to the first, or "master," menu.

Our menu selections are always numbered, and a carriage return must be entered every time a selection is made. Occasionally, you may come across a program that requires a carriage return in response to certain questions and none in others. For example, in one place in the program you may have to enter a 1 (CR), while in another similar situation a simple 1 is all that is necessary. The computer automatically adds the (CR). Such a program is a prime example of inconsistent programming, and most users find this inconsistency annoying, irritating, and irrational. Unless a program like this one is used on a daily basis, the user is always entering the unneeded carriage return and accidently causing an undesired action.

Remember, user-friendly software contains no surprises or hidden pitfalls and is consistent. A good menu should provide your users with the visibility necessary to really understand where they are going and why.

Programmer Features

The menu routine can be written by using either one of two methods. Method 1 automatically chains to and runs another program, while method 2 causes a branch to a routine within the current program.

In method 1 the programmer includes the names of the programs to be executed with the actual menu text. The help screen display routine from Chapter 4 is modified to extract the names and not display them.
Method 2 requires either an ON GOSUB or an ON GOTO statement to branch to the appropriate routine within the current program. For either option the programmer simply passes the name of the desired menu screen to the menu routine and it takes over from there.

The program names may be placed on any line of the screen file. A name must be between two # symbols, begin on column 1, and contain the option number in columns 2 and 3. For example,

```
#01 ACCOUNTS RECEIVABLE#
#02 GENERAL LEDGER#
#03 PAYROLL#
```

This listing means that accounts receivable is menu option 1, general ledger is option 2, etc.

The # symbol was chosen in our program to avoid a conflict with the < and > symbols used for the data entry screen, just in case a program contains both a menu and a data entry screen. The choice of symbols is completely arbitrary, and if these symbols conflict with the way you use the data entry or menu screens, then change the symbols. To avoid problems, try to be consistent with your choice of symbols so that all the screens of a given type use the same style.

**MENU PROGRAM**

A flowchart of the menu program is shown in Fig. 6.1. This program is very simple. It first uses the help screen display subroutine to display the text and to strip out the options and the names of the programs to chain to or run. It then asks the user to select an option. After verifying that a valid option number has been entered, the menu either runs the requested program or branches somewhere within the current program, depending on what line 22160 (see below) of the menu contains.

The program listing is as follows:

```
22000 REM MENU DRIVER *
22005 REM
22010 REM
22015 REM
```
The menu routine contains a feature that terminates execution and returns to BASIC. This task is accomplished by using the word END in place of the program name. Line 22150 tests to see if the selected option con-
The menu routine also contains a feature that allows you to call HELP. This task is accomplished by using the word HELP in place of the program name. Line 22152 tests to see if the selected option consists of HELP for the program name; if it does, then the program clears the screen and loads and displays the requested HELP screen.

The default option number assigned on line 22125 is 0. We usually use 0 (CR) and ESC to mean the same thing—go back to the previous menu. Thus those users not familiar enough with the programs to remember to use ESC can still exit the programs with a single keystroke, a carriage return (since 0 is defaulted). Also, since this line always has the same meaning, it is positioned as the last option on the list instead of the first option. When it is at the end of the list, the user is not forced to read it every time a menu appears. This feature is a small touch, but it shows that you have put some thought into the menus and attempt to make them consistent and easy to use.
The menu routine uses the LINE$( ) array. This array must be dimensioned to at least the maximum option number before the menu routine is called. If you are also using the data entry screen, it is not necessary to dimension the array twice. If needed, be sure to place it at the front of the program so that it only executes once.

Other features of the menu program are described in the following subsections.

**Chaining or Branching**

Line 22160 currently is

```
22160 PRINT DSK$; "RUN "; LINE$(X): REM CHAIN TO THE REQUESTED PROGRAM
```

This line will cause the menu to run another program once a valid option has been selected. If you wish to branch to a point within the current program, this line should be changed as follows:

```
22160 ON X GOSUB 1000,2000,3000,ETC: REM BRANCH TO SUBROUTINE
```

where 1000, 2000, 3000, ETC are the subroutine entry points. Do not forget to change these numbers to the correct line numbers in your program.

You will also have to add some lines at the end of your subroutines to either terminate the program on the return or redisplay the menu. For example,

```
22165 END : REM TERMINATE EXECUTION
```

will terminate the program, and

```
22165 GOTO 22000 : REM REDISPLAY THE MENU
```

will cause the menu to be redisplayed.

If you are branching to subroutines and not chaining to any other programs, then you do not need to put any program names in your menu screen. It can contain just the text you wish to display.

Finally, line 22145 must be deleted if you are branching since you are not checking program names and ITEMS% is not being set properly.
Accepting a Program Name

When the screen display routine encounters a # symbol, it branches to line 22180, where the program name is extracted and inserted into the LINE$( ) array. ITEM% contains the value of the largest option found. The program listing for this routine is as follows:

```
22180 REM ACCEPT A CHAIN NAME
22185 REM
22190 Z = VAL ( MID$ (A$,2,2)) : REM GET THE OPTION NUMBER
22195 IF ITEM% < Z THEN ITEM% = Z: REM ADJUST COUNTER
22200 LINES$(Z) = MID$ (A$,4) : REM MOVE THE FILE NAME
22205 ROW% = ROW% + 1 : REM INCREMENT COUNTER
22210 RETURN
22215 REM
22220 REM *****************************************
22225 REM
```

Modifications to the Help Screen Display Routine

Two lines must be added to the existing help screen display routine from Chapter 4 to enable it to process menus. They are as follows:

```
21237 IF ROW% = LINE% THEN PRINT DSK$; "CLOSE "; HELP$: RETURN :
  REM 21237 ALL DONE
21242 IF MID$ (A$,1,1) = "#" THEN GOSUB 22180: GOTO 21237 :
  REM 21242 PROCESS MENU FILE NAMES
```

Using the Help Subroutine

The menu system employs the help routine for user on-line help in the same way that the data entry system does. If the user enters ^Q, then on line 22135 the text

" HELP "

is added to the end of the menu screen name, SCREEN$. Since the help system is a fundamental part of the menu system, you should use it.
Figure 6.2 illustrates one method of entering a menu. This menu is designed to chain to other programs or terminate execution and return to BASIC. Note that both the title and the number 0 will be highlighted since they have been enclosed within ^ symbols.

**FIG. 6.2** Sample menu screen saved as CHAP6 MENU

```
^ACCOUNTING MASTER MENU^

PLEASE SELECT OPTION -

1. ACCOUNTS RECEIVABLE PROCESSING

#01 AR MENU#

2. PAYROLL PREPARATION

#02 PAYROLL MENU#

3. GENERAL LEDGER TRANSACTIONS AND REPORTS

#03 GL MENU#

4. INVENTORY MAINTENANCE

#04 INV MENU#

5. ACCOUNTS PAYABLE PROCESSING

#05 AP MENU#

^0^. ESC - EXIT TO BASIC

#00 END#
```
TEST POINT

First, create a menu to chain onto some programs you previously created or the programs developed in the earlier chapters (which is what is done on the optionally available floppy). Run the program and make sure that the screen is displayed properly and that the correct program is chained.

Second, change lines 22160 and 22165 presented earlier to branch to some small subroutine that you create. These subroutines can do something clever, like clear the screen and print “option 1,” etc. After the subroutine executes, the menu should be redisplayed.

Enter these lines for testing:

```
100 REM TEST ROUTINE FOR MENU DRIVER
110 REM
120 REM
130 DIM LINE$(10)
140 SCREEN$ = "CHAP6 MENU"
150 GOTO 22000: REM DISPLAY THE MENU AND CHAIN
160 END
```

USER INSTRUCTIONS

The following subsections contain the user instructions for the menu system. Please add these instructions to your user's manual.

Using a Menu

A menu is a list of actions that the computer can perform. After reading the available options, select the one you want by entering its number followed by a carriage return. For example, if you were looking at the following screen and you wished to run the general ledger program, you would enter 3 (CR).

```
ACCOUNTING MASTER MENU

PLEASE SELECT OPTION -

1. ACCOUNTS RECEIVABLE PROCESSING
```
2. PAYROLL PREPARATION

3. GENERAL LEDGER TRANSACTIONS AND REPORTS

4. INVENTORY MAINTENANCE

5. ACCOUNTS PAYABLE PROCESSING

0. ESC - RETURN TO PREVIOUS MENU

Requesting Help

If you cannot decide which option to select or you do not understand what is wanted from you, help can be requested by entering a ^Q (control Q) in response to PLEASE SELECT OPTION. The help text contains a detailed explanation of what each option will do. After the help text has been displayed, the menu will be redisplayed.

Exiting, or Which Way Is Out?

To return to the previous menu and eventually back to BASIC, either enter 0 (CR) or simply strike the ESC (escape) key. [Note: The default option is 0; therefore striking the RETURN key is the same as entering 0 (CR).]

COMPLETE MENU PROGRAM

The complete listing of the menu program is as follows:

????? DIM LINE$(10) (NOTE: DON'T FORGET TO INCLUDE THIS DIMENSION STATEMENT AT THE FRONT OF YOUR PROGRAM IF IT IS NOT INCLUDED IN ANOTHER ROUTINE)
IF ROW% = LINE% THEN PRINT DSK$;"CLOSE ";HELP$: RETURN :
REM 21237 ALL DONE

IF MID$(A$,1,1) = "#" THEN GOSUB 22180: GOTO 21237 :
REM 21242 PROCESS MENU FILE NAMES

REM TEST ROUTINE FOR MENU DRIVER
REM
REM
DIM LINE$(10)
SCREEN$ = "CHAP6 MENU"
GOTO 22000: REM DISPLAY THE MENU AND CHAIN
END

REM MENU DRIVER *
REM
REM DISPLAYS MENU AND CHAINS TO PROGRAM OF USERS CHOICE
REM
REM IMPORTANT VARIABLES USED:
REM LINE$( ) CONTAINS NAME OF PROGRAM TO CHAIN TO
REM
DSK$ = CHR$(4) : REM DISK ^D
ITEMS% = 0 : REM NUMBER OF OPTIONS
NOPAUSE% = 1 : REM INFORM HELP SYSTEM NOT TO PAUSE
HELP$ = SCREEN$ : REM SEND THE MENU NAME
GOSUB 21000 : REM SHOW THE SCREEN USING HELP ROUTINE
NOPAUSE% = 0 : REM RESTORE THE FLAG FOR OTHERS
REM
REM ASK FOR OPTION
REM
VTAB 3
POKE 36, 10 : REM HTAB
PRINT "PLEASE SELECT OPTION"
ROW% = 3
COL% = 35
MASK$ = "##" : REM ALLOW HELP REQUEST
ENTRY$ = "0"
GOSUB 50000 : REM ACCEPT OPTION
IF HELP% THEN HELP$ = SCREEN$ + " HELP": GOSUB 21000 :
GOTO 22000: REM RESPONDED TO HELP REQUEST
X = VAL (ENTRY$) : REM OPTION SELECTED
IF X > ITEMS% THEN GOTO 22000: REM BAD ENTRY
IF LINE$(X) = "END" THEN HOME : END : REM RETURN TO BASIC
IF LINE$(X) = "HELP" THEN HELP% = 1:HELP$ = SCREEN$ + " HELP": GOSUB
21000: GOTO 22000: REM RESPONDED TO HELP REQUEST
PRINT : REM CLEAR FOR DISK COMMAND
PRINT DSK$; "RUN ";LINE$(X): REM CHAIN TO THE REQUESTED PROGRAM
REM
REM *************
REM ACCEPT A CHAIN NAME
REM
Z = VAL ( MID$ (A$,2,2)) : REM GET THE OPTION NUMBER
IF ITEM% < Z THEN ITEM% = Z: REM ADJUST COUNTER
LINE$(Z) = MID$ (A$,4) : REM MOVE THE FILE NAME
ROW% = ROW% + 1 : REM INCREMENT COUNTER
RETURN
REM
REM *************
Generating reports is a time-consuming programming task. Creating a dump of the data is not time-consuming; creating polished and meaningful reports is what takes the time. This chapter deals with the problem of report generation. First, we discuss some philosophical aspects of report creation, making some suggestions. Then we present and explain a program designed to make the task of actually creating a report easier for you, the programmer.
PHILOSOPHICAL CONSIDERATIONS

We have all heard it said many times that "data goes into the machine; information comes out." A report containing raw data is usually of little value, unless the only purpose of the report is to record all data, as in a scientific experiment notebook. But a programmer typically is trying to change data into information, and this process can be broken into some definite steps of action. We will describe these steps, giving an example and our recommendations, in the following subsections.

Steps in Report Generation

The first thing to do is to consider what information is to be presented and what data this information is generated from. How much of the data is needed to support the information directly? How much can be left out or output in a separate report (produced only when required) to be used to verify the accuracy of data entry or collection?

Second, consider who the report is for. Is it for the vice-president of marketing, the engineering staff, the secretarial pool, or the maintenance team? Even though they are using the same data/information, these users may have individual requirements and formats. Also, each group looks at the data from a different perspective.

Third, consider how the report is going to be used. Several different presentations of the same data/information may be required.

An Example

To illustrate what we mean, we'll look at a very common example that most of us come into contact with at one time or another: the quarterly stockholder's report for a corporation. It comes out in one style and one format for everyone. Whether or not you understand it, that's how you get it. It is usually dressed up on slick printing stock with many color photographs. This trick is known as camouflage—no one reads the numbers anyway, right? If the data were really important, we suspect, it would not be presented only in standard CPA jargon and format. However, the information could be presented in several formats within the
same report so that the stockholders would have a clear understanding of the company's financial condition and changes without having to learn how to decipher the data.

In an improved reporting format the standard balance sheet and profit/loss statement could be supplied in the report for reference and for those who prefer to read them in their formal format. For the rest of us the key points could be called out and explained as follows:

1. Our new warehouse in Irvine is now open and has been fully stocked with 1,000,000 Widgets, raising our inventory value by $Z and the overhead by $T.

2. A 1000-acre plot surrounding our Silicon Valley plant has been sold, reducing the value of our physical assets by $X and increasing our cash holdings by $Y.

3. The 10,000,000 model 1963 Widgets stored in Alaska have been donated to the local junior college, giving us a tax credit of $C and an inventory write-off of $D, and freeing 60,000 square feet of needed warehouse space.

This type of disclosure tells the stockholders what has transpired. In fact, it may tell them too much. For this reason you must consider the three points listed previously and zero in on your target reader when you write your report. The corporations may have decided that they will present their reports only one way because of the huge and varied audience they address. You, on the other hand, have a smaller audience for your reports, and hence you can be more specific in what you report and how you present it.

**Recommendations**

So, if designing a good report requires addressing the three points, which focus on user requirements, it follows that you should talk to the potential users and survey their needs. Their responses will vary from "We do not have any idea" or "We do not care" to very specific and well-thought-out requirements. Since these reports are for users, their requirements should receive your thoughtful attention. For those of you unfortunate
enough to get the “no help” answers, try to consider the user’s viewpoint. Then try to visualize the finished product, and create a sample report. Once you provide the users with a sample, they unfailingly have no problem coming up with criticisms (they will call them suggestions).

Actually, trying to make all users happy is an impossible task, and some political skills will be required. For instance, you may want to remind (inform, educate) the users that the computer does not do the organization or report preparation. You must design it for the computer, and the user’s cooperation is necessary and appreciated.

**SIMPLE REPORT GENERATOR**

Getting information into the computer is usually easier than getting it out, because you may take a single page of input data and create twenty pages of output information. Designing and programming twenty pages of output is a lot of work. The program presented in this chapter will be helpful for outputting many types of reports. (However, it may be totally worthless for others.)

The report generator is based on a concept similar to that used for the data entry screen. The text file contains both the regular text to appear on the page and information describing the variable and its position on the printed page. In cases where data is being extracted from a file and listed one line at a time down the page, a simple loop may still be the best way to create the report.

**Design**

As in the data entry screen, the report generator uses an array of variables. This situation is a simple one and will generally only be useful on pages containing a small amount of information. You can improve on this technique by adding additional markers to represent special or recurring variables in your report. For example, you might use markers for general ledger account numbers or inventory part numbers.
Building the Program

The program presented in this chapter does not use any of the subroutines from the previous chapters. The easiest way to enter the program is to use the screen editor from Chapter 3 (which creates a T, or text, file) and then EXEC the file into memory, creating an A, or Applesoft, file to be saved to disk—and be sure to use a new name!

The output to the printer portion of the program may require special commands to be passed depending on the type of printer used. One of the most popular makes, the Epson, requires the code CONTROL I 80 N to be sent to the printer to initialize it before printing more than 40 columns (as many other printers require). The “80” can be any number between 1 and 255 or the limit of the printer, whichever is smaller. The most common value used is 80. Many special features such as different character fonts and bold printing are possible. See your printer manual for the special codes it requires.

Programmer Features

With our program we want the programmer to be able to create and modify a report layout without having to modify the BASIC program used to create the report. The text editor from Chapter 3 can be used to create and edit the report. In the text file describing the report the number of the array element to be printed is placed between < and > symbols. For example, <23> means “print array element 23 at this location.” Also, the & symbol at the end of the line is used to indicate that two or more lines are to be concatenated before printing.

There are two possible ways of inserting information into an existing line of text: expand the line to fit new data or truncate the data to fit existing space. If the information to be inserted into the line is too long or too short to fill the space between the < and > symbols, the line is adjusted so that the data fits exactly the space described.

A simple code is used to tell the report generator which method is desired. If a space is included between the symbols, then a short line will be filled with spaces and not expanded. For example, suppose we wish to insert “Marty and Alan” into these lines:
<1> are two great guys who write strange books.
<1 > are two great guys who write strange books.
<1 > 100 200 3000 100

These lines become

Marty and Alan are two great guys who write strange books.
Marty are two great guys who write strange books.
Marty and Alan 100 200 3000 100

The first example contains no spaces between the < and > symbols. Therefore the line is expanded to accept the entire insert without extra spaces.

The second example has the two < and > symbols, the array reference number (one digit or space), and two spaces, totaling five spaces for the mask. Therefore, the report generator only accepts the first five characters of the insert.

The third example has a mask size in excess of the size of the insert. Thus the report generator prints the entire insert along with the remaining spaces to complete the mask.

As these examples show, if you want the entire insert and aren’t sure how long it is going to be, either leave no spaces or far too many.

**PROGRAM FEATURES**

The flowchart for the report generator is shown in Fig. 7.1. After the text file describing the report page is opened and the length of the file is read, the file is processed one line at a time. Each line is processed one character at a time, looking for a < symbol. If the character is not a <, then it is printed. Once a < is encountered, the array element number is determined and the decision is made about whether or not the line is to be expanded. Then the data is printed.
We have included a couple of other variations in the program, which is listed at the end of the chapter. First, we have included lines that cause a single page to be selected from a group of pages (lines 30025 and 30065). If you have only one page, you will not need this option.

The second option allows you to print the report either to the disk or directly to a printer by setting DEVICE%. If DEVICE% equals 0, you will
print to the disk; otherwise, the report will be sent to a printer port. Why would you want to send the report to the disk? Convenience is the main reason. A report can be created and stored on the disk much faster than the average printer can print. Therefore, the operator can use the computer again sooner, and the report can be printed later, perhaps at a more convenient time. If a large number of reports are being created and user input is required in between each report, a lot of time can be wasted waiting for the printer to finish.

**Explanation of Variables**

The report subroutine requires two arrays to be dimensioned, PAGE$( ) and LINE$( ). PAGE$( ) contains the names of the report pages to be printed and should be dimensioned to at least the maximum number of report definition pages required. LINE$( ) contains the actual data to be printed. It must be dimensioned to at least the maximum number of data points used.

**Report Generator Test Program**

The report generator can be tested with the following routines:

```basic
100 REM TEST ROUTINE FOR DATA PRINT
110 REM
120 DEVICE% = 1 : REM FOR PRINTER IN SLOT #1
130 DIM LINE$(10) : REM NAME OF REPORT
140 DIM PAGE$(10) : REM YOU PROVIDE THIS PAGE
150 PAGE$(1) = "CHAP7A RPT" : REM TRY THE FIRST PAGE
160 PAGE$(2) = "CHAP7B RPT" : REM PRINT A PAGE
170 PAGE% = 1
180 GOSUB 30000
190 PRINT
200 PRINT "DONE"; CHR$(7) : REM RING THE BELL
210 PRINT
220 END
230 REM
240 REM
250 REM
```

172
**FIG. 7.2** Sample report

**MASK**

<table>
<thead>
<tr>
<th>Field</th>
<th>Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>&lt;1&gt;</td>
</tr>
<tr>
<td>STREET</td>
<td>&lt;2&gt;</td>
</tr>
<tr>
<td>ZIP CODE</td>
<td>&lt;3&gt;</td>
</tr>
<tr>
<td>AMOUNT</td>
<td>&lt;4&gt;</td>
</tr>
</tbody>
</table>

<1> LIVE AT <2> AND OWES & US $ <4>

**OUTPUT**

<table>
<thead>
<tr>
<th>Field</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>JOHN JONES</td>
</tr>
<tr>
<td>STREET</td>
<td>1234 FIRST STREET</td>
</tr>
<tr>
<td>ZIP CODE</td>
<td>1000</td>
</tr>
<tr>
<td>AMOUNT</td>
<td>3000</td>
</tr>
</tbody>
</table>

JOHN JONES LIVES AT 1234 FIRST STREET AND OWES US $ 3000

35000 REM DATA FOR TEST (CHAP7A RPT)
35010 REM
35020 LINE$(1) = "JOHN JONES"
35030 LINE$(2) = "1234 FIRST STREET"
35040 X = 1000
35050 LINE$(3) = STR$(X)
35060 LINE$(4) = STR$(X * 3)
35070 RETURN
35080 REM ********************
35090 REM
35100 REM
BASIC BUSINESS SUBROUTINES FOR THE APPLE II AND IIe

The first routine is the calling or "main" program. Lines 150 and 160 name two report pages assigned to the array PAGE$( ). The second routine sets up the LINE$( ) array for printing. In this program the values of the array are assigned directly. In your programs you will usually use variables.

The first report, CHAP7A RPT, is shown in Fig. 7.2. The second report and the data assignment subroutine (line 35000) are left as exercises for you to do.

Since there is only one report page, PAGE% on line 170 is set equal to 1. If you add a second report, this line can be replaced with:

170 INPUT "ENTER PAGE NUMBER: " PAGE%

COMPLETE REPORT GENERATOR PROGRAM

Here is the listing for the report generator program:

100 REM TEST ROUTINE FOR DATA PRINT
110 REM
120 DEVICE% = 1 : REM FOR PRINTER IN SLOT #1
130 DIM LINE$(10)
140 DIM PAGE$(10)
150 PAGE$(1) = "CHAP7A RPT" : REM NAME OF REPORT
160 PAGE$(2) = "CHAP7B RPT" : REM YOU PROVIDE THIS PAGE
170 PAGE% = 1 : REM TRY THE FIRST PAGE
180 GOSUB 30000 : REM PRINT A PAGE
190 PRINT
200 PRINT "DONE"; CHR$ (7) : REM RING THE BELL
210 PRINT
220 END
230 REM
240 REM
250 REM

30000 REM DATA PRINT SUBROUTINE
30005 REM
30010 REM THIS FILLS IN A DATA PAGE AND SENDS IT TO THE PRINTER OR DISK
CHAPTER 7  REPORT GENERATION

30015  REM
30020  REM BE SURE TO ADD SPECIAL PRINTER COMMANDS
30025  SCREEN$ = PAGE$(PAGE%) : REM THE PAGE TO FILL
30030  HOME  : REM INFORM THE USER OF PRINTING
30035  VTAB 10
30040  POKE 36, 10  : REM HTAB
30045  PRINT "PROCESSING " ; SCREEN$
30050  VTAB 12
30055  POKE 36, 15  : REM HTAB
30060  PRINT "PLEASE WAIT"
30065  ON PAGE% GOSUB 35000, 35500: REM FILL LINE$( ) WITH VALUES
30070  DSK$ = CHR$ (4) : REM DISK COMMAND CODE
30075  PRINT DSK$; "OPEN "; SCREEN$; " REPORT": REM OPEN THE PAGE FOR INPUT
30080  IF DEVICE% <> 0 THEN PRINT DSK$; "PR "; DEVICE%; GOTO 30115: REM SEND DIRECTLY TO THE PRINTER
30085
30090  REM PRINT IT TO A DISK FILE
30095
30100  PRINT DSK$; "OPEN "; SCREEN$; " REPORT"
30105  PRINT DSK$; "DELETE "; SCREEN$; " REPORT": REM ERASE ANY OLD FILE
30110  PRINT DSK$; "OPEN "; SCREEN$; " REPORT": REM OPEN A CLEAN NEW FILE
30115  PRINT DSK$; "READ "; SCREEN$ : REM GET THE NUMBER OF LINES
30120  INPUT ITEMS%
30125  REM
30130  REM
30135  ITEMS% = ITEMS% - 1  : REM DECREMENT COUNTER
30140  IF ITEMS% < 0 THEN GOTO 30475: REM ALL DONE SO EXIT
30145  PRINT DSK$; "READ "; SCREEN$ : REM READ THE DATA PAGE
30150  INPUT ENTRY$: REM GET A LINE FROM THE DATA PAGE
30155  IF A$ <> "&" THEN C$ = "": REM CLEAR THE OUTPUT LINE
30160  IF A$ = "&" THEN GOSUB 30240: REM STRIP LEADING SPACES
30165  REM
30170  REM PROCESS ENTRY$ ONE CHAR AT A TIME LOOKING FOR INSERTS
30175  REM
30180  FOR X = 1 TO LEN (ENTRY$): REM STEP DOWN THE LINE
30185  A$ = MID$ (ENTRY$, X, 1) : REM GET ONE CHARACTER
30190  IF A$ = "<" THEN GOSUB 30275: GOTO 30205: REM DO AN INSERTION
30195  IF A$ = "&" THEN X = LEN (ENTRY$): GOTO 30205: REM CONTINUATION SYMBOL
30200  C$ = C$ + A$  : REM BUILD THE OUTPUT STRING
30205  NEXT X
30210 IF DEVICE% = 0 THEN PRINT DSK$; "WRITE "; SCREEN$; " REPORT": REM
SEND OUTPUT TO DISK
30215 IF A$ <> "&" THEN PRINT C$: REM OUTPUT THE STRING
30220 GOTO 30135
30225 REM *******************************
30230 REM
30235 REM
30240 REM REMOVE THE LEADING SPACES
30245 REM
30250 IF MID$ (ENTRY$, 1, 1) = " " THEN ENTRY$ = MID$ (ENTRY$, 2):
   GOTO 30250: REM FOUND ONE SO REMOVE IT
30255 RETURN
30260 REM *******************************
30265 REM
30270 REM
30275 REM INSERT A FIELD INTO THE LINE
30280 REM
30285 REM TWO MODES:
30290 REM 1) INSERT AND CONCATENATE
30295 REM 2) INSERT BUT DO NOT CONCATENATE
30300 REM MODE 1 IS USED WHEN NO SPACES EXIST BETWEEN <> SYMBOLS
30305 REM MODE 2 IS USED IF A SPACE EXISTS BETWEEN <> SYMBOLS
30310 REM
30315 REM
30320 REM
30325 REM
30330 REM FIRST REMOVE THE <> SYMBOLS
30335 REM
30340 Y = 0 : REM DEFAULT TO MODE 1
30345 X = X + 1
30350 IF X > LEN (ENTRY$) THEN A$ = "": RETURN : REM
   ONLY ONE SYMBOL SO ERROR
30355 B$ = MID$ (ENTRY$, X, 1) : REM GET ONE CHARACTER
30360 A$ = A$ + B$
30365 IF B$ < > "">" THEN GOTO 30345: REM NOT DONE SO GET ANOTHER CHARACTER
30370 REM LOOK FOR A SPACE TO SET MODE
30375 FOR Z = 1 TO LEN (A$)
30380 IF A$ = " " THEN Y = Z: Z = LEN (A$)
30385 NEXT Z
30390 IF Y = 0 THEN Z = VAL (MID$ (A$, 2, LEN (A$) - 1))
30395 IF Y < > 0 THEN Z = VAL (MID$ (A$, 2, Y - 1)): REM GET THE FIELD NUMBER
30400  \texttt{C$ = C$ + LINE$(Z)$} : \texttt{REM ADD THE FIELD}
30405  IF \texttt{Y = 0} THEN \texttt{A$ = ": RETURN} : \texttt{REM MODE 1 SO ALL DONE}
30410  REM
30415  REM MODE 2 SELECTED SO WE MUST FILL WITH SPACES
30420  REM
30425  IF LEN (\texttt{LINE$(Z)$}) = LEN (\texttt{A$}) \texttt{THEN RETURN} : \texttt{REM NOTHING TO CLEAR}
30430  \texttt{Y = LEN (A$) - LEN (LINE$(Z)$)}
30435  FOR \texttt{Z = 1 TO Y}
30440  \texttt{C$ = C$ + " "} : \texttt{REM ADD THE SPACES}
30445  NEXT \texttt{Z}
30450  \texttt{A$ = "}
30455  RETURN : \texttt{REM ALL DONE}
30460  REM ******************** *************************************************************************
30465  REM
30470  REM
30475  REM CLOSE EVERYTHING AND RETURN
30480  REM
30485  \texttt{PRINT DSK$; "CLOSE "; SCREEN$}
30490  IF \texttt{DEVICE$} = 0 \texttt{THEN PRINT DSK$; "CLOSE "; SCREEN$; " REPORT"}
30495  IF \texttt{DEVICE$} < > 0 \texttt{THEN PRINT DSK$; "PR#O": REM RETURN TO SCREEN OUTPUT}
30500  \texttt{HOME} : \texttt{REM CLEAR THE SCREEN}
30505  RETURN
30510  REM ******************************************************************************
30515  REM
30520  REM

35000  REM DATA FOR TEST (CHAP7A RPT)
35010  REM
35020  \texttt{LINE$(1)$ = "JOHN JONES"}
35030  \texttt{LINE$(2)$ = "1234 FIRST STREET"}
35040  \texttt{X = 1000}
35050  \texttt{LINE$(3)$ = STR$ (X)}
35060  \texttt{LINE$(4)$ = STR$ (X * 3)$}
35070  RETURN
35080  REM ****************************
35090  REM
35100  REM
INTRODUCTION

Thus far, several flexible subroutines have been developed. This chapter will illustrate how easy it is to use these subroutines to create larger programs. The sample program developed in this chapter will maintain a personal appointment calendar. This program is a moderate-sized program, but it is easily and quickly constructed by using the building blocks developed in the previous chapters. We will use the same methods to create the personal calendar program that we used in the previous chapters.

In addition to using the routines developed in the previous chapters, the personal calendar program introduces a few new routines and concepts. A subroutine is used to calculate the number of days a given date is
from the beginning of the year. Another subroutine uses a random-access file to store the appointment information.

In the following subsections we design the input screens and the reports. Then in succeeding sections we create the various program modules necessary to connect everything together.

**Building the Program**

The calendar program builds on the routines that have been developed in previous chapters. The routines of this chapter can be combined with those of Chapters 2, 4, 5, 6, and 7 by the EXEC technique described in Chapter 3.

Here is the method to use to build the calendar program:

1. Start with a fresh, initialized disk for the program of this chapter.
2. Transfer copies of the programs developed in Chapters 2, 4, 5, 6, and 7 to your new disk (A, or Applesoft, copies), using DOS. Merge them and save the results under the name CAL.A.
3. Using the screen editor from Chapter 3, type in the program lines presented in this chapter and save them (these lines will be T, or text, files) under the name CAL.T.
4. Using DOS, load CAL.A into memory and EXEC the file of this chapter (CAL.T), or as much of it as you have entered so far, into memory. Now the two programs are merged and can be saved again on disk as Applesoft files, ready to run the next time you load them.
5. Repeat steps 3 and 4 each time you enter more of the program from this chapter.

**Design**

The personal calendar program displays, accepts, and prints hourly appointment information. The features of this program can best be seen by looking at the various input screens and reports.

The menu, shown in Fig. 8.1, lists the available options: (1) review, (2) print, (3) help, and (4) exit. If options 1 and 2 are selected, then the pro-
**Fig. 8.1** Appointment calendar menu

```
^APPOINTMENT CALENDAR MENU^

1. REVIEW DAY'S APPOINTMENTS
2. PRINT ONE DAY'S APPOINTMENTS
3. PRINT GROUP OF DAYS
4. ^HELP^
^0. EXIT TO BASIC^
```

The program will ask for the date to be reviewed or printed, as shown in Fig. 8.2. If option 3 is selected, then the range of dates of interest is entered, as illustrated in Fig. 8.3.

The appointment data entry screen is shown in Fig. 8.4. It is designed to accept hourly appointments between 8 A.M. and 5 P.M. The report generator template, shown in Fig. 8.5, is very similar to the data entry screen.

Since any date may be edited, random-access files are used to store the data on the disk.

**Fig. 8.2** Single date entry screen

```
^APPOINTMENT DATE^

1. MONTH OF APPOINTMENT       <##>
2. DAY OF MONTH                 <##>
```
**FIG. 8.3** Data entry screen for group of dates

```
^APPOINTMENT CALENDAR^

1. BEGINNING MONTH <##>
2. DAY OF BEGINNING MONTH <##>
3. LAST MONTH <##>
4. DAY OF LAST MONTH <##>
```

**FIG. 8.4** Data entry screen for appointments

```
^APPOINTMENT CALENDAR^

1. 8 AM <AAAAAAAAAAAAAAAAAAAAAAAAAAAAA>
2. 9 AM <AAAAAAAAAAAAAAAAAAAAAAAAAAAAA>
3. 10 AM <AAAAAAAAAAAAAAAAAAAAAAAAAAAAA>
4. 11 AM <AAAAAAAAAAAAAAAAAAAAAAAAAAAAA>
5. 12 PM <AAAAAAAAAAAAAAAAAAAAAAAAAAAAA>
6. 1 PM <AAAAAAAAAAAAAAAAAAAAAAAAAAAAA>
7. 2 PM <AAAAAAAAAAAAAAAAAAAAAAAAAAAAA>
8. 3 PM <AAAAAAAAAAAAAAAAAAAAAAAAAAAAA>
9. 4 PM <AAAAAAAAAAAAAAAAAAAAAAAAAAAAA>
10. 5 PM <AAAAAAAAAAAAAAAAAAAAAAAAAAAAA>
```
CHAPTER 8  PERSONAL CALENDAR: A SAMPLE PROGRAM

FIG. 8.5 Appointment report generator template

^APPOINTMENT CALENDAR^

8 AM <1>
9 AM <2>
10 AM <3>
11 AM <4>
12 PM <5>
1 PM <6>
2 PM <7>
3 PM <8>
4 PM <9>
5 PM <10>

BASIC CALENDAR PROGRAM

The calendar program, flowcharted in Fig. 8.6, uses the following subroutines:

- Menu,
- Accept single date,
- Load data from disk,
- Edit/review appointment data,
- Write data to disk,
- Print data,
- User help text.
Most of these routines are based on subroutines developed in the previous chapters.

In the following subsections the personal calendar program will be created one menu option at a time, beginning with the menu itself. Before you begin editing the program, however, you should merge the subroutines from Chapters 2, 4, 5, 6, and 7.
Menu

The menu uses the menu system from Chapter 6 to display the options, accept an option, and branch to the requested routine. This program could be considered the master or main program because it calls the other routines.

100    REM
110    REM  CHAPTER 8 PERSONAL CALENDAR
120    REM
130    REM
140    REM
150    REM  THIS MAINTAINS AN ANNUAL PERSONAL CALENDAR. IT IS
160    REM  AN EXAMPLE OF HOW A LARGE PROGRAM CAN BE BUILT
170    REM  FROM THE SUBROUTINES DEVELOPED THROUGHOUT THE BOOK.
180    REM
190    REM
200    LAST% = 10    : REM  NUMBER OF FIELDS ON THE DATA ENTRY SCREEN
210    DIM LINE$(LAST%): REM  THESE ARE FOR DATA ENTRY ROUTINE
220    DIM SROW%(LAST%)
230    DIM SCOL%(LAST%)
240    DIM SMASK%(LAST%)
250    REM
260    REM  CALL THE MENU ROUTINE
270    REM
280    DSK$ = CHR$(4)    : REM  ^D
290    DEVICE% = 1    : REM  PRINTER SLOT NUMBER
300    PRINT DSK$;"OPEN CALENDAR,L280" : REM  OPEN THE CALENDAR FILE
310    PRINT
320    SCREEN$ = "CHAP8 MENU"    : REM  NAME OF MENU SCREEN
330    REM  THE MENU ROUTINE WILL BRANCH TO THE APPROPRIATE ROUTINE
340    REM  AND RETURN HERE OR IT WILL END
350    REM
360    GOSUB 22000    : REM  THE MENU ROUTINE
370    GOTO 320    : REM  TRY AGAIN
380    REM
390    REM
400    REM
In addition to using the above program, you must change some lines in the original menu subroutine. Line 22145 must be deleted. Lines 22150 and 22165 must be changed to

```
22150 IF X = 0 THEN HOME : END : REM RETURN TO BASIC
22165 RETURN
```

Also, instead of chaining to another program, you change line 22160 to a branch,

```
22160 ON X GOSUB 1000,2000,3000,4000
```

where the subroutines perform as follows:

1. **1000** edit/reviews a day's appointments,
2. **2000** prints a single day's appointments,
3. **3000** prints a group of dates,
4. **4000** provides user help text.

The help text is shown later in this chapter (in the “Option 4: Help” subsection) and is saved on the disk as CHAP8 MENU HELP.

**TEST POINT**

When you execute the program, it should display the menu of Fig. 8.1. The only option, (0), should be working. Verify that it does clear the screen, and stop the program.

**Option 1: Edit/Review a Day's Appointments**

Option 1 combines two features: editing and reviewing the data. In some programs you may want to separate these features. For example, you may have several people using a program and you may not want all of them to be able to edit the data. If you are not concerned about keeping...
people from editing, then we feel it is more convenient to keep both fea­
tures combined. Since this program is a personal calendar program, any­
one who has access to the calendar can change it.

The program for option 1, shown below, is quite simple:

```plaintext
1000 REM OPTION 1 EDIT/REVIEW ONE DATE
1010 REM
1020 GOSUB 7000 : REM ACCEPT THE DATE
1030 GOSUB 8000 : REM LOAD FROM DISK
1040 SCREEN$(1) = "CHAP8 ENTRY" : REM DATA SCREEN NAME
1050 PAGE% = 1 : REM USE THE FIRST PAGE
1060 GOSUB 20000 : REM EDIT THE DATA
1070 GOSUB 9000 : REM WRITE TO DISK
1080 RETURN
1090 REM
1100 REM ********************************************
1110 REM
```

Subroutines 7000, 8000, and 9000 are described in succeeding sections.
Subroutine 20000 is the data entry system from Chapter 5.

Additionally, a data entry screen, shown in Fig. 8.4, and the help
text, Fig. 8.7, must be entered.

### Accepting a Date

Two functions are performed by subroutine 7000: It accepts a month and
day from the user, and then it calculates the number of days this date is
from the beginning of the year.

The number of days between any two dates can be found by calculat­
ing a value for both dates, using the following formulas, and then sub­
tracting these values (subroutine 7500). The formula for dates in January
and February is

\[
\text{value} = 365 \text{year} + \text{day} + 31 \text{month} - 1 + \text{INT}[\text{year} - 1/4]
\]

\[
- \text{INT}[.75 \{\text{INT}[\text{year} - 1/100] + 1\}]
\]
For dates between March and December the formula is

\[
\text{value} = 365(\text{year}) + \text{day} + 31(\text{month} - 1) + \text{INT}[\{(\text{year} - 1)/4\}
\]
\[
- \text{INT}(0.75\{\text{INT}[(\text{year} - 1)/100] + 1\}) - \text{INT}[0.4(\text{month}) + 2.3].
\]

The terms in the above equations have the following meanings:

- Year is the calendar year (e.g., 1983).
- Month is the number of the month (e.g., March is 3).
- Day is the number of the day's date (e.g., 31).
- \text{INT} is the \text{BASIC} integer command.

\text{INT} The \text{INT}(N) command returns the integer (whole number) part of the argument \(N\).

\textbf{EXAMPLE}

\begin{verbatim}
100  X = 100.25
110  PRINT "INTEGER " ; INT(X)
\end{verbatim}

When this program is executed, the computer will print

\texttt{INTEGER 100}

The program listing for the subroutine that accepts a date is as follows:

```
7000  REM
7010  REM ACCEPT A DATE
7020  REM
7030  REM
7040  PAGE% = 2 : REM ASSIGN TO DATA PAGE 2
7050  SCREEN$(PAGE%) = "CHAP8 DATE" : REM DATA SCREEN NAME
7060  GOSUB 20000 : REM DATA ENTRY ROUTINE
7070  GOSUB 7500 : REM CALCULATE DAY NUMBER
7080  RETURN
7090  REM
7100  REM
7110  REM
7500  REM  CALCULATE DAY NUMBER
```
CHAPTER 8 PERSONAL CALENDAR: A SAMPLE PROGRAM

7510 YY = 1983 : REM CURRENT YEAR
7520 IF (MM = 0) OR (DD = 0) THEN DAY = 1:RETURN: REM ERROR SO FORCE 1
7530 IF (MM > 12) OR (DD > 31) THEN DAY = 1:RETURN: REM BAD DATE ENTERED
7540 JAN1 = 365 * YY + INT((YY - 1) / 4) - INT(.75 * (INT((YY - 1) / 100) + 1)) : REM VALUE FOR JAN 1, YY
7550 IF MM > 3 THEN GOTO 7590 : REM USE SECOND EQUATION
7560 DAY = 365 * YY + DD + 31 * (MM - 1) + INT((YY - 1) / 4) - INT(.75 * (INT((YY - 1) / 100) + 1)) - JAN1: REM DAYS FROM JAN 1
7570 RETURN
7580 REM
7590 REM NOW FOR MARCH THRU DECEMBER
7600 REM
7610 DAY = 365 * YY + DD + 31 * (MM - 1) + INT(YY / 4) - INT(.75 * (INT(YY / 100) + 1)) - INT(.4 * MM + 2.3) - JAN1: REM FOR MARCH THRU DECEMBER
7620 RETURN
7630 REM
7640 REM **************
7650 REM

The data entry routine of Chapter 5 (line 7060, GOSUB 20000) calls routine 26000 (shown below) to fill the LINE$( ) array with the original or default values and routine 26500 (below) to move LINE$( ) values into the regular variable names. There are no default values in this routine, so routine 26000 merely clears the LINE$( ) elements used. We could have put these routines closer (with lower line numbers) to the front of the program; but since the original program used these addresses, we decided to keep them as they were.

Subroutine 26000 is as follows:

26000 REM SINGLE DATE PAGE
26010 REM
26020 LINE$(1) = ""
26030 LINE$(2) = ""
26040 RETURN
26050 REM
26060 REM **************
26070 REM
26500 REM RESTORE SINGLE DATE DATA
26510 REM

189
TEST POINT

Execute the program and verify that option 1 allows you to enter a month and day. While in the data entry screen program, test the help feature and the various editing commands available.

Random-Access Files

To edit any date's data, we must use random-access disk files. A random-access file, as the name suggests, is a data file that can be read from or written to in any order, i.e., randomly. Up to this point all the files used were sequential or text files. A sequential file is read in exactly the same order it is written in. (Applesoft has a command called POSITION that allows you to randomly access sequential files. It is difficult to use, however, and we recommend that if you need to access files randomly, then you use random-access files.)

Random-access files require a little more programming and bookkeeping than sequential files require, because the computer stores random files slightly differently than it stores sequential files. The following analogy will help explain the difference. A sequential file can be compared with a stack of papers. Such a stack is very compact, but it is difficult to insert or remove papers from the stack unless they happen to be next to each other in the stack. In contrast, a random-access file can be thought of as a stack of identical shoe boxes numbered sequentially, 1,2,3,... Each box contains pieces of paper. Some of the boxes are full, some partially full, and some empty. When the contents of a box are to be read or changed, you tell your assistant to go get box number XX. The shoe boxes take up more room than the stack of papers, but with the boxes you can generally get to a particular set of papers faster.
In the computer, sequential files are written to the disk with the text packed tightly together (the stacks of paper). Random files are written in pieces called records (the shoe boxes); all the records in a file use exactly the same amount of space. When a random file is opened for processing, the computer is told how large the records are. The computer must also be told which record is to be accessed. The length of the data stored in a record must be less than or equal to the record size. Since, in general, the data is smaller than the record size, some space is wasted. However, since the computer can easily calculate exactly where the beginning of each record is, it can quickly read or write the data.

In the personal calendar program we wish to store ten lines of data with a maximum length of 25 characters each. In determining the record size, we must count the carriage return at the end of each line. Also, since we write text between double quotation marks, an additional two bytes are needed for each line. The record size is therefore

\[
10(25 + 1 + 2) = 280 \text{ bytes.}
\]

The records are consecutively numbered, beginning with 1. In our program January 1 is record 1, January 2 is record 2, etc.

Random-access files should be initialized before they are used. A file is initialized by writing blank or null lines into every record. The following program initializes a one-week calendar:

```
100 REM CHAPTER 8 CALENDAR INITIALIZATION ROUTINE
110 REM
120 REM CREATES AND CLEARS CALENDAR FILE
130 REM
140 HOME
150 PRINT "CREATING CALENDAR FILE"
160 NULL$ = CHR$(34) + CHR$(34): REM TWO DOUBLE QUOTES MAKES NULL STRING
170 DAYS = 7 : REM EXAMPLE FOR ONE WEEK ONLY
180 DSK$ = CHR$(4) : REM ^D
190 PRINT DSK$; "OPEN CALENDAR,L280": REM OPEN BYTE RECORD SIZE = 280
200 FOR X = 1 TO DAYS
210 PRINT DSK$; "WRITE CALENDAR,R";X
220 FOR Y = 1 TO 10
```
BASIC BUSINESS SUBROUTINES FOR THE APPLE II AND IIe

230 PRINT NULL$ :REM WRITE OUT NULL STRING
240 NEXT Y
250 NEXT X
260 PRINT DSK$; "CLOSE CALENDAR"
270 END

Adjust the number of days in the routine above to the size of calendar you want. But note that a full, one-year calendar will fill most of a blank disk. So do not try to put a calendar on a disk already containing other files.

The following subroutines will read and write the random files. Note that when a line is written, it is surrounded by two quotation marks, CHR$(34). This feature is necessary if commas are to be used in the text. In Applesoft BASIC, fields can be on the same line if terminated with a comma. Therefore commas can be used in the text only if the text is written between quotation marks.

The read and write subroutines are as follows:

8000 REM LOAD A DATE'S DATA
8010 REM
8020 PRINT : REM JUST TO MAKE SURE DISK COMMANDS WORK
8030 PRINT DSK$; "READ CALENDAR,R"; DAY
8040 FOR X = 1 TO 10
8050 INPUT LINE$(X)
8060 NEXT
8070 PRINT DSK$ : REM TURN DISK 10 OFF
8080 RETURN
8090 REM
8100 REM ****************
8110 REM
9000 REM WRITE DATE'S DATA TO DISK
9010 REM
9020 PRINT : REM MAKE SURE DISK COMMANDS WORK
9030 PRINT DSK$; "WRITE CALENDAR,R"; DAY: REM SET RECORD NUMBER
9040 FOR X = 1 TO 10
9050 PRINT CHR$(34);LINE$(X); CHR$(34): REM SAVE WITH QUOTE MARKS
9060 NEXT
9070 PRINT DSK$: REM TURN DISK 10 OFF
Calendar Help Screens

The following illustrations show the four help screens that will be displayed. Please enter them, using the editor, as four separate files with the names specified.

1. File 'CHAP8 MENU HELP'

^APPPOINTMENT MENU HELP^

THIS PROGRAM MAINTAINS A PERSONAL APPOINTMENT CALENDAR. YOU CAN EDIT, REVIEW OR PRINT ANY DAY'S APPOINTMENTS.

^OPTION 1^ ALLOWS YOU TO EDIT OR REVIEW ANY SINGLE DAY'S SCHEDULE.

^OPTION 2^ PRINTS A SINGLE DAY'S APPOINTMENTS.

^OPTION 3^ PRINTS SEVERAL DAYS APPOINTMENTS.

^OPTION 4^ DISPLAYS THIS HELP PAGE.

^OPTION 0^ STOPS THE PROGRAM AND RETURNS TO BASIC.

^SELECT OPTION, ENTER ITS NUMBER AND STRIKE RETURN^
2. File 'CHAP8 DATE HELP'

^APPOINTMENT DATE HELP^

PLEASE ENTER THE MONTH AND DAY YOU
WISH TO EDIT OR REVIEW. IF YOU ENTER
AN IMPROPER DATE THE COMPUTER WILL
ASSUME THAT YOU WISH TO EDIT JANUARY 1.

3. File 'CHAP8 2DAYS HELP'

^PRINTING GROUP OF DATES^

THIS SECTION PRINTS A GROUP OF DAYS
APPOINTMENTS. ENTER THE MONTH AND DAY
YOU WISH TO BEGIN PRINTING AND THE
LAST MONTH AND DAY YOU WISH TO PRINT.

4. File 'CHAP8 ENTRY HELP'

^PERSONAL CALENDAR^

THIS IS A PERSONAL CALENDAR PROGRAM.
YOU MAY ENTER TEXT ON ANY LINE. TO
EXIT STEP THROUGH THE LINES, USING
THE RETURN KEY, UNTIL THE CHANGES LINE
AND ENTER 0 (CR).

**Calendar Data Entry Subroutine**

The data entry system needs the name of the entry screen and two sub-
routines to move the data between the regular variable names and
LINES$(    ). Since we are not manipulating the data other than with the
data entry system, we are taking a shortcut here and not assigning regular variable names. Therefore the two subroutines become RETURNs, as follows:

```basic
25000 REM SETUP FOR PAGE 1 25010 RETURN : REM NONE NEEDED
25020 REM 25030 REM ***************
25040 REM 25050 REM RESTORE FOR PAGE 1 25060 RETURN : REM NONE NEEDED
25070 REM 25080 REM ***************
25090 REM 25100 REM
```

**TEST POINT**

In the menu, select option 1. In the date scene, enter a date in January and verify that appointment information can be entered and edited. After the appointments have been edited, the program returns to the menu. Repeat this test now to verify that the data has been properly written to the disk and read from the disk. Finally, repeat the test again, and enter ^Q and verify that the help routine is working.

**Option 2: Print a Day's Appointments**

Option 2 uses some of the previous routines plus the report generator of Chapter 7. The report generator needs a report template name and a routine to move the data between the regular variable names and the LINE$( ) array. As in the data entry screen, since we are not manipulating the data, we can leave it in the LINE$( ) array. We take another shortcut here and use the data entry routine 25000. Hence the report generator program is quite small:

```basic
2000 REM PRINT A SINGLE DAY 2010 REM
2020 GOSUB 7000 : REM ACCEPT DATE
2030 GOSUB 8000 : REM LOAD FROM DISK
```
Line 30065 of the report generator becomes

```
30065 ON PAGE% GOSUB 25000 : REM FILL THE LINE$( ) ARRAY
```

And line 30487 below is added to force the printer to eject the page (form feed) after each day:

```
30487 PRINT CHR$(12) : REM PRINT A FORM FEED
```

Your printer's slot number is entered in the following line 290. Verify that the line is correct.

```
290  DEVICE% = 1 : REM PRINTER SLOT NUMBER
```

**TEST POINT**

Run the program and select option 2. If everything goes according to plan, the report should appear on your printer, and the program should return to the menu.

**Option 3: Print a Group of Dates**

Option 3 is only a little more complicated than option 2. In option 3 subroutine 10000 (shown below) is called to accept two dates instead of one, and then the program loops until all the days have been printed.

Here is option 3:

```
3000 REM PRINT SEVERAL DAYS APPOINTMENTS
3010 REM
```
CHAPTER 8 PERSONAL CALENDAR: A SAMPLE PROGRAM

3020 GOSUB 10000 : REM GET TWO DATES
3030 PAGE% = 1 : REM FOR REPORT GENERATOR
3040 PAGE$(1) = "CHAP8 RPT"
3050 FOR DAY = D1 TO D2
3060 PRINT
3070 PRINT DSK$; "READ CALENDAR,R"; DAY : REM READ IN THE DATA
3080 GOSUB 8000 : REM LOAD THE DATA
3090 GOSUB 30000 : REM PRINT A REPORT
3100 NEXT DAY
3110 RETURN
3120 REM
3130 REM ***************
3140 REM

10000 REM ACCEPT TWO DATES
10010 REM
10020 PAGE% = 3 : REM FOR DATA ENTRY ROUTINE
10030 SCREEN$(PAGE%) = "CHAP8 2DAYS"
10040 GOSUB 20000 : REM DATA ENTRY ROUTINE
10050 MM = VAL (LINE$(1))
10060 DD = VAL (LINE$(2))
10070 GOSUB 7500 : REM CALCULATE DAY NUMBER
10080 D1 = DAY
10090 MM = VAL (LINE$(3))
10100 DD = VAL (LINE$(4))
10110 GOSUB 7500 : REM SECOND DAY
10120 D2 = DAY
10130 RETURN
10140 REM
10150 REM ***************
10160 REM

TEST POINT

Execute and select option 3. Enter two dates about three days apart and verify that three reports are printed. Repeat and test the help system. You may wish to enter several days’ appointments to make sure that the correct data is printed.
Option 4: Help

Option 4 uses the help subsystem and is a short routine. Enter the help text and the following routine:

4000 REM
4010 REM MENU HELP OPTION
4020 REM
4030 HELP$ = "CHAP8 MENU HELP" : REM HELP SCREEN NAME
4040 GOSUB 21000 : REM HELP SUBROUTINE
4050 RETURN : REM ALL DONE
4060 REM
4070 REM
4080 REM

TEST POINT

Select option 4 from the menu and the menu help text should appear.

Summary

You now have a working personal calendar program built from reusable subroutines. The number of new program lines required to create this program is small compared with its overall size. We hope this example has shown you the value of a library of subroutines.

USER INSTRUCTIONS

The calendar program maintains a daily personal calendar. You can edit, review, or print any day’s or group of days’ schedules. Help is available with every screen.
This menu is the first screen seen:

APPOINTMENT CALENDAR MENU

1. REVIEW DAY'S APPOINTMENTS
2. PRINT ONE DAY'S APPOINTMENTS
3. PRINT GROUP OF DAYS
4. HELP
0. EXIT TO BASIC

The following subsections contain the user instructions for the personal calendar system. Please add these instructions to your user's manual.

**Option 1: Edit/Review a Day's Appointments**

Option 1 allows you to edit or review any day's appointments. After the option is selected, you will be asked to enter a month and a day. The appointments for this day will be displayed and can be edited if desired.

**Option 2: Print a Day's Appointments**

Option 2 asks for the month and day to be printed. Make sure that the printer is on line.

**Option 3: Print a Group of Dates**

In option 3 you will be asked to enter the beginning month and day and the last month and day to be printed. These appointments will be printed one day to a page. Make sure that the printer is on line.
Option 4: Help

When you select option 4, a brief description of the various options is presented.

COMPLETE PERSONAL CALENDAR PROGRAM

Here is the listing for the complete personal calendar program:

```
100 REM
110 REM CHAPTER 8 PERSONAL CALENDAR
120 REM
130 REM
140 REM
150 REM THIS MAINTAINS AN ANNUAL PERSONAL CALENDAR. IT IS
160 REM AN EXAMPLE OF HOW A LARGE PROGRAM CAN BE BUILT
170 REM FROM THE SUBROUTINES DEVELOPED THROUGHOUT THE BOOK.
180 REM
190 REM
200 LAST% = 10 : REM NUMBER OF FIELDS ON THE DATA ENTRY SCREEN
210 DIM LINE$(LAST%) : REM THESE ARE FOR DATA ENTRY ROUTINE
220 DIM SROW%(LAST%)
230 DIM SCOL%(LAST%)
240 DIM SMASK%(LAST%)
250 REM
260 REM CALL THE MENU ROUTINE
270 REM
280 DSK$ = CHR$(4) : REM °D
290 DEVICE% = 1 : REM PRINTER SLOT NUMBER
300 PRINT DSK$;"OPEN CALENDAR,L280" : REM OPEN THE CALENDAR FILE
310 PRINT
320 SCREEN$ = "CHAP8 MENU" : REM NAME OF MENU SCREEN
330 REM THE MENU ROUTINE WILL BRANCH TO THE APPROPRIATE ROUTINE
340 REM AND RETURN HERE OR IT WILL END
350 REM
360 GOSUB 22000 : REM THE MENU ROUTINE
370 GOTO 320 : REM TRY AGAIN
```
CHAPTER 8  PERSONAL CALENDAR: A SAMPLE PROGRAM

380 REM
390 REM
400 REM
1000 REM  OPTION 1 EDIT/REVIEW ONE DATE
1010 REM
1020 GOSUB 7000  : REM  ACCEPT THE DATE
1030 GOSUB 8000  : REM  LOAD FROM DISK
1040 SCREEN$(1) = "CHAP8 ENTRY"  : REM  DATA SCREEN NAME
1050 PAGE% = 1  : REM  USE THE FIRST PAGE
1060 GOSUB 20000  : REM  EDIT THE DATA
1070 GOSUB 9000  : REM  WRITE TO DISK
1080 RETURN
1090 REM
1100 REM ********************
1110 REM
2000 REM  PRINT A SINGLE DAY
2010 REM
2020 GOSUB 7000  : REM  ACCEPT DATE
2030 GOSUB 8000  : REM  LOAD FROM DISK
2040 PAGE% = 1
2050 PAGE$(1) = "CHAP8 RPT"  : REM  NAME OF REPORT
2060 GOSUB 30000  : REM  REPORT GENERATOR
2070 RETURN
2080 REM
2090 REM ********************
2100 REM
3000 REM  PRINT SEVERAL DAYS APPOINTMENTS
3010 REM
3020 GOSUB 10000  : REM  GET TWO DATES
3030 PAGE% = 1  : REM  FOR REPORT GENERATOR
3040 PAGE$(1) = "CHAP8 RPT"
3050 FOR DAY = D1 TO D2
3060 PRINT
3070 PRINT DSK$; "READ CALENDAR,R"; DAY: REM  READ IN THE DATA
3080 GOSUB 8000  : REM  LOAD THE DATA
3090 GOSUB 30000  : REM  PRINT A REPORT
3100 NEXT DAY
3110 RETURN
3120 REM
3130 REM ********************
BASIC BUSINESS SUBROUTINES FOR THE APPLE II AND IIe

3140 REM
4000 REM
4010 REM MENU HELP OPTION
4020 REM
4030 HELP$ = "CHAPS MENU HELP" : REM HELP SCREEN NAME
4040 GOSUB 21000 : REM HELP SUBROUTINE
4050 RETURN : REM ALL DONE
4060 REM
4070 REM
4080 REM
7000 REM
7010 REM ACCEPT A DATE
7020 REM
7030 REM
7040 PAGE% = 2 : REM ASSIGN TO DATA PAGE 2
7050 SCREEN$(PAGE%) = "CHAPS DATE" : REM DATA SCREEN NAME
7060 GOSUB 20000 : REM DATA ENTRY ROUTINE
7070 GOSUB 7500 : REM CALCULATE DAY NUMBER
7080 RETURN
7090 REM
7100 REM
7110 REM
7500 REM CALCULATE DAY NUMBER
7510 YY = 1983 : REM CURRENT YEAR
7520 IF(MM = 0) OR (DD = 0) THEN DAY = 1:RETURN: REM ERROR SO FORCE 1
7530 IF (MM > 12) OR (DD > 31) THEN DAY = 1:RETURN: REM BAD DATE ENTERED
7540 JAN1 = 365 * YY + INT((YY - 1) / 4) - INT(.75 * (INT((YY - 1) / 100) + 1)) : REM VALUE FOR JAN 1, YY
7550 IF MM > 3 THEN GOTO 7590 : REM USE SECOND EQUATION
7560 DAY = 365 * YY + DD + 31 * (MM - 1) + INT((YY - 1) / 4) - INT(.75 * (INT((YY - 1) / 100) + 1)) - JAN1: REM DAYS FROM JAN1
7570 RETURN
7580 REM
7590 REM NOW FOR MARCH THRU DECEMBER
7600 REM
7610 DAY = 365 * YY + DD + 31 * (MM - 1) + INT(YY / 4) - INT(.75 * (INT(YY / 100) + 1)) - INT(.4 * MM + 2.3) - JAN1: REM FOR MARCH THRU DECEMBER
7620 RETURN
7630 REM
CHAPTER 8 PERSONAL CALENDAR: A SAMPLE PROGRAM

7640 REM ***************
7650 REM
8000 REM LOAD A DATE'S DATA
8010 REM
8020 PRINT : REM JUST TO MAKE SURE DISK COMMANDS WORK
8030 PRINT DSK$; "READ CALENDAR,R"; DAY
8040 FOR X = 1 TO 10
8050 INPUT LINE$(X)
8060 NEXT
8070 PRINT DSK$ : REM TURN DISK IO OFF
8080 RETURN
8090 REM
8100 REM ***************
8110 REM
9000 REM WRITE DATE'S DATA TO DISK
9010 REM
9020 PRINT : REM MAKE SURE DISK COMMANDS WORK
9030 PRINT DSK$; "WRITE CALENDAR,R"; DAY : REM SET RECORD NUMBER
9040 FOR X = 1 TO 10
9050 PRINT CHR$(34);LINE$(X);CHR$(34): REM SAVE WITH QUOTE MARKS
9060 NEXT
9070 PRINT DSK$ : REM TURN DISK IO OFF
9080 RETURN
9090 REM
9100 REM ***************
9110 REM
10000 REM ACCEPT TWO DATES
10010 REM
10020 PAGE% = 3 : REM FOR DATA ENTRY ROUTINE
10030 SCREEN$(PAGE%) = "CHAP8 2DAYS"
10040 GOSUB 20000 : REM DATA ENTRY ROUTINE
10050 MM = VAL (LINE$(1))
10060 DD = VAL (LINE$(2))
10070 GOSUB 7500 : REM CALCULATE DAY NUMBER
10080 D1 = DAY
10090 MM = VAL (LINE$(3))
10100 DD = VAL (LINE$(4))
10110 GOSUB 7500 : REM SECOND DAY
10120 D2 = DAY
10130 RETURN
20000 REM DATA ENTRY * SCREEN PROCESSOR
20005 REM
20010 REM
20015 REM
20020 REM DISPLAYS SCREEN, LOADS MASK DATA
20025 REM DISPLAYS DEFAULT VALUES
20030 REM EDITS AND SAVES VALUES
20035 REM
20040 REM VARIABLES USED:
20045 REM LINE$() HOLDS EDIT DATA
20050 REM ITEM% NUMBER TO EDIT
20055 REM PAGE% PAGE TO EDIT
20060 REM SROW%( ) FIELD ROW NUMBER
20065 REM SCOL%( ) FIELD COL NUMBER
20070 REM SMASK$() FIELD MASK$
20075 REM SCREEN$() NAME OF SCREEN
20080 REM LINE% CURRENT LINE BEING EDITED
20085 REM
20090 REM
20095 REM
20100 REM
20105 REM EDIT A DATA SCREEN
20110 REM
20115 REM SHOW THE DATA SCREEN
20120 REM
20125 ITEM% = 0 : REM CLEAR NUMBER OF ITEMS
20130 NOPAUSE% = 1 : REM INFORM HELP SCREEN NOT TO PAUSE
20135 COL% = 1 : REM RESET POSITION COUNTER
20140 HELP$ = SCREEN$(PAGE% ) : REM NAME OF SCREEN TO PROCESS
20145 GOSUB 21000 : REM SHOW SCREEN
20150 NOPAUSE% = 0 : REM RESET TO PAUSE
20155 ON PAGE% GOSUB 25000,26000,27000 : REM SHOW DEFAULT VALUES
20160 GOSUB 20295 : REM EDIT DATA
20165 GOSUB 20350 : REM SHOW DEFAULT VALUES
20170 ON PAGE% GOSUB 25500,26500,27500 : REM SAVE THE EDITED DATA
20175 IF HELP% > 0 THEN HELP$ = HELP$ + " HELP": GOSUB 21000 :
GOTO 20000 : REM SHOW HELP SCREEN AND START OVER
20180  RETURN
20185  REM
20190  REM  *******************
20195  REM
20200  REM
20205  REM  SET FIELD PARAMETERS
20210  REM
20215  ITEMS% = ITEMS% + 1  : REM  INC FIELD COUNTER
20220  SMASK$(ITEMS%) = ""  : REM  CLEAR IT
20225  SROW%(ITEMS%) = ROW% + 1  : REM  CURRENT ROW NUMBER
20230  SCOL%(ITEMS%) = COL%  : REM  CURRENT COLUMN NUMBER
20235  PRINT " " ;  : REM  PRINT SPACE
20240  Z = Z + 1
20245  ENTRY$ = MID$(A$,Z,1)  : REM  GET ONE CHARACTER
20250  IF ENTRY$ = ">" THEN GOTO 20270  : REM  ARE WE AT END OF MASK?
20255  SMASK$(ITEMS%) = SMASK$(ITEMS%) + ENTRY$  : REM  ADD TO FIELD MASK
20260  COL% = COL% + 1  : REM  INC COL CNT
20265  GOTO 20235  : REM  GET NEXT CHAR
20270  ENTRY$ = " "  : REM  CLEAR ENTRY$
20275  RETURN
20280  REM
20285  REM  *******************
20290  REM
20295  REM  SHOW DEFAULT VALUES
20300  REM
20305  FOR X = 1 TO ITEMS%
20310  VTAB SROW%(X)  : REM  LINE
20315  POKE 36, SCOL%(X)  : REM  COLUMN - HTAB
20320  PRINT LINE$(X);  : REM  DATA
20325  NEXT X
20330  RETURN
20335  REM
20340  REM  *******************
20345  REM
20350  REM  EDIT THE DATA FIELDS
20355  REM
20360  LINE% = 1  : REM  START IN DATA FIELD
20365  ENTRY$ = LINE$(LINE%)  : REM  FIELD
20370  ROW% = SROW%(LINE%)  : REM  ROW
20375  COL% = SCOL%(LINE%)  : REM  COL
20380  MASK$ = SMASK$(LINE%)  : REM  MASK
BASIC BUSINESS SUBROUTINES FOR THE APPLE II AND IIe

20385 GOSUB 50000 : REM EDIT FIELD
20390 LINE$(LINE%) = ENTRY$ : REM SAVE THE EDITED DATA FIELD
20395 IF HELP% > 0 THEN RETURN : REM HELP REQUESTED IN THE FIELD
20400 IF (CTRL% = 3) AND (LINE% > 1) THEN LINE% = LINE% - 1 :
       GOTO 20365 : REM UP ARROW
20405 IF CTRL% = 3 THEN GOTO 20365 : REM UP ARROW BUT ALREADY AT TOP
20410 IF CTRL% = 27 THEN GOTO 20425 : REM ESC SO GO TO BOTTOM
20415 IF LINE% < ITEMS% THEN LINE% = LINE% + 1; GOTO 20365; REM MOVE DOWN A LINE
20420 REM
20425 REM VERIFY ENTRIES
20430 REM
20435 VTAB 24 : REM GOTO BOTTOM
20440 HTAB 10
20445 PRINT "CHANGE WHICH ITEM?";
20450 MASK$ = "## " : REM ALLOW HELP AND UP TO 99 FIELDS
20455 ENTRY$ = "0 " : REM DEFAULT
20460 ROW% = 24
20465 COL% = 30
20470 GOSUB 50000 : REM EDIT DATA
20475 IF CTRL% = 3 THEN LINE% = ITEMS%: GOTO 20365: REM UP ARROW
20480 LINE% = VAL (ENTRY$) : REM LINE TO EDIT
20485 IF LINE% = 0 THEN RETURN : REM ALL DONE WITH THIS SCREEN
20490 IF (LINE% <= ITEMS%) AND (LINE% > 0) THEN GOTO 20365 : REM EDIT THE REQUESTED FIELD
20495 GOTO 20425 : REM BAD ENTRY
20500 REM
20505 REM ***************
20510 REM
21262 IF ENTRY$ = "< " THEN GOSUB 20205 : REM ASSIGN FIELD CHARACTERISTICS
21263 IF ENTRY$ <> CHR$ (13) THEN COL% = COL% + 1 : REM INC COLUMN
       POSITION COUNTER
21286 COL% = 1 : REM RESET COLUMN COUNTER
21457 IF NOPAUSE% <> 0 THEN PRINT DSK$; "CLOSE ";SCREEN$: RETURN : REM
       RETURN WITHOUT A PAUSE
51124 IF KEY% = 10 THEN CTRL% = 2 : REM ^J LINE FEED 51124
51126 IF KEY% = 11 THEN CTRL% = 3 : REM ^K UP ARROW EXIT 51126
51184 IF KEY% = 27 THEN CTRL% = 27 : REM ESC EXIT 51184
21000 REM SHOW HELP * SCREEN DISPLAY ROUTINE
21005 REM
21010  REM
21015  REM
21020  REM DISPLAYS SCREEN AND USES INVERSE
21025  REM WILL PAUSE WHEN THE SCREEN IS FULL
21030  REM PROVIDES USER TIME TO READ
21035  REM
21040  REM
21045  REM IMPORTANT VARIABLES USED:
21050  REM  LINE%   NUMBER OF LINES TO DISPLAY
21060  REM  Y       INVERSE FLAG
21065  REM  ROW%   TOTAL NUMBER OF LINES DISPLAYED
21070  REM  XX      NUMBER OF LINES IN CURRENT SCREEN
21075  REM  NOPAUSE% 1 = DO NOT PAUSE AT END OF PAGE. 0 = PAUSE
21080  REM  HELP$   NAME OF HELP TEXT FILE
21085  REM
21090  REM LOAD AND DISPLAY THE SCREEN
21095  REM
21100  REM SCREEN READ ONE CHARACTER AT A TIME
21110  REM INVERSE TOGGLED ON '^[ ' CHARACTER
21115  REM
21120  REM
21125  REM
21130  REM
21135  ROW% = 0 : REM CLEAR COUNTER
21140  XX = 0
21145  DSK$ = CHR$ (4) : REM DISK ^D
21150  PRINT : REM CLEAR ANY DSK COMMANDS
21155  PRINT DSK$; "OPEN ";HELP$
21160  PRINT DSK$; "READ ";HELP$
21165  REM
21170  REM READ THE NUMBER OF LINES ON THE SCREEN
21175  REM
21180  INPUT LINE%
21185  REM
21190  REM CLEAR THE COUNTERS
21195  REM
21200  ROW% = 0
21205  REM
21210  Y = 0 : REM INVERSE FLAG
21215  REM
21220  REM
21225  REM INPUT THE SCREEN
21230 REM
21235 HOME : REM CLEAR SCREEN
21237 IF ROW% = LINE% THEN PRINT DSK$; "CLOSE "; HELP$: RETURN : REM ALL DONE
21240 INPUT A$: REM GET A TEXT LINE
21242 IF MID$(A$,1,1) = "#" THEN GOSUB 22180: GOTO 21237: REM 21242
21245 PROCESS MENU FILE NAMES
21250 FOR Z = 1 TO LEN(A$)
21255 ENTRY$ = MID$(A$,Z,1)
21265 IF ENTRY$ = "@" THEN GOSUB 21430: REM PAUSE WANTED
21268 IF ENTRY$ = "^" THEN GOSUB 21380: REM TOGGLE INVERSE VIDEO
21275 PRINT ENTRY$; NEXT Z
21280 PRINT
21285 ROW% = ROW% + 1 : REM INCREMENT LINE COUNTER
21290 XX = XX + 1 : REM INCREMENT THIS PAGE LINE COUNTER
21295 IF (ROW% = LINE%) OR (XX = 22) THEN GOSUB 21430: REM DO I PAUSE?
21300 IF ROW% = LINE% THEN PRINT DSK$; "CLOSE "; HELP$: RETURN : REM RETURN TO THE CALLER
21305 REM
21310 GOTO 21240 : REM GET NEXT CHR
21315 REM
21320 REM ******************
21325 REM
21330 REM
21335 REM TOGGLE INVERSE ON/OFF
21340 REM
21345 ENTRY$ = "" 
21350 IF Y > 0 THEN Y = 0: NORMAL : RETURN : REM CLEAR INVERSE
21355 INVERSE : REM TURN INVERSE ON
21360 Y = 1 : REM SET FLAG
21365 RETURN
21370 REM
21375 REM
21380 REM PAUSE AND ASK FOR MORE?
21385 REM
21390 REM PAUSE AND ASK FOR MORE?
21395 REM
21400 REM
21405 REM
21410 REM
21415 REM
21420 REM ******************
21425 REM
21430 REM PROGRAMMER DOES NOT WANT PAUSE
21435 PRINT : REM CLEAR GET COMMAND
21440 REM
21445 IF NOPAUSE% > 0 THEN RETURN : REM PROGRAMMER DOES NOT WANT PAUSE
21450 PRINT : REM CLEAR GET COMMAND
21455 PRINT DSK$: REM TURN READ OFF
CHAPTER 8  PERSONAL CALENDAR: A SAMPLE PROGRAM

21457  IF NOPAUSE% <> 0 THEN PRINT DSK$;"CLOSE ";SCREEN$: RETURN : REM
21460   VTAB 23: REM  PAUSE LINE
21465  ENTRY$ = ": REM  MAKE SURE NOTHING HERE
21470  INPUT "DO YOU WISH MORE? ":ENTRY$
21475  HOME : REM  CLEAR THE SCREEN
21480  IF ENTRY$ = "N" THEN YY = LINE%: RETURN : REM  THEY WANT OUT
21485  PRINT DSK$;"READ ";HELP$ : REM  TURN DISK INPUT BACK ON
21490  XX = 0 : REM  RESET PAGE LINE COUNTER
21495  ENTRY$ = "" : REM  CLEAR ANSWER
21500  RETURN : REM  GET NEXT LINE
21505  REM
21510  REM  ***************
21515  REM

22000  REM  MENU DRIVER *
22005  REM
22010  REM
22015  REM
22020  REM  DISPLAYS MENU AND CHAINS TO PROGRAM OF USERS CHOICE
22025  REM
22030  REM
22035  REM  IMPORTANT VARIABLES USED:
22040  REM  LINE$() CONTAINS NAME OF PROGRAM TO CHAIN TO
22045  REM
22050  DSK$ = CHR$ (4) : REM  DISK ^D
22055  ITEMS% = 0 : REM  NUMBER OF OPTIONS
22060  NOPAUSE% = 1 : REM  INFORM HELP SYSTEM NOT TO PAUSE
22065  HELP$ = SCREEN$ : REM  SEND THE MENU NAME
22070  GOSUB 21000 : REM  SHOW THE SCREEN USING HELP ROUTINE
22075  NOPAUSE% = 0 : REM  RESTORE THE FLAG FOR OTHERS
22080  REM
22085  REM  ASK FOR OPTION
22090  REM
22095  VTAB 3
22100  POKE 36, 10 : REM  HTAB
22105  PRINT "PLEASE SELECT OPTION"
22110  ROW% = 3
22115  COL% = 35
22120  MASK$ = "##": REM  ALLOW HELP REQUEST
22125  ENTRY$ = "0"
```
22130 GOSUB 50000  REM ACCEPT OPTION
22135 IF HELP% THEN HELP$ = SCREEN$ + " HELP": GOSUB 21000:
     GOTO 22000: REM RESPONDED TO HELP REQUEST
22140 X = VAL (ENTRY$)  REM OPTION SELECTED
22150 IF X = 0 THEN HOME: END: REM RETURN TO BASIC
22152 IF LINE$(X) = "HELP" THEN HELP% = 1:HELP$ = SCREEN$ + " HELP": GOSUB
     21000: GOTO 22000: REM RESPONDED TO HELP REQUEST
22155 PRINT  REM CLEAR FOR DISK COMMAND
22160 ON X GOSUB 1000,2000,3000,4000
22165 RETURN
22170 REM ********************
22175 REM
22180 REM ACCEPT A CHAIN NAME
22185 REM
22190 Z = VAL (MID$ (A$,2,2))  REM GET THE OPTION NUMBER
22195 IF ITEM% < Z THEN ITEM% = Z: REM ADJUST COUNTER
22200 LINE$(Z) = MID$ (A$,4)  REM MOVE THE FILE NAME
22205 ROW% = ROW% + 1  REM INCREMENT COUNTER
22210 RETURN
22215 REM
22220 REM ********************
22225 REM
25000 REM SETUP FOR PAGE 1
25010 RETURN  REM NONE NEEDED
25020 REM
25030 REM ********************
25040 REM
25500 REM RESTORE FOR PAGE 1
25510 RETURN  REM NONE NEEDED
25520 REM
25530 REM ********************
25540 REM
26000 REM SINGLE DATE PAGE
26010 REM
26020 LINE$(1) = ""
26030 LINE$(2) = ""
26040 RETURN
26050 REM
26060 REM ********************
```
CHAPTER 8  PERSONAL CALENDAR: A SAMPLE PROGRAM

26070 REM
26500 REM  RESTORE SINGLE DATE DATA
26510 REM
26520 MM = VAL (LINE$(1))  : REM  MONTH
26530 DD = VAL (LINE$(2))  : REM  DAY
26540 RETURN
26550 REM
26560 REM  ***************
26570 REM
27000 REM
27010 REM
27020 FOR X = 1 TO 4
27030 LINE$(X) = ""  : REM  NULL STRING
27040 NEXT X
27050 RETURN
27060 REM
27070 REM  ***************
27080 REM
27500 REM  CHANGE NOTHING FOR TWO DATES
27510 REM
27520 RETURN
27530 REM
27540 REM  ***************
27550 REM
30000 REM  DATA PRINT SUBROUTINE
30005 REM
30010 REM  THIS FILLS IN A DATA PAGE AND SENDS IT TO THE PRINTER OR DISK
30015 REM
30020 REM  BE SURE TO ADD SPECIAL PRINTER COMMANDS
30025 SCREEN$ = PAGE$(PAGE%)  : REM  THE PAGE TO FILL
30030 HOME  : REM  INFORM THE USER OF PRINTING
30035 VTAB 10
30040 POKE 36, 10  : REM  HTAB
30045 PRINT "PROCESSING ";SCREEN$
30050 VTAB 12
30055 POKE 36, 15  : REM  HTAB
30060 PRINT "PLEASE WAIT"
30065 ON PAGE$ GOSUB 25000
30070 DSK$ = CHR$ (4)  : REM  DISK COMMAND CODE
30075  PRINT DSK$;"OPEN ";SCREEN$: REM OPEN THE PAGE FOR INPUT
30080  IF DEVICE% <> 0 THEN PRINT DSK$;"PR"];DEVICE%: GOTO 30115: REM
SEND DIRECTLY TO THE PRINTER
30085  REM
30090  REM PRINT IT TO A DISK FILE
30095  REM
30100  PRINT DSK$;"OPEN ";SCREEN$; " REPORT"
30105  PRINT DSK$;"DELETE ";SCREEN$; " REPORT": REM ERASE ANY OLD FILE
30110  PRINT DSK$;"OPEN ";SCREEN$; " REPORT": REM OPEN A CLEAN NEW FILE
30115  PRINT DSK$;"READ ";SCREEN$: REM GET THE NUMBER OF LINES
30120  INPUT ITEMS%
30125  REM
30130  REM
30135  ITEMS% = ITEMS% - 1 : REM DECREMENT COUNTER
30140  IF ITEMS% < 0 THEN GOTO 30475: REM ALL DONE SO EXIT
30145  PRINT DSK$;"READ ";SCREEN$: REM READ THE DATA PAGE
30150  INPUT ENTRY$: REM GET A LINE FROM THE DATA PAGE
30155  IF A$ < > "&" THEN C$ = "": REM CLEAR THE OUTPUT LINE
30160  IF A$ = "&" THEN GOSUB 30240: REM STRIP LEADING SPACES
30165  REM
30170  REM PROCESS ENTRY$ ONE CHAR AT A TIME LOOKING FOR INSERTS
30175  REM
30180  FOR X = 1 TO LEN (ENTRY$) : REM STEP DOWN THE LINE
30185  A$ = MID$ (ENTRY$,X,1) : REM GET ONE CHARACTER
30190  IF A$ = "<" THEN GOSUB 30275: GOTO 30205: REM DO AN INSERTION
30195  IF A$ = "&" THEN X = LEN (ENTRY$): GOTO 30205: REM CONTINUATION
SYMBOL
30200  C$ = C$ + A$ : REM BUILD THE OUTPUT STRING
30205  NEXT X
30210  IF DEVICE% = 0 THEN PRINT DSK$;"WRITE ";SCREEN$; " REPORT": REM
SEND OUTPUT TO DISK
30215  IF A$ < > "&" THEN PRINT C$: REM OUTPUT THE STRING
30220  GOTO 30135
30225  REM ***************
30230  REM
30235  REM
30240  REM REMOVE THE LEADING SPACES
30245  REM
30250  IF MID$ (ENTRY$,1,1) = " " THEN ENTRY$ = MID$ (ENTRY$,2):
GOTO 30250: REM FOUND ONE SO REMOVE IT
30255  RETURN
CHAPTER 8  PERSONAL CALENDAR: A SAMPLE PROGRAM

30260 REM  ***************
30265 REM
30270 REM
30275 REM INSERT A FIELD INTO THE LINE
30280 REM
30285 REM TWO MODES:
30290 REM 1) INSERT AND CONCATENATE
30295 REM 2) INSERT BUT DO NOT CONCATENATE
30300 REM MODE 1 IS USED WHEN NO SPACES EXIST BETWEEN <> SYMBOLS
30305 REM MODE 2 IS USED IF A SPACE EXISTS BETWEEN <> SYMBOLS
30310 REM
30315 REM
30320 REM
30325 REM
30330 REM FIRST REMOVE THE <> SYMBOLS
30335 REM
30340 Y = 0 : REM DEFAULT TO MODE 1
30345 X = X + 1
30350 IF X > LEN (ENTRY$) THEN A$ = "": RETURN : REM ONLY ONE SYMBOL SO ERROR
30355 B$ = MID$ (ENTRY$,X,1) : REM GET ONE CHARACTER
30360 A$ = A$ + B$
30365 IF B$ < > "">" THEN GOTO 30345: REM NOT DONE SO GET ANOTHER CHARACTER
30370 REM LOOK FOR A SPACE TO SET MODE
30375 FOR Z = 1 TO LEN (A$)
30380 IF A$ = " " THEN Y = Z:Z = LEN (A$)
30385 NEXT Z
30390 IF Y = 0 THEN Z = VAL ( MID$ (A$,2, LEN (A$) - 1))
30395 IF Y < > 0 THEN Z = VAL (MID$ (A$,2,Y - 1)): REM GET THE FIELD NUMBER
30400 C$ = C$ + LINE$(Z) : REM ADD THE FIELD
30405 IF Y = 0 THEN A$ = "": RETURN : REM MODE 1 SO ALL DONE
30410 REM
30415 REM MODE 2 SELECTED SO WE MUST FILL WITH SPACES
30420 REM
30425 IF LEN (LINE$(Z)) = LEN (A$) THEN RETURN : REM NOTHING TO CLEAR
30430 Y = LEN (A$) - LEN (LINE$(Z))
30435 FOR Z = 1 TO Y
30440 C$ = C$ + " " : REM ADD THE SPACES
30445 NEXT Z
30450 A$ = ""
30455 RETURN : REM ALL DONE
30460  REM  ***************
30465  REM
30470  REM
30475  REM  CLOSE EVERYTHING AND RETURN
30480  REM
30485  PRINT DSK$;"CLOSE ";SCREEN$
30487  PRINT CHR$(12) : REM  FORM FEED
30490  IF DEVICE% = 0 THEN PRINT DSP$;"CLOSE ";SCREEN$; " REPORT"
30495  IF DEVICE% <> 0 THEN PRINT DSK$;"PR#O": REM  RETURN TO SCREEN OUTPUT
30500  HOME                     : REM  CLEAR THE SCREEN
30505  RETURN
30510  REM  ***************
30515  REM
30520  REM

50000  REM  BASIC LINE EDITOR
50005  REM
50010  REM
50015  REM  THIS IS A BASIC LINE EDITOR
50020  REM
50025  REM  THE PROGRAMMER CALLS IT USING THE FOLLOWING VARIABLES
50030  REM
50035  REM  ROW% => SCREEN LINE NUMBER
50040  REM  COL% => SCREEN COLUMN NUMBER
50045  REM  ENTRY$ => TEXT TO BE EDITED
50050  REM  MASK$ => DATA TYPE TO BE ALLOWED
50055  REM  WHERE:
50060  REM  A = ALPHANUMERIC
50065  REM  # = NUMBER FIELD ONLY
50070  REM  Y = YES/NO FIELD
50075  REM  Q = HELP REQUEST OK, USE IN ANY CHARACTER
50080  REM  THE LENGTH OF MASK$ IS THE MAXIMUM LENGTH OF THE INPUT STRING
50085  REM
50090  REM
50095  REM
50100  PLACE% = 1                     : REM  SET THE STARTING POSITION
50105  REM
50110  FILL$ = "."                   : REM  DISPLAY DOTS
50115  HELP% = 0                     : REM  CLEAR THE HELP FLAG
50120  CTRL% = 0                     : REM  CLEAR THE EXIT FLAG
GOSUB 52130 : REM DISPLAY ENTRY$
GOSUB 50165 : REM EDIT THE STRING
FILL$ = " " : REM CLEAR THE SCREEN
GOSUB 52130 : REM DISPLAY ENTRY$
RETURN : REM GO BACK TO CALLER
REM **************
REM EDIT THE ENTRY$ FIELD
REM POSITION THE CURSOR
VTAB ROW% : REM VERTICAL POSITION
GOSUB 52000 : REM PRINT THE CHARACTER IN INVERSE
REM ACCEPT A KEY FROM THE KEYBOARD
KEY% = PEEK (49152) : REM TEST FOR INPUT
IF KEY% < 128 THEN GOTO 50210 : REM LOOP UNTIL ENTRY
REM IF HERE THEN A KEY PUSHED
XX = PEEK (49168) : REM CLEAR KEYBOARD
KEY% = KEY% - 128 : REM STRIP OFF FLAG BIT
REM PROCESS THE KEY
GOSUB 50295 : REM KEY% PROCESSOR
IF HELP% > 0 THEN RETURN : REM HELP REQUESTED BY USER
IF CTRL% > 0 THEN GOSUB 52070: RETURN : REM CONTROL KEY EDIT
GOTO 50210 : REM GET THE NEXT KEY
REM ****************************
TEST FOR CONTROL KEY
IF KEY% < = 31 THEN GOSUB 51000: RETURN : REM PROCESS AND RETURN
REM MUST BE AN ALPHANUMERIC
REM TEST THE MASK TO DETERMINE DATA TYPE
BASIC BUSINESS SUBROUTINES FOR THE APPLE II AND IIe

50335 IF MID$(MASK$,PLACE%,1) = "A" THEN GOSUB 50900: RETURN
50340 IF MID$(MASK$,PLACE%,1) = "#" THEN GOSUB 50390: RETURN
50345 IF MID$(MASK$,PLACE%,1) = "Y" THEN GOSUB 50440: RETURN
50355 REM
50360 REM BAD MASK CHARACTER
50365 REM
50370 RETURN
50375 REM
50380 REM ************************************
50385 REM
50390 REM ACCEPT A NUMBER
50395 REM
50400 REM TEST TO SEE IF IT IS A VALID NUMERIC TYPE OF CHARACTER
50405 REM
50410 IF (KEY% < 45) OR (KEY% > 57) THEN RETURN: REM BAD KEY
50415 IF KEY% = 47 THEN RETURN : REM BAD KEY ALSO
50420 GOSUB 50900 : REM GOOD KEY SO ACCEPT IT
50425 RETURN
50430 REM ************************************
50435 REM
50440 REM TEST FOR YES OR NO
50445 IF (KEY% < > 89) AND (KEY% < > 78) THEN RETURN : REM BAD KEY
50450 GOSUB 50900 : REM ACCEPT IT
50455 RETURN
50460 REM ************************************
50465 REM
50900 REM PRINT KEY% AND ADD TO ENTRY$
50905 REM
50910 IF INSERT% = 1 THEN GOSUB 51315 : REM INSERT A SPACE
50915 TXTSIZE% = LEN (ENTRY$) : REM MAKE SURE WE HAVE CORRECT TXTSIZE%
50920 REM ADD TO END OF ENTRY
50925 IF PLACE% > TXTSIZE% THEN ENTRY$ = ENTRY$ + CHR$(KEY%): GOTO 50945
50926 REM ADD AS FIRST CHARACTER
50930 IF PLACE% = 1 THEN ENTRY$ = CHR$(KEY%) + MID$(ENTRY$ , PLACE% + 1) : GOTO 50945
50932 REM ADD AS LAST CHARACTER
50935 IF PLACE% = TXTSIZE% THEN
50936 ENTRY$ = LEFT$(ENTRY$ , PLACE% - 1) + CHR$(KEY%): GOTO 50945
50936 REM ADD IN THE MIDDLE SOMEWHERE
50940 ENTRY$ = LEFT$(ENTRY$ , PLACE% - 1) + CHR$(KEY%) + MID$(ENTRY$ , PLACE% + 1)
50945 TXTSIZE% = LEN (ENTRY$)
50946 REM IF TOO BIG TRUNCATE IT
CHAPTER 8  PERSONAL CALENDAR: A SAMPLE PROGRAM

50950  IF TXTSIZE% > MAXSIZE% THEN ENTRYS$ = LEFT$(ENTRY$, MAXSIZE%)
50955  REM
50960  REM NEED TO MOVE RIGHT ONE PLACE
50965  REM
50970  GOSUB 51425  : REM RIGHT ARROW
50975  RETURN
50980  REM
50985  REM ***************
50990  REM
51000  REM PROCESS A CONTROL KEY
51005  REM
51010  REM EXIT KEYS SUCH AS RETURN SET CTRL%
51015  REM
51020  REM
51025  REM ^A = 1 > PREVIOUS WORD
51030  REM ^D = 4 > DELETE THIS CHARACTER
51035  REM
51040  REM ^F = 6 > FILL WITH A SPACE
51045  REM ^H = 8 > LEFT ARROW
51050  REM ^N = 14 > SKIP TO END
51055  REM ^Q = 17 > HELP REQUEST
51060  REM ^U = 21 > RIGHT ARROW
51065  REM ^W = 23 > NEXT WORD
51070  REM ^Y = 25 > ERASE TO END
51075  REM IGNORE ALL OTHER KEYS
51080  REM
51085  CTRL% = 0  : REM CLEAR EXIT FLAG
51090  IF KEY% = 6 THEN GOSUB 51280 : RETURN : REM INSERT
51095  INSERT% = 0  : REM TURN INSERT OFF
51100  IF KEY% = 1 THEN GOSUB 51555 : REM PREVIOUS WORD
51105  REM
51110  IF KEY% = 4 THEN GOSUB 51210 : REM DELETE
51115  REM
51120  IF KEY% = 8 THEN GOSUB 51380 : REM LEFT ARROW
51125  REM
51130  IF KEY% = 13 THEN CTRL% = 1: REM RETURN KEY
51135  REM CHECK HELP REQUEST
51140  IF KEY% = 17 THEN HELP% = 1: RETURN : REM HELP REQUEST
51145  REM
51150  IF KEY% = 14 THEN GOSUB 51680: REM GOTO END
51155  REM
51160 IF KEY% = 21 THEN GOSUB 51425: REM RIGHT ARROW
51165 REM
51170 IF KEY% = 23 THEN GOSUB 51475: REM NEXT WORD
51175 REM
51180 IF KEY% = 25 THEN GOSUB 51630: REM ERASE TO END
51184 IF KEY% = 27 THEN CTRL% = 27 : RETURN : REM 51184 ESC
51185 REM
51190 RETURN
51195 REM
51200 REM *******************
51205 REM
51210 REM DELETE AND PACK
51215 REM
51220 TXTSIZE% = LEN (ENTRY$)
51225 IF TXTSIZE% = 0 THEN RETURN : REM NOTHING TO DELETE
51230 IF TXTSIZE% = 1 THEN ENTRY$ = ": PLACE% = 1: GOTO 51250: REM DELETE LINE
51235 IF PLACE% = 1 THEN ENTRY$ = MID$ (ENTRY$,2): GOTO 51250
51240 IF PLACE% > = TXTSIZE% THEN
   ENTRY$ = LEFT$(ENTRY$,TXTSIZE% - 1): PLACE% = PLACE% - 1: GOTO 51250
51245 ENTRY$ = LEFT$(ENTRY$, (PLACE% - 1)): MID$ (ENTRY$, PLACE% + 1)
51250 GOSUB 52130 : REM PRINT NEW STRING
51255 GOSUB 52000 : REM PRINT INVERSE
51260 RETURN
51265 REM
51270 REM *********************
51275 REM
51280 REM TOGGLE THE INSERT MODE
51285 REM
51290 IF INSERT% = 1 THEN INSERT% = 0: RETURN : REM TURN IT OFF
51295 INSERT% = 1 : REM TURN IT ON
51300 RETURN
51305 REM *********************
51310 REM
51315 REM INSERT A CHARACTER
51320 REM
51325 REM
51330 REM IS IT THE FIRST CHARACTER?
51335 IF PLACE% = 1 THEN ENTRY$ = " " + ENTRY$: GOTO 51350
51340 REM INSERT IN THE MIDDLE
51345 ENTRY$ = LEFT$(ENTRY$,PLACE% - 1) + " " + MID$(ENTRY$, PLACE%)
51350 GOSUB 52130 : REM PRINT THE FIELD
51355  GOSUB 52070 : REM REPOSITION CURSOR
51360  RETURN
51365  REM
51370  REM **************
51375  REM
51380  REM LEFT ARROW
51385  REM
51390  GOSUB 52070 : REM DISPLAY NORMAL
51395  IF PLACE% > 1 THEN PLACE% = PLACE% - 1 : REM MOVE LEFT ONE
51400  GOSUB 52000 : REM DISPLAY INVERSE
51405  RETURN
51410  REM
51415  REM **************
51420  REM
51425  REM RIGHT ARROW
51430  REM
51435  IF MID$ (ENTRY$,PLACE%,1) = "" THEN RETURN
51440  GOSUB 52070 : REM DISPLAY AS NORMAL
51445  IF PLACE% < MAXSIZE% THEN PLACE% = PLACE% + 1
51450  GOSUB 52000 : REM DISPLAY AS INVERSE
51455  RETURN
51460  REM
51465  REM **************
51470  REM
51475  REM SKIP TO NEXT WORD
51480  REM
51485  REM
51490  IF PLACE% = > TXTSIZE% THEN RETURN : REM ALREADY AT END
51495  GOSUB 52070 : REM REMOVE CURSOR
51500  PLACE% = PLACE% + 1 : REM LOOK FOR FIRST SPACE
51505  IF PLACE% = TXTSIZE% THEN GOTO 51530
51510  IF MID$ (ENTRY$,PLACE%,1) <> " " THEN GOTO 51500: REM IS IT A SPACE?
51515  PLACE% = PLACE% + 1 : REM MOVE RIGHT ONE
51520  IF PLACE% = TXTSIZE% THEN GOTO 51530
51525  IF MID$ (ENTRY$,PLACE%,1) = " " THEN GOTO 51515: REM SKIP OVER SPACES
51530  GOSUB 52000 : REM DISPLAY CURSOR
51535  RETURN
51540  REM
51545  REM **************
51550  REM
51555  REM SKIP TO PREVIOUS WORD
51560 REM
51565 IF PLACE% = 1 THEN RETURN : REM AT THE FRONT ALREADY
51570 GOSUB 52070 : REM REMOVE CURSOR
51575 PLACE% = PLACE% - 1 : REM LOOK FOR SPACE
51580 IF PLACE% = 1 THEN GOTO 51610: REM FORCE MOVE AT LEAST ONE SPACE
51585 IF MID$(ENTRY$,PLACE%,1) = " " THEN GOTO 51575: REM SKIP OVER SPACES
51590 PLACE% = PLACE% - 1
51595 IF PLACE% = 1 THEN GOTO 51610
51600 IF MID$(ENTRY$,PLACE%,1) <> " " THEN GOTO 51590: REM IS IT A SPACE?
51605 PLACE% = PLACE% + 1 : REM POSITION OVER FIRST LETTER
51610 GOSUB 52000 : REM DISPLAY THE CURSOR
51615 RETURN
51620 REM
51625 REM **************
51630 REM ERASE TO END OF LINE
51635 REM
51640 IF PLACE% = 1 THEN ENTRY$ = "": GOTO 51650: REM ERASE WHOLE LINE
51645 ENTRY$ = LEFT$(ENTRY$,PLACE% - 1)
51650 GOSUB 52130 : REM PRINT THE FIELD
51655 GOSUB 52000 : REM DISPLAY THE CURSOR
51660 RETURN
51665 REM
51670 REM **************
51675 REM
51680 REM SKIP TO END OF LINE
51685 REM
51690 GOSUB 52070 : REM MOVE THE CURSOR
51695 PLACE% = LEN (ENTRY$) + 1
51700 IF PLACE% > MAXSIZE% THEN PLACE% = MAXSIZE%: REM DO NOT GO PAST END
51705 GOSUB 52000 : REM SHOW THE CURSOR
51710 RETURN
51715 REM
51720 REM **************
51725 REM
52000 REM PRINT CHARACTER IN INVERSE
52005 REM THIS GIVES THE ILLUSION OF CURSOR MOVEMENT
52010 REM
52012 VTAB ROW% : REM POSITION CURSOR
52015 POKE 36, (COL% + PLACE% - 1) : REM HTAB
52020 INVERSE : REM REVERSE VIDEO
52025 XX$ = MID$(ENTRY$,PLACE%,1): REM MOVE FOR THE NEXT IF
IF XX$ = "" THEN XX$ = " ": REM IF NULL MAKE IT A SPACE
PRINT XX$; : REM PRINT THE INVERSE
NORMAL : REM RESTORE TO NORMAL VIDEO
POKE 36, (COL% + PLACE% - 1) : REM REPOSITION THE CURSOR - HTAB
RETURN
REM
***************
REM
REM POSITION AND DISPLAY NORMAL
REM
VTAB ROW% : REM POSITION CURSOR
POKE 36, (COL% + PLACE% - 1) : REM HTAB
XX$ = MID$ (ENTRY$,PLACE%, 1) : REM PRINT ONE LETTER
IF XX$ = "" THEN XX$ = FILL$ : REM IF NULL THEN MAKE IT A SPACE
PRINT XX$;
REM
POKE 36, (COL% + PLACE% - 1) : REM REPOSITION THE CURSOR
RETURN
REM
 ***************
REM
DISPLAY TEXT$
REM FILL$ IS THE FILL CHARACTER
REM TXTSIZE% IS THE LENGTH OF ENTRY$
REM MAXSIZE% IS THE MAXIMUM ALLOWED LENGTH
REM
TXTSIZE% = LEN (ENTRY$) : REM HOW LONG IS THE CURRENT FIELD?
MAXSIZE% = LEN (MASK$) : REM WHAT IS MAX LENGTH ALLOWED?
REM
REM IS ENTRY$ TOO LONG?
REM
IF TXTSIZE% > MAXSIZE% THEN
ENTRY$ = LEFT$ (ENTRY$,MAXSIZE%):TXTSIZE% = MAXSIZE%
REM
REM POSITION THE CURSOR
REM
VTAB ROW% : REM ROW POSITION
POKE 36, COL% : REM COLUMN NUMBER - HTAB
REM
REM PRINT THE TEXT
52225   REM
52230   PRINT ENTRY$;          : REM NO LINE FEED
52235   REM
52240   REM PRINT THE FILL CHARACTER
52245   REM
52250   IF TXTSIZE% = MAXSIZE% THEN RETURN : REM NO FILL$ TO PRINT
52255   FOR XX = TXTSIZE% TO MAXSIZE% - 1
52260   PRINT FILL$;
52265   NEXT XX
52270   RETURN                   : REM ALL DONE
52275   REM
52280   REM ***************
52285   REM
INDEX

ANUM%, 67
Apple II family differences, 10–11
ASCII character codes, table of, 32
Auto line numbering, 93

CATALOG, 94–95
Character-accept routine, 38–39
CLOSE, 89
COL%, 22
Command display and processor, 82
Concatenate two lines, 77
Control characters, 35, 39
Cursor movement, 41–42

Data entry screen, 125
Date algorithm, 187
Default values, 127
DELETE, 86–87
Deleting a character, 45
Deleting a line, 79
DEVICE%, 171
DIM, 67
Displaying a cursor, 33
Display subroutine, 25, 38, 47, 68
EDIT command, 85
Editing capabilities of the Apple, 17
Editor features, 16
Edit subroutine, 30, 139–142
80 column card, 10
End of line, 44
ENTRY$, 22, 38
Erase to end of line, 48
ESC key, 49, 80
EXEC, 96

Field, 15
Field parameters, 136
INDEX

FILL$, 22
FIRST%, 67
FLASH, 10
FOR-NEXT, 27
FRE(O), 82

Garbage variables, 116
GOSUB-RETURN, 2–3

Help, 111, 158
HELP$, 116
HOME, 68

INPUT, 13–14
Input keys, processing the, 37
Inserting a blank line, 71
Inserting characters, 46
INSERT%, 38
Integer, 22
INVERSE, 11, 34, 119

Jump to home page, 76
Jump to last page, 72

KEY%, 39
LAST%, 67
LCOL%, 66
LEFT$, 46
LEN, 26
Line editor, 14
Line numbering, 5
LINE$, 67
LROW%, 66

MASK$, 22
MAXSIZE%, 38
Menu system, 151
Merging programs, 96
MIDS, 36
MLINE%, 67, 74
Moving down a line, 74
Moving up a line, 75
Multiple statements on one line, 7

Next word, 42
NOPAUSE%, 116

NORMAL, 34
OPEN, 87
PAGE%, 134
Pause subroutine, 118
PLACE%, 34, 41
Previous word, 43
PRINT, 34
Processing a key, 35
Programming style, 4–7

Random access files, 190
READ, 92
Remark (REM) statements, 6–7
Reports, philosophical, 165
ROW%, 22

Screen editor, 61
Scroll down a page, 78,
Scroll up a page, 78
Space savings in programming, 9
Speeding up programs, 10
Structure of the book, 11–12
Subroutines
defined, 2
display, 25, 38, 47, 68
edit, 30, 139
and line numbers, 5
pause, 118
text loading, 91
text saving, 86

Tab stops, 73
Testing, 8, 23
Text loading subroutine, 91
Text saving subroutine, 86

User-friendly programs, 8

VAL(), 85
Variable-exchange routine, 130
Variable names, 4
Variables, explanation of, 22–23
VTAB, 30

WRITE, 88
Other books in the Microcomputer Books Series are available from your local computer store or bookstore. For more information write:

**General Books Division**
Addison-Wesley Publishing Company, Inc.
Reading, Massachusetts 01867
(617) 944-3700

<table>
<thead>
<tr>
<th>ISBN</th>
<th>Title</th>
<th>Authors</th>
</tr>
</thead>
<tbody>
<tr>
<td>[10483]</td>
<td>Database for the IBM PC</td>
<td>Sandra L. Emerson and Marcy Darnovsky</td>
</tr>
<tr>
<td>[11358]</td>
<td>Database: A Primer</td>
<td>C. J. Date</td>
</tr>
<tr>
<td>[01245]</td>
<td>1-2-3 Go!</td>
<td>Julie Bingham</td>
</tr>
<tr>
<td>[10242]</td>
<td>Executive VisiCalc for the Apple Computer</td>
<td>Roger E. Clark</td>
</tr>
<tr>
<td>[10241]</td>
<td>Executive SuperCalc</td>
<td>Roger E. Clark</td>
</tr>
<tr>
<td>[15895]</td>
<td>Introduction to the Lisa</td>
<td>Arthur Naiman</td>
</tr>
<tr>
<td>[10187]</td>
<td>How to Choose Your Small Business Computer</td>
<td>Mark Birnbaum and John Sickman</td>
</tr>
<tr>
<td>[10355]</td>
<td>CP/M and the Personal Computer</td>
<td>Thomas A. Dwyer and Margot Critchfield</td>
</tr>
<tr>
<td>[05248]</td>
<td>Executive Computing</td>
<td>John M. Neivson</td>
</tr>
<tr>
<td>[05092]</td>
<td>Microcomputer Graphics</td>
<td>Roy E. Meyers</td>
</tr>
<tr>
<td>[06599]</td>
<td>Basic Money: Managing Personal Finances on Your Microcomputer</td>
<td>Charles Seiter</td>
</tr>
<tr>
<td>[07769]</td>
<td>Discovering Apple Logo</td>
<td>David D. Thornburg</td>
</tr>
<tr>
<td>[11208]</td>
<td>The Beginner's Guide to Computers</td>
<td>Robin Bradbeer, Peter DeBono, and Peter Laurie</td>
</tr>
<tr>
<td>[16482]</td>
<td>Astounding Games for Your Apple Computer</td>
<td>Hal Renko and Sam Edwards</td>
</tr>
<tr>
<td>[11507]</td>
<td>Dr. C. Wacko Presents Applesoft BASIC and the Whiz-Bang Miracle</td>
<td>David Heller and John Johnson</td>
</tr>
<tr>
<td>[14775]</td>
<td>Applesoft BASIC Toolbox</td>
<td>Larry G. Wintermeyer</td>
</tr>
<tr>
<td>[14652]</td>
<td>Introducing Logo</td>
<td>Peter Ross</td>
</tr>
<tr>
<td>[10341]</td>
<td>Pascal: A Problem Solving Approach</td>
<td>Elliot B. Koffman</td>
</tr>
<tr>
<td>[08296]</td>
<td>Pascal for FORTRAN Programmers</td>
<td>Robert Weiss and Charles Seiter</td>
</tr>
<tr>
<td>[06577]</td>
<td>Pascal for BASIC Programmers</td>
<td>Charles Seiter and Robert Weiss</td>
</tr>
<tr>
<td>[06516]</td>
<td>Using BASIC on the IBM PC</td>
<td>Angela and Michael Trombetta</td>
</tr>
<tr>
<td>[05464]</td>
<td>Pascal for the IBM Personal Computer</td>
<td>Ted G. Lewis</td>
</tr>
<tr>
<td>[05209]</td>
<td>Assembly Language for the Applesoft Programmer</td>
<td>Clarence W. Finley, Jr., and Roy E. Myers</td>
</tr>
<tr>
<td>[01589]</td>
<td>BASIC and the Personal Computer</td>
<td>Thomas A. Dwyer and Margot Critchfield</td>
</tr>
<tr>
<td>[05157]</td>
<td>Expanding and Maintaining Your Apple Personal Computer</td>
<td>James Morrison</td>
</tr>
</tbody>
</table>
Alan G. Porter and Martin G. Rezmer

BASIC Business Subroutines
for the
Apple II and IIe

FOR PROFESSIONALS WHO PROGRAM IN BASIC—
SOLUTIONS TO COMMON BUSINESS PROGRAMMING PROBLEMS

Does this describe you?
You're a professional, not a professional programmer.
Standard documentation does not fill your needs.
You know what you want your Apple Computer to do, but it won't do it.

BASIC Business Subroutines takes frequent business programming problems, describes how to solve them, and gives exact solutions in Applesoft BASIC. Solutions are provided in subroutines (program modules) which are easy to transport between programs and easy to modify to match your particular problem. Each solution builds upon techniques previously explained, and all the techniques are brought together in a single program at the end.

With these solutions, your software will take less time to write and, finally, will be just what you need.

Alan Porter is a California-based microcomputer consultant. Martin Rezmer has, for the last five years, been a computer store owner and consultant.

Cover design by Marshall Henrichs
Apple puzzle courtesy of Mag-Nif, Inc.'s, "Adam's Apple," New York, NY

ADDISON-WESLEY PUBLISHING COMPANY

ISBN 0-201-05663-1