SAMS

22384

BASIC Tricks for the TI-99/4A™

Allen Wyatt

BASIC Tricks for the TI-99/4A™

Allen Wyatt has been actively involved with the microcomputer industry for six years and is currently software development supervisor for Sams Software in Indianapolis, Indiana. Mr. Wyatt has had extensive experience in computer consulting and software development.

He has written several commercial software packages utilizing many of the same techniques detailed in *BASIC Tricks for the TI-99/4A*. The broad range of computer programs runs the gamut from small system data bases to games and utilities.

Besides being a computer author, Allen is a devoted family man and active church member. He uses his personal computers to assist him in all of these areas. At home, his family spends many hours using the computer every day.

BASIC Tricks for the TI-99/4A™

by
Allen Wyatt

Howard W. Sams & Co., Inc.
4300 West 62nd Street
Indianapolis, Indiana 46268 USA

FIRST EDITION
FIRST PRINTING—1984

International Standard Book Number: 0-672-22384-4
Library of Congress Catalog Card Number: 84-50802

Edited by *Susan Pink*

Printed in the United States of America.

Preface

BASIC Tricks for the TI-99/4A has several purposes. As best as I can determine, they are as follows:

1. To encourage programmers and computer users to think logically.
2. To provide a source of efficient subroutines for the aspiring programmer.
3. To provide a fix for computer junkies.
4. To make money for the author.

This book is dedicated to my parents, James and Virginia Wyatt. They are just now beginning to understand what I have known all along—that computers offer the most fun and the best chance for self-education available. I believe that they are part of a growing number of people who have been bitten by the computer bug.

ALLEN WYATT

A NOTE TO THE READER

The programs in this book were not written as applications software but as educational examples of what your personal computer can do. All of the programs have been tested and work on the machine configuration for which they were designed. The programs, or subroutines, are unprotected. This means that you can modify them to better understand how they work or to fit a different machine configuration.

What is a Combo Pack?

A Combo Pack, like this package, is a step beyond your average technical book. While most books give you programming examples through printed listings (which we do here), Combo Packs provide the book and the listings recorded on magnetic media, either diskette, cassette tape, or both.

Every effort has been made to be clear, concise, and informative about how these programs and routines work. If you experience any difficulty with the software operations, the solution can be found in the book or in your computer manuals.

We are rather proud of the time and effort that went into preparing the Combo Pack. If you have purchased the Combo Pack and have enjoyed using it, let us know your thoughts. Your comments will be valuable in preparing future Combo Packs.

LOADING INSTRUCTIONS

The cassette accompanying this Combo Pack contains the subroutine listings and/or program listings printed in the book.

To load a cassette file from this tape, perform the following steps:

1. Put the cassette into the cassette recorder.
2. Position the tape at the beginning of the subroutine or program you wish to load.
3. Type **OLD CS1**
   Press <ENTER>
4. Follow the directions as they appear on your video screen.

This will cause the next program on the tape to load into the computer's memory. When the program is loaded, it is ready to be used as described in the book.

The following list shows the listing names and tape counter positions for the contents of the cassette tape. These numbers are approximate and may vary from recorder to recorder. They should, however, assist you in locating the programs you are searching for.

Tape Directory

Listing #	Program Name	Counter Location
2-1	Numeric Input Routine	4
2-2	Payroll Program	10
2-3	Alphanumeric Input Routine	18
2-4	Payroll Program	24
3-1	Rounding Examples	34
4-1	Dollars & Cents Routine	40
4-2	Dollars & Cents Routine	44
4-3	Dollars & Cents Routine	51
4-4	Dollars & Cents with Commas	59
5-1	Report Headings Routine	64
5-2	Payroll Program with Reports	69
6-1	Date Analysis	85
6-2	Days of Week	93
6-3	Birthday Program	99
7-1	Time Routine	113
7-2	Time Routine (Hours, Minutes, Seconds)	121
7-3	Convert to Time	129
8-1	Upper Case Convert	136
8-2	Case Conversion Sample	142
8-3	Lower Case Convert	154
9-1	Substitution Sort	160
9-2	Modified Substitution Sort	167
9-3	Sort Comparison	173
9-4	Shell Sort	183
9-5	Sort Comparison	193
9-6	Quicksort	203
9-7	Sort Comparison	212
10-1	Menu Generator	230

Introduction

One day I was browsing through the offerings at the book store in my hometown mall, and I noticed something interesting. There were no books to help a computer user write BASIC routines that were useful, quick, and efficient.

Most of the books said, in effect, "Here are the tools, BASIC and your computer. They are great." But no book demonstrated how to make the personal computer do great things, although one did show me how to program a few outstanding games of tic-tac-toe.

I hope this book will go a long way toward filling this need. That is not to say that it will be a cure-all, or the cat's meow. Probably far from it. We may not even see the cat begin to stir until the coming generation fully unleashes the power of the personal computer.

However, I think that this book can be useful. It is organized so that you can start just about anywhere. You can read it as a novel or you can use it as a reference manual.

This book assumes that you know BASIC. I know what they say about assuming, but, again, there are many who already know BASIC, but can't quite figure out how to make it do all the neat tricks we've heard it can do.

Hence the name, *BASIC Tricks for the TI-99/4A*, and since this book is written specifically for the TI-99/4A, it would be helpful for you to have access to that computer. The version of BASIC employed here is straight TI BASIC. You can use Extended BASIC, but in writing the book, I thought it best to use a common version of the language. If you have experienced problems with any of this thus far, I am sure that your local department, discount, or computer store would just love for you to wander in to discuss programming and maybe buy a computer. If you do order a computer or two, tell them I sent you. Then they will love me too.

There are a lot of people, so-called "micro professionals", who may scoff at a book like this, because they feel that BASIC is too slow and

elementary, or that machine code is more elegant and powerful. To them I can only say—"Ha!" BASIC is fine for many everyday applications, and if we do what we do well, it doesn't matter what programming language we use.

The first chapter gives you a short history of the development of BASIC. You may find this chapter interesting, although not essential to using BASIC effectively. If my computer science professor heard me say that, he would give me a retroactive "F" in my Intro To Data Processing class. Oh well, the secret is out. You don't need to know the history of tools to know how to build a house.

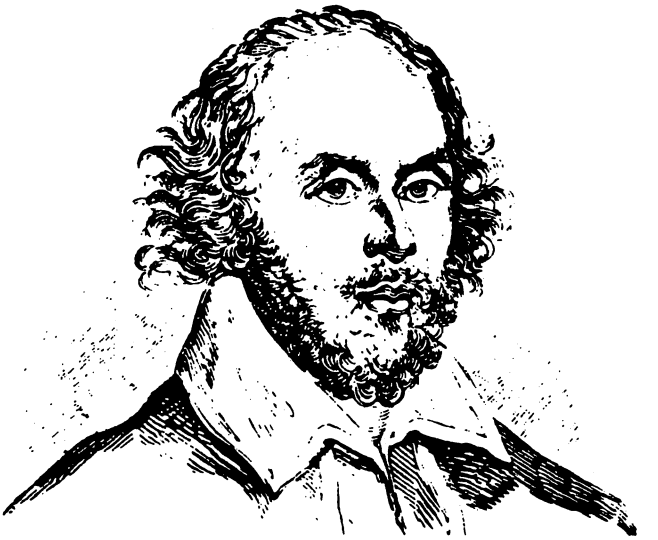

Fig. I-1. Abner Fritzbingle, the first computer science teacher (Whatsamatter U, March, 1919).

The remaining chapters are self-contained lessons on various applications routines. They can be read in or out of sequence.

Each chapter includes variable tables for each subroutine. They will help you convert these routines for use in your programs. The tables are included immediately after each listing.

Finally, it should be noted that there is a difference between *working* BASIC, *effective* BASIC, and *efficient* BASIC. Each represents a new level of programming expertise that encompasses the previous level, while doing the same task faster and better. I hope this book will help you master each level.

That's it for the introduction. Let's get on with the rest of the book.

Contents

CHAPTER 6

CHAPTER 7

CHAPTER 8

CHAPTER 9

CHAPTER 10

CHAPTER 11

Chapter 1

A Brief History of BASIC

What follows is not an in-depth history of BASIC, but rather a brief history of its development. This discussion includes some material on the TI-99/4A. This is, then, sort of your basic brief BASIC history.

BASIC was developed at Dartmouth College by Drs. John Kemeny and Thomas Kurtz primarily as a teaching language for a time-sharing environment. By 1965 it was commercially available.

BASIC is an acronym for *B*eginner's *A*ll-purpose *S*ymbolic *I*nstruction *C*ode. It is a compilation of elements found in both FORTRAN and ALGOL. Since its introduction, the simple structure and plain-English commands of BASIC have made it the most-used computer language in the world. Its many versions are used on all types of computers.

One of BASIC's chief advantages is that it is highly user-interactive. Each BASIC instruction is interpreted (parsed) when the program is run rather than when it's compiled. Before BASIC, the instructions in most languages had to be compiled (translated to machine language) to be understood by the computer. As a result, programs written in BASIC can be developed more quickly than programs written in earlier languages. Recently, several computer manufacturers have introduced both interpreter and compiler versions of BASIC for their machines. Thus, not only can a program be developed more quickly in BASIC, it can also be compiled to execute faster.

BASIC's simplicity and popularity have, however, led to the development of many different versions of the original language. No one version of BASIC is widely accepted; each hardware manufacturer changes the language slightly to take advantage of their machines' particular features.

In many ways, the *de facto* standard of BASIC is the Microsoft version. In one form or another, the different versions of the language developed by this company have touched almost every personal computer.

Microsoft began writing its version of BASIC in the mid-1970's. It has evolved considerably since then, but somewhere during development it crossed over from a "minimal" to an "extended" language. What was still needed was a version of BASIC that was a minimum acceptable implementation of the language.

The American National Standards Institute, ANSI, decided on a minimal subset of the BASIC versions that were available in the late 1970's. It retained many useful features of the language, but did not specify any one hardware configuration with machine-dependant enhancements. This version is known as ANSI Minimal BASIC.

TI BASIC is a superset of ANSI Minimal BASIC. ANSI's standard is used as the basis for implementation, but other commands have been added to allow the language to use special features available on the TI-99/4A. In addition two enhancements were made—in memory allocation, to allow larger arrays with up to three dimensions to be programmed, and in program entry, to make available a built-in line editor.

Texas Instruments built TI BASIC into the TI-99/4A on ROM chips, or Read Only Memory. As a result, to use and program the TI-99/4A you need only to plug the computer in, turn it on, and flip a switch. This feature has made the TI-99/4A a very popular computer. Of course, its low retail price may have had something to do with it.

In addition to such features, all part of standard TI BASIC, Texas Instruments has also produced an enhanced version of BASIC available on a separate cartridge. This BASIC has more features and capabilities than the standard version of BASIC. But it does cost more and, therefore, not everyone will have access to it.

However, standard TI BASIC is adequate for most programming needs. In this book, we will be working with TI BASIC to develop routines and programs that exemplify how to use the language to solve specific tasks.

Chapter 2

Selective Input

Most programs have one thing in common—they require human input to make them valuable. For example, what good would a payroll program be unless the paymaster could tell the computer how many hours, and at what rate, John Doe worked last week?

Although input makes a program valuable, it also can create potential problems. Not everyone thinks like you and me, and some have no idea what a program really wants when it asks a question. For instance, consider the following short program:

```
10 INPUT "HOW MANY HOURS: ":H
20 INPUT "WHAT RATE OF PAY: ":R
30 P=H*R
40 PRINT "CORRECT PAY IS: ";P
```

Looks like a whiz-bang program, right? Wrong! At this very moment, there is some paymaster in Podunksville who entered STANDARD as his answer to the first question. We don't have to consider the second question, because the computer will generate an error after the invalid answer to the first question.

What is the best way to handle this type of problem? The first, and most important, way is to never use numeric variables in inputs because they can create problems. Did you know that when using the TI-99/4A, pressing

ENTER when you are prompted for a numeric variable will cause an error? After all, ENTER is not a number.

Therefore, the first step is to write the program using string variables, and then convert the strings to their numeric equivalents. But if you do this using the VAL statement, as shown in the following program, an error will still be generated if the paymaster enters STANDARD for his response.

```
10  INPUT "HOW MANY HOURS: ":H$
15  H=VAL(H$)
20  INPUT "WHAT RATE OF PAY: ":R$
25  R=VAL(R$)
30  P=H*R
40  PRINT "CORRECT PAY IS: ";P
```

This error occurs because BASIC does not recognize the direct conversion of a string containing any non-numeric characters to a number when using the VAL statement. We need a different way to input information; one that will accept virtually any input without generating an error.

The following program accepts the input as a string and analyzes it quickly for non-numeric characters. If there are any non-numeric characters, their value is set to zero, and program execution continues. This result, though not ideal, is better than an execution error.

```
10  INPUT "HOW MANY HOURS: ":H$
12  A$=H$
14  GOSUB 60
16  H=A
20  INPUT "WHAT RATE OF PAY: ":R$
22  A$=R$
24  GOSUB 60
26  R=A
30  P=H*R
40  PRINT "CORRECT PAY IS: ";P
50  END
60  FOR J=1 TO LEN(A$)
65  A=ASC(SEG$(A$,J,1))
70  IF (A<48)+(A>57) THEN 90
75  NEXT J
80  A=VAL(A$)
85  GOTO 95
90  A=0
95  RETURN
```

Although the program is longer, there is no execution error when a user answers incorrectly. For instance, you won't get an error if our paymaster enters STANDARD for the number of hours. However, you also won't get any pay because the numeric value of STANDARD is zero, and, as we all know, zero multiplied by anything else is still zero.

Now comes the interesting part. How do we check to make sure that the user has input a valid entry? First, we have to follow a very simple set of guidelines:

1. Decide what input is wanted.
2. Develop a routine to get only that type of input.
3. Check to make sure the input is within predetermined bounds.
4. Test the heck out of it.
5. Let someone else test the heck out of it.
6. Never assume that the routine is foolproof, because some fool is bound to prove you wrong.

Simple, right? The first step is to determine what type of input you want. This can be either numeric or alphanumeric input. We will discuss both types in just a moment, but first, let's look at one of the neatest commands since sliced bread (how's that for a mixed metaphor?).

THE CALL KEY STATEMENT

BASIC has a neat little internal subroutine that will allow your computer to do wonderful things—the CALL KEY statement. This subroutine allows you to test if the keyboard is being pressed at the moment the KEY routine is called. The correct format for the statement is:

```
100 CALL KEY(T,A,S)
```

There are three variables in the statement, and each has individual significance. T is used to designate the keyboard type. The type you specify determines what type of values are returned in the next variable, A. This is the corresponding key code for the key pressed, if any. Finally, S will have one of three values: it will be 1 if a new key has been pressed, -1 if the key is being held down, and 0 if no key is being pressed.

For our purposes, it is best to use a keyboard type of 0. This will return ASCII codes for the keys pressed. For instance, pressing the letter G would set variable A equal to 71, and the status flag S equal to 1. A quick program to show how the CALL KEY statement works is shown below:

```
100 CALL KEY(0,A,S)
110 IF S=0 THEN 100
120 PRINT A,CHR$(A)
130 GOTO 100
```

Once this program is run, the only way to exit is to press FCTN CLEAR. This is because the program is an endless loop that cycles while waiting for someone to press any key on the keyboard. If you do press a key, the key's associated keyboard code, or ASCII code since we specified a keyboard type of 0, is printed along with the key's character.

It is clear from this simple exercise why use of keyboard type 0 was recommended. The keyboard codes that are returned when you specify keyboard type 0 can then be translated directly into characters by use of the CHR$ statement.

Well, how do we make use of all this information in an input routine? Since we already started with an example of inputting numerics, we will continue with that. Later we will discuss alphanumeric input.

NUMERIC INPUT ROUTINES

By using the CALL KEY statement, and slightly modifying the example that was shown a short while ago, we can develop a basis for our numeric input routine:

```
100 CALL KEY(0,A,S)
110 IF S=0 THEN 100
115 IF (A<48)+(A>57) THEN 100
120 PRINT CHR$(A);
130 GOTO 100
```

This set of instructions, in only three lines, does quite a bit. First of all, line 100 gets a character from the user's input. Then line 115 checks the character to make sure that it is a valid number. The ASCII value for the number 0 is 48, and for 9 it is 57. If the keypress had a value that was less than 0 (A<48) or greater than 9 (A>57), then the keypress was out of range. If this is the case, the keypress is ignored, and the computer goes back to wait for another keypress.

Finally, if the keypress was acceptable, the computer prints what was typed with the statement in line 120. The semi-colon at the end of this statement keeps all the printed characters on the same line.

Now, we need to add a few more lines. We need to accumulate, in a handy place, the answer that the user is typing. We also need to check to see if the user typed an ENTER, or if the first typed character was a

negative sign, and if a decimal point has been entered. With these features, our routine now looks like this:

Listing 2-1

```
1 REM      LISTING 2-1
100 B$=""
105 D=0
110 C=LEN(A$)+3
115 PRINT A$;
120 GOTO 130
125 CALL SOUND(100,1000,10)
130 CALL KEY(0,A,S)
135 IF S<>1 THEN 130
140 IF A=13 THEN 235
145 IF A<>8 THEN 195
150 IF LEN(B$)=0 THEN 125
155 IF SEG$(B$,LEN(B$),1)<>"." THEN 165
160 D=0
165 B$=SEG$(B$,1,LEN(B$)-1)
170 J=LEN(B$)-1+C
175 IF LEN(B$)=0 THEN 185
180 CALL HCHAR(24,J,ASC(SEG$(B$,LEN(B$),1)))
185 CALL HCHAR(24,J+1,32)
190 GOTO 130
195 IF (B$="")*(A=45)THEN 220
200 IF ((A<48)+(A>57))*(A<>46)THEN 125
205 IF (A=46)*(D=1)THEN 125
210 IF A<>46 THEN 220
215 D=1
220 B$=B$&CHR$(A)
225 CALL HCHAR(24,C+LEN(B$)-1,A)
230 GOTO 130
235 B=0
240 IF (B$="")+(B$=".")THEN 250
245 B=VAL(B$)
250 RETURN
```

Table 2-1. Entry and Exit Variables for Numeric Input Routine

On entry:	On exit:
A$—User prompt	A$—User prompt B$—String entered by user B —Numeric value of string

Notice our fundamental routine? It has been shifted around a bit, but all we did was add an initialization line to clear the variables that will be used in the routine (lines 100-110). Then we print the question prompt, which should be contained in A$ when this routine is entered.

After getting the keypress and making sure that a new key was pressed (lines 130-135), line 140 checks if the keypress is the ENTER key. If it is,

execution branches to line 235 which assigns the actual value that was input to variable B and returns from the routine.

Lines 145 through 190 handle the keypress if it was a backspace, CHR$(8). After all, everyone makes mistakes. If it was pressed, line 150 checks that there is something there to backspace over. If not, a beep is sounded and the routine waits for the next keypress.

Table 2-2. Variable Table for Numeric Input Routine

Variable	Type	Purpose	Used in lines
A	Numeric	Keypress character	130, 140, 145, 195, 200, 205, 210, 220, 225
A	String	User prompt	110, 115
B	String	User input	100, 150, 155, 165, 170, 175, 180, 195, 220, 225, 240, 245
B	Numeric	String value	235, 245
C	String	Prompt length	110, 170, 225
D	Numeric	Decimal flag	105, 205, 215
J	Numeric	String position	170, 180, 185
S	Numeric	Keyboard status	130, 135

If there is something to backspace over, lines 155 and 160 handle the instances when the user wants to backspace over the decimal point in the number. If the program is erasing the decimal point, the numeric flag D is set to 0 and the rightmost character of B$ is dropped. Lines 180 and 185 print the new last character of the input string and blank out the old last character. Then the routine goes back to wait for another keypress.

Line 195 checks if the character pressed was a minus sign. If it was the minus sign and if it will be the first character of the entry, it is accepted. Notice that B$ will be null (equal to nothing) only if we are analyzing the first character that is pressed.

A routine is also included to check that only one decimal point was entered in the number. Lines 200 through 210 check, and set the flag, if a decimal point has been entered. A decimal point will only be allowed if D is still zero. When one decimal point has been entered, D is equal to one, and we continue on our merry way, accepting numbers only. As pointed

out, all this changes if we backspace over the decimal point, in which case D is set to zero again so we can accept another decimal point.

Finally, line 220 adds the keypress to B$. Then we print the key pressed and go back to get another keypress at line 130.

The whole idea behind this routine is to accept only numeric input; a negative sign and any legal number or decimal point (only one) will be accepted.

Will this work under all circumstances? Probably not, because users can be very resourceful in finding illegal entries to make programs bomb. However, it is more reliable and sophisticated than the INPUT statement.

There are a few other statements that can be added to this routine to tailor it to your specific needs. You could add a statement at the end of the routine to check to make sure that the input is within limits. For instance, if you do not want a number smaller than 0, or larger than 999.99, take out the checks for a negative sign, and then check to make sure that the length of B$ never goes beyond six characters.

How will the numeric input routine fit into our payroll program that we started to develop earlier? Like this:

Listing 2-2

```
1 REM      LISTING 2-2
2 CALL CLEAR
90 GOTO 1000
100 B$=""
105 D=0
110 C=LEN(A$)+3
115 PRINT A$;
120 GOTO 130
125 CALL SOUND(100,1000,10)
130 CALL KEY(0,A,S)
135 IF S<>1 THEN 130
140 IF A=13 THEN 235
145 IF A<>8 THEN 195
150 IF LEN(B$)=0 THEN 125
155 IF SEG$(B$,LEN(B$),1)<>"." THEN 165
160 D=0
165 B$=SEG$(B$,1,LEN(B$)-1)
170 J=LEN(B$)-1+C
175 IF LEN(B$)=0 THEN 185
180 CALL HCHAR(24,J,ASC(SEG$(B$,LEN(B$),1)))
185 CALL HCHAR(24,J+1,32)
190 GOTO 130
195 IF (B$="")*(A=45)THEN 220
200 IF ((A<48)+(A>57))*(A<>46)THEN 125
205 IF (A=46)*(D=1)THEN 125
210 IF A<>46 THEN 220
215 D=1
220 B$=B$&CHR$(A)
225 CALL HCHAR(24,C+LEN(B$)-1,A)
230 GOTO 130
```

cont. on next page

Listing 2-1—cont.

```
235 B=0
240 IF (B$="")+(B$=".")THEN 250
245 B=VAL(B$)
250 RETURN
1000 A$="HOW MANY HOURS: "
1010 GOSUB 100
1020 H=B
1030 A$="WHAT RATE OF PAY: "
1040 GOSUB 100
1050 R=B
1060 P=H*R
1070 PRINT "CORRECT PAY IS:";P
9999 END
```

Table 2-3. Variable Table for Payroll Program

Variable	Type	Purpose	Used in lines
A	Numeric	Keypress character	130, 140, 145, 195, 200, 205, 210, 220, 225
A	String	User prompt	110, 115, 1000, 1030
B	String	User input	100, 150, 155, 165, 170, 175, 180, 195, 220, 225, 240, 245
B	Numeric	String value	235, 245, 1020, 1050
C	String	Prompt length	110, 170, 225
D	Numeric	Decimal flag	105, 205, 215
H	Numeric	Hours	1020, 1060
J	Numeric	String position	170, 180, 185
P	Numeric	Pay	1060, 1070
R	Numeric	Rate	1050, 1060
S	Numeric	Keyboard status	130, 135

We have renumbered the earlier part of the program and added it to the end of our subroutine. This was not done to confuse you, but to conform to a standard required of those who want to write good, efficient BASIC code. All common subroutines should be located near the beginning of the program, with all controlling routines residing after them. Believe it or not, this actually helps the program execute faster! Not a whole lot faster on a small program, but when using BASIC, every little bit helps.

ALPHANUMERIC INPUT ROUTINES

Now let's talk about alphanumeric input. Again, we could use the standard INPUT statement, but it doesn't allow several things. First, if you input commas or colons you will get an input error warning message. At the least, this can be disconcerting, and can leave a user wondering what to do next.

Another minus for the INPUT statement—you can't limit input length while the user is typing an entry. Wouldn't it be great if you could have a bell ring every time a key was pressed when you were beyond the legal length for the entry; or if all characters beyond that point were ignored?

Well, we can do all of this with minor modifications to our numeric input routine. We will use the same basic instructions, but with slightly different input checks. The changed routine looks like this:

Listing 2-3

```
1 REM      LISTING 2-3
300 B$=""
305 C=LEN(A$)+3
310 PRINT A$;
315 GOTO 325
320 CALL SOUND(100,1000,10)
325 CALL KEY(0,A,S)
330 IF S<>1 THEN 325
335 IF A=13 THEN 400
340 IF A<>8 THEN 380
345 IF LEN(B$)=0 THEN 320
350 B$=SEG$(B$,1,LEN(B$)-1)
355 J=LEN(B$)-1+C
360 IF LEN(B$)=0 THEN 370
365 CALL HCHAR(24,J,ASC(SEG$(B$,LEN(B$),1)))
370 CALL HCHAR(24,J+1,32)
375 GOTO 325
380 IF (A<32)+(A>126)THEN 320
385 B$=B$&CHR$(A)
390 CALL HCHAR(24,C+LEN(B$)-1,A)
395 GOTO 325
400 PRINT
405 RETURN
```

Table 2-4. Entry and Exit Variables for Alphanumeric Input Routine

On entry:	On exit:
A$—User prompt	B$—User input string

This routine is simpler than the numeric routine used earlier because we can accept a wider range of characters than before. We have enhanced the

Table 2-5. Variable Table for Alphanumeric Input Routine

Variable	Type	Purpose	Used in lines
A	Numeric	Keypress character	325, 335, 340, 380, 385, 390
A	String	User prompt	305, 310
B	String	Input string	300, 345, 350, 355, 360, 365, 385, 390
C	Numeric	Prompt length	305, 355, 390
J	Numeric	String position	355, 365, 370
S	Numeric	Keyboard status	325, 330

input capabilities by allowing the user to enter commas, colons, quotes, and all other printable characters. While it may be possible to do this by having the user put quote marks around their responses, it is simpler and friendlier to take care of that with a routine like this.

So far, so good. We are getting the keypress, checking it, and adding it to our input string. It would be nice, however, to add a length check. This can be accomplished by adding one line of code:

```
381 IF LEN(B$)=UL THEN 320
```

This line will make sure that the length of the entry doesn't go beyond a pre-determined boundary. The length should be specified by setting UL to an upper limit before entering the routine. If the user tries to enter any character over the limit except a backspace or a RETURN, the bell will ring and the keypress is ignored.

Let's put the basic alphanumeric input routine to work in our payroll program. Now it's all starting to look pretty good:

LISTING 2-4

```
1 REM      LISTING 2-4
2 CALL CLEAR
90 GOTO 1000
100 B$=""
105 D=0
110 C=LEN(A$)+3
115 PRINT A$;
120 GOTO 130
125 CALL SOUND(100,1000,10)
130 CALL KEY(0,A,S)
135 IF S<>1 THEN 130
```

```
140 IF A=13 THEN 235
145 IF A<>8 THEN 195
150 IF LEN(B$)=0 THEN 125
155 IF SEG$(B$,LEN(B$),1)<>"." THEN 165
160 D=0
165 B$=SEG$(B$,1,LEN(B$)-1)
170 J=LEN(B$)-1+C
175 IF LEN(B$)=0 THEN 185
180 CALL HCHAR(24,J,ASC(SEG$(B$,LEN(B$),1)))
185 CALL HCHAR(24,J+1,32)
190 GOTO 130
195 IF (B$="")*(A=45)THEN 220
200 IF ((A<48)+(A>57))*(A<>46)THEN 125
205 IF (A=46)*(D=1)THEN 125
210 IF A<>46 THEN 220
215 D=1
220 B$=B$&CHR$(A)
225 CALL HCHAR(24,C+LEN(B$)-1,A)
230 GOTO 130
235 B=0
240 IF (B$="")+(B$=".")THEN 250
245 B=VAL(B$)
250 RETURN
300 B$=""
305 C=LEN(A$)+3
310 PRINT A$;
315 GOTO 325
320 CALL SOUND(100,1000,10)
325 CALL KEY(0,A,S)
330 IF S<>1 THEN 325
335 IF A=13 THEN 400
340 IF A<>8 THEN 380
345 IF LEN(B$)=0 THEN 320
350 B$=SEG$(B$,1,LEN(B$)-1)
355 J=LEN(B$)-1+C
360 IF LEN(B$)=0 THEN 370
365 CALL HCHAR(24,J,ASC(SEG$(B$,LEN(B$),1)))
370 CALL HCHAR(24,J+1,32)
375 GOTO 325
380 IF (A<32)+(A>126)THEN 320
385 B$=B$&CHR$(A)
390 CALL HCHAR(24,C+LEN(B$)-1,A)
395 GOTO 325
400 PRINT
405 RETURN
1000 A$="EMPLOYEE NAME: "
1010 GOSUB 300
1020 N$=B$
1030 A$="HOW MANY HOURS: "
1040 GOSUB 100
1050 H=B
1060 A$="WHAT RATE OF PAY: "
1070 GOSUB 100
1080 R=B
1090 P=H*R
1100 PRINT "CORRECT PAY FOR ";N$;" IS";P
9999 END
```

Now, if our Podunksville paymaster enters the name DOE, JOHN, there won't be a problem. The routines allow for this without generating an error message that would confuse the user.

Table 2-6. Variable Table for Payroll Program

Variable	Type	Purpose	Used in lines
A	Numeric	Keypress character	130, 140, 145, 195, 200, 205, 210, 220, 225, 325, 335, 340, 380, 385, 390
A	String	User prompt	110, 115, 305, 310, 1000, 1030, 1060
B	String	User input	100, 150, 155, 165, 170, 175, 180, 195, 220, 225, 240, 245, 300, 345, 350, 355, 360, 365, 385, 390, 1020
B	Numeric	String value	235, 245, 1020, 1050
C	String	Prompt length	110, 170, 225, 305, 355, 390
D	Numeric	Decimal flag	105, 205, 215
H	Numeric	Hours	1050, 1090
J	Numeric	String position	170, 180, 185, 355, 365, 370
N	String	Employee name	1020, 1100
P	Numeric	Pay	1090, 1100
R	Numeric	Rate	1080, 1090
S	Numeric	Keyboard status	130, 135, 325, 330

INPUT CONCLUSIONS

Now, a final comment or two—life requires constant tradeoffs. This is a good place to give an example of one of those little tradeoffs.

Since we have gained so much latitude with regard to user input, we have to give up something somewhere, right? Right! By using input routines such as those listed, we lose some program speed. The user won't usually notice it when typing an entry, but the slow-down is there, nonetheless.

Because so much string manipulation is being done with each keypress, your TI-99/4A has quite a bit of program overhead to go through. This

shouldn't be a big problem, unless your program is very long or uses quite a bit of memory.

So much for the trade-off. In our next chapter we will look at how to round numbers. You may finally get to discover why all your teachers stressed math.

Chapter 3

Rounding

Rounding is the art of making numbers come out to the nearest "something." Rounding is usually used with money figures, and that "something" can be the nearest dollar, penny, or fraction of a penny.

There are many other uses for rounding. For example, you may want to round a number to the nearest hundred for a calculation, or, when calculating disk space from within a configuration program, you may want to round down to make sure that there is enough room on the disk for the records.

The rounding technique is a simple one. The basic rounding equation looks like this:

```
N = INT( N / D + K ) * D
```

There are only three variables used—N, D, and K. Variable N is the number we want to round. It should be set to a value before entering this formula, and after exiting it will be equal to the rounded value that we want.

Variable D is the divisor and multiplier because it serves as both at different times within the equation. This variable determines to how many decimal places, or to what precision, N will be rounded. In a moment we will look at the effect that various values of D have on N.

Variable K is the "kicker". That sounds like a descriptive term, doesn't it? Well, that is exactly what it does. It "kicks" our number either up, down, or to the nearest figure. Its effect will be seen in the examples.

To use this formula in a program, we would first need to decide to how many places we wanted our numbers rounded. If we were only working with dollars and cents, this would be a simple decision; we would need to round our figures to two decimal places. This, again, determines what we would use for the value of D in our formula.

Some common values for D, based on rounding to powers of 10, are shown in Table 3-1.

Table 3-1. Rounding Divisors & Multipliers

Divisor/ Multiplier	Result rounded to nearest:
0.0001	4 decimal places
0.001	3 decimal places
0.01	2 decimal places
0.1	1 decimal place
1	whole number
10	ten
100	hundred
1000	thousand
10000	ten thousand

For dollars and cents, use a value of .01 for D. For K, there are usually only 3 different values used when rounding to powers of 10. When K is equal to 1, we are rounding up. If K is equal to .5, we are rounding to the nearest figure. If K is equal to 0, we are rounding down. Notice that if K was equal to 0, it could be left out of the equation completely.

ROUNDING TO THE NEAREST FIGURE

To round a dollar and cents figure to the nearest penny, we could use the following line of code:

```
100 N = INT( N / .01 + .5 ) * .01
```

Instead of using a line of BASIC code similar to the one listed above, we could just as easily have used the DEF FN statement. This would allow us to define our rounding formula as an integral function, as follows:

```
100 DEF RD( N ) = INT( N / .01 + .5 ) * .01
```

The advantage is that you can later call this by simply setting a number equal to RD(x), where you substitute the number to be rounded for x. For instance, such a statement could look like this:

```
200 A = RD(A)
```

Both routines, whether as a callable subroutine, or as an independent function, will round the number. If you follow through on the logic, the routine takes a number that you feed to it (within the variable N), and then rounds it. Perhaps it is best to show an example.

First we substitute an unrounded dollar figure (157.19832 in this example) for the variable N. Our equation looks like this:

```
N = INT(157.19832 / .01 +.5) * .01
```

Next, by performing the calculation within the parentheses (remember that within the computer, division will be done before addition), our formula will look like this:

```
N = INT(15720.332) * .01
```

Next, the INT function is executed. This function will truncate all the numbers following the decimal point of the argument value. Therefore, during the next step the formula looks like this:

```
N = 15720 * .01
```

Multiplying the number by .01 is the last operation. The answer is:

```
N = 157.2
```

Upon completion, our number is rounded to the nearest penny. This is great for printing dollar figures without trailing fractions of pennies that only the pickiest accountants would be interested in anyway. In a later chapter, we will discuss ways of formatting the rounded numbers so they look nice and neat.

ROUNDING UP

Using the dollars and cents example, we see that it is easy to round up. The statement to do this is:

```
100 N = INT(N / .01 + 1) * .01
```

This routine will always round a number to the next highest penny. Great for bankers figuring interest due, huh? Well, the proof, using the previous example, works like this:

```
N = INT ( 157.19832 / .01 + 1 ) * .01
N = INT ( 15719.832 + 1 ) * .01
N = INT ( 15720.832 ) * .01
N = 15720 * .01
N = 157.2
```

This works fine with this example, but what about a number that normally would have been rounded down? Let's examine it using 12.00001103 in the formula:

```
N = INT ( 12.00001103 / .01 + 1 ) * .01
N = INT ( 1200.001103 + 1 ) * .01
N = INT ( 1201.001103 ) * .01
N = 1201 * .01
N = 12.01
```

Again, our figure, although just slightly over $12.00 to begin with, was rounded to the next highest cent.

ROUNDING DOWN

Rounding down uses the same procedure as rounding up, except there is no kicker added to the formula. The kicker in the above equation, the number 1, was added to force a round up. With no kicker, the line of code to round down would look like this:

```
100 N = INT ( N / .01 ) * .01
```

As a proof, let's use our original number, since that is a number that is usually rounded up:

```
N = INT ( 157.19832 / .01 ) * .01
N = INT ( 15719.832 ) * .01
N = 15719 * .01
N = 157.19
```

The number has been rounded down to the next lowest penny.

So far, we have discussed rounding to a penny, whether it be nearest, up, or down. This is also referred to as rounding to two decimal places. You can round to more (or fewer) decimal places by simply increasing (or reducing) the number of zeros in the divisor, and consequently the multiplier.

ODD ROUNDING

In our examples thus far, we have rounded only to powers of 10. We could just as easily round to a different value to achieve different results. For instance, you may want (for some unknown reason) to round to the nearest 50, or to the nearest 25, or to the nearest 30. All it takes is using slightly different numbers in the formula.

If you want to experiment with different values for rounding, try the following short program. It will run through a few quick rounding formulas:

Listing 3-1

```
1 REM      LISTING 3-1
2 CALL CLEAR
10 FOR J=1 TO 100 STEP .26
20 RESTORE
30 FOR K=1 TO 6
40 READ D
50 N1=INT(J/D+1)*D
60 N2=INT(J/D+.5)*D
70 N3=INT(J/D)*D
80 PRINT " NUMBER:";J
90 PRINT " FACTOR:";D
100 PRINT "      UP:";N1
110 PRINT "    DOWN:";N2
120 PRINT "NEAREST:";N3
130 PRINT
140 NEXT K
150 PRINT
160 PRINT
170 NEXT J
180 DATA .001,.01,.1,3,10,11
190 END
```

Table 3-2. Variable Table for Rounding Example

Variable	Type	Purpose	Used in lines
D	Numeric	Unrounded number	40, 50, 60, 70, 90
J	Numeric	Loop counter	10, 50, 60, 70, 80, 170
K	Numeric	Loop counter	30, 140
N1	Numeric	Rounded number	50, 100
N2	Numeric	Rounded number	60, 110
N3	Numeric	Rounded number	70, 120

When you run the program, it goes by on your screen rather quickly, so you may want to modify it to print only a portion of the output at a time. If this type of program really does something for you, you could change the print statements to make the output go to your printer so that you will have a copy of various numbers rounded in different ways.

The program can also be changed by using different step values in line 10, or by using different DATA elements to experiment with different rounding capabilities.

As a final note, even computers are not perfect. The TI-99/4A does strange things with some numbers, as you may notice on your output. Because of the internal representation of some numbers, they are never quite derived in the expected manner. Do not despair! We all have to work within the limits of a general-purpose computer. Such is life.

Chapter 4

Dollars and Cents

How to make sense of dollars and cents? What a perplexing question. In this chapter, you will see how to format your numeric output.

First of all, if you haven't noticed, numbers come in all shapes and sizes. There are big numbers (1.5623 E 23), small numbers (1.5623 E -23), and some in between (Ø). All of these numbers are useful, and indeed necessary for various applications, but they need to be formatted for a professional appearance on reports, as well as the video screen.

The biggest problem is the numbers to the right of the decimal point, because when your decimal points are not in line, it looks like a chicken tracked all over your report. This can be very disconcerting. Well, never fear! There is a solution.

If you ask a non-TI-99/4A user for the solution, they will probably laugh smugly and point to their version of BASIC which has formatted output built in. Well, the TI-99/4A doesn't, and that's that, and you have to work with what you have.

The first rule of printing numbers is to not print numbers—print only strings because it is much easier to format and position strings than numbers. For instance, the following program prints just numbers. Try running it, and take a look at what is displayed:

```
100 FOR J = 1 TO 50 STEP 5.25
110 PRINT J
120 NEXT J
```

This program prints output all over the place! Fig. 4-1 shows what you should have seen on your video screen. We got all the numbers we asked for, but they would have looked nicer lined up in standard column order. There is no intrinsic way to accomplish this on the TI-99/4A.

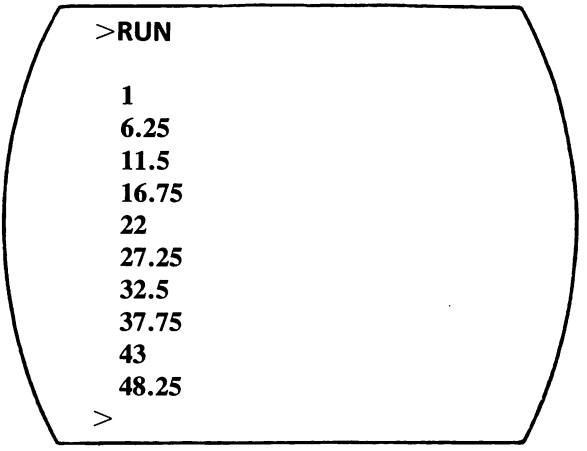

Fig. 4-1.

FORMATTED OUTPUT

The first step when creating a formatting routine is to convert all numbers to strings, and then print the strings. The strings should all be a uniform length, and each number should be right-justified within the string. This is not as difficult as it may sound. Our program, with the following subroutine, does all of this:

Listing 4-1

```
1 REM      LISTING 4-1
2 CALL CLEAR
5 GOTO 100
10 N=INT(N/.01+.5)*.01
20 A$=SEG$("          "&STR$(N),LEN(STR$(N))+1,10)
30 RETURN
100 FOR J=1 TO 50 STEP 5.25
110 N=J
120 GOSUB 10
130 PRINT A$
140 NEXT J
```

Table 4-1. Entry and Exit Variable Table for Dollars & Cents Routine Example

On entry:	On exit:
N—Number to be rounded and formatted	A$—Formatted string N —Rounded number

Table 4-2. Variable Table for Dollars & Cents Routine Example

Variable	Type	Purpose	Used in lines
A	String	Formatted output	20, 130
J	Numeric	Loop counter	100, 110, 140
N	Numeric	Work number	10, 20, 110

Also included is a rounding function in line 10 (see Chapter 3) that rounds the numbers to no more than two decimal places. When this program is run, your screen should appear similar to Fig. 4-2. The figures are lined up on the right, and each figure ends in the tenth column on the screen. This is a very good start. The only remaining problem is that the numbers are still not lined up by decimal points, because each number does not have the same number of digits to the right of the decimal point. We need a way to make sure that trailing zeros are included in each number so that two decimal places are displayed.

```
>RUN

          1
       6.25
       11.5
      16.75
         22
      27.25
       32.5
      37.75
         43
      48.25
>
```

Fig. 4-2.

Let's change this routine slightly, and see if this can be straightened out.

Listing 4-2

```
1 REM      LISTING 4-2
2 CALL CLEAR
5 GOTO 100
10 N=INT(N/.01+.5)*.01
15 A$=STR$(N)
20 IF N<>0 THEN 30
25 A$=""
30 IF INT(N)<>0 THEN 40
35 A$="0"&A$
40 IF N<>INT(N)THEN 55
45 A$=A$&".00"
50 GOTO 65
55 IF ASC(SEG$(A$,LEN(A$)-2,1))=46 THEN 65
60 A$=A$&"0"
65 A$=SEG$("          "&A$,LEN(A$)+1,10)
70 RETURN
100 FOR J=1 TO 50 STEP 5.25
110 N=J
120 GOSUB 10
130 PRINT A$
140 NEXT J
```

Table 4-3. Entry and Exit Variable Table for Dollars & Cents Routine Example

On entry:	On exit:
N—Number to be rounded and formatted	A$—Formatted string N —Rounded number

Table 4-4. Variable Table for Dollars & Cents Routine Example

Variable	Type	Purpose	Used in lines
A	String	Formatted output	15, 25, 35, 45, 55, 60, 65, 130
J	Numeric	Loop counter	100, 110, 140
N	Numeric	Work number	10, 15, 20, 30, 40, 110

We added a few lines to make the routine more useful. Line 15 converts N to a string (A$), then the program checks to see if N is a zero. If it is, A$ is set to zero so that all of the following manipulations will work correctly.

Next, in line 30, the program checks if N is between zero (inclusive) and one. If it is, a leading zero is added to the string. In line 40, the

program checks if N is a whole number. If it is, then our job is simple. All that remains to be done is to add the decimal point and two zeros. Then execution skips to line 65 where the string is filled out to 10 spaces.

If N is not a whole number, we should only have to add, at most, one zero. However, the number may already have two significant digits. It would be counter-productive to add a zero to a number that already is justified to two decimal places. Line 55 checks to see if the third character from the right is a decimal point. The ASCII value for a decimal point is 46. If it is a decimal point (as is the case with a number already containing two significant digits), execution branches to line 65.

However, if there is only one significant digit, a zero is added to the end of the string to make sure that there are two places to the right of the decimal point.

All routine execution goes through line 65, which adds enough spaces to the left of the string to make sure that it is 10 characters long. This right-justifies the number within the string.

When this program is run, as shown in Fig. 4-3, the output is in columns. After all, isn't that what we wanted in the first place?

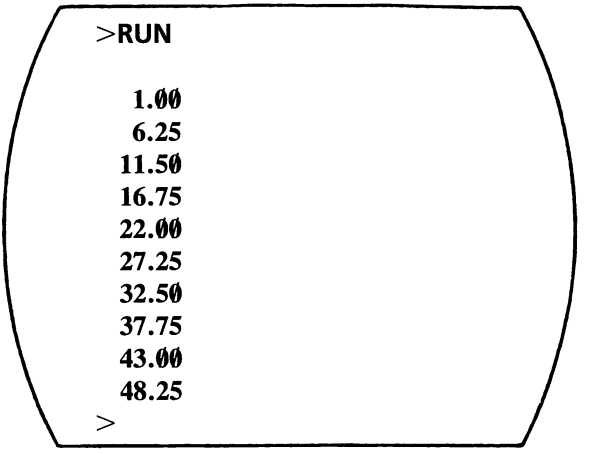

Fig. 4-3.

For most applications, you can use this routine with no modifications. However, if you will be processing numbers greater than 9999999.99, make your formatted strings longer than 10 characters. Simply change line 35 to reflect a string size larger than 10.

Other special cases are processing negative numbers, or formatting numbers using commas. Read on.

FORMATTING NEGATIVE NUMBERS

Negative numbers can be formatted in one of two ways. The negative sign can either precede or trail the number. Both methods are acceptable for reports.

To format with a preceding negative sign, the above routine will work fine if the number does not exceed 10 places, including the negative sign. When we convert the number to a string, the sign is converted as well. By using this routine with no modification, the smallest negative number that could be used is -999999.99. Again, if you need to allow for smaller numbers, simply change line 65 to reflect a larger string.

If you want trailing negative signs, a change will need to be made to this routine. This is done by using a sign string and an absolute value of the number to be formatted, as shown below:

Listing 4-3

```
1 REM   LISTING 4-3
2 CALL CLEAR
5 GOTO 100
10 N=INT(N/.01+.5)*.01
11 NS$=" "
12 IF N>=0 THEN 15
13 NS$="-"
14 N=ABS(N)
15 A$=STR$(N)
20 IF N<>0 THEN 30
25 A$=""
30 IF INT(N)<>0 THEN 40
35 A$="0"&A$
40 IF N<>INT(N)THEN 55
45 A$=A$&".00"
50 GOTO 65
55 IF ASC(SEG$(A$,LEN(A$)-2,1))=46 THEN 65
60 A$=A$&"0"
65 A$=SEG$("          "&A$,LEN(A$)+1,9)&NS$
70 RETURN
100 FOR J=-25 TO 25 STEP 5.25
110 N=J
120 GOSUB 10
130 PRINT A$
140 NEXT J
```

Table 4-5. Entry and Exit Variable Table for Dollars & Cents Routine Example

On entry:	On exit:
N—Number to be rounded and formatted	A$—Formatted string N —Rounded number

Table 4-6. Variable Table for Dollars & Cents Routine Example

Variable	Type	Purpose	Used in lines
A	String	Formatted output	15, 25, 35, 45, 55, 60, 65, 130
J	Numeric	Loop counter	100, 110, 140
N	Numeric	Work number	10, 12, 14, 15, 20, 30, 40, 110
NS	String	Sign string	11, 13, 65

We added lines 11 through 14 and changed line 65. The additional lines set a negative sign equal to a space or a minus sign, depending on whether N is less than 0 or not. If N is less than zero, it is changed to a positive number for the remainder of the calculations by using the ABS function.

In line 65, the sign string is appended to the formatted string. If the number is negative it will have a trailing minus sign.

Note that in line 65 we reduced, by one digit, the size of the number that could be sucessfully formatted. This allows the sign string to fill up the format to 10 characters. It is just as easy to use any other size string by changing 9 to some other number.

If you run this routine, the output would look similar to Fig. 4-4.

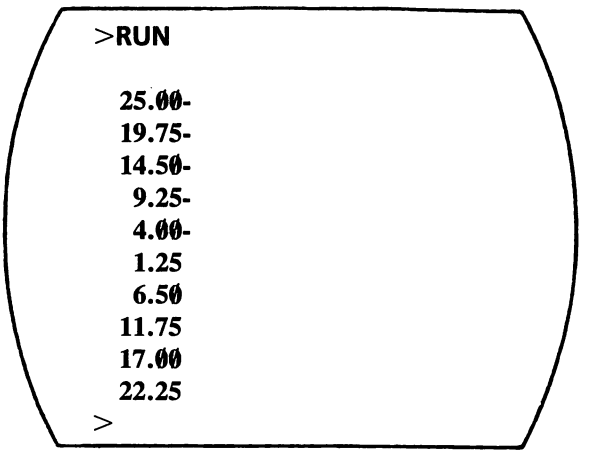

Fig. 4-4.

FORMATTING WITH COMMAS

Commas in formatted numbers have a good side and a bad side. They look great on some reports, but they can be a real bear to use and, depending on the size of the numbers being formatted, they may slow down the processing because of the amount of string manipulation that's needed.

To include commas, it is best to drastically alter our subroutine to allow separate processing of the decimal and whole parts of the number. This makes it easier to insert commas. The subroutine, when written the following way, does this rather nicely:

Listing 4-4

```
1 REM      LISTING 4-4
2 CALL CLEAR
5 GOTO 100
10 N=INT(N/.01+.5)*.01
13 NS$=" "
16 IF N>=0 THEN 25
19 NS$="-"
22 N=ABS(N)
25 N1=INT(N)
28 N2=N-N1
31 W$=STR$(N1)
34 W1$=""
37 IF N1<>0 THEN 43
40 W$="0"
43 D$=STR$(N2)
46 IF N2<>0 THEN 52
49 D$=".00"
52 IF LEN(D$)>2 THEN 58
55 D$=D$&"0"
58 IF LEN(W$)<4 THEN 70
61 W1$=","&SEG$(W$,LEN(W$)-2,3)&W1$
64 W$=SEG$(W$,1,LEN(W$)-3)
67 GOTO 58
70 W1$=W$&W1$
73 A$=SEG$("          "&W1$&D$,LEN(W1$&D$)+1,9)&NS$
76 RETURN
100 FOR J=-10000 TO 10000 STEP 1500.25
110 N=J
120 GOSUB 10
130 PRINT A$
140 NEXT J
```

Table 4-7. Entry and Exit Variables for Dollars & Cents with Commas Routine

On entry:	On exit:
N—Number to be formatted	A$—Formatted output N —Rounded number

Table 4-8. Variable Table for Dollars & Cents with Commas Routine

Variable	Type	Purpose	Used in lines
A	String	Formatted output	73, 130
D	String	Temporary string	43, 49, 52, 55, 73
J	Numeric	Loop counter	100, 110, 140
N	Numeric	Work number	10, 16, 22, 25, 28, 110
N1	Numeric	Integer value	25, 28, 31, 37
N2	Numeric	Integer value	28, 43, 46
NS	String	Sign string	13, 19, 73
W	String	Temporary string	31, 40, 58, 61, 64, 70
W1	String	Temporary string	34, 61, 70, 73

This is the super-duper do-it-all routine that inserts commas, justifies, and adds trailing negative signs if necessary. Lines 58 through 67 add commas by successively breaking apart the whole string into three-digit parts and adding them to the target string. In line 73 the formatting is completed.

Because we are still limiting our output to a total of nine absolute digits (the tenth is used for the sign), the largest number we can format with this routine is 99999.99. Anything above that will be truncated on the left, with only the nine right digits showing. If you anticipate using larger numbers, increase the string limiter in line 73 from a 9 to a larger digit.

We have also changed lines 100 through 130 to use larger numbers and negatives, to show the full formatting potential of the routine. When the program is run, the output should be similar to Fig. 4-5. In the next chapter, we will use this technique to format report lines.

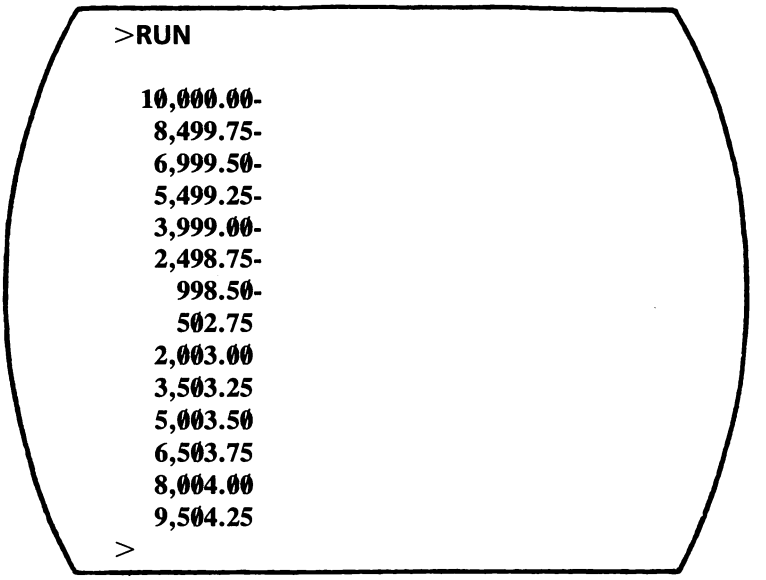
>RUN
10,000.00-
8,499.75-
6,999.50-
5,499.25-
3,999.00-
2,498.75-
998.50-
502.75
2,003.00
3,503.25
5,003.50
6,503.75
8,004.00
9,504.25
>

Fig. 4-5.

Chapter 5

Report Formatting

Report formatting is an important aspect in the success of a program. Usually, the first contact a person has with a program is the written output, and the readability of that output may color their feelings about your program.

If the report that your program generates is either too short or too long, you have failed the user. The report should adequately cover the information needs of the user without burying him in a mountain of useless numbers or paper.

Many times the word "report" conjures up visions of a stack of paper or printers disgorging numbers at an ever increasing rate. Reports, however, are not limited to hardcopy (paper) output. They can be printed either to the screen or to the printer. It is a big plus for a program if it handles both types of output because it shows planning on the part of the programmer. Your program should have separate sections for both screen and printer output. This is because screen reports only need to be formatted for 28 column lines, while printed reports generally need to be formatted for 80 or 132 column lines.

STANDARDIZATION

Reports consist of two general parts—headings, which appear on each page, and individual lines that make up the body of the report. We will cover each area in detail.

```
                                    GOLIATH WIDGET CORPORATION
Week Ending 02/31/1984                                Prepared by

                                                  ----------
Line  Metropolitan Location          Date      10000     11000
-------------------------------------------------------------
```

Fig. 5-1. Sample 132-column report heading.

It is easier to read output, both video and printed, that is standardized. This means that all associated output begins at a certain column, or that the output is all left or right justified. Justification is the process used to make text either begin at a specified left margin (left justification), or end at a specified right margin (right justification), or both (fill justification). It is beyond the scope of this book to discuss fill justification, so we will only discuss the first two topics, as well as standardization.

The first step, when planning your program (you do plan your programs, don't you?), is to decide on the output format. When planning a report, there are many questions that need to be answered:

1. What should appear on the heading?
2. Should the heading be centered?
3. Should a date and page numbers be included?
4. What size paper will this be printed on?
5. How wide is your report going to be?
6. How many lines for the heading?
7. How many columns?
8. What should the columns contain?
9. What order should the data be presented in?
10. What should the column headings say?
11. Should the report be single or double spaced?
12. How many blank lines should there be at the top of the screen?
13. How many blank lines should there be at the bottom of the screen?

As you can see, there are quite a few questions to consider, and your needs may raise questions that were not touched on here. It is best to answer such questions in the planning stage of the program. The answers that you give may vary considerably from report to report, depending on factors such as the complexity of the report subject, value of the report, and intended report audience.

Because these are individual questions, we will only consider the two basic components of a report. You can then use those sections that are needed for your reports.

```
--------- WEEKLY SUMMARY RECAPITULATION
          Johnathan Doe                                          Page 1

-------------------- Accounts -----------------------------
          12000     17500     20100     23000     24000     25100     Comments
-------------------------------------------------------------------------------
```

REPORT HEADINGS

Report headings are the lines that identify the report. The heading should contain information that will help the reader understand the contents of the report. Generally, a heading contains several standard items—a title, date, page number, column headings, and divider. A sample report heading for a 132-column report appears in Fig. 5-1. Each part of the heading is distinct and easily recognized by the reader.

The placement of these items in a heading depends, in large measure, on the report's width. On narrow reports, you may want your heading several lines deep. On a wide report, the heading may be only one line. Most headings are a total (counting blank lines) of approximately 5 to 6 lines. Again, this may vary under any given conditions. For instance, the report heading in Fig. 5-2 is the same as that referred to earlier, but it has been formatted for an 80-column report.

```
                GOLIATH WIDGET CORPORATION - WEEKLY SUMMARY RECAP
Week Ending 02/31/1984                                          Page 1
--------------------------------------------------------------------------------
Line   Location          Date      Account      Amount       Comments
--------------------------------------------------------------------------------
```

Fig. 5-2. Sample 80-column report heading.

Printing the heading should be handled by a separate subroutine so it can be easily called at the beginning of each page of output. A typical heading subroutine might look like this:

Listing 5-1

```
1 REM       LISTING 5-1
50 P=P+1
55 CALL CLEAR
60 PRINT TAB(2);"SAMPLE TECHNICAL REPORT"
65 PRINT DATE$;
70 PRINT TAB(20);"PAGE";P
75 PRINT :HD$
80 FOR J=1 TO 28
85 PRINT "=";
90 NEXT J
95 PRINT ::
100 LC=6
105 RETURN
```

Table 5-1. Entry and Exit Variables for Report Headings Routine

On entry:	On exit:
DATE$—Date for report	LC—Number of lines printed
HD$ —Column headings	P —New page value
P —Last page #	

Table 5-2. Variable Table for Report Headings Routine

Variable	Type	Purpose	Used in lines
DATE	String	Date	65
HD	String	Column headings	75
J	Numeric	Loop counter	80, 90
LC	Numeric	Line counter	100
P	Numeric	Page number	50, 70

This routine does several important things. First, it increments the page counter in line 50 to produce a consecutive page number on each page of the report. At some point before entering this heading routine for the first time, you will need to make sure that variable P gets set to zero so that the first page will always be 1. Generally, it is a good idea to begin your reports with page one so that the reader doesn't think that you are only providing a partial report.

The next item handled is the report title. This is centered for 80 columns by using the TAB command in the print statement. For instance, **PRINT TAB(10); A$** will cause the variable A$ to be printed beginning at the tenth column of the output line.

Centering the report title gives it added importance and draws the reader's attention to the top and middle of the page. In addition, the title of the report is important enough that the reader should not have to search for it. Here we have both centered the title, as well as given it a full line.

The next line contains only two items, but for most reports, they are only slightly less important than the title. The date appears to the left side of the line, and the page number appears to the right. The page number is set each time the heading routine is accessed, but the date will have to be set before entering the routine.

Next is a blank line and the column headers. The variable HD$ was used for the headers line, but this could be easily changed to individual

print commands. In general, I like to use the string method because it makes it easier to change the headings, and the routine will look less cluttered.

Before entering this routine, HD$ needs to be set equal to the column headings that you want printed. Grab your pencil and paper and write out the column headings you want. Be sure to space correctly between the columns, and then type it into the program. Chances are, your column heading line will need to be changed later, so don't throw your notes away yet.

Finally, there is a dividing line of equal signs and then a blank line. These separate the heading from the rest of the report.

The last part of the subroutine sets the line counter to six, the number of individual lines that have been printed by this routine. What this counter is set to will vary with the number of lines printed by your heading subroutine. The importance of this in the overall report will be discussed shortly.

None of the routines presented here are cast in concrete, and they can be changed to reflect your specific needs. The important thing is that you develop a routine that will print clear, concise report headings.

After working out, in theory, your heading routine, it is best to make a few test runs printing only the heading. You can then determine if it will fit your needs, and you can make changes before you are too far into actual report development. After all, let's face the fact that if we get something to work the first time around, it usually means we have overlooked an obvious flaw that will make the program bomb later. This is just an irrefutable law of programming.

When your report headings are completed to your satisfaction, it is time to proceed to the body of the report.

LINE PREPARATION

The body of any report is usually made up of individual data lines, repeated over and over again.

Preparing those lines for output is simply a matter of collecting the data you want to print, and printing it at the right position. To illustrate this, we need to pull together a program that will include all the subroutines we have developed up to this point. This program will allow a user to input individual payroll information, and print out the resultant information in an 8Ø-column format. This will not be your basically marketable (or even useful) payroll program. However, it will illustrative. The program would look like this:

Listing 5-2

```
1 REM      LISTING 5-2
2 CALL CLEAR
10 HD$="EMPLOYEE NAME                HOURS          RA
   TE              PAY"
20 OPEN #1:"RS232.BA=9600.DA=8",OUTPUT
90 GOTO 1000
100 B$=""
105 D=0
110 C=LEN(A$)+3
115 PRINT A$;
120 GOTO 130
125 CALL SOUND(100,1000,10)
130 CALL KEY(0,A,S)
135 IF S<>1 THEN 130
140 IF A=13 THEN 235
145 IF A<>8 THEN 195
150 IF LEN(B$)=0 THEN 125
155 IF SEG$(B$,LEN(B$),1)<>"." THEN 165
160 D=0
165 B$=SEG$(B$,1,LEN(B$)-1)
170 J=LEN(B$)-1+C
175 IF LEN(B$)=0 THEN 185
180 CALL HCHAR(24,J,ASC(SEG$(B$,LEN(B$),1)))
185 CALL HCHAR(24,J+1,32)
190 GOTO 130
195 IF (B$="")*(A=45)THEN 220
200 IF ((A<48)+(A>57))*(A<>46)THEN 125
205 IF (A=46)*(D=1)THEN 125
210 IF A<>46 THEN 220
215 D=1
220 B$=B$&CHR$(A)
225 CALL HCHAR(24,C+LEN(B$)-1,A)
230 GOTO 130
235 B=0
240 IF (B$="")+(B$=".")THEN 250
245 B=VAL(B$)
250 RETURN
300 B$=""
305 C=LEN(A$)+3
310 PRINT A$;
315 GOTO 325
320 CALL SOUND(100,1000,10)
325 CALL KEY(0,A,S)
330 IF S<>1 THEN 325
335 IF A=13 THEN 400
340 IF A<>8 THEN 380
345 IF LEN(B$)=0 THEN 320
350 B$=SEG$(B$,1,LEN(B$)-1)
355 J=LEN(B$)-1+C
360 IF LEN(B$)=0 THEN 370
365 CALL HCHAR(24,J,ASC(SEG$(B$,LEN(B$),1)))
370 CALL HCHAR(24,J+1,32)
375 GOTO 325
380 IF (A<32)+(A>126)THEN 320
385 B$=B$&CHR$(A)
390 CALL HCHAR(24,C+LEN(B$)-1,A)
395 GOTO 325
400 PRINT
405 RETURN
445 REM      ROUNDING ROUTINE
```

```
450 N=INT(N/.01+.5)*.01
455 RETURN
495 REM       ROUTINE TO CONVERT NUMBERS FOR OUTPUT
500 N=INT(N/.01+.5)*.01
505 NS$=" "
510 IF N>=0 THEN 525
515 NS$="-"
520 N=ABS(N)
525 N1=INT(N)
530 N2=N-N1
535 W$=STR$(N1)
540 W1$=""
545 IF N1<>0 THEN 555
550 W$="0"
555 D$=STR$(N2)
560 IF N2<>0 THEN 570
565 D$=".00"
570 IF LEN(D$)>2 THEN 580
575 D$=D$&"0"
580 IF LEN(W$)<4 THEN 600
585 W1$=","&SEG$(W$,LEN(W$)-2,3)&W1$
590 W$=SEG$(W$,1,LEN(W$)-3)
595 GOTO 580
600 W1$=W$&W1$
605 A$=SEG$("           "&W1$&D$,LEN(W1$&D$)+2,9)&NS$
610 RETURN
695 REM       ROUTINE TO PRINT REPORT HEADINGS
700 P=P+1
710 PRINT #1:TAB(28);"PAYROLL WORKSHEET REPORT"
715 PRINT #1:DATE$;TAB(72);"PAGE";P
720 PRINT #1::HD$
730 FOR J=1 TO 80
735 PRINT #1:"=";
750 NEXT J
755 PRINT #1::
760 LC=6
765 RETURN
795 REM       ROUTINE TO PRINT INDIVIDUAL REPORT LINE
800 PRINT #1:N$;
805 I=LEN(N$)
810 FOR J=I TO 24
815 PRINT #1:" ";
820 NEXT J
825 N=H
830 GOSUB 500
835 PRINT #1:A$;"      ";
840 N=R
845 GOSUB 500
850 PRINT #1:A$;"      ";
855 N=PA
860 GOSUB 500
865 PRINT #1:A$
870 LC=LC+1
875 IF LC<60 THEN 890
880 PRINT #1:CHR$(12)
885 GOSUB 700
890 RETURN
995 REM       MAIN CONTROL ROUTINES
1000 P=0
1010 DATE$="01/01/99"
1020 GOSUB 700
1030 A$="EMPLOYEE NAME: "
1040 GOSUB 300
```

cont. on next page

Listing 5-2—cont.

```
1050 N$=B$
1060 IF B$="" THEN 1170
1070 A$="HOW MANY HOURS: "
1080 GOSUB 100
1090 H=B
1100 A$="WHAT RATE OF PAY: "
1110 GOSUB 100
1120 R=B
1130 PA=H*R
1140 GOSUB 800
1150 PRINT ::
1160 GOTO 1030
1170 PRINT #1:CHR$(12)
1175 CLOSE #1
9999 END
```

Table 5-3. Variable Table for Payroll Program

Variable	Type	Purpose	Used in lines
A	Numeric	Keypress character	130, 140, 145, 195, 200, 205, 210, 220, 225, 325, 335, 340, 380, 385, 390
A	String	User prompt & return string	110, 115, 305, 310, 605, 835, 850, 865, 1030, 1070, 1100
B	Numeric	String value	235, 245, 1090, 1120
B	String	User input	100, 150, 155, 165, 170, 175, 180, 195, 220, 225, 240, 245, 300, 345, 350, 355, 360, 365, 385, 390, 1050, 1060
C	String	Prompt length	110, 170, 225, 305, 355, 390
D	Numeric	Decimal flag	105, 205, 215
D	String	Temporary	555, 565, 570, 575, 605
DATE	String	Date	715, 1010
H	Numeric	Hours	825, 1090, 1130
HD	String	Headers	10, 720
I	Numeric	Temporary	805, 810

Table 5-3—cont. Variable Table for Payroll Program

Variable	Type	Purpose	Used in lines
J	Numeric	String position & loop counter	170, 180, 185, 355, 365, 370, 730, 750, 810, 820
LC	Numeric	Line counter	760, 870, 875
N	Numeric	Operation number	450, 500, 510, 520, 525, 530, 825, 840, 855
N	String	Employee name	800, 805, 1050
N1	Numeric	Integer value	525, 535, 545
N2	Numeric	Integer value	530, 555, 560
NS	String	Sign indicator	505, 515, 605
P	Numeric	Page counter	700, 715, 1000
PA	Numeric	Pay	855, 1130
R	Numeric	Rate	840, 1120, 1130
S	Numeric	Keyboard status	130, 135, 325, 330
W	String	Temporary string	535, 550, 580, 585, 590, 600
W1	String	Temporary string	540, 585, 600, 605

If you run this program you may be surprised, because it does a nice job of printing a usable report, and it even handles multiple pages. Most of the routines in this program are presented elsewhere in this book, so we will only concentrate on lines 800 through 899. These program lines print the individual lines that make up the body of the report.

The routines in this program were designed to direct output to the TI Impact Printer through a serial interface. If you are using a different type of interface or printer, you will need to make a change in line 20. This is the only place where the printer file is opened, so one change should do it all.

Line 800 prints the employee's name, and a FOR NEXT loop in lines 805 through 820 positions the cursor for printing the next variable. Then we convert the number we want printed to a formatted string, and print it. The balance of the routine repeats this process for the rate and the total pay. In line 870 we increment the line counter (L). If it is above a certain

number of lines (60 in this program) we execute our subroutine to do headings again for the next page.

This routine can be made more complex for other reports, depending on the data to be printed. Another change that could be implemented for other reports is a line that will keep a running total of the amounts printed. Then, when the printing of data is complete, you can print the totals for the report.

SPECIAL LINE POSITIONING

There is one last area about reports to cover. When printing information, it may be desirable to left justify, center, or right justify your output.

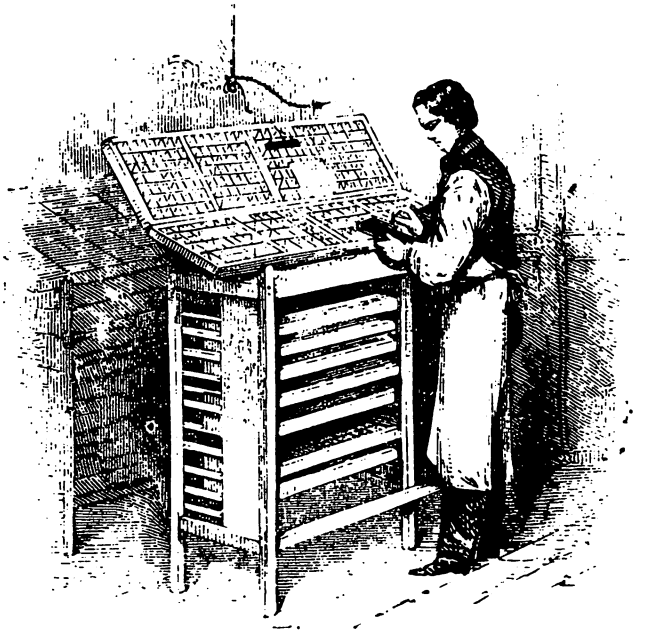

Fig. 5-3. Early manual line formatter.

Left justification is simple because all you need to do is position the cursor to the column where you want to begin printing, and then print it. If you are printing from the left column use the TAB statement, otherwise use a FOR NEXT loop to space correctly between fields on the same line. Simple, huh?

Well, centering variable length information is a little trickier, but if it's handled in a common subroutine it can be made simpler. As with all of our routines so far, you will need to be sure that you make the common variable (A$) equal to the string to be centered. The routine looks like this:

```
100  PRINT TAB(14 - LEN(A$) / 2); A$
110  RETURN
```

The heart of the routine is the formula **14 - LEN(A$) / 2.** Simply stated, this routine takes half the length of the string to be centered, subtracts it from the center point of the output line, and moves the cursor to that column. The string is then printed.

This centering routine is for the 28-column screen on the TI-99/4A; to make it work for any other size output line, all you need to do is change 14 to a number that represents half of the length of your output line. For example, use 40 for 80-column centering, 36 for 72-column centering, 66 for 132-column centering, etc.

This routine will not center a string that is longer than the line on which you want it centered. This will cause the program to try to TAB a negative amount. Since this can't be done, the TI-99/4A will begin printing at the leftmost line position.

Centering isn't too bad now, is it? Neither is right justification, as long as you approach it in the same manner. All right justification means is positioning each string to be printed so that it ends at a specific column. Here is a routine that will handle that:

```
100  PRINT TAB(27 - LEN(A$)); A$
110  RETURN
```

Simple, right? Right! All that needs to be done is to subtract the length of the string to be printed from a number that represents the column where you want the string to end.

I suppose it goes without saying that if you want the printing to end at a different place than column 27, all you need to do is change the number 27 to a number that represents the rightmost printing position. But I'll remind you anyway.

Well, now you know the deep, dark secrets about how to print reports and strings to get the output appearance that you want. All that needs to be done now is to arrange them in the order necessary to get the report that you want. The creative process that you go through may be the toughest part of programming.

In later chapters we will be discussing routines that can be helpful when formatting output for your report. Stick around, OK?

Chapter 6

Working with Dates

In this chapter, we will only be working with the kind of dates that determine our present position in relation to the continuum that stretches from yesterday through tomorrow.

Many programs need to work with dates—either inputting, manipulating, or printing them. We will discuss how to standardize dates, how to let the user input dates, how to manipulate them, and how to print them. First, however, we should give some general purpose information concerning dates.

DATES, CALENDARS, AND DATING CONVENTIONS

Our present calendar is the Gregorian calendar, named after Pope Gregory XIII. It was adopted by most of the world in 1582, and corrected errors in the previous (Julian) calendar. The transition between the two was rather abrupt, with 10 days dropped from the middle of October to correct differences between the sun's position and the calendar. After all, that had to be easier than moving the earth or sun to match the calendar.

So, October 5, 1582 became October 15, 1582, and most of the world adopted the new calendar system. Most, that is, except for certain Germanic nations that adopted it in approximately 1700. Great Britain did not

change until 1752, Russia until 1918, and Turkey was the last nation to switch in 1927.

The Gregorian calendar is based largely on the Julian calendar with 12 months of varying length. However, the new calendar corrected cumulative errors in the Julian calendar by changing the method of figuring leap year days. Instead of a leap year every 4 years, the Gregorian calendar allowed for a leap year every 4 years except in century years. In these years, there would be no leap year unless the year was divisible by 400. For example, 1900 was not a leap year, since it is not divisible by 400, but 2000 will be a leap year because it is.

A lot of the problems with checking and comparing dates has to do with leap years. Before discussing these problems, however, we should discuss standardization of dates.

DATE STANDARDIZATION

If you haven't noticed, nearly everyone uses a different notation for dates. For instance, each of the following signifies the same date:

August 18, 1983
18 August 1983
8/18/83
08/18/83
18/8/83
18/08/83

To complicate matters further, the year portion of each date could have four digits (1983) instead of two (83), or each date could have a different field separator, such as a hyphen (-) instead of the slash (/). If you allow the users of your program to input dates in any format they want, you will usually get one more format than you were able to think of. This, in turn, will cause your program to go bonkers.

I rest my case for date standardization. It should be obvious that if you are going to do any type of manipulation with dates, you need to have a standard that the user will follow.

The best way to proceed is to determine what the standard is for other programs that the user is running. It is frustrating to have to remember from program to program how to input dates. For a user, it should be as painless as possible to input information, particularly dates. Further, if the user is familiar with one type of notation, he is going to loathe any change. If possible, determine what the user wants and is used to doing.

If no standard for inputting dates is evident, it is up to you to determine

one. Remember, however, that your preference may not be the market preference.

Most computer programs use a MM/DD/YY or MM/DD/YYYY format for inputting dates. Computers work with numbers better than names of months, and allowing the user to spell out the name of the month will usually open the door to frustration because of possible misspellings.

INPUTTING DATES

Now that you have decided on a standard, let's look at a method of inputting dates. Since the MM/DD/YY format is the most common, we will use that in our discussions and examples. If you decide to use some other format, you will have to change the routine to reflect your format.

Basically, all we have to do is allow the user to input a string of characters, break them up to check their validity, and then put them back together in a standardized format.

The prompt that you present the user with should also show the format the program expects the response to follow. For instance, presenting the user with DATE (MM/DD/YY): leaves little room for doubt as to what is expected. The actual input routine then used should be rather selective. Chapter 2 presents a string input routine that will work just fine.

After getting the input from the user, the next step is to break it apart to check its validity. For instance, you may have someone who actually believes that there are 31 days in September. Since we all know that this only happens once every 327 years, we have to gently remind the user to re-enter the date. The following routine will break up the string to check for the validity of a date. It is then formatted to the standard (MM/DD/YY), and returns to the caller. On entry, A$ should contain the string to be analyzed. On a successful exit, MM contains the number of the month (1-12), DD contains the day (1-valid for month, e.g. 28, 29, 30, 31), and YY contains the year. The re-formatted date is also back in A$.

Listing 6-1

```
1 REM      LISTING 6-1
100 MM=0
105 DD=0
110 YY=0
115 K=LEN(A$)
120 IF (K<6)+(K>8)THEN 210
125 FOR J=1 TO K
130 IF (ASC(SEG$(A$,J,1))<48)+(ASC(SEG$(A$,J,1))>57)THE
    N 140
135 GOTO 150
```

cont. on next page

Listing 6-1—cont.

```
140 DD=YY
145 YY=J+1
150 NEXT J
155 IF DD=0 THEN 210
160 MM=VAL(SEG$(A$,1,DD-2))
165 DD=VAL(SEG$(A$,DD,YY-DD-1))
170 YY=VAL(SEG$(A$,YY,K-YY+1))
175 IF (MM<1)+(MM>12)THEN 210
180 IF (DD<1)+(DD>31)THEN 210
185 IF (MM=2)*((INT(YY/4)*4)<>YY)*((INT(YY/400)*400)<>Y
    Y)*(DD>28)THEN 210
187 IF (MM=2)*(DD>29)THEN 210
190 IF ((MM=4)+(MM=6)+(MM=9)+(MM=11))*(DD>30)THEN 210
195 A$=SEG$("00"&STR$(MM),LEN(STR$(MM))+1,2)&"/"&SEG$("
    00"&STR$(DD),LEN(STR$(DD))+1,2)&"/"
200 A$=A$&SEG$("00"&STR$(YY),LEN(STR$(YY))+1,2)
205 GOTO 215
210 A$=""
215 RETURN
```

Table 6-1. Entry and Exit Variables for Date Analysis Routine

On entry:	On exit:
A$—Preliminary date	A$ —Formatted date DD —Day MM—Month YY —Year

Table 6-2. Variable Table for Date Analysis Routine

Variable	Type	Purpose	Used in lines
A	String	Date (in & out)	115, 130, 160, 165, 170, 195, 200, 210
DD	Numeric	Day	105, 140, 155, 160, 165, 180, 185, 187, 190, 195
J	Numeric	Loop counter	125, 130, 145, 150
K	Numeric	Length of input	115, 120, 125, 170
MM	Numeric	Month	100, 160, 175, 185, 187, 190, 195
YY	Numeric	Year	110, 140, 145, 165, 170, 185, 200

There are several things to note about this routine. First, the length of A$ is checked upon entry. If it is less than 6 characters, or greater than 8 characters, the user has made an invalid date entry. This is because the smallest possible date in the MM/DD/YY format would be something similar to 1/1/83. This is only six characters. If the month and day were two characters each, the string would still only be 8 characters long. Anything over that doesn't fit the format.

If the date falls outside of these length specifications, execution branches to line 210. All our early exits from this routine will exit by way of line 210. You can test for a correct completion of this routine by checking to see if A$ is set to anything. If it isn't, you should have the program loop back to get the date again.

The next several instructions are used to break A$ apart so it can be completely analyzed. Lines 125 through 150 loop through the entire A$ variable to find the delimiters. A delimiter is a non-numeric divider between the components of the date. In the string 01/02/84, the slashes are delimiters, and this routine will find them. Then DD (day variable) is assigned equal to YY (year variable), and YY is set equal to the character position following the position of the delimiter within the string.

This process is repeated until the program reaches the end of the string. Upon exiting this loop, if DD is still equal to 0 we know the user did not enter a complete date. If this is the case, the routine is exited early by way of line 210.

Line 160 sets MM equal to the first part of the string, which should be the month. Lines 165 and 170 set DD and YY equal to the value of the two consecutive characters following the delimiters. Remember that DD and YY were previously set equal to the character position following each delimiter during the analysis loop.

Line 175 checks that the month is a valid number. If it is too small or large, a branch is made to line 210 for an early routine exit. Lines 180 through 187 then make sure that the day entered is valid for the month that was entered. Line 185 even allows for leap years based on the year entered. As you remember from our discussion earlier, we have to check for years divisible by 4 and centuries divisible by 400 to accurately figure the date. If an illegal date is detected due to entering the wrong day (based on the month entered), then the routine branches to line 210.

Finally, lines 195 and 200 reassemble A$ to exactly 8 characters by padding each component of the date with a leading zero, if needed. It also uses dashes as delimiters, although this could easily be changed to some other character if desired. After this, execution is returned to the caller.

By using this routine, several things are accomplished. First of all, you

can find month, day, and year values for future use in calculations. Second, each date will be formatted exactly the same. In fact, it is possible for someone to enter a date such as 6:1*84, and the program will return it as 06/01/84. This standardization is imperative for efficient program execution.

MANIPULATING DATES

To manipulate a date, it needs to be converted to a single number for comparison, addition, or whatever. This would be rather simple to do, if it weren't for leap years. However, the following formula will accomplish this task quickly:

```
10 DEF DATE = 365 * YY + DD + 31 * (MM - 1) + ((MM > 2)
   * ( INT (.4 * MM + 2.3))) + INT ((YY + (MM < 3)) / 4) - INT
   (.75 * ( INT ((YY + (MM < 3)) / 100 + 1)))
```

Pretty long equation, right? Well, I thought it best to keep it one equation and make it a function so that it could be utilized anywhere within a program. Now, to use the function, use N = DATE, and N will be made equal to the unique value corresponding to the date entered.

When using this statement, it is assumed that MM, DD, and YY have already been set to valid (checked) values. YY can be set to either a two-digit value or a four-digit value. If you are sure that all dates entered will be within the same century, two-digit values for years would be fine. However, if this is not the case, four-digit values are required. Otherwise, the calculations will not be accurate. In any case, the year should never be less than 1582, chiefly because that is the year that the Gregorian calendar was adopted, and this formula is based on that calendar.

As an overview, this formula converts any given date to a number that represents the number of total days in the date. To understand this formula completely, let's look at each part of it, beginning with the formula portion of the statement itself. It is assumed that you already know the purpose of defining functions and how to use them.

```
10 DEF DATE = 365 * YY + DD + 31 * (MM - 1) + ((MM > 2)
   * ( INT (.4 * MM + 2.3))) + INT ((YY + (MM < 3)) / 4) - INT
   (.75 * ( INT ((YY + (MM < 3)) / 100 + 1)))
```

This part of the formula converts years to days, and adds the partial portion of the month to the number. Then the number of days in the completed months of the year is added to that number.

```
10  DEF DATE = 365 * YY + DD + 31 * (MM - 1) + ((MM >
    2) * ( INT (.4 * MM + 2.3))) + INT ((YY + (MM < 3)) / 4) -
    INT (.75 * ( INT ((YY + (MM < 3)) / 100 + 1)))
```

In the previous part of the formula, we based all months on 31 days. This part of the formula corrects that by subtracting the correct number of days from our number. This, however, should only happen if the date is later than or equal to March 1st, so we do a check during this part of the equation. If it is false, then this whole part of the equation will be equal to zero. Only if it is true (MM will be greater than 2) will this portion of the formula be utilized.

```
10  DEF DATE = 365 * YY + DD + 31 * (MM - 1) + ((MM >
    2) * ( INT (.4 * MM + 2.3))) + INT ((YY + (MM < 3)) / 4) -
    INT (.75 * ( INT ((YY + (MM < 3)) / 100 + 1)))
```

This part of the equation adds the number of leap days that have occurred so far. If the date being analyzed is in January or February, YY is decremented by one so that we are not counting a leap day that hasn't happened yet.

```
10  DEF DATE = 365 * YY + DD + 31 * (MM - 1) + ((MM >
    2) * ( INT (.4 * MM + 2.3))) + INT ((YY + (MM < 3)) / 4) -
    INT (.75 * ( INT ((YY + (MM < 3)) / 100 + 1)))
```

This final part of the formula corrects for the century years that do not contain leap days. It also takes into account the fact that we may be analyzing a January or February date in a century year by decrementing YY if this is the case.

This formula is accurate, as long as certain conditions are met. First of all, you should make sure that MM, DD, and YY are valid figures. If they were derived after completing the date input routine discussed earlier in this chapter, then they should be correct.

Another consideration is that the year variable (YY) should never be equal to zero. So that this doesn't happen, it is a good idea, though not necessary, to always use four-digit years. In this way you will be sure to have unique numbers. This routine is only accurate for dates after October 15, 1582. Usually, however, you will not have someone entering an earlier date. Remember that there were no actual dates between 10/5/1582 and 10/14/1582.

Some examples of unique numbers associated with certain dates are shown in Fig. 6-1. These numbers were derived by using this formula.

Date	Unique #
06/11/1956	714576
05/25/1983	724420
02/29/1984	724700
03/01/1984	724701
11/05/1984	724950
12/31/1999	730484
01/01/2000	730485
08/12/2013	735457

Fig. 6-1. Unique Numbers for Specific Dates.

Once this formula has been executed, the unique number can be stored so that it can later be compared to unique numbers from other dates. In this way, you can get an idea of how many days are between dates, or which date came first, etc.

DAYS OF THE WEEK

Once you have performed the above function on a date, you can do one more function to determine the day of the week based on what date was entered. The following series of lines will do it for you:

Listing 6-2

```
1 REM      LISTING 6-2
50 RESTORE 80
55 FOR J=0 TO 6
60 READ W$(J)
65 NEXT J
70 D=INT(7*(N/7-INT(N/7))+.5)
75 PRINT W$(D)
80 DATA SATURDAY,SUNDAY,MONDAY,TUESDAY,WEDNESDAY,THURSD
    AY,FRIDAY
```

Table 6-3. Entry and Exit Variables for Day of Week Routine

On entry:	On exit:
N—Unique calculated date value	D —Numeric day of week W$(*)—Days of week

Table 6-4. Variable Table for Day of Week Routine

Variable	Type	Purpose	Used in lines
D	Numeric	Day of week	70, 75
J	Numeric	Loop counter	55, 60, 65
N	Numeric	Unique date	70
W$(*)	String	Days of week	60, 75

This routine finds the remainder after dividing evenly by 7. The remainder is then rounded so that it returns a number between 0 and 6. These represent the days of the week between Saturday and Friday.

Pretty simple, right? Well, how could this be used in a program? It would be useful in a business program to have a user enter today's date when they start using the program. Then you would be able to return a standardized date and even the day of the week. This is not the only use. If you think about it, there are many other uses of these routines.

PRINTING DATES

There are many ways to print dates. The best way is the one of the most use to your intended audience.

For most people the format MM/DD/YY will be sufficient. However, you may want to use the name of the month in the format—September 22, 1983. This is easy if you have previously analyzed the date. Set up an alphanumeric array with the names of the months of the year in a 1 through 12 matrix. Then print the name of the month using MM as a subscript. This is basically the same thing that was done with the day of the week.

The most important consideration when printing dates is to be consistent. Consistency, in whatever format you are using, looks professional.

PULLING IT ALL TOGETHER

So far, we have covered a lot of ground concerning dates. If we put these routines together, we can create a useful (albeit rather simplistic) program.

Listing 6-3

```
1 REM      LISTING 6-3
10 DEF DATE=365*YY+DD+31*(MM-1)+((MM>2)*(INT(.4*MM+2.3)
   ))+INT((YY+(MM<3))/4)-INT(.75*(INT((YY+(MM<3))/100+1)))
15 DATA SATURDAY,SUNDAY,MONDAY,TUESDAY,WEDNESDAY,THURSD
   AY,FRIDAY
20 RESTORE 15
25 FOR J=0 TO 6
30 READ W$(J)
35 NEXT J
40 GOTO 1000
100 MM=0
105 DD=0
110 YY=0
115 K=LEN(A$)
120 IF (K<8)+(K>10)THEN 210
125 FOR J=1 TO K
130 IF (ASC(SEG$(A$,J,1))<48)+(ASC(SEG$(A$,J,1))>57)THE
   N 140
135 GOTO 150
140 DD=YY
145 YY=J+1
150 NEXT J
155 IF DD=0 THEN 210
160 MM=VAL(SEG$(A$,1,DD-2))
165 DD=VAL(SEG$(A$,DD,YY-DD-1))
170 YY=VAL(SEG$(A$,YY,K-YY+1))
175 IF (MM<1)+(MM>12)THEN 210
180 IF (DD<1)+(DD>31)THEN 210
182 IF LEN(STR$(YY))<>4 THEN 210
185 IF (MM=2)*((INT(YY/4)*4)<>YY)*((INT(YY/400)*400)<>Y
   Y)*(DD>28)THEN 210
187 IF (MM=2)*(DD>29)THEN 210
190 IF ((MM=4)+(MM=6)+(MM=9)+(MM=11))*(DD>30)THEN 210
195 A$=SEG$("00"&STR$(MM),LEN(STR$(MM))+1,2)&"/"&SEG$("
   00"&STR$(DD),LEN(STR$(DD))+1,2)&"/"
200 A$=A$&SEG$("0000"&STR$(YY),LEN(STR$(YY))+1,4)
205 GOTO 215
210 A$=""
215 RETURN
300 B$=""
305 C=LEN(A$)+3
310 PRINT A$;
315 GOTO 325
320 CALL SOUND(100,1000,10)
325 CALL KEY(0,A,S)
330 IF S<>1 THEN 325
335 IF A=13 THEN 400
340 IF A<>8 THEN 380
345 IF LEN(B$)=0 THEN 320
350 B$=SEG$(B$,1,LEN(B$)-1)
355 J=LEN(B$)-1+C
360 IF LEN(B$)=0 THEN 370
365 CALL HCHAR(24,J,ASC(SEG$(B$,LEN(B$),1)))
370 CALL HCHAR(24,J+1,32)
375 GOTO 325
380 IF (A<32)+(A>126)THEN 320
385 B$=B$&CHR$(A)
390 CALL HCHAR(24,C+LEN(B$)-1,A)
395 GOTO 325
400 PRINT
```

```
405 RETURN
1000 CALL CLEAR
1010 PRINT "ENTER TODAY'S DATE"
1020 A$="(MM/DD/YYYY): "
1030 GOSUB 300
1040 A$=B$
1050 GOSUB 100
1060 IF A$<>"" THEN 1090
1070 CALL SOUND(100,1000,10)
1080 GOTO 1010
1090 D1$=A$
1100 D1=DATE
1110 PRINT :"ENTER YOUR BIRTHDAY"
1120 A$="(MM/DD/YYYY): "
1130 GOSUB 300
1140 A$=B$
1150 GOSUB 100
1160 IF A$<>"" THEN 1180
1165 CALL SOUND(100,1000,10)
1170 GOTO 1110
1180 D2$=A$
1190 D2=DATE
1200 W1=INT(7*(D1/7-INT(D1/7))+.5)
1210 W2=INT(7*(D2/7-INT(D2/7))+.5)
1220 DIF=ABS(D1-D2)
1230 CALL CLEAR
1240 PRINT "TODAY IS ";W$(W1);", ";D1$::
1250 PRINT "YOU WERE BORN ON ";W$(W2);", ";D2$::
1260 PRINT "THAT WAS ";STR$(DIF);" DAYS AGO."
9999 END
```

Table 6-5. Variable Table for Birthday Program

Variable	Type	Purpose	Used in lines
A	String	Prompt & Work string	115, 130, 160, 165, 170, 210, 305, 310, 1020, 1040, 1060, 1090, 1120, 1140, 1160, 1180
A	Numeric	ASCII keypress	325, 335, 340, 380, 385, 390
B	String	Input string	300, 345, 350, 355, 360, 365, 385, 390, 1040, 1140
C	Numeric	Screen pointer	305, 355
D1	String	Today's date	1090, 1240
D1	Numeric	Unique value of today's date	1100, 1200, 1220
D2	String	Birthday	1180, 1250
D2	Numeric	Unique value of birthday	1190, 1210, 1220

Table 6-5—cont. Variable Table for Birthday Program

Variable	Type	Purpose	Used in lines
DATE	Numeric Function	Derive unique number from date	10, 1100, 1190
DD	Numeric	Day	10, 105, 140, 155, 160, 165, 180, 185, 187, 190, 195
DIF	Numeric	Date difference	1220, 1260
J	Numeric	Loop counter & string pointer	25, 30, 35, 125, 130, 145, 150, 355, 365, 370
K	Numeric	Length of string	115, 120, 125, 170
MM	Numeric	Month	10, 100, 160, 175, 185, 187, 190, 195
S	Numeric	Keyboard status	325, 330
W(*)	String	Days of week	30, 1240, 1250
W1	Numeric	Day of week	1200, 1240
W2	Numeric	Day of week	1210, 1250
YY	Numeric	Year	10, 110, 140, 145, 165, 170, 182, 185, 200

This program requests two dates from a user—today's date in lines 1010 through 1080 and the user's birthdate in lines 1110 through 1170. The program analyzes each date as it is entered (lines 100 through 215) to determine if it is in the correct format. If not, then a bell is sounded and the questions are repeated.

Finally, the dates are compared (lines 1100 and 1190) and the days of the week that correspond to those dates are figured (lines 1200 and 1210). The results are then printed and the program ends.

This is a good example of how to use the information covered in this chapter. Notice also that the input routine contained in lines 300 through 405 is the one that was developed in Chapter 2.

In the next chapter, time will be analyzed using your computer.

Chapter 7

Time and Time Again

Have you ever noticed that we have a strange way of telling time in our society? Time is based on sixties. There are sixty seconds in a minute, and sixty minutes in an hour. After that it gets really messed up. Well, a computer can handle all sorts of messy items and details, as long as it is programmed correctly.

In this chapter we will discuss all the nitty-gritty details that have to do with hours, minutes, and seconds. We will not discuss how to time events, because software timers are generally very inaccurate (particularly over long periods of time). Our attention, however, will be directed towards manipulating units of time—in other words, once you have a beginning and ending time, how to calculate the intervals in between.

INPUTTING TIME

For most applications that require calculations with time, having a user input a beginning and ending time will be sufficient. As in inputting dates (see Chapter 6), the entry that the user makes will need to be analyzed to make sure that it is formatted correctly and within bounds.

If a more accurate (or trustworthy) inputting of time is needed, there are numerous peripherals available that provide a direct input of time to a program. These are usually called chronographs or clock cards. The use

Fig. 7-1. Father Time.

and interfacing of these devices is beyond the scope of this book. The multitude of peripherals that exist for the purpose of inputting time to a computer would make such a task monumental.

The keyboard input of the time string can best be done by using the string input routine (see Chapter 2). When analyzing it, however, a specialized routine will be needed. The complexity of this routine depends, in large part, on the accuracy needed for later calculations. For instance, if you only want accuracy to the nearest minute, then you only need to expect a user to input time as hours and minutes. If, however, you want to account for seconds, then the user will also need to enter those as well.

In this section, examples will be given of routines that may be used for both degrees of accuracy. First, let's look at an example of a routine used to analyze strings that consist of hours and minutes separated by a colon, in the format HH:MM T. The variable T stands for either an A or a P, depending on whether or not the time being entered is AM or PM.

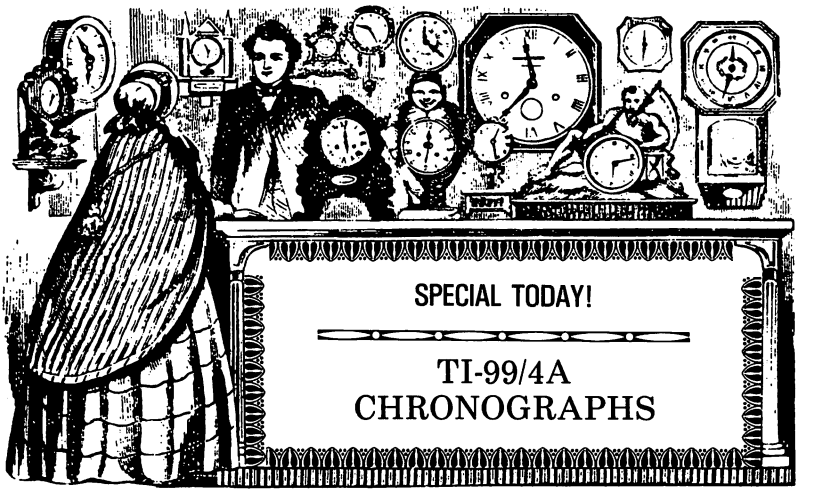

Fig. 7-2. Typical computer chronograph store.

Listing 7-1

```
1 REM      LISTING 7-1
100 IF LEN(A$)<4 THEN 220
105 IF (SEG$(A$,LEN(A$),1)<>"A")*(SEG$(A$,LEN(A$),1)<>"
    P")THEN 220
110 N=0
115 FOR J=1 TO LEN(A$)
120 K=ASC(SEG$(A$,J,1))
125 IF (K>47)*(K<58)THEN 150
130 M=N
135 N=J+1
140 IF M=0 THEN 150
145 J=LEN(A$)
150 NEXT J
155 IF M=0 THEN 220
160 IF N-M<2 THEN 220
165 H=VAL(SEG$(A$,1,M-2))
170 M=VAL(SEG$(A$,M,N-M-1))
175 IF (M<0)+(M>59)THEN 220
180 IF (H<1)+(H>12)THEN 220
185 A$=SEG$("00"&STR$(H),LEN(STR$(H))+1,2)&":"&SEG$("00
    "&STR$(M),LEN(STR$(M))+1,2)&" "&SEG$(A$,LEN(A$),1)
190 T=H*60+M
195 IF T<720 THEN 205
200 T=T-720
205 IF SEG$(A$,LEN(A$),1)="A" THEN 215
210 T=T+720
215 RETURN
220 A$=""
225 GOTO 215
```

Table 7-1. Entry and Exit Variables for Time Routine (Hours and Minutes)

On entry:	On exit:
A$ —String to be analyzed Format: HH:MM T HH—Hours MM—Minutes T—A or P	A$ —Return string Format: HH:MM T HH—Hours MM—Minutes T—A or P If equal to "" (null) then error in input string. H —Hours M —Minutes T —Total minutes

Table 7-2. Variable Table for Time Routine (Hours and Minutes)

Variable	Type	Purpose	Used in lines
A	String	Entry/Exit	100, 105, 115, 120, 145, 165, 170, 185, 205, 220
H	Numeric	Hours	165, 180, 185, 190
J	Numeric	Loop counter	115, 120, 135, 145, 150
K	Numeric	ASCII values	120, 125
M	Numeric	Minutes	130, 140, 155, 160, 165, 170, 175, 185, 190
N	Numeric	End delimiter	110, 130, 135, 160, 170
T	Numeric	Total minutes	190, 195, 200, 210

Before an explanation of the routine's operation, it is useful to mention the variables used. A$ is used both as input and the primary return value. This is the string to be checked, and it is the completely formatted string that is returned when the routine is completed. The variables H, M, and T represent hours, minutes and total minutes, respectively. Total minutes is the total amount of time represented by the input string and calculated

according to a 24-hour clock. Thus, T can vary from 0 to 1439 (number of minutes in a day, minus one).

Now for the routine itself. First, line 100 checks that the length of the string entered is valid. If it is not at least four characters in length, execution is transferred to line 220 where a null string is returned.

Line 105 makes sure that the rightmost character of the string entered is an A or a P. This signifies AM or PM, and saves the user from having to manually calculate time based on 24 hours. If the input string does not end with either of these characters, an early exit is taken through line 220.

Lines 110 through 170 do the actual work of breaking down the string to component parts. Each character of the string is examined to see if it is a non-numeric character. The first non-numeric character is assumed to be the delimiter between hours and minutes. If such a delimiter is found, M is set equal to the character position following the non-numeric character.

This loop is completed in line 150. Upon completion, if M still equals zero, an early exit is taken. This is because M will only be equal to zero if no delimiter was found in the preceding loop. Again, this is an invalid format, so an early exit is necessary.

Line 165 sets H equal to the number of hours in the input string, and line 170 sets M equal to the minute value of the input string. These values are then checked in lines 175 and 180 to make sure they are valid. If the minutes are not between 0 and 59, or if the hours are less than 1 or greater than 12, an early exit is taken.

By the time the program executes line 185, the input string has been completely checked and is valid. Line 185 puts the derived values back together in a standard format. The user could enter the time as 1/1P, and when line 185 is executed the return string will be equal to 01:01 P. This re-formatted string is assigned to A$, the same variable that was the input string. This helps conserve variable space and also means you won't have to redefine any variables after returning from this routine.

Lines 170 and 180 calculate the total number of minutes that was input. This figure is assigned to variable T, and can be used for later calculations of elapsed and cumulative time.

After this routine is called and executed, the length of A$ should be checked. If the length is equal to zero (null), then an error was detected in the user's entry. At this point, the time entry should be requested again. Checking the length of A$ is the quickest way to determine the validity of the time entered by the user.

A final note is in order. In analyzing a user's input, it is assumed that noon is entered as 12:00 P, and midnight as 12:00 A. Obviously, if these are confused by a user, later calculations will be inaccurate.

HOURS, MINUTES AND SECONDS

The preceding routine works fine when you are figuring hours and minutes, and will be adequate for the majority of applications. A few people also need to figure seconds, though. The following listing shows the routine slightly modified to account for seconds:

Listing 7-2

```
1 REM      LISTING 7-2
100 IF LEN(A$)<6 THEN 255
105 B$=SEG$(A$,LEN(A$),1)
110 IF (B$<>"A")*(B$<>"P")THEN 255
115 N=0
120 FOR J=1 TO LEN(A$)
125 K=ASC(SEG$(A$,J,1))
130 IF (K>47)*(K<58)THEN 160
135 M=S
140 S=N
145 N=J+1
150 IF M=0 THEN 160
155 J=LEN(A$)
160 NEXT J
165 IF M=0 THEN 255
170 IF (N-S)<2 THEN 255
175 IF (S-M)<2 THEN 255
180 H=VAL(SEG$(A$,1,M-2))
185 M=VAL(SEG$(A$,M,S-M-1))
190 S=VAL(SEG$(A$,S,N-S-1))
195 IF (S<0)+(S>59)THEN 255
200 IF (M<0)+(M>59)THEN 255
205 IF (H<1)+(H>12)THEN 255
210 A$=SEG$("00"&STR$(H),LEN(STR$(H))+1,2)&":"&SEG$("00
    "&STR$(M),LEN(STR$(M))+1,2)&":"
215 A$=A$&SEG$("00"&STR$(S),LEN(STR$(S))+1,2)&" "&B$
220 T=H*60+M
225 IF T<720 THEN 235
230 T=T-720
235 IF B$="A" THEN 245
240 T=T+720
245 T=T*60+S
250 RETURN
255 A$=""
260 GOTO 250
```

Since this routine is basically the same as the one used to calculate hours and minutes, only the changed lines will be examined.

Line 100, which still checks the length of the user's entry, now allows a minimum string length of six characters. Thus, 1:0:0A is the minimum that can be entered.

The verification loop in lines 115 through 175 has been expanded to include the variable S which is used to denote seconds. Here we will be searching for two delimiters, instead of only one. If both delimiters are found (M will no longer be equal to zero when this happens), the loop is terminated.

Table 7-3. Entry and Exit Variables for Time Routine (Hours, Minutes and Seconds)

On entry:	On exit:
A$ —String to be analyzed Format: HH:MM:SS T HH—Hours MM—Minutes SS—Seconds T—A or P	A$ —Return string Format: HH:MM:SS T HH—Hours MM—Minutes SS—Seconds T—A or P If equal to "" (null) then error in input string. H —Hours M —Minutes S —Seconds T —Total seconds

Table 7-4. Variable Table for Time Routine (Hours, Minutes and Seconds)

Variable	Type	Purpose	Used in lines
A	String	Entry/Exit	100, 105, 120, 125, 155, 180, 185, 190, 210, 215, 255
B	String	AM/PM marker	105, 110, 215, 235
H	Numeric	Hours	180, 205, 210, 220
J	Numeric	Loop counter	120, 125, 145, 155, 160
K	Numeric	ASCII value	125, 130
M	Numeric	Minutes	135, 150, 165, 175, 180, 185, 200, 210, 220
N	Numeric	End delimiter	115, 140, 145, 170, 190
S	Numeric	Seconds	140, 170, 175, 185, 190, 195, 215, 245
T	Numeric	Total seconds	220, 225, 230, 240, 245

Line 195 checks that the user has input a valid number of seconds. If S is less than 0 or greater than 59, the routine is exited.

Line 215 now allows the inclusion of seconds in the formatted output string. If 1:0:0A was the input string, line 215 will format it to 01:00:00 A.

Finally, line 245 converts all of the calculated minutes to seconds. To prepare for any manipulations, we need to use the smallest common denominator for the calculated totals. If hours and minutes are all that are being worked with, everything is converted to minutes. This is how the totals were handled in the previous section. If seconds are included, everything needs to be converted to seconds.

MANIPULATING TIME

Once the "time strings" have been converted to similar units of time, they can then be manipulated like any other numbers. The most frequent manipulations, of course, would be subtraction and addition. Be careful, when subtracting, to use the absolute value of the result. Differentials in time are usually expressed as positive numbers.

After you have manipulated the figures, you may want to convert back to hours, minutes and seconds. This is very easy to do, as shown by the following steps:

Listing 7-3

```
1 REM      LISTING 7-3
100 S=N
105 M=INT(S/60)
110 S=S-(M*60)
115 H=INT(M/60)
120 M=M-(H*60)
125 A$=SEG$("00"&STR$(H),LEN(STR$(H))+1,2)&":"&SEG$("00
    "&STR$(M),LEN(STR$(M))+1,2)&":"
130 A$=A$&SEG$("00"&STR$(S),LEN(STR$(S))+1,2)
135 RETURN
```

Table 7-5. Entry and Exit Variables for Convert-to-Time Routine

On entry:	On exit:
N—Value to be converted	A$—Formatted time string
	H —Hours
	M —Minutes
	N —Original number
	S —Seconds

Table 7-6. Variable Table for Convert-to-Time Routine

Variable	Type	Purpose	Used in lines
A	String	Formatted time	125, 130
H	Numeric	Hours	115, 120, 125
M	Numeric	Minutes	105, 110, 115, 120, 125
N	Numeric	Input figure	100
S	Numeric	Seconds	100, 105, 110, 130

This routine allows all three gradations of time. If you were only working to the nearest minute, you could substitute M for S, and H for M, and then drop the final original references to H. An alternate solution would be to multiply N by 60 prior to entering this routine. This would convert the total minutes to total seconds.

Fig. 7-3. Mother Time.

TIME OUTPUT

Printing the time is easy when you use the above routines because the formatted time is always in A$. The only thing that needs to be done is to determine where to print A$. For printing reports, you may find Chapter 6 useful.

If the hours and minutes routine is used, the output is always seven characters long. Output from the hours, minutes, and seconds routine is always ten characters long.

Working with time is easy if you do all your input, analysis, manipulation, and output in modular subroutines. Many programs can be enhanced by the use of time and time-related figures. Most notable among these are payroll, time-keeping, and cost-accounting programs that require direct time input from the user.

In the next chapter we will be looking at upper and lower cases.

Chapter 8

Character Cases

This could be the beginning of a great (but cheap) detective novel, *Character Cases I Have Known and Loved.* Sounds like a best seller, right? Well, this won't be quite as dramatic. However, it will be very useful in your programming efforts.

The TI-99/4A is equipped to handle both upper and lower case characters. On the display screen, upper case appears as capital letters, and lower case appears as slightly squashed capital letters. This distinction may not be readily apparent to the computer neophyte, so your program should take into account the fact that a user can enter both upper and lower case letters.

THE PROBLEM IN CASE

The problem arises with string handling, and particularly ordering, because any given string may contain both upper and lower case characters. For example, you have a program that accepts customer names from a user, and later you need to search that data for a specific name. Following is a partial list of some of the name strings that the file could contain:

BAKER, MARTHA
Doe, John
JOHNSON, DOUGLAS
Jones, Barbara
Mitchell, Dennis
Pease, Debra
Porter, Thomas
Thackery, Richard
Thomas, Albert
WYATT, ALLEN

STANDARD ASCII CHARACTER CODES

ASCII CODE	CHARACTER	ASCII CODE	CHARACTER
32	(space)	81	Q
33	! (exclamation point)	82	R
34	" (quote)	83	S
35	# (number or pound sign)	84	T
36	$ (dollar)	85	U
37	% (percent)	86	V
38	& (ampersand)	87	W
39	' (apostrophe)	88	X
40	((open parenthesis)	89	Y
41	) (close parenthesis)	90	Z
42	* (asterisk)	91	[(open bracket)
43	+ (plus)	92	\ (reverse slant)
44	, (comma)	93	] (close bracket)
45	− (minus)	94	^ (exponentiation)
46	. (period)	95	_ (line)
47	/ (slant)	96	` (grave)
48	0	97	a
49	1	98	b
50	2	99	c
51	3	100	d
52	4	101	e
53	5	102	f
54	6	103	g
55	7	104	h
56	8	105	i
57	9	106	j
58	: (colon)	107	k
59	; (semicolon)	108	l
60	< (less than)	109	m
61	= (equals)	110	n
62	> (greater than)	111	o
63	? (question mark)	112	p
64	@ (at sign)	113	q
65	A	114	r
66	B	115	s
67	C	116	t
68	D	117	u
69	E	118	v
70	F	119	w
71	G	120	x
72	H	121	y
73	I	122	z
74	J	123	{ (left brace)
75	K	124	¦
76	L	125	} (right brace)
77	M	126	~ (tilde)
78	N	127	DEL(appears on screen as a blank.)
79	O		
80	P		

Fig. 8-1. ASCII character chart.

Overall, the list looks about average. Some entries are all caps, and some are caps and lower case. This is not, in itself, a problem. A typical problem surfaces when you try to search for specific information. For instance, if you ask a user to enter a name to look for, and he enters THOMAS, Albert, you may run into a problem in the search. This is because the name, as it appears in the file, is really Thomas, Albert.

At this point it is helpful to look at an ASCII character chart. Notice in Fig. 8-1 that the ASCII character set includes all kinds of characters. We are just interested, however, in the alphameric (letters only) characters, both upper and lower case.

Each character has a corresponding number value so that the computer can manipulate it. The computer uses these numbers, instead of the letters themselves.

Fig. 8-2 shows the numeric conversion of both the user's entry and the file entry for Albert Thomas. The computer uses the numbers, not the letters, to make comparisons. By comparing the numbers it is obvious that the two strings do not match.

User:	T	H	O	M	A	S	,	A	l	b	e	r	t
Codes:	84	72	79	77	65	83	44	65	108	98	101	114	116
File:	T	h	o	m	a	s	,	A	l	b	e	r	t
Codes:	84	104	111	109	97	115	44	65	108	98	101	114	116

Fig. 8-2. Names in ASCII Code.

It should be obvious by now that the only way to make the computer think like a human (in this instance) is to convert each string so that it has the greatest likelihood of being matched. This is primarily accomplished by converting both the source and object strings to either upper or lower case.

LOWER TO UPPER CASE

By using the ASCII numeric representation of data, instead of actual letters, you can quickly manipulate the characters to the form needed. The following subroutine uses this principle to convert mixed-case input to all upper case.

This subroutine takes the original string (A$) and converts it, one character at a time, to upper case. Line 100 sets the resulting string (B$) to null, and also checks that the input string is at least one character in length.

Listing 8-1

```
1 REM      LISTING 8-1
100 B$=""
105 IF LEN(A$)=0 THEN 140
110 FOR J=1 TO LEN(A$)
115 A=ASC(SEG$(A$,J,1))
120 IF (A<97)+(A>122)THEN 130
125 A=A-32
130 B$=B$&CHR$(A)
135 NEXT J
140 RETURN
```

Table 8-1. Entry and Exit Variables for Upper Case Converter Routine

On entry:	On exit:
A$—String to convert	A$ —Original string B$ —Upper case string

Table 8-2. Variable Table for Upper Case Converter Routine

Variable	Type	Purpose	Used in lines
A	String	Input string	105, 110, 115
A	Numeric	ASCII values	115, 120, 125, 130
B	String	Return string	100, 130
J	Numeric	Loop counter	110, 115, 135

Line 110 starts the loop that derives the ASCII value of each successive character in A$. Line 120 then checks to see if the character is lower case (remember the ASCII chart). If it isn't lower case, then the case conversion is skipped. If it is lower case, then line 125 subtracts 32 from the ASCII value of the character. This effectively changes the character from lower to upper case. Finally, line 130 converts the number back to a character and adds it to B$. The loop is repeated until the process is finished, and the routine returns to line 140.

Although this routine is rather short, it is very effective. It is also useful because it is much easier to compare strings when everything is all either in upper case or in lower case.

In our earlier example, we could have used this routine on the search name input by the user to ensure that it was all upper case. Then, as each name was extracted from the name list, we could have used this routine on

it to do a quick conversion. To expand on this point, the conversion routine could be used in this way:

Listing 8-2

```
1 REM      LISTING 8-2
10 DIM F$(10)
20 RESTORE
30 FOR J=1 TO 10
40 READ F$(J)
50 NEXT J
60 GOTO 1000
100 B$=""
105 IF LEN(A$)=0 THEN 140
110 FOR J=1 TO LEN(A$)
115 A=ASC(SEG$(A$,J,1))
120 IF (A<97)+(A>122)THEN 130
125 A=A-32
130 B$=B$&CHR$(A)
135 NEXT J
140 RETURN
300 B$=""
305 C=LEN(A$)+3
310 PRINT A$;
315 GOTO 325
320 CALL SOUND(100,1000,10)
325 CALL KEY(0,A,S)
330 IF S<>1 THEN 325
335 IF A=13 THEN 400
340 IF A<>8 THEN 380
345 IF LEN(B$)=0 THEN 320
350 B$=SEG$(B$,1,LEN(B$)-1)
355 J=LEN(B$)-1+C
360 IF LEN(B$)=0 THEN 370
365 CALL HCHAR(24,J,ASC(SEG$(B$,LEN(B$),1)))
370 CALL HCHAR(24,J+1,32)
375 GOTO 325
380 IF (A<32)+(A>126)THEN 320
385 B$=B$&CHR$(A)
390 CALL HCHAR(24,C+LEN(B$)-1,A)
395 GOTO 325
400 PRINT
405 RETURN
1000 CALL CLEAR
1010 PRINT "ENTER NAME TO SEARCH FOR:"
1020 A$=""
1030 GOSUB 300
1040 IF B$="" THEN 9999
1050 E$=B$
1060 A$=B$
1070 GOSUB 100
1080 N$=B$
1090 FOR K=1 TO 10
1100 A$=F$(K)
1110 GOSUB 100
1120 IF N$=B$ THEN 1150
1130 NEXT K
1140 GOTO 1180
1150 PRINT ::"NAME ENTERED: ";E$
1160 PRINT "NAME MATCHED: ";F$(K)::
```

cont. on next page

Listing 8-2—cont.

```
1170 GOTO 1010
1180 PRINT ::"SORRY, NAME NOT FOUND!":
1185 CALL SOUND(100,1000,10)
1190 GOTO 1010
1200 DATA "BAKER, MARTHA","Doe, John"
1210 DATA "JOHNSON, DOUGLAS","Jones, Barbara"
1220 DATA "Mitchell, Dennis","Pease, Debra"
1230 DATA "Porter, Thomas","Thackery, Richard"
1240 DATA "Thomas, Albert","WYATT, ALLEN"
9999 END
```

Table 8-3. Variable Table for Case Conversion Sample Program

Variable	Type	Purpose	Used in lines
A	Numeric	ASCII value of keypress	115, 120, 125, 130, 325, 335, 340, 380, 385, 390
A	String	Entry/Prompt	105, 110, 115, 305, 310, 1020, 1060, 1100
B	String	Work string	100, 130, 300, 345, 350, 355, 360, 365, 385, 390, 1040, 1050, 1060, 1080, 1120
C	Numeric	Screen position	305, 355, 390
E	String	What user entered	1050, 1150
F(*)	String	Names array	10, 40, 1100, 1160
J	Numeric	Loop counter	30, 40, 50, 110, 115, 135, 355, 365, 370
K	Numeric	Loop counter	1090, 1100, 1130, 1160
N	String	Converted entry	1080
S	Numeric	Keyboard status	325, 330

Notice that this program is written with the names in DATA statements. It could just as easily have been written to get information from a disk or cassette file. To experiment with the program, simply change the information contained in the DATA statements (lines 1200 through 1240) to the information that you want to search.

UPPER TO LOWER CASE

To convert everything to lower case, only two lines need to be changed. Here is the same routine, modified to convert to lower case.

Listing 8-3

```
1 REM      LISTING 8-3
100 B$=""
105 IF LEN(A$)=0 THEN 140
110 FOR J=1 TO LEN(A$)
115 A=ASC(SEG$(A$,J,1))
120 IF (A<65)+(A>90)THEN 130
125 A=A+32
130 B$=B$&CHR$(A)
135 NEXT J
140 RETURN
```

Table 8-4. Entry and Exit Variables for Lower Case Converter Routine

On entry:	On exit:
A$—String to convert	A$ —Original string B$ —Lower case string

Table 8-5. Variable Table for Lower Case Converter Routine

Variable	Type	Purpose	Used in lines
A	String	Input string	105, 110, 115
A	Numeric	ASCII values	115, 120, 125, 130
B	String	Return string	100, 130
J	Numeric	Loop counter	110, 115, 135

By changing lines 120 and 125, we can check to see if the character being examined is upper case. If it is, then we add 32 to the decimal ASCII value to make it lower case.

Converting to lower case will work just as well for comparison purposes. As stated before, the only prerequisite is that both strings being compared are the same case, either upper or lower.

CASE CONCLUSIONS

This type of a routine will work well on the TI-99/4A, and becomes more important as the user has increased interaction with your program. If you do not have a routine such as this built in, searching for information may be more difficult than it should be.

While it is not impossible to correctly search or sort without using one of these routines, their use makes a program more professional and user-friendly.

Chapter 9

Sorting

Sorting is a field unto itself. Entire volumes have been written about various sorting algorithms. There are specialists who do nothing except work with sort programs. This is not one of those books, nor am I one of those people. I figure that this is all right, though, because I doubt if you are one of those people either. Neither of us has to be unless we are planning a future in Systems Design and Sort Technique.

If you are like me, all you need to do is periodically sort a short- to medium-length list of information. BASIC implementations of popular sorting algorithms will do fine for such applications. It should be noted, however, that under various conditions, BASIC sorting is rather slow when compared to machine language sorting. If speed is not one of your major concerns, however, BASIC routines will work just fine.

There are two general approaches to sorting—the in-memory (internal) sort and the disk (external) sort. In-memory sorting is well-suited for most BASIC applications of short- to medium- length lists. Longer lists require a sectional sorting technique with intermediate data saves to disk. Notice that I mention disks, and not cassettes. External sorts rarely, if ever, are written to use cassette tape. The sequential nature of tape and its slow access speed make tape a totally unsuitable media for sorting purposes.

External sorting, because it is an advanced technique, will not be discussed in this book. For a discussion of more advanced sorting applica-

tions, I suggest you visit your local computer store or library. Sorting techniques have not changed much over the last several years. You should be able to find a suitable book on the subject in either of these places.

SORTING FUNDAMENTALS

The main idea in sorting is to place items in a pre-determined order based on a common quality. For example, a gym teacher has a class of 30 students, and he wants them to line up in alphabetical order according to their last name. Or he may want to line them up according to height or age. Name, height, and age are called keys, because they determine what order the resulting line-up, or list, will be in.

Once you determine a key, you also need to know if the list will be in ascending or descending order—ordered from least to greatest value, or vice versa. In the case of the gym teacher, ascending order is from A to Z, shortest to tallest, or youngest to oldest, depending on the selected key. If he arranged the line in descending order, it would be the reverse of the ascending ordered line.

The sorting process is nothing more than comparing items of an original unordered list and exchanging them with other items, to get the list closer to its final sorted position. The number of elements to be sorted is usually expressed with the variable N, and all our routines will use N to denote the number of elements in the arrays to be sorted.

There have been many sorting algorithms designed, each supposedly more efficient than preceding methods. Each sorting method is adept at general list sorting, and each sorting algorithm is outstanding in sorting an unordered list that meets certain requirements. Some algorithms, for example, are great at sorting partially ordered lists, or changing ascending ordered lists to descending order. However, these same algorithms do poorly when used to sort randomly distributed lists. The actual sorting method you choose should depend, in large part, on the type and order of data you need to sort.

There are minor differences when sorting lists of numbers and lists of alphanumeric data. All of the examples in this chapter will only deal with sorting strings. To apply the same techniques to numeric data, use numeric variables in place of the string variables.

As you are deciding on a sort routine to use, you must determine what best fits your needs. Certain "specialty" sorts may work better for your needs. In this chapter, however, we will only be discussing a few of the many algorithms available. Each of these will be general-purpose sorting algorithms. The first to be discussed will be the Substitution Sort.

SUBSTITUTION SORTING

The most simple sorting technique is the Substitution Sort, sometimes called a "Bubble Sort". This technique works by comparing each string to every other string in the list to find which one belongs where. As each comparison is made, it is determined whether a "switch" of the data at the two locations should occur or not. If so, then the values are exchanged and the process continues. Each time a complete pass is made through the unsorted list, one more element is positioned in its correct sorted position. Therefore, we would assume that one pass of the list is required for each element in the original list. Thus, if the original list contains eight elements, then eight passes will be required to correctly sort the list.

In fact, that many passes are not required. For a list of eight elements, only seven passes are required, because the last pass actually places the final two elements in their correct positions. An eighth pass would be redundant. Fig. 9-1 demonstrates the way a Substitution Sort would work with a list of eight elements.

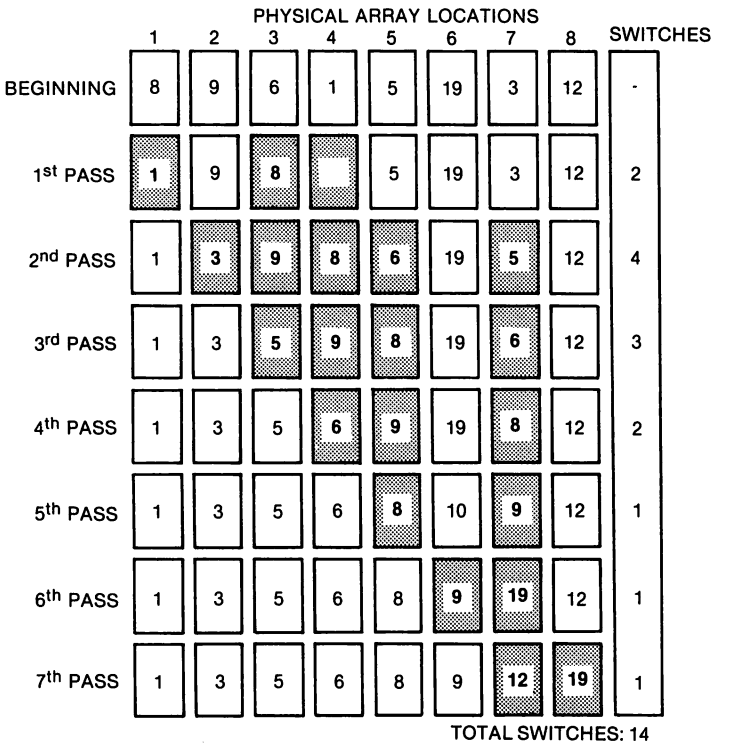

Fig. 9-1. Sorting Sequence for Substitution Sort. Switches have been highlighted.

Notice that each time through the list, an element at the front of the list is placed in correct position. We can then ignore these elements, because we know that they are positioned correctly and need no more comparisons. This means that the first time through the list there are seven comparisons required, six the second time, five the third, and so on until all passes have been completed. The number of comparisons for a Substitution Sort, then, is **(N-1)!** (called N-1 factorial). This is a mathematical way of saying what was expressed earlier. This concept of the number of comparisons required is displayed in Fig. 9-2.

$$(N-1)! = 7 + 6 + 5 + 4 + 3 + 2 + 1 = 28$$

Fig. 9-2. Method for Figuring Factorials for Numbers. This represents the number of comparisons required by the Substitution Sort algorithm.

Implementing the Substitution Sort is simple when using FOR NEXT loops. The following routine will sort an alphanumeric array in ascending order:

Listing 9-1

```
1 REM      LISTING 9-1
100 FOR J=1 TO N-1
105 FOR K=J+1 TO N
110 IF (S$(J)<=S$(K))THEN 130
115 T$=S$(J)
120 S$(J)=S$(K)
125 S$(K)=T$
130 NEXT K
135 NEXT J
140 RETURN
```

This routine is not very long. It is the "quick and dirty" type of sort that will work best on short lists. Notice that there is no wasted time in the routine, because there are no instances of a string being compared to itself. The two loops are always offset by one array element. This allows for the most efficient use of this algorithm.

Table 9-1. Entry and Exit Variables for Substitution Sort Routine

On entry:	On exit:
N —Highest element of array S$(*)—Array to be sorted	S$(*)—Sorted array

Table 9-2. Variable Table for Substitution Sort Routine

Variable	Type	Purpose	Used in lines
J	Numeric	Loop counter	100, 105, 110, 115, 120, 135
K	Numeric	Loop counter	105, 110, 120, 125, 130
N	Numeric	Highest element of array	100, 105
S(*)	String	Array to sort	110, 115, 120, 125
T	String	Temporary string	115, 125

There is no need to compare those elements of the array that we know to have been correctly placed. Thus, the K loop (which does the actual comparisons) uses J + 1 as a bottom boundary on each iteration.

The drawback of this routine is that it tends to be rather slow, mainly because there is quite a bit of "switching" that goes on. If you had an array of 100 elements, only one element can be put in order with each pass through the array. However, it is highly possible that there were 5, 10, 20, or more switches in order to place that one string. This slows things down because when that much string manipulation is done, BASIC has more overhead work, such as "garbage collection." This is the process of freeing up string storage space periodically.

SPEEDING UP SUBSTITUTION

As the list of elements to be sorted becomes longer, it proves increasingly critical to speed up the sorting process. This can be done in any of three ways. You can decrease the number of comparisons, decrease the number of exchanges, or use different variables. The best way to speed up the Substitution Sort is to use different variables (which in effect reduces the number of exchanges necessary).

BASIC works quite a bit faster with numeric variables, because there is no garbage collection, and there are fewer bytes to move. If the sorting routine were to be modified to allow for the switching of numeric pointers instead of the actual strings, then it would theoretically be faster. It would be possible to improve the efficiency to only one switch per pass, maximum. This is done by using pointers, and by only switching strings at the end of the K loop.

This is how the Substitution Sort would appear if we were to introduce these modifications:

Listing 9-2

```
1 REM      LISTING 9-2
100 FOR J=1 TO N-1
105 J1=J
110 FOR K=J+1 TO N
115 IF (S$(J1)<=S$(K))THEN 125
120 J1=K
125 NEXT K
130 IF J1=J THEN 150
135 T$=S$(J)
140 S$(J)=S$(J1)
145 S$(J1)=T$
150 NEXT J
155 RETURN
```

Table 9-3. Entry and Exit Variables for Modified Substitution Sort Routine

On entry:	On exit:
N —Highest element of array S$(*)—Array to be sorted	S$(*)—Sorted array

Table 9-4. Variable Table for Modified Substitution Sort Routine

Variable	Type	Purpose	Used in lines
J	Numeric	Loop counter	100, 105, 110, 130, 135, 140, 150
J1	Numeric	Low string pointer	105, 115, 120, 130, 140, 145
K	Numeric	Loop counter	110, 115, 120, 125
N	Numeric	Highest element of array	100, 110
S(*)	String	Array to sort	115, 135, 140, 145
T	String	Temporary string	135, 145

The biggest change here is the addition of J1 as a string pointer. It starts in line 100 as being equal to J, and then is reset equal to K each time a string is encountered that is of lesser value than the one to which it previously pointed. This is continued until the "K loop" is finished, and then the actual string switching takes place in lines 135 through 145. To save time, the switching only takes place if a switch is needed. In other words, there is only a switch if J1 does not equal J. If this check was not done, it would be possible to switch in place. This is wasteful of time and string space.

Functionally, this routine is the same as the straight Substitution Sort. Successive array elements are compared to find the lowest valued elements. Then they are placed in their correct sorted order. The difference is that they are not placed in that order until each pass is completed. Thus, where it was possible (with a list of 100 items) to have 10, 20, or 30 exchanges per pass, we will only have 1 exchange per pass with this modified version of the algorithm.

This routine is faster than the unmodified Substitution Sort. The following program illustrates this. The first part of the program creates an array of 100 random string elements. The second part makes a second array equal to the first. Finally, each of the duplicate arrays is sorted. One is sorted by the regular Substitution Sort, and the other is sorted by the modified version. As each section of the program is completed, a notice is displayed, and the bell is sounded. This way you can use your watch to time each sorting method.

Listing 9-3

```
1 REM      LISTING 9-3
100 DIM S$(100),S1$(100)
110 N=100
120 CALL CLEAR
130 FOR J=1 TO N
140 PRINT "WORKING ON";J
150 J1=INT(RND*15)
160 IF J1<2 THEN 150
170 FOR K=1 TO J1
180 K1=RND*91
190 IF (K1<32)+(K1>90)THEN 180
200 S$(J)=S$(J)&CHR$(K1)
210 NEXT K
220 S1$(J)=S$(J)
230 NEXT J
240 CALL CLEAR
250 CALL SOUND(100,1000,10)
260 PRINT "STARTING SUBSTITUTION SORT"
270 FOR J=1 TO N-1
280 FOR K=J+1 TO N
290 IF S$(J)<=S$(K)THEN 330
300 T$=S$(J)
310 S$(J)=S$(K)
320 S$(K)=T$
330 NEXT K
340 NEXT J
350 CALL SOUND(100,1000,10)
360 PRINT "DONE WITH SUBSTITUTION SORT"
370 CALL SOUND(100,1000,10)
380 PRINT ::"STARTING MODIFIED":"SUBSTITUTION SORT"
390 FOR J=1 TO N-1
400 J1=J
410 FOR K=J+1 TO N
```

cont. on next page

Listing 9-3—cont.

```
420 IF (S1$(J1)<=S1$(K))THEN 440
430 J1=K
440 NEXT K
450 IF J1=J THEN 490
460 T$=S1$(J)
470 S1$(J)=S1$(J1)
480 S1$(J1)=T$
490 NEXT J
500 CALL SOUND(100,1000,10)
510 PRINT "DONE WITH MODIFIED":"SUBSTITUTION SORT"
```

Table 9-5. Variable Table for Substitution Sort Test Program

Variable	Type	Purpose	Used in lines
J	Numeric	Loop counter	130, 140, 200, 230, 270, 280, 290, 300, 310, 340, 390, 400, 410, 450, 460, 470, 490
J1	Numeric	Low element pointer	150, 160, 170, 400, 420, 430, 450, 470, 480
K	Numeric	Loop counter	170, 210, 280, 290, 310, 320, 330, 410, 420, 430, 440
K1	Numeric	ASCII character	180, 190, 200
N	Numeric	Upper array limit	110, 130, 270, 280, 390, 410
S(*)	String	Array to sort	100, 200, 220, 290, 300, 310, 320
S1(*)	String	Array to sort	100, 220, 420, 460, 470, 480
T	String	Temporary string	300, 320, 460

In case you didn't have a watch handy, I'll tell you what mine told me. By timing both methods, using duplicate arrays to sort, and under the same conditions, there was almost a 50% increase in speed by using the second modified routine.

Obviously, if these routines are implemented in another program, the speed of the routines may differ dramatically. The speed is dependent on many things, including the size of the program, the number of string variables in memory, the number of items to sort, the ordering of the

original list, and whether the routine is "straight-line" or a subroutine. In this example, we made the sorting algorithms straight-line in order to make them as fast as possible.

SHELL SORT

This algorithm, introduced by D. L. Shell (thus the name) in July of 1959, is different than the Substitution Sort. It differs in that it relies on comparisons and exchanges of array elements that are not immediate neighbors. Comparisons are done at a pre-determined distance between elements. As an example, if the distance between elements was 4, then element 1 is compared with element 5, 2 with 6, 3 with 7, and so on.

Much has been said and written about Shell Sort, and I will not go into detail here. The greatest debate has been over the determination of an appropriate starting distance between items to be compared. Shell, in his original algorithm, used an interval equal to one-half of the total number of elements in the array. Each successive pass cut the distance in half, until an interval of only one element was reached. This required only a small number of passes to sort a relatively large array. As an example, it only takes one more pass to sort 200 items than it does to sort 100 items.

In our example, we will use the distance proposed by Shell. The following subroutine is the implementation of Shell Sort:

Listing 9-4

```
1 REM      LISTING 9-4
100 I=N
105 I=INT(I/2)
110 IF I=0 THEN 160
115 FOR J=I TO N
120 T$=S2$(J)
125 FOR K=J-I TO 1 STEP -I
130 IF T$>S2$(K)THEN 145
135 S2$(K+I)=S2$(K)
140 NEXT K
145 S2$(K+I)=T$
150 NEXT J
155 GOTO 105
160 RETURN
```

Table 9-6. Entry and Exit Variables for Shell Sort Algorithm

On entry:	On exit:
N —Highest element of array S2$(*) —Array to be sorted	S2$(*)—Sorted array

Table 9-7. Variable Table for Shell Sort Algorithm

Variable	Type	Purpose	Used in lines
I	Numeric	Distance pointer	100, 105, 110, 115, 125, 135, 145
J	Numeric	Loop counter	115, 120, 125, 150
K	Numeric	Loop counter	125, 130, 135, 140, 145
N	Numeric	Highest array element	100, 115
S2(*)	String	Sort array	120, 130, 135, 145
T	String	Temporary	120, 130, 145

The distance between comparisons for each pass of the array is set in line 105. Initially, this distance is equal to one-half of the total number of elements in the array. In lines 115 through 150, we execute a loop that runs the actual comparisons and exchanges string values if necessary.

Shell Sort is relatively simple, yet it offers a significant speed advantage over the earlier sorts that were introduced. It is best used on short- to medium-length lists of data.

SHELL SORT COMPARISON

Next we will compare the performance of Shell Sort with the Substitution Sorts under similar conditions. The following program has the Shell Sort algorithm appended, and gives the same type of output as the earlier comparison program. Get your watch ready.

Listing 9-5

```
1 REM      LISTING 9-5
100 DIM S$(100),S1$(100),S2$(100)
110 N=100
120 CALL CLEAR
130 FOR J=1 TO N
140 PRINT "WORKING ON";J
150 J1=INT(RND*15)
160 IF J1<2 THEN 150
170 FOR K=1 TO J1
180 K1=RND*91
190 IF (K1<32)+(K1>90)THEN 180
200 S$(J)=S$(J)&CHR$(K1)
210 NEXT K
220 S1$(J)=S$(J)
230 S2$(J)=S$(J)
240 NEXT J
```

```
250 CALL CLEAR
260 CALL SOUND(100,1000,10)
270 PRINT "STARTING SUBSTITUTION SORT"
280 FOR J=1 TO N-1
290 FOR K=J+1 TO N
300 IF S$(J)<=S$(K)THEN 340
310 T$=S$(J)
320 S$(J)=S$(K)
330 S$(K)=T$
340 NEXT K
350 NEXT J
360 CALL SOUND(100,1000,10)
370 PRINT "DONE WITH SUBSTITUTION SORT"
380 CALL SOUND(100,1000,10)
390 PRINT ::"STARTING MODIFIED":"SUBSTITUTION SORT"
400 FOR J=1 TO N-1
410 J1=J
420 FOR K=J+1 TO N
430 IF (S1$(J1)<=S1$(K))THEN 450
440 J1=K
450 NEXT K
460 IF J1=J THEN 500
470 T$=S1$(J)
480 S1$(J)=S1$(J1)
490 S1$(J1)=T$
500 NEXT J
510 CALL SOUND(100,1000,10)
520 PRINT "DONE WITH MODIFIED":"SUBSTITUTION SORT"
530 CALL SOUND(100,1000,10)
540 PRINT ::"STARTING SHELL SORT"
550 I=N
560 I=INT(I/2)
570 IF I=0 THEN 670
580 FOR J=I TO N
590 T$=S2$(J)
600 FOR K=J-I TO 1 STEP -I
610 IF T$>S2$(K)THEN 640
620 S2$(K+I)=S2$(K)
630 NEXT K
640 S2$(K+I)=T$
650 NEXT J
660 GOTO 560
670 CALL SOUND(100,1000,10)
680 PRINT "DONE WITH SHELL SORT"
```

The execution of the Substitution Sorts has slowed down somewhat in this comparison. The coding was not changed, but the memory of the computer has more variables to contend with. This added overhead, as noted earlier, can slow down performance of sorting routines.

Table 9-9 shows the comparative times for a sample run of this program. Because the random strings may be generated differently on each run, your times may be different than those noted.

So far, we have looked at how to speed up a sort by reducing the number of actual switches. The next step would be to speed it up by reducing the total number of comparisons required. This would take a completely different type of sorting algorithm, however. The one that shall be introduced here is called Quicksort.

Table 9-8. Variable Table for Sort Comparison Program

Variable	Type	Purpose	Used in lines
I	Numeric	Distance pointer	550, 560, 570, 580, 600, 620
J	Numeric	Loop counter	130, 140, 200, 240, 280, 290, 300, 310, 320, 350, 400, 410, 420, 460, 470, 480, 500, 580, 590, 600, 650
J1	Numeric	Low element pointer	150, 160, 170, 410, 430, 440, 460, 480, 490
K	Numeric	Loop counter	170, 210, 290, 300, 320, 330, 340, 420, 430, 440, 450, 600, 610, 620, 630, 640
K1	Numeric	ASCII character	180, 190, 200
N	Numeric	Upper array limit	110, 130, 280, 290, 400, 420, 550, 580
S(*)	String	Array to sort	100, 200, 220, 230, 300, 310, 320, 330
S1(*)	String	Array to sort	100, 220, 430, 470, 480, 490
S2(*)	String	Sort array	100, 230, 590, 610, 620, 640
T	String	Temporary string	310, 330, 470, 590, 610, 640

Table 9-9. Sorting Times for Comparison. All Tests Done with 100 Randomly Generated Elements

Sort Type	Sample Time	Improvement
Substitution Sort	217 seconds	—%
Modified Substitution Sort	91 seconds	58.06%
Shell Sort	58 seconds	36.26%

QUICKSORT

This sorting algorithm, devised by C.A.R. Hoare in 1962, is quite elegant. The idea behind Quicksort is to exchange non-adjacent elements of an array to achieve a more nearly sorted array. Partitioning is used to accomplish this sorting. This may sound confusing, but it actually works as implemented.

For example, suppose that you have the same gym class that was introduced earlier in the chapter. Class members were lined up in no specific order, and the gym teacher wanted to have them lined up according to height. Well, if he were trying to do this in the same method that Quicksort does, he would divide (partition) the class in half, and compare members from each half of the class. A member of the bottom half is compared to a member of the top half. The gym teacher knows that the bottom half of the class will contain all of the shorter members. If comparison of two members shows the taller member in the bottom half of the class, then a trade is made in order to at least get the members in the right half of the class. This process continues until the entire class is in correct order.

This example has been greatly generalized and over-simplified. When using arrays, Quicksort is more complex. It continues to partition the list until there is only one element per partitioned sublist. Then it works its way back up the partitions until they are all done. If each partition is in order, then the entire array will be in order.

To accomplish the sort, we must choose some element of the array to be placed in its correct final position. Then the remaining elements are arranged so that they are either less than or greater than the original chosen element.

This chosen element is called a pivot, and there are many theories on the best way to choose it for the optimal performance of Quicksort. The best performance will occur when the value of the pivot is the median value of the partition being sorted. In practice, however, there is no way to ensure this without extensive testing. Such testing can slow down the overall performance of the sort, particularly with a medium-length list. For our purposes, we shall use the first sequential element of the partition as the pivot.

Once the pivot is selected, the next step is to scan forward from there until we find a value greater than the pivot value. Then we scan backwards from the end of the partition until a value is found that is less than the value of the pivot. Then we exchange the lower and higher values, since they are both in the theoretical "wrong half" of the partition.This process is continued until the forwards and backwards comparison pointers pass

each other. At this point, the pivot value and the final forward value are exchanged, and it is assumed that the pivot value is now in its final sorted position. Then the whole process of selecting a pivot is repeated again and again until the whole array is completely sorted.

For those of you who are mathematically inclined, perhaps an explanation using variables would be helpful. Assume that the upper and lower bounds of our alphanumeric array are P and Q, such that it appears as S3$(P) . . . S3$(Q). On the first pass through the array with Quicksort,

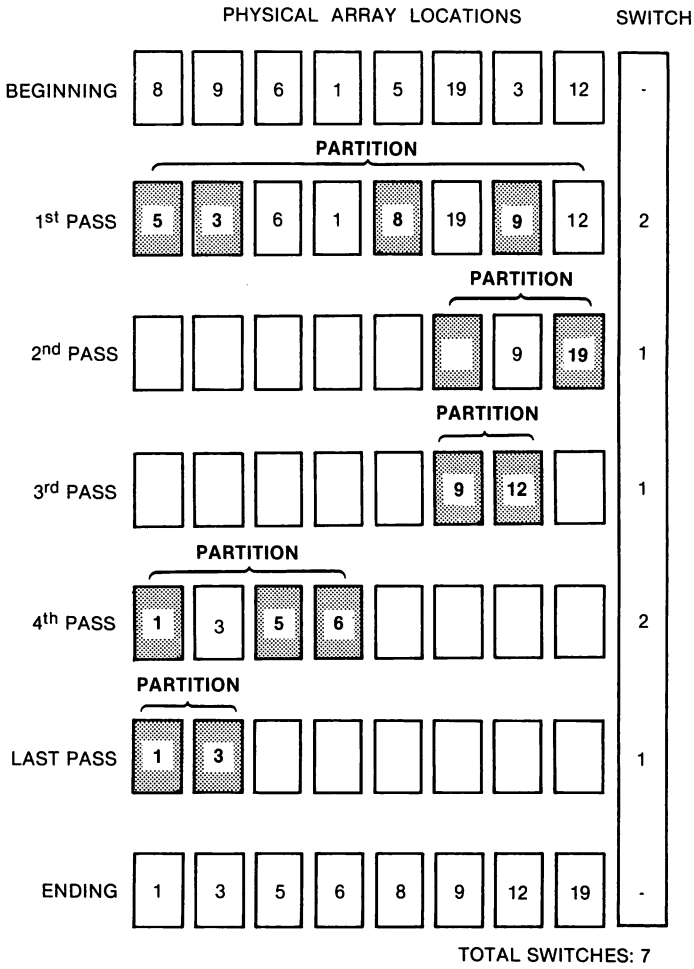

Fig. 9-3. Sorting Sequence for Quicksort. Switches have been highlighted.

the pivot point string, denoted as X$, will be equal to S3$(P). We will use I and J as our scanning variables. These are respectively set as equal to P + 1 and Q. Then we start a process of incrementing I to look for an array element whose value is greater than or equal to X$. When we find one, we start decrementing J until we find an array element whose value is less than or equal to X$. When that is found, we switch S3$(I) and S3$(J). If J is still greater than I, we go back to the point where I was being incremented to look for the next array value higher than or equal to X$. The process is repeated until J is less than I, at which point X$ and S3$(I) are switched in position.

This may all sound more confusing than it actually is. Fig. 9-3 graphs this process with an array of 8 elements. If you keep studying the process, it will become clearer and clearer.

The following subroutine shows the BASIC implementation of a Quicksort variation:

Listing 9-6

```
1 REM       LISTING 9-6
100 P=1
105 Q=N
110 T0=0
115 IF P>=Q THEN 240
120 V$=S3$(P)
125 I=P
130 J=Q+1
135 J=J-1
140 IF S3$(J)>V$ THEN 135
145 I=I+1
150 IF (S3$(I)<V$)*(I<N)THEN 145
155 IF J<=I THEN 180
160 T$=S3$(I)
165 S3$(I)=S3$(J)
170 S3$(J)=T$
175 GOTO 135
180 S3$(P)=S3$(J)
185 S3$(J)=V$
190 IF (J-P)>=(Q-J)THEN 215
195 ST(T0+1)=J+1
200 ST(T0+2)=Q
205 Q=J-1
210 GOTO 230
215 ST(T0+1)=P
220 ST(T0+2)=J-1
225 P=J+1
230 T0=T0+2
235 GOTO 115
240 IF T0=0 THEN 265
245 Q=ST(T0)
250 P=ST(T0-1)
255 TO =T0-2
260 GOTO 115
265 RETURN
```

Table 9-10. Entry and Exit Variables for Quicksort Algorithm

On entry:		On exit:
N	—Highest element of array	S3$(*)—Sorted array
ST(*)	—Stack array (empty, but dimensioned to proper size)	
S3$(*)	—Array to be sorted	

Table 9-11. Variable Table for Quicksort Algorithm

Variable	Type	Purpose	Used in lines
I	Numeric	Forward counter	125, 145, 150, 155, 160, 165
J	Numeric	Backward counter	130, 135, 140, 155, 165, 170, 180, 185, 190, 195, 205, 220, 225
N	Numeric	Total array elements	105, 150
P	Numeric	Lower partition pointer	100, 115, 120, 125, 180, 190, 215, 225, 250
Q	Numeric	Upper partition pointer	105, 115, 130, 190, 200, 205, 245
S3(*)	String	Sort array	120, 140, 150, 160, 165, 180, 185
ST(*)	Numeric	Stack	195, 200, 215, 220, 245, 250
T	String	Temporary	160, 170
T0	Numeric	Stack pointer	110, 195, 200, 215, 220, 230, 240, 245, 250, 255
V	String	Temporary	120, 140, 150, 185

While this routine takes more code than the Substitution and Shell Sorts, it is still compact. By studying this listing, it should be fairly clear as to how Quicksort is implemented. Take particular care to apply this listing to the process shown in Fig 9-4. This will help clear away some of the fog, if it is still there.

Lines 100 through 110 do nothing more than set the entry variables for

the routine. P is set equal to the lowest element of the array, and Q is set equal to the highest. T0 is a stack pointer, and is initially set to zero. The stack is used to save the beginning and ending pointers of the partitions that still need to be sorted.

Line 115 checks to see if the current partition has been sorted. If it is (P > = Q) then execution skips to line 240 where the routine checks to see if another unfinished partition is waiting in the wings.

Lines 120 through 155 are the heart of the actual sort. These lines carry out the process of selecting a pivot (V$) and then scanning forwards from the front of the partition for elements that are less than or equal to the pivot, and backwards from the back of the partition for elements that are greater than or equal to the pivot.

When both of these are found, the elements are switched in lines 160 through 170, and the process continues until the pointers (I and J) "pass" each other. When they do, then the values of the pivot and S3$(J) are exchanged.

Next, a new partition is determined, and the process repeated. The partition currently not being sorted is saved on the stack for future reference.

QUICKSORT COMPARISON

This routine works quickly for most sorting needs. It is interesting, however, to compare it against the earlier sorting methods. By using the same comparison program that was introduced earlier, and appending Quicksort, we can get an idea of how the routine fares. The following program will illustrate the comparison:

Listing 9-7

```
1 REM      LISTING 9-7
100 DIM S$(100),S1$(100),S2$(100),S3$(100)
110 N=100
120 CALL CLEAR
130 FOR J=1 TO N
140 PRINT "WORKING ON";J
150 J1=INT(RND*15)
160 IF J1<2 THEN 150
170 FOR K=1 TO J1
180 K1=RND*91
190 IF (K1<32)+(K1>90)THEN 180
200 S$(J)=S$(J)&CHR$(K1)
210 NEXT K
220 S1$(J)=S$(J)
230 S2$(J)=S$(J)
```

cont. on next page

Listing 9-7—cont.

```
240 S3$(J)=S$(J)
250 NEXT J
260 CALL CLEAR
270 CALL SOUND(100,1000,10)
280 PRINT "STARTING SUBSTITUTION SORT"
290 FOR J=1 TO N-1
300 FOR K=J+1 TO N
310 IF S$(J)<=S$(K)THEN 350
320 T$=S$(J)
330 S$(J)=S$(K)
340 S$(K)=T$
350 NEXT K
360 NEXT J
370 CALL SOUND(100,1000,10)
380 PRINT "DONE WITH SUBSTITUTION SORT"
390 CALL SOUND(100,1000,10)
400 PRINT ::"STARTING MODIFIED":"SUBSTITUTION SORT"
410 FOR J=1 TO N-1
420 J1=J
430 FOR K=J+1 TO N
440 IF (S1$(J1)<=S1$(K))THEN 460
450 J1=K
460 NEXT K
470 IF J1=J THEN 510
480 T$=S1$(J)
490 S1$(J)=S1$(J1)
500 S1$(J1)=T$
510 NEXT J
520 CALL SOUND(100,1000,10)
530 PRINT "DONE WITH MODIFIED":"SUBSTITUTION SORT"
540 CALL SOUND(100,1000,10)
550 PRINT ::"STARTING SHELL SORT"
560 I=N
570 I=INT(I/2)
580 IF I=0 THEN 680
590 FOR J=I TO N
600 T$=S2$(J)
610 FOR K=J-I TO 1 STEP -I
620 IF T$>S2$(K)THEN 650
630 S2$(K+I)=S2$(K)
640 NEXT K
650 S2$(K+I)=T$
660 NEXT J
670 GOTO 570
680 CALL SOUND(100,1000,10)
690 PRINT "DONE WITH SHELL SORT"
700 CALL SOUND(100,1000,10)
710 PRINT "STARTING QUICKSORT"
720 P=1
730 Q=N
740 T0=0
750 IF P>=Q THEN 1000
760 V$=S3$(P)
770 I=P
780 J=Q+1
790 J=J-1
800 IF S3$(J)>V$ THEN 790
810 I=I+1
820 IF (S3$(I)<V$)*(I<N)THEN 810
830 IF J<=I THEN 880
```

```
840 T$=S3$(I)
850 S3$(I)=S3$(J)
860 S3$(J)=T$
870 GOTO 790
880 S3$(P)=S3$(J)
890 S3$(J)=V$
900 IF (J-P)>=(Q-J)THEN 950
910 ST(T0+1)=J+1
920 ST(T0+2)=Q
930 Q=J-1
940 GOTO 980
950 ST(T0+1)=P
960 ST(T0+2)=J-1
970 P=J+1
980 T0=T0+2
990 GOTO 750
1000 IF T0=0 THEN 1050
1010 Q=ST(T0)
1020 P=ST(T0-1)
1030 T0=T0-2
1040 GOTO 750
1050 CALL SOUND(100,1000,10)
1060 PRINT "FINISHED WITH QUICKSORT"
```

Table 9-12. Variable Table for Sort Comparison Program

Variable	Type	Purpose	Used in lines
I	Numeric	Distance pointer	560, 570, 580, 590, 610, 630, 770, 810, 820, 830, 840, 850
J	Numeric	Loop counter	130, 140, 200, 250, 290, 300, 310, 320, 330, 360, 410, 420, 430, 470, 480, 490, 510, 590, 600, 610, 660, 780, 790, 800, 830, 850, 860, 880, 890, 900, 910, 930, 960, 970
J1	Numeric	Low element pointer	150, 160, 170, 420, 440, 450, 470, 490, 500
K	Numeric	Loop counter	170, 210, 300, 310, 330, 340, 350, 430, 440, 450, 460, 610, 620, 630, 640, 650
K1	Numeric	ASCII character	180, 190, 200
N	Numeric	Upper array limit	110, 130, 290, 300, 410, 430, 560, 590, 730, 820

Table 9-12—cont. Variable Table for Sort Comparison Program

Variable	Type	Purpose	Used in lines
P	Numeric	Lower partition pointer	720, 750, 760, 770, 880, 900, 950, 970, 1020
Q	Numeric	Upper partition pointer	730, 750, 780, 900, 920, 930, 1010
S(*)	String	Array to sort	100, 200, 220, 230 240, 310, 320, 330, 340
S1(*)	String	Array to sort	100, 220, 440, 480, 490, 500
S2(*)	String	Sort array	100, 230, 600, 620, 630, 650
S3(*)	String	Sort array	760, 800, 820, 840, 850, 860, 880, 890
ST(*)	Numeric	Stack array	910, 920, 950, 960, 1010, 1020
T	String	Temporary string	320, 340, 480, 600, 620, 650, 840, 860
T0	Numeric	Stack pointer	740, 910, 920, 950, 960, 980, 1000, 1010, 1020, 1030
V	String	Temporary string	760, 800, 820, 890

The approximate results of the program run are shown in Table 9-13. These times were derived using my handy-dandy wristwatch, so those of you with deluxe chronographs may get more accurate times. The point is, however, that we have developed several sorting techniques that fare quite well.

Table 9-13. Sorting Times for Comparison. All Tests Done with 100 Randomly Generated Elements

Sort Type	Sample Time	Improvement
Substitution Sort	282 seconds	—%
Modified Substitution Sort	115 seconds	59.22%
Shell Sort	70 seconds	39.13%
Quicksort	33 seconds	52.86%

Your program results may vary depending on other program factors. When you run the program, you may notice a decrease in time efficiency for all of the earlier algorithms. Such is the price of added overhead! This points out that the Quicksort algorithm is more efficient, even with greater memory overhead, and shows a significant performance increase over earlier methods. Also, as the size of the array to be sorted increases, using an algorithm like Quicksort becomes more critical.

SORTING CONCLUSIONS

The Quicksort algorithm works very well on medium to long lists. For shorter lists, the modified Substitution Sort or the Shell Sort may work better. You may want to experiment to find out which type of algorithm will work best for you.

If you enjoy working with sorting techniques, and sorting intrigues you, then you may want to search out some additional material on the subject. Your local library is a good place to start. Many computer magazines often carry articles on sorting—it seems to be a favorite topic of computerists.

Chapter 10

Program Menus

If you walk into a restaurant (and the hostess doesn't ignore you), you will be seated and handed a menu. Without a menu, you may find it extremely difficult to find out what the restaurant has to offer so you can order.

It is just the same with program menus. If you write a program without a menu, it is difficult (at best) for a user to discover how to use the program. Without a menu, you can generate a negative feeling about your program.

MENU COMPONENTS

Menus consist of several items. Ordinarily, there is a series of choices (that's one item) that are displayed on the screen (that's two). Then, there is a way for the user to enter his choice for which option to initiate (that's three). Other than that, there isn't much more to a menu.

In this chapter, we will be going over each section of a menu, and then present a menu routine that will handle all aspects of menu display and selection. This will be a full-screen menu. There are many programs on the market today that have partial-screen menus, but these are not often as clear as those with full screens.

Fig. 10-1 shows a sample menu from the TI-99/4A. You will probably notice it as the menu that appears when you first turn your computer on. This is somewhat similar, in function, to the type of menu that will be developed at the end of this chapter. The actual design, however, will be different.

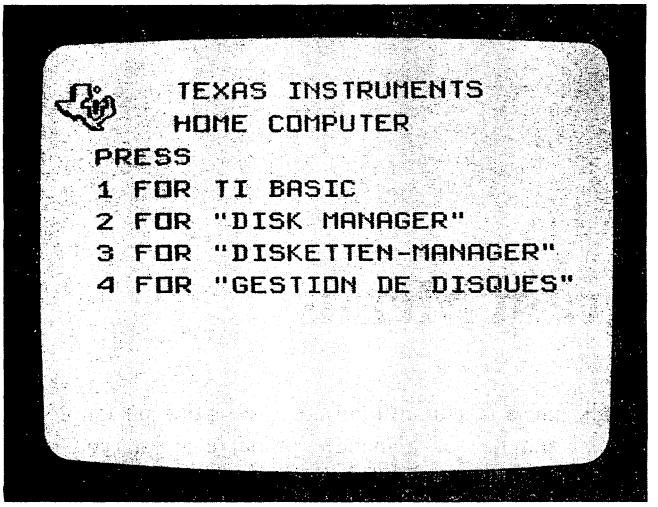

Fig. 10-1. Sample Menu.

MENU CHOICES

The choices in your menu should be clear, concise, and to the point. They should represent all the main functions and operations of your program. For instance, you may have developed a program that files names of individuals. In a menu for such a program, you decide on what choices to offer. They may be basic operations, such as:

1. Enter Names
2. Change Names
3. Delete Names
4. Exit Program

All the basic functions are covered here. Notice that this menu offers a way to quit the program. Menus that do not offer a way to exit the program can be confusing, not to mention frustrating, for the user.

This menu can be expanded to include additional operations. For ex-

ample, if you want to add a print and display routine, expand the menu to signify the new choice.

It is not necessary to outline on one menu the program's every function. It is logical to have sub-menus from a main menu. This allows your program to be broken up into distinct units.

As an example, we may have several operations that do routine chores, such as purging files, backing up disks, creating new disks, displaying system parameters, etc. Instead of putting each of these on the main menu, why not allocate a menu just for these items, and access it from a main menu choice called "System Operations"? This is logical, and it gives the user the idea that although these are important system functions, they are not in the "mainstream" of program activity.

MENU DISPLAY

This area of menu development is just as important, and in many instances more difficult, than deciding on the choices to be offered on the menu.

Most menus are displayed on one screen. It is cumbersome to have a menu occupy more than one screen. The first item on a menu is the title, or heading. It lets the user know what the choices represent. This heading can be nothing more than the title of the program, or it can be the heading for the section of the program currently executing.

At a minimum, the heading should contain this title. It serves as a road sign for the novice user, and it also reassures the experienced operator. Other items, such as underlines and the date, can also be included in a heading. To a large degree, it depends on the needs of the users.

The next portion of the display is the choices. They should be displayed in a clear, readable fashion, but your ability to do this may vary from program to program. It depends (largely) on the menu choices being displayed. For instance, there may be five choices for the menu, but each choice could be rather long. There is a difference between how EXIT would be displayed, and how TRANSVERSE POLYNOMIAL COMPUTATION would be displayed. If you horizontally center one, then the other might look awkward. You have to try to strike a happy medium.

The choices should be set off from the left margin of the screen by at least a few spaces. This distinguishes them from the heading, and gives the screen a more balanced appearance. The routine presented later in this chapter begins all choices at the seventh print position on each line.

Try to keep the screen neat and uncluttered. Having 40 possible com-

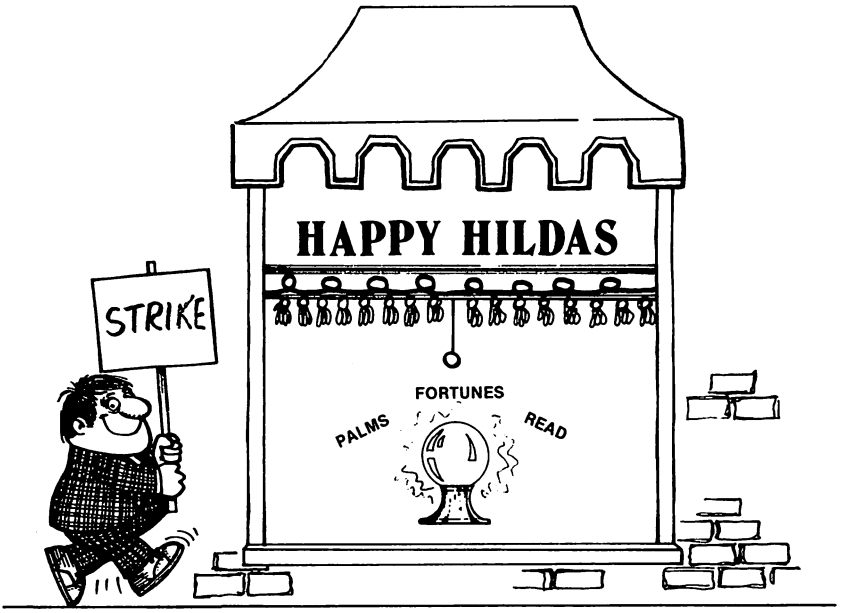

Fig. 10-2. Striking a happy medium.

mands listed on the screen can get confusing, particularly to the first-time user. Well-planned displays are inviting and don't scare people off.

Actually, this is where our happy medium makes another appearance. One of the reasons for computer phobia is a screen that jumps out at you, instead of one that invites you in. If there are too many choices on the screen the user may feel intimidated. Conversely, if the choices do not include all of the operations available from the menu, the user may feel lost.

CHOICE SELECTION

Selecting an option from a menu should be as easy as pressing one or two keys, if possible. Single-key input is easy enough for the programmer to do, and it is a nice touch.

Single-key choice selection is no real problem as long as you remember to limit the number of choices on a menu. For instance, it is a good idea to limit the number of choices to 9. If you go over 9, group several choices into sub-menus. Then all a user needs to do is press a number between 1 and 9 to initiate a selection.

The prompt for a choice should be short and to the point. *FUNCTION CHOICE:* or *ENTER SELECTION:* may sound unimaginative, but they do communicate what is desired of the user. These are excellent prompts, as long as no other action than pressing a key is required of the user. If your routine requires that the user press ENTER to initiate the option, then you may need to include that information as part of your prompt or general display. It is best to clarify the choices available. The routine at the end of this chapter requires that the user press ENTER. All necessary information is displayed on the menu.

The actual placement of the prompt line depends on how the rest of the menu is structured. The standard place is near the bottom of the screen. This lets a user's eyes follow from the choices to the prompt in a linear fashion, instead of bouncing around the screen looking for what's next.

THE MENU ROUTINE

This routine is structured to present a series of up to nine choices to a user, and then request a response. It allows a number from the screen to be entered to select your choice; when you press ENTER your choice is executed. You may have seen similar menus on commercial products.

Listing 10-1

```
1 REM      LISTING 10-1
100 CALL CLEAR
102 R=1
104 C=5
106 A$="MAIN PROGRAM MENU"
108 GOSUB 186
110 R=2
112 C=6
114 A$="---------------"
116 GOSUB 186
118 R=22
120 C=6
122 A$="PRESS SELECTION"
124 GOSUB 186
126 R=23
128 C=6
130 A$="RETURN TO ACCEPT"
132 GOSUB 186
134 FOR K=0 TO NP
136 R=6+K
138 C=6
140 A$=" "&STR$(K)&") "&M$(K)
142 GOSUB 186
144 NEXT K
146 R=6+CH
```

cont. on next page

Listing 10-1—cont.

```
148 C=6
150 A$=">"
152 GOSUB 186
154 TD=CH
156 CALL KEY(0,A,S)
158 IF S<>1 THEN 156
160 IF A=13 THEN 180
162 A=A-48
164 IF (A<0)+(A>NP)THEN 182
166 CH=A
168 IF A=TD THEN 156
170 R=6+TD
172 C=6
174 A$=" "
176 GOSUB 186
178 GOTO 146
180 RETURN
182 CALL SOUND(100,1000,10)
184 GOTO 156
186 FOR J=1 TO LEN(A$)
188 CALL HCHAR(R,C+J-1,ASC(SEG$(A$,J,1)))
190 NEXT J
192 RETURN
```

Table 10-1. Entry and Exit Variables for Menu Generator Subroutine

On entry:	On exit:
M$(*)—Array containing menu choices NP —Number of menu choices	CH—Menu choice desired

There are several requirements before entering this routine. You must set up the actual wording of the menu choices in an array, M$(*). For example, if "ENTER NAMES" is your first menu choice, you would assign M$(1) equal to "ENTER NAMES". This would be done for each of your menu choices. Also, you need to set the upper limit pointer, NP, equal to the number of choices on the menu. This is used in error and boundary checking.

When the routine returns to the caller, the value of the chosen function will be assigned to CH. This will be a number between 1 and whatever upper limit you have assigned to NP. At this point, you can use a statement such as **ON CH GOTO X, Y, Z** to branch to the various function handlers. Of course, X, Y, and Z should all be replaced with appropriate line numbers.

In the next chapter, we will deal with error routines in a program.

Table 10-2. Variable Table for Menu Generator Subroutine

Variable	Type	Purpose	Used in lines
A	Numeric	Keypress	156, 160, 162, 164, 166, 168
A	String	Temporary	106, 114, 122, 130, 140, 150, 174, 186, 188
C	Numeric	Column	104, 112, 120, 128, 138, 148, 172, 188
CH	Numeric	Menu choice desired	146, 154, 166
J	Numeric	Loop counter	186, 188, 190
K	Numeric	Loop counter	134, 136, 140, 144
M(*)	String	Menu text	140
NP	Numeric	Number of menu choices	134, 164
R	Numeric	Row	102, 110, 118, 126, 136, 146, 170, 188
TD	Numeric	Last choice	154, 168, 170

Chapter 11

Error Handling

All things in life hold the opportunity for mistakes. That is why there are erasers on pencils and error messages on computers. Knowing how to handle an eraser is one thing, but knowing how to handle an error on the computer is another. In this chapter we will discuss specific errors and what you can do about them.

Many types of computers only generate errors when you actually execute your program. The TI-99/4A, however, generates errors at three distinct times—during program entry, program initialization, and program execution. These are known as different error types, and we will discuss each of these shortly.

A big difference between the TI-99/4A and other computers is that the TI-99/4A does not allow you to trap errors while the program is executing. Some may view this as a disadvantage, but it does have one advantage, in that it forces you to make sure that your program is completely debugged. This chapter has been written to give you a better understanding of the possible errors on the TI-99/4A, so you can then make the appropriate corrections in your programming.

TYPE I ERROR MESSAGES

The first error level, or Type I errors, are syntactical input errors, or errors that were generated as you input program lines into the computer.

As each line is typed, and you press ENTER, BASIC does a quick check to make sure that what you typed makes sense. This saves you time and effort later on when running the program, because the earlier an error is detected, the greater the chance that it won't be compounded.

If, after pressing ENTER, you find that a Type I error message has been generated, it means that the computer did not accept what you typed. In other words, if you were entering a new line, you will have to retype it because the computer ignored it. If you were retyping a line that already existed, then the old line stays intact. The line you are trying to replace is only changed when the TI-99/4A does not detect any syntactical errors.

There are six different Type I error messages, and each will be reviewed shortly. The best way to avoid these types of errors is to learn the capabilities of BASIC, and then not enter information that you know will generate the errors.

Each of the following error types shows the actual message in **boldface type**; when the message is displayed on the screen it is preceded by an asterisk (*) and a beep is sounded. There is no line number given with the message, because the number only applies to the line that you tried to enter.

BAD LINE NUMBER happens if you enter an illegal line number, such as one less than 0 or greater than 32767.

BAD NAME occurs if you enter a variable name that is over 15 characters long.

CAN'T DO THAT happens when you try to have the computer do something it figures that it can't (makes perfect sense, right?). It occurs if you try to use a command in immediate mode (without a line number) that can only be used under program control, or if you try to use a command within a program line that can only be used in immediate mode.

INCORRECT STATEMENT simply means the syntax of what you entered is incorrect. For instance, you did not put quote marks around a literal.

LINE TOO LONG is supposedly an error that comes up if you try to enter a line that is too long. I don't think I have ever seen it, because the TI-99/4A beeps at you and refuses to accept any input past 112 characters.

MEMORY FULL means you have entered a program that is too big for the computer. The best solution is to optimize your coding or break the program down into smaller portions.

Notice that the error messages presented here indicate that the actual checking that is done is very rudimentary and cursory. That is why, after entering a line and pressing ENTER, the computer can give you a deter-

mination very quickly as to whether the line is acceptable or not. If the error checking at this stage was more complex, it would make entering program lines very unwieldy because of the time involved.

TYPE II ERROR MESSAGES

When you have entered your program, and then type RUN, the computer goes through some internal routines to set up the addresses and allocate the space it needs to execute your program. During this time, it is possible for errors to be generated. These are termed Type II errors.

The errors that are generated at this point are the ones that were not so obvious during program entry. Naturally, this type of error checking, along with the necessary program setup, takes a noticable amount of time. That is why there is a delay between giving the command RUN and the beginning of program execution.

There are six different Type II error messages, and they are printed on the screen with a preceding asterisk (*) and a beep is heard. Some Type II error messages give line numbers, and some don't. On those that don't, you, as the programmer, will need to try to find where the actual error occurred. This may be rather difficult and time consuming when working with a large program.

BAD VALUE means you have dimensioned an array too large or too small. The maximum array dimension is 32769 elements, and the minimum is either 0 or 1, depending on what you have previously set the OPTION BASE equal to.

CAN'T DO THAT has to do with the OPTION BASE statement and where it is placed in the program. If it is placed after an array it either explicitly or implicitly dimensioned, or if there are two OPTION BASE statements, this error will occur.

FOR-NEXT ERROR means that you have a FOR without a NEXT, or a NEXT without a FOR. This error gives a line number if there is a NEXT without a FOR, but does not give a line number if there is a FOR without a NEXT.

INCORRECT STATEMENT means a latent syntax error has been detected. These are usually ones that cannot be detected upon program entry, and include things like NEXT statements without variables, or improper use of a DIM statement.

MEMORY FULL Guess what? Your program is trying to use more data space than is available in the computer. As with the Type I error, you are going to have to optimize your code and reduce the data storage requirements.

NAME CONFLICT simply means that you are trying to use a variable one way, when it has already been defined another way. For example, you are trying to dimension the same variable twice, or a variable that has been dimensioned as an array is later referenced as a simple variable (or vice-versa).

Virtually every one of these errors is generated if something is wrong with the way your program allocates memory, or the way it uses FOR NEXT loops. By paying special attention to these program areas, you can eliminate a large portion of Type II errors.

TYPE III ERROR MESSAGES

Type II errors are generated immediately after giving the RUN command, but before actual program execution begins. Type III errors are generated after the actual program execution has commenced. They cause the program to stop working (usually called "crash" and "burn") and can cause a grown programmer to cry.

Many versions of BASIC offer ways to trap errors so that they can be compensated for under program control. In fact, TI Extended BASIC has this capability. However, this is not the case with straight TI BASIC.

If an error occurs during program execution, the error message is printed on the screen preceded by an asterisk (*), a beep is heard, and the program stops running. In most of these error messages a line number indicating where the error occurred is specified.

Several of these errors issue only a "soft error". This means that a warning is issued, and program execution continues. As these may make further problems with the program you are running, they will still need programmer attention.

BAD ARGUMENT occurs when you try to execute a statement that requires an argument, and the argument is not acceptable to the computer. For instance, trying to execute PRINT VAL("123ABC") will generate this error.

BAD LINE NUMBER is generated when you try to branch to a non-existent line number.

BAD NAME is generated when you try to CALL a non-existent subprogram. For instance, while CALL CLEAR will work, CALL LCEAR will not.

BAD SUBSCRIPT happens if you try to access a subscript that is outside the dimensioned range of an array. Also, if you type a statement like DIMA(5) you will get this error because there is no space between the M and the A. All arrays, unless explicitly defined, have an implied dimen-

sion of 10 elements. If you try to access anything outside of this range, without an explicit dimension statement, then this error will occur.

BAD VALUE happens when a value in your program statement is outside the acceptable bounds for that value; for instance, using a color value that is out of bounds, or using a statement similar to CHR$(-5).

CAN'T DO THAT is caused by trying to execute something beyond the limits of the computer; for example, trying to execute a RETURN when no GOSUB was encountered.

DATA ERROR simply means that you ran out of data. There must be at least one DATA element for each READ statement. It could also mean that you simply forgot to put commas between each DATA element, in which case the computer would think you had one long element.

FILE ERROR occurs when you are trying to do something improper in regards to a file (makes sense, doesn't it?). Usually this occurs when you are trying to do a file operation on a file that has not been opened, or was not opened for that operation.

INCORRECT STATEMENT is a latent syntax error that escaped earlier detection, as with the Type II errors. It is usually caused by missing necessary parameters or reserved wording that is out of order.

INPUT ERROR is caused by input that is not within acceptable bounds for the variable type it is to be assigned to, or by input that is too long. This is only a warning error if input is to come from the keyboard. If input comes from any other source, a hard error is generated and program execution stops.

I/O ERROR is generated when something goes wrong when saving or retrieving information from disk or tape. Actually, when this error is generated, you are given quite a bit of information to work with. This is because a two-digit error number is also printed that tells which specific operation caused the problem (first digit) and what type of error occurred (second digit).

The possible first digit values and their meanings are shown in Fig. 11-1, and those for the second digit are shown in Fig. 11-2.

MEMORY FULL occurs when your program has defined too much data storage area. You need to split the program, or cut down on your variable requirements.

NUMBER TOO BIG is a soft (warning) error only, and is generated when the program is trying to assign a number that is too big or too small to a numeric variable. Program execution will continue with the variable set to the machine's numeric limit.

STRING-NUMBER MISMATCH is caused by using a number to perform a function that requires strings, or vice versa.

Code	Operation Causing Error
0	OPEN
1	CLOSE
2	INPUT
3	PRINT
4	RESTORE
5	OLD
6	SAVE
7	DELETE

Fig. 11-1. I/O Error Codes (First Digit).

Code	Operation Causing Error
0	Device name not found
1	Device write protected
2	Bad OPEN attribute
3	Illegal operation
4	Out of space
5	End of file
6	Device error
7	File error

Fig. 11-2. I/O Error Codes (Second Digit).

ERROR CONCLUSIONS

I hope these insights into how and why the TI-99/4A generates errors will be helpful. It is good to know that there may be reasons why the computer stopped. It sure beats feeling that the computer does not like you personally.

Glossary

If you are looking for highly technical definitions of computer terms you should probably invest in a computer dictionary. But then, that is not the main reason you bought this book, is it?

Seriously, though, this glossary should give you enough information to at least enlighten you as to the meaning of some words and terms that may seem foreign at first.

ADDRESS is a unique number associated with a specific computer memory location.

ALGORITHM is a set of instructions to do a set task. A example from mathematics is to define the task, such as finding the area of a square; the algorithm to do this would be height multiplied by width.

ALPHAMERIC is a seldom-used term that denotes a string composed of letter characters only.

ALPHANUMERIC is a term used to describe a series of characters that may be either alphabetic (letters), numeric (numbers), or symbolic (such as control characters).

ARGUMENT is what users of competing personal computers usually end up in. It is also the data entered into a program that is used in a calculation or procedure to formulate an output.

ARRAY is an organized arrangement of data items. For instance, the letters FI may represent a single variable, but FI(9) represents the ninth element of a data array. Each element is addressable by changing the subscript within the parentheses.

ASCEND means to rise, or go up. In sorting, ascending order indicates that the sorted items will be ordered from lowest value to highest value.

ASCII is the American Standard Code for Information Interchange. It is a method used by the vast majority of mini- and microcomputers to encode characters through the arrangement of the individual bits of a byte.

ASSEMBLER is a program used to translate a series of commands and directives into the actual machine language codes needed to run directly on a computer processor.

BACKUP is a term that refers to the process of making a copy of valuable information in case the original copy is damaged or destroyed.

BASE usually refers to a mathematical numbering system. For instance, base 10 (or decimal), allows 10 different digits (0 through 9) in each number position. In contrast, base 16 (hexadecimal) allows 16 digits (0 through F) per number position.

BASIC stands for Beginner's All-purpose Symbolic Instruction Code. The most widely used computer language in the mini- and microcomputer markets.

BINARY is a number system based on powers of 2. The only digits in binary are 0 and 1. It is of practical use in computers where electronic circuits can only be off (0) or on (1).

BIT is a single binary digit. It is also the smallest indivisible unit of information that is understood by a computer.

BRANCH means to change the program execution from the normal contiguous series of steps to another predetermined step. Branching is usually achieved by use of a GOTO or GOSUB instruction from BASIC.

BUG is an undesirable pest that sometimes sneaks into the best of programs. Upon detection, bugs can sometimes hide again and should be documented thoroughly.

BYTE is a group of bits that collectively represent either a letter (on 8-bit computers) or a word (on larger processors).

CARRIAGE RETURN is the term given to the ASCII code that causes the computer to return to the leftmost column. It is also used to signify an end of input. Usually this is the ASCII code 13 ($D). It is generated from a keyboard by pressing the key marked ENTER (RETURN on some computers).

CASE is a term used to describe whether capital letters are used or not. Upper case means a capital letter, and is the opposite of lower case letters.

CHARACTER is either a letter, space, number or special symbol. Each character requires one byte of computer memory.

CHIP is short for "microchip" and refers to a collection of solid state circuits in one device. Each chip is generally designed and created to perform a specific task.

COMPILE means taking source code and converting it to object code. Translating from a higher-level language (such as Assembler) to instructions that the computer can understand directly (such as machine code).

COMPILER is the program that compiles, or translates, source code to object code. See COMPILE.

COMPUTER is an electronic device that performs calculations and pre-determined instructions at a very fast rate. Depending on the way the computer is programmed, this may or may not be of use to humans. Computers generally fall into one of three classifications. These are micros, minis, and mainframes.

CONCATENATION is the process of joining two or more alphanumeric strings to make one string.

CONTROL CHARACTER is a special two-key combination of keystrokes that directs the computer to do something special. On the TI-99/4A, control characters are generated by holding down the key marked CTRL and pressing any other alphameric key at the same time. Some control characters are used so frequently that they have been assigned to one specific key, for example, ENTER and LEFT ARROW.

CPU stands for Central Processing Unit, the heart of any computer system.

CROSS-ASSEMBLER is a program that executes under the control of one type of microprocessor to create program code that will ultimately be executed under the control of a different microprocesser.

CRT stands for Cathode Ray Tube, and is "computerese" for a video terminal or computer monitor.

CURSOR is a video marker appearing on the computer monitor that lets the user know where the next input or output is to occur. The cursor is generally a small blinking block or underline character.

DATA is nothing more than information. The word "data" is used to save space and ink.

DATA FORMAT are the guidelines and rules that dictate the order, style, and condition of information that must be adhered to when supplying data for a computer program.

DEFAULT is a term used to describe the assumed value that is accepted when none is offered in its place.

DEMODULATE is the process of converting signal tones to electrical impulses. Used primarily over phone lines. See MODEM.

DESCEND means to fall, or go down. In sorting, it means ordering the elements to be sorted from greatest to least value.

DISKETTE is a semi-permanent storage device for electronic information. Disks come in many different sizes, but most all are round, hence the name disks. Early attempts by the government to create a square-disk standard failed after wasting an appropriate amount of taxpayer money on feasibility studies. Subsequent efforts in this area led to the removal of the center hole and marketing them as Mag-cards for word processors. The most popular sizes today for disks are 8 inch, 5¼ inch, and 3½ inch.

DOCUMENTATION is a classy word used to describe the set of written or printed instructions that accompany a program to explain how to use it. There has been no successful approach to teach how to use documentation.

DOS is short for Disk Operating System. This is the computer program that regulates all interaction with the disk drives and the information stored therein.

DP is the computerese term for Data Processing which is the computerese term for Information Juggling.

ELEMENT, when used in relation to programming, is a single item of a larger data array. For instance, FI(9) is a single element variable of a larger array of data.

ENTRY refers to the answers that you give to questions within a program. Most entries are completed (ended) with a carriage return. This is accomplished by pressing the ENTER key.

FIELD is a term used to describe a single piece of information. Names and dates are examples of possible fields.

FILE is a collection of related records. These comprise a logical block of information that has a specific name that can be accessed by a user.

FLAG is a marker that is set within a program to indicate the presence or absence of a condition. Most often used to signal that some other event or process is to take place.

FLOWCHART is a graphic depiction of the logical process that takes place within a computer program. Primarily used by programmers and systems analysts. Good programming theory dictates that flowcharting be used as a step to completely document a program. The flowchart should be developed before a program is actually coded so that any logic errors may be uncovered and corrected with a minimum of effort.

FORMAT is the form or condition that an item should be patterned after. See DATA FORMAT.

FORMATTING is a process whereby a blank disk is organized to allow the orderly storing of information for future retrieval and use.

HACKERs are sometimes viewed as social mutants. They are fanatic programmers (see PROGRAMMER) who use computers to the exclusion of all else.

HARD COPY is printed output from a computer.

HARDWARE is the term that describes the physical circuits, chips, cards, and other items that make up a computer system.

HEADING is that portion of a report that identifies the contents of the report. It may include any information that would be of use to the user. This can include items such as titles, date, page numbers, column headings, etc.

HEXADECIMAL is the numbering system most often used in computers to represent equivalent binary data in a human-readable fashion. It is based on powers of 16, with each number position able to contain one of 16 digits, 0 through F.

INPUT is data that the computer receives from an outside source such as a keyboard, modem, or disk controller.

INTERFACE is the process of connecting or communicating between computer peripherals or between computers and humans. Interfacing may require special hardware (such as cables or modems) or software.

INTERPRETER is a program that translates each executable program step to machine code at execution time. Most versions of BASIC used on microcomputers are interpreter BASIC.

JUSTIFY means to line up all text to a certain column. It is usually used in connection with word processing. Left justify means all lines of text begin at a specific column. Right justify means to force each line of text to end at a specific column. Fill justify means to insert spaces in individual lines of text to make sure that the lines begin and end at the same columns as other lines of text in the paragraph.

KEYBOUNCE is the rapid opening and closing of a keyboard relay after the original opening and closing. Many times this results in the appearance of two of the same characters when only one was pressed.

MAINFRAME is a real big computer.

MENU is a series of choices. The choices represent the sum of all possible functions at the time the menu is presented.

MICRO means real small. In the world of computers, micro generally refers to the smallest member of the computer threesome. As a rule of thumb, if the computer can fit on a desk, it is usually a micro.

MINI means small. The middle member of the computer trilogy. Minis traditionally have less computing power than mainframes, but slightly more than micros. Buying a mini may require mortgaging your kids and subletting your house, so many people have turned to micros to do more and more.

MODEM is a device most often used to connect your computer to the phone lines so that you can call other computers and run up your phone bill. Modem is a term that means MODulate - DEModulate. Many people think that modems were created by the phone company to "hook" computerists and thus generate another source of revenue.

MODULATE means to convert electrical impulses to signal tones. These tones are used to transfer information over telephone lines. See MODEM.

MOTHER BOARD is the term given to the main circuit board of a microcomputer. It contains most, if not all, of the circuits and chips required to make the computer work.

NULL means nothing. Literally, that is what it means. When a string is equal to null, it has no length or no assignment.

PARSE means to interpret an instruction for syntactical format.

PERIPHERALS are pieces of equipment attached to a computer to help it do a specific task. Printers, disk drives, video monitors, and modems are all examples of peripherals.

POINTERS are markers used by a computer program to specify certain important limits or parameters.

PRINTER is a device used to translate electronically stored information to a permanent, printed state.

PROGRAM is a series of instructions designed to make a computer perform a specific task.

PROGRAMMER is that breed of person who enters a program into a computer for hopefully error-free execution.

RAM is Random Access Memory—it is used by the computer to store information and programs. All data stored in RAM will disappear when the computer power is turned off.

RECORD is a collection of related fields. A group of records makes up a file.

REPORT is an organized group of information used as functional output from a program.

ROM is Read Only Memory. It is computer memory used to store information permanently. It does not go away when the power is turned off.

RUN TIME signifies that period of time during which a program is being executed. In a BASIC program, it is the time after the RUN statement has been issued, and before control has been returned to the user.

SOFTWARE refers to the programs (see PROGRAM) used on a computer. Basically, this is an interchangeable term for program.

STATIC is the great nemesis of computers. Static is that little bit of electricity that causes a spark when touching a conductive surface after crossing a dry, carpeted room. It also has an untoward effect on computer data. The voltage spikes from static can erase data on disks, or in computer chips.

STRING is a collection of characters. The notation A$ is usually pronounced A string. String is short for alphanumeric string, which signifies what we just said.

SUBROUTINE is a group of computer instructions designed to collectively perform a set task. The subroutine may be "called" from any place necessary within a program, usually by a GOSUB command, and is terminated by the RETURN statement.

SYNTAX refers to the rules that govern the specific format that computer commands must follow.

TEXT FILE is a file (see FILE) saved to a disk as a series of ASCII characters, as opposed to saving it as a series of compressed coded characters. Also called a data file or an ASCII file.

TIME SHARING means to concurrently share CPU time between several different terminals. Usually accomplished on larger computers.

USER FRIENDLY means that the hardware or software interfaces well with a user. All this means is the degree of ease with which the system can be used. If a piece of software is described as being user friendly, it is easy to learn and use. Usually user friendliness cannot be added after a system has been started. It needs to be designed in from the beginning.

UTILITY is a multi-meaning word. When used as a noun, it describes a class of programming tools used to help a programmer become more productive. Generally this is accomplished through simplifing some task or helping the programmer keep track of complex areas such as variables, line referencing, and program editing. If used as a verb, "utility" is a term describing the functional use that a program has to a user. If a program has a utilitarian value, then it is of great use to the user.

SAMS

BASIC Tricks for the TI-99/4A™

Go one step beyond the fundamentals of BASIC programming on the TI-99/4A and learn to perform the "tricks" which will make your programs more useful and efficient. In clear, concise fashion, this book guides you through the logic creation and integration of some 35 routines—including routines designed to:

- Give reports a professional appearance with rounded and aligned numbers, column headings, and centered or justified lines
- Allow you to input and print times and dates in a standard format and use them in program calculations
- Create menu screens for your programs
- Sort array variables by any one of four different methods and compare the time required by each
- Convert names or other input containing upper and lower case characters to all upper or all lower case

Howard W. Sams & Co., Inc.
4300 West 62nd Street, Indianapolis, Indiana 46268 U.S.A.

$9.95/22384

ISBN: 0-672-22384-8