

THE

from DATAMOST, INC.

ELEMENTARY

11 + = x - / \$14.95

TI-99/4A

BY
William B. Sanders

0000

PLAY

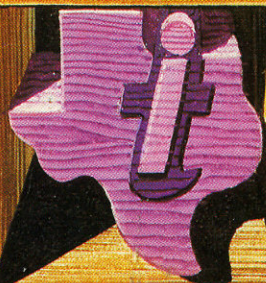
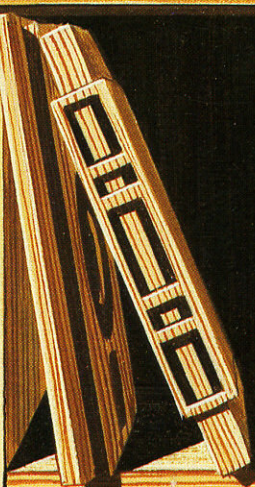
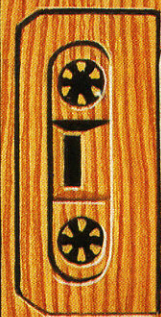
FF

RW

PAUSE

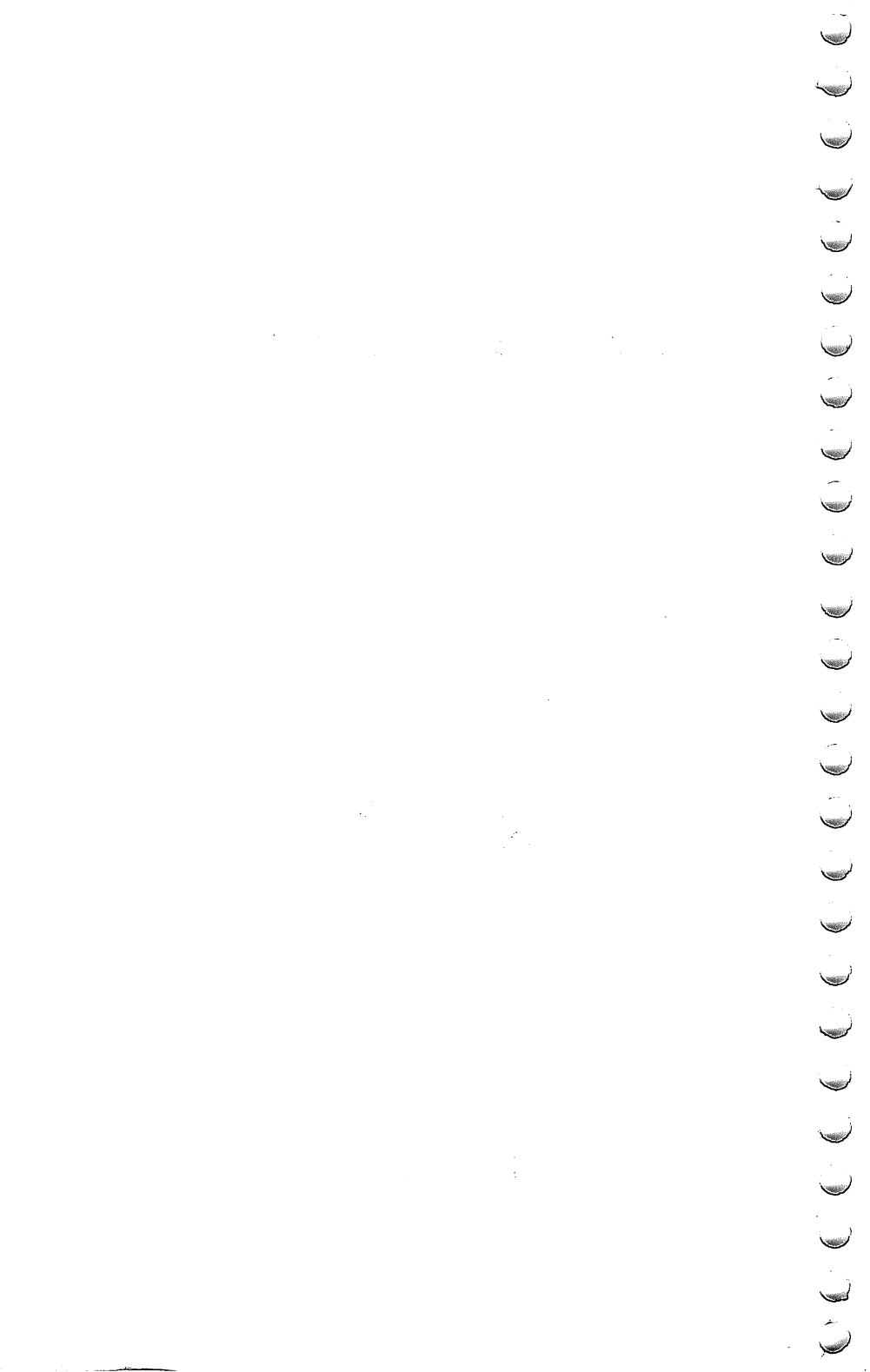
STOP

REC



Y	U	I	O
G	H	J	K
B	N	M	,

The Elementary TI-99/4A



The Elementary TI-99/4A

by

William B. Sanders, Ph.D.

San Diego State University

**Illustrations by
Martin Cannon**



8943 Fullbright Avenue
Chatsworth, CA 91311-2750
(213) 709-1202



ISBN 0-88190-247-0

This manual is published and copyrighted by DATAMOST Inc. Copying, duplicating, selling or otherwise distributing this product is hereby expressly forbidden except by prior written consent of DATAMOST Inc.

The word TI-99/4A and the TI logo are registered trademarks of Texas Instruments Inc.

Texas Instruments Inc. was not in any way involved in the writing or other preparation of this manual, nor were the facts presented here reviewed for accuracy by that company. Use of the term TI-99/4A should not be construed to represent an endorsement, official or otherwise, by Texas Instruments Inc.

Copyright 1983 DATAMOST Inc.

ACKNOWLEDGEMENTS

Several people helped directly or indirectly in the creation of **ELEMENTARY TI-99/4A**. First and foremost, I owe a great deal to Eric Goetz. Eric taught me more about programming than anyone else; especially about the importance of good algorithms in programming. Having only 16K of RAM memory in the standard TI-99/4A, a good algorithm is indeed worth a thousand bytes of memory! Secondly, Bill Parker got across the point of structured programming to me better than anyone else ever has. Done correctly, structured programming makes tasks easier, not more difficult. Finally, the folks at Texas Instruments supplied me with all the necessary hardware and a good deal of software for preparing this book. Especially helpful was Jon Campbell of TI who took the time to make sure everything got to me on time and in the right place. Likewise, Texas Instruments was both helpful and patient in providing me with answers to several questions I asked. No one could have received better support, and I am grateful for theirs.

Dave Gordon of **DATAMOST INC.** provided a world of support for the book's production. Marcia Carrozzo edited the manuscript for style and consistency, making the work a good deal clearer. She also had to learn about using the TI-99/4A to make sure that what was in the manuscript worked on the computer. Also, Marcia's strong background in math was very helpful for improving many of the programs. Martin Cannon did the art work in a way that communicated ideas creatively and visually. He gave life to the notion that a picture is worth a thousand words. The rest of the staff at **DATAMOST** were equally helpful and friendly.

Finally, my wife Eli and sons Billy and David, and even our dog Cassiopeia, put up with the inconvenience of a writer in the house. To every one of these people I owe a debt of gratitude, but as in all such efforts, if anything goes wrong, it is only the author who is to blame. Therefore, while I happily give those who assisted credit, any of the book's shortcomings are the sole responsibility of the author.

TABLE OF CONTENTS

Preface	9
chapter 1 — Introduction	11
Hardware	13
Software	14
Hooking Up Your TI-99/4A and Peripheral Equipment	15
Power On!	23
Bootting Disks	26
LOADing and RUNning from Tape	30
The TI-99/4A Keyboard	34
Summary	37
chapter 2 — Getting Started	38
Your Very First Command!	38
Your Very First Program!	40
Setting Up a Program	41
Using Your Editor: Fixing Mistakes on the Run	48
Elementary Math Operations	54
Summary	57
chapter 3 — Moving Along	59
Variables	59
Input and Output I/O	67
Looping with FOR/NEXT	75
Summary	80
chapter 4 — Branching Out	81
Branching	82
Relationals	86
Subroutines	88
Computed GOTO and GOSUB	91
Arrays	96
Summary	103
chapter 5 — Organizing the Parts	104
Formatting Text	104
Unraveling Strings	107
String Formatting	107
Setting Up Data Entry	117
Setting Up Data Manipulation	119
Organizing Output	122
Scroll Control	125
More PRINT Formatting	127
Summary	128

Chapter 6 — Some Advanced Topics	130
The ASCII Code and CHR\$ Functions	131
CALL	134
Missiles and Music: CALL SOUND	143
Summary	149
Chapter 7 — Using Graphics	151
Screen Graphics	151
Making Color	152
Bit Graphics	163
Multiple Character Graphics	171
Joystick Control	174
CALL GCHAR	179
Summary	181
Chapter 8 — Data and Text Files	182
Data Files and Your Cassette	182
OPEN, INPUT#, PRINT# and CLOSE	184
Sequential Files and the Disk System	191
Summary	198
Chapter 9 — You and Your Printer	200
Printing Text on Your Printer	202
CHR\$ to the Rescue	204
Tab Stops on your Printer	209
Printing Graphics	212
Making Your Own Graphic Characters	
on the Printer	212
Printer Graphic Utilities	216
Summary	219
Chapter 10 — Program Hints and Help	220
TI-99/4A User Groups	221
TI-99/4A Magazines	222
TI-99/4A Speaks Many Languages	223
Sort Routines	228
Utility Programs	231
Word Processors	231
Data Base Programs	235
Business Programs	236
Graphics Packages	239
Hardware	240
Summary	241
TI-99/4A Command Examples	243
Index	253

PREFACE

My first formal introduction to the workings of a computer was in 1966. At that time our wise mentor told us that if we learned the lowest level operations of a computer, we would be set for life. As a result of this philosophy, we were taught how to do everything from counting in binary and conversion to octal to the essentials of FORTRAN. The problem was that we never really sat down and programmed at a terminal. So while we had a terrific theoretical understanding of the workings of computers, we did not learn very much about actual programming.

Since that time, both computers and the people who use them have changed. To learn how to use a computer, it is unnecessary to learn everything about how they work or the theory behind their operation. It is true that by having a detailed understanding of the theory and operation of computers one can do more with them, but it is something that does not have to be done at the outset. One can learn how to program, and at a later date learn the more technical details of a computer's operation. After all, most people learn to drive without knowing the intricacies of the internal combustion engine of their automobile.

Another major change in computers has been in the transition from "mainframes" and "terminals" to small "individual" computers. Your TI-99/4A is not merely a terminal; it is a whole computer. Therefore, you are not dependent on using a piece of a larger computer, but you get the whole thing all to yourself. As a result, you are not subject to a set of policies and regulations for getting "on line" or paying for the time you use. You make your own policies and are the captain of your own computer ship. It is unnecessary to spend a lot of time discussing the organizational aspects of accessing the CPU (Central Processing Unit), time-sharing, and so forth. We will go right to the heart of the matter, programming YOUR computer.

The purpose of this book is primarily to teach you how to work your computer and program in the language called BASIC. It is *ELEMENTARY*. So, while you will learn a great deal, don't expect to learn everything about working with your TI-99/4A. Once you are finished with this book, you will realize how much more you can do with your computer, and the more you learn, the more you will find to learn. By following the instructions and keying in the examples, you will learn how to write programs with most of the instructions in the standard version of BASIC on your TI-99/4A.

As a final note, don't expect to learn everything right away. Be patient with yourself and your computer and you will be amazed at how much you will learn. If you do not understand a command or a procedure, you can always come back to it later. Try different things and play with your programs. Think up different projects you would like your computer to do and then try writing a program to do what you want. By all means, do not be afraid to attempt anything. With each step or attempt you will make some progress. While it may be slow at times, the accumulated knowledge will eventually lead to understanding.

CHAPTER 1

Introduction

This book is intended to help you operate your new TI-99/4A computer, get started programming and make life with your computer easier. It is not for professional programmers or more advanced applications. It is only the first step, and it is for BEGINNERS on the TI-99/4A computer. Everything will be kept on an introductory level but, by the time you are finished, you should be able to write and use programs.



To best use ELEMENTARY TI-99/4A it is suggested that you start at the beginning and work your way through step-by-step. I have tried to arrange the book so that each part and section logically follows the one preceding it. Skipping around might result in your not understanding some important aspect of the computer's operation. The only exception to this rule is the last chapter where I have put a number of suggestions for programs you might want to buy in order to help you write programs (called UTILITY PROGRAMS). Also, there are descriptions of programs for doing other things such as business, word processing and so forth. When you're finished with this chapter, it would be a good idea to take a quick peek at some of the programs described in the last chapter to see if any of them fit your needs while you're learning about your TI-99/4A. You don't have to purchase any programs but, depending on your interests and needs, you will find some of them very useful.



The first thing to learn about your computer is that it will not “bite” you. It requires a certain amount of care. There are ways you can destroy diskettes, tapes and information but, by following a few simple rules, you should be all right. All of us have used sophisticated electronic equipment, such as our stereos, televisions and video-tape recorders; there is a certain amount of care they require. Otherwise, there is no need to fear them. Likewise, your computer is electronic. If you pour water or other liquids on the computer while the power is on, you’re likely to damage it. Using reasonable care, go ahead and put it to use. Remember, it is virtually impossible to write a program which will harm the hardware (or electronic circuits) in your machine. At worst, one of your programs might erase the information on a tape or diskette. Throughout this book there will be tips about how to do things the right and wrong way but, for the most part, treat your computer as you would your microwave oven, garage door opener or radio — with care but without fear.

At this stage of the game it is unnecessary to learn a lot of computer jargon, but some of this jargon is necessary to help you understand how your computer operates. As we go on, more new terms will be introduced but in general the text will be plain English. Nevertheless, you should know the following just to get started:

HARDWARE

Hardware refers to the machine and all of its electronic parts. Basically, everything from the keyboard to the wires and little black chips in your computer is considered “hardware.” You will also hear the term, “firmware.” This is another type of hardware on which programs are written. Called “proms” or “eproms,” these chips have information stored in them just as tapes and disks do. Firmware is either inside your computer or in cartridges or boards you plug into your TI-99/4A. A biological analogy of hardware is the physical body, most importantly the brain, and firmware is a like inherent intelligence or transplanted intelligence.

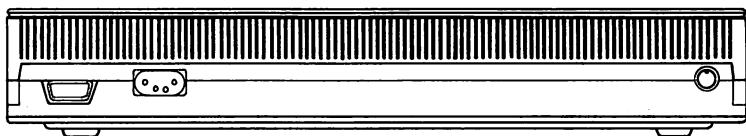
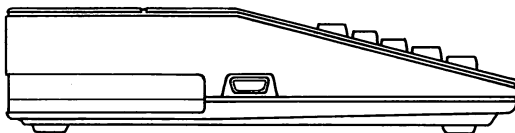
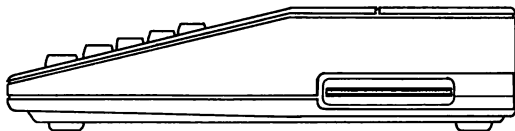
SOFTWARE

Software consists of the programs which tell the computer to do different things. Whatever goes into the computer's memory is software. It is analogous to the mind or ideas. Treating the hardware as the brain, any idea which goes into the hardware is the software. Software is to computers what records are to stereos. Software operates either in Random Access Memory (RAM) or Read Only Memory (ROM). (Firmware is hardware with "burned in" software.)

RAM You may hear people talk about expanding their RAM. This is the part of the computer's memory into which you can enter information in the form of data and programs. The more memory you have, the larger the program and more data you can enter. Think of RAM as a warehouse. When you first turn on your computer the warehouse is just about empty, but as you run programs and enter information, the warehouse begins filling up. The larger the warehouse the more information you can store there; when it is full, you have to stop. TI-99/4A's come with 16K of RAM. The "K" for computerists refers to kilobytes or thousands-of-bytes, but the actual number is 1024 bytes. (The new disk storage systems are measured in megabytes or millions-of-bytes — 1,024,000 bytes to be precise. The next time you're at a cocktail party, mention megabytes and you'll really impress everyone.) For now, all you need to know about bytes is that they are a measure of storage in computers. The more bytes, the more room you have. Think of them in the same way you would gallons, inches or meters — simply a unit of measure.

ROM A second type of computer memory is ROM, meaning "Read Only Memory." This type of memory is "locked" into your computer's chips. Your TI-99/4A's programming language, called BASIC, is stored in ROM. The difference between ROM and RAM is that whenever you turn off your computer, all information in RAM evaporates, but ROM keeps all of its information. Don't worry, though, you can save whatever is in RAM on diskettes and tape and get it back. We'll see how that is done later.

Now that you know a few terms and enough not to fear your computer, let's get it cranked up and running. If you already have your computer all hooked up and working properly, you can skip the next section and go directly to the **POWER ON!** section of this chapter.



Hooking Up Your TI-99/4A and Peripheral Equipment

The *last* thing you should do after reading this section is plug in your TI-99/4A and turn it on. Everything else should be done *first*. If you bought your computer without a tape recorder or a disk drive, it will work fine, but you will need a tape recorder or a disk drive to save information. If you have just the computer, skip to the section on hooking up your TV set to the computer.

Tape Recorder

If you are using a tape recorder, either with or without a disk operating system, hooking it up is quite simple. On the left side of your TI Program Recorder are four holes (ports) into which you plug your connecting cable. The ports are color coded for the white, red and black jacks. First, insert the white jack into the white port, the red jack into the red port and then, to the left of the red jack, insert the black plug. (There should be one port left with nothing in it to the left of the white jack.) Take the other end of the cable and insert it into the port in the *back* of your computer right next to where your power cord is connected. (There is another 9 pin port on the side of your computer, but do not connect it there.) That is all there is to it! Your cassette recorder is now ready to operate. Use ordinary cassette tapes - usually 5-10 minute tapes are the best.

Disk Drive

With the TI-99/4A you can use the TI disk drive. To connect your disk drive you will need the TI Expansion System. This system allows you to attach all kinds of peripherals to your TI-99/4A with only a single connection to your computer. Let's take it step by step.

1. Remove the lid from the Expansion System. (In the back of the system are two pressure latches. Just press them in, lift the back of the top and then slide the top back and up.)
2. Before you insert the disk drive Controller, it would be a good idea to practice connecting and disconnecting the cable that comes with the Controller. Later you will have to connect the cable to the disk drive Controller through a little opening in the back of the cavity where the disk drive goes. Insert the disk drive Controller into Slot #8.
3. Remove the metal shield that covers the disk drive port. It is held in by two screws on the top and bottom of the Expansion System.

4. Take the cable you practiced connecting to the disk drive Controller and, reaching back into the disk drive cavity of the Expansion System, connect the cable to the controller. This is a tricky operation, but it can be done. (If you cannot do it. Remove the disk drive Controller and practice some more.)
5. Reach inside the disk drive port and *carefully* pull out the power connection cord and plug it into the back of the disk drive. The cord has different colored wires. (The connection is difficult to find on the back of the disk drive, but it is there under some wires.)
6. Connect the flat ribbon cable to the back of the disk drive. Note the location of the "slit" in the flat connection part on the back of the disk drive and the corresponding "plug" in the cable connection.
7. Slip the disk drive into the cavity with the red light of the disk drive toward the TOP of the Expansion System.
8. Put two short screws into the top of the Expansion System and the long screw into the bottom to secure the disk drive.

TV or Monitor

In order to see what's going on in your computer, you need a TV set. On some computers it is necessary to purchase an RF modulator, but your TI-99/4A comes with an RF modulator you attach to your TV set. Just plug one end of the connecting cable that comes with your TI-99/4A into the jack in the back of your computer, directly behind the "1" key, and the other end into the box that you attach to your TV. The box is attached to the antenna leads marked VHF on the back of your TV set, and the switch on the box is flipped to MODULATOR. Finally, there is a switch on the bottom of the black box you attached to your TV set. Switch it to channel 3 or 4 depending on what channel is free in your area. Then set your TV dial to

channel 3 or channel 4. If you're not certain which channel should be used, try both of them. It won't hurt you TV or computer if you have the wrong one. Once you've found your proper channel, you are all set.

Another option you can use with your TI-99/4A is a monitor instead of a TV set. Basically, a monitor is the same as a TV except it has higher resolution. It is quite useful if you're doing a lot of word processing. The TI Color Monitor comes with a special cable that connects to the same port as your TV cable, in the back of your computer. If you use another brand of monitor, the 5-pin DIN audio cable found in stereo and electronic stores can be used to connect your computer to a monitor. One end of the cable is 5-pin DIN, and the other end is an RCA standard male plug. The following descriptions of monitors and TV sets are the range of video devices you can use with your TI-99/4A.

Types of TV Sets

TVs come in a "jillion" different shapes, sizes, etc.; either a color or black and white set will work fine. **BE CAREFUL** in the selection of the TV set you buy! Not all televisions work well with your TI-99/4A; so ask *before* you buy. When I bought my TV set, a color one for the graphics, I simply looked at the color TVs being used on the computers in the stores and bought the same make and model at an "El Cheapo" discount house. An inexpensive way to get clear text is to purchase a black and white set. It has better resolution than a color set, is less expensive and is good for word processing. Best of all, you can get one for as little as \$50 and used ones for even less. Whatever the case, check to make sure that the TV set you purchase will work with your TI-99/4A.

Types of Monitors

Green screen This type of monitor gives a green on black display and can be bought for between \$100 and \$200. The green and black display is quite good for doing word process-

ing and non-graphic programming since it is easy on the eyes. However, since this display presents only green and black, it is not too good for color graphics. Monitors also come with amber or blue screens, but the green screens are the most popular.

Black and white This monitor is essentially the same as the green screen, but is in black and white instead of black and green. It is more expensive than black and white TV sets, and while it gives better resolution than a television set, the extra cost may not be worth the difference. If you are considering the purchase of a black and white monitor, compare the resolution with a black and white TV set first to see if the extra cost is justified.

Color This type of monitor is the most expensive, but for people who work a lot with graphics, it is probably worth the added cost. The color monitor provides the high resolution for seeing graphics in detail. Since TI makes an excellent color monitor especially for the TI-99/4A, your best bet for a color monitor would be the TI.

PRINTERS

This section simply tells you how to hook up your printer and a little about the different kinds of printers. If your printer is already hooked up and working, take a look at Chapter 9 for tips on maximizing your printer's use.

TYPES OF PRINTERS

There are three basic kinds of printers - dot matrix, letter quality and thermal. However, for specialized use there are also devices called plotters, ink-jet printers, line printers, laser printers and drum rotate printers. For heavy business use or specialized applications, you may want to ask your dealer about these other ones not described below.

DOT MATRIX First, the most popular kind of printer is the dot matrix printer. This printer has a number of little pins which are fired to form little dots that print out as text or graphics. The advantage of dot matrix printers is their relatively low cost and the fact that many of them can do both text and graphics. The improved quality in the text printing of dot matrix printers gives an almost letter quality product and usually can give you several different type faces. In Chapter 9 there are several examples of different printing modes on dot matrix printers. We will be using the TI-99/4 Printer for our examples since it is directly compatible with the TI-99/4A. You will need the RS232 Interface Card plugged in your Expansion System. This card can accommodate both serial and parallel printers. The TI-99/4 Printer can be used with either serial or parallel connections, but we will deal with the serial connection since the cable that comes with the printer is for serial.

LETTER QUALITY Second, for people whose major use of their computers is to do word processing, there are letter quality printers. Most of these are daisy wheel printers and type characters in much the same way as a typewriter. Each symbol has a molded image like those found on typewriter heads. These printers are not good for graphics, but for the user who wants top-notch looking letters, manuscripts, reports and other written documents, these types of printers are the best. Letter quality printers tend to be relatively expensive so for most written materials dot matrix printers are fine. The thing to do before you buy is compare. Special interfaces will be needed to connect a letter quality printer to your TI-99/4A; so make sure you get a demonstration with the correct interface before buying a printer.

THERMAL Third, for those people who are really on a budget, there are thermal printers. These printers work with a special kind of paper, usually on a roll, and make a picture of what is on the computer screen. They can easily handle both text and graphics, but the quality of output is relatively low and the paper is very expensive. The best feature of these printers is their small size and light weight; for people who travel with their computers and need print-outs, they can be

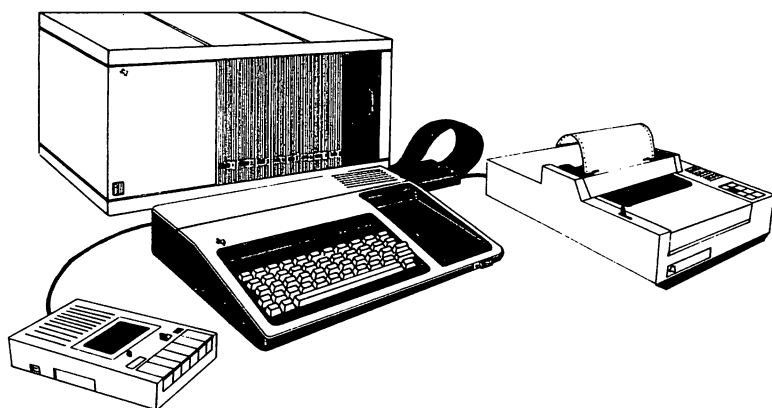
handy. Like dot matrix and letter quality printers, make sure the thermal can interface to your TI-99/4A before making a purchase.

FREE ADVICE

Before you buy a printer, decide what you will need it for and then look at the features of the different kinds before buying!!! And by all means, ask to see a demonstration on a TI-99/4A. Never let a salesperson convince you a certain printer will work without seeing a demonstration. Even a salesperson with the best intentions (e.g., they think a certain printer is the best for your needs) may not realize that the model cannot be interfaced to your machine. Only a demonstration is sufficient to remove all doubts! The RS232 Module can connect to both serial and parallel printers, but special cables are required. The parallel connection to the RS232 Module is different from most other parallel connectors, so be sure to get the correct type of cable connections if you plan to use a parallel printer.

CAUTION

NEVER insert or remove cables or interfaces to your computer while the **POWER IS ON!** Even if you are rich and can afford to buy new chips every time you blow them by messing with the hardware on your TI-99/4A while the power is on, you might give yourself the **SHOCK** of a lifetime by doing so.



Other Gadgets

Besides the disk drive, TV/monitor and printer, most new users do not have anything else to hook up at this point, so you can skip on to the next section. However, if you plan on expanding your TI-99/4A or have other gadgets you bought with your system, you had better read the following section.

Many Ports of Call

The nicest feature of the TI-99/4A is its expandability and adaptability. The Expansion System allows you to connect all kinds of things to your computer to enhance your system.

Modem A MODEM is a device which allows your computer to communicate with other computers over telephone lines. These devices usually require that you hook up your telephone to a part of the modem, or place the phone in an acoustic sender/receiver. The Telephone Coupler can be used with the TI-99/4A simply by connecting it to the RS-232 Interface and placing your phone's handset on it. Not only can the modem be used to call up computer bulletin boards, but you can access such information centers as The Source to get everything from weather reports to airline tickets!

More Wonderful Gadgets There are numerous other cartridges and interfaces to make the TI-99/4A into a multifaceted machine. Special interfaces will allow you to access and use a variety of peripherals such as various disk drive systems, printers and devices made for other computers. So while the TI-99/4A is a terrific microcomputer all by itself, it is fully expandable to make it even better.

POWER ON!

SYSTEM CHECK-OUT

Now that you have your TI-99/4A all set to go, you simply plug it in, along with your TV or monitor, disk drive and printer, turn on the power and let her rip! If you have an expansion system, use the following sequence to turn everything on:

1. Turn on Expansion System.
2. Turn on Computer and TV/monitor.

On the left-hand side of your Expansion System you will find the ON/OFF power switch. Flip it to the ON position. On the right-hand side in the front of your computer is a power switch. Slide it to the right and a RED light on your computer will turn on. If everything is connected, your TV screen will display the following:

=COLOR BAR=



TEXAS INSTRUMENTS
HOME COMPUTER

READY-PRESS ANY KEY TO BEGIN

=COLOR BAR=

[c]1981 TEXAS INSTRUMENTS

If you have a color TV/monitor, the bars across the screen will be in several different colored blocks. On a black and white TV or non-color monitor, they will appear as different shades. Do as it says and press a key.

Now your screen should read:



TEXAS INSTRUMENTS
HOME COMPUTER
PRESS
1 FOR TI BASIC

NOTE: If you have your disk system connected and the DISK MANAGER cartridge installed, you will get a different menu, but you still press "1" to get TI BASIC.

Go ahead and press the 1 key in the upper left hand corner of your keyboard. The screen will go blank, and in the lower left hand corner you should see

TI BASIC READY



Directly below the TI BASIC READY, you will see a little black blinking square. It is called the cursor, indicating your computer is waiting for you to press some keys and tell it what to do. Press the ENTER key several times and the message on your screen will scroll off the top. That done, you know your keyboard and computer are all set. We will return to the keyboard in a bit, but first let's check out your printer, disk drive and/or cassette tape recorder. (Skip the sections that do not apply to your system.)

Printer Check Out

To see if your printer is working correctly, turn off all the power to your computer, printer and Expansion System. Connect the cable to the RS232 Module and the printer. Put the ribbon and some paper into your printer. Then:

1. Turn on the power to your printer.
2. Turn on the power to your Expansion System.
3. Turn on your computer and TV/monitor.

Now, key in the following program *exactly* as it appears below. First write in the word NEW and press ENTER. (<ENTER> means press the button marked ENTER.)

```
10 OPEN #1:"RS232" <ENTER>
20 PRINT <ENTER>
30 PRINT #1:"MY PRINTER IS WORKING!"
  <ENTER>
40 PRINT #1:"My lower case is working." <ENTER>
50 CLOSE #1 <ENTER>
```

Make certain you have written the program as it appears above. If there are even minor differences, change it so that it is precisely the same. Now key in the word RUN on your computer and <ENTER>. If your printer is attached properly, it will print out the message,

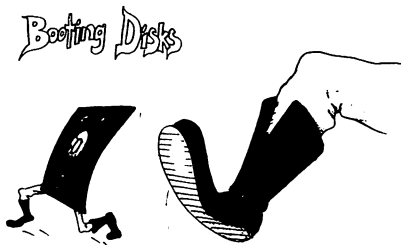
MY PRINTER IS WORKING!

My lower case is working.

If an error message jumps on the screen, it means that you wrote the little test program improperly; so go back and do it again. If the system "hangs up" - the screen goes blank and nothing happens - check to make sure the printer is turned on and is ON LINE. If it still doesn't work, turn off the printer, Expansion System and the computer and review the steps for hooking up your printer.

BOOTING DISKS

Assuming your disk drive is working correctly, let's "boot" a diskette on your TI disk drive. First make sure the Disk Manager cartridge is installed in the cartridge slot next to the keyboard. Turn on your computer (or press FCTN and = for QUIT). Next, take a blank diskette and insert it in the disk drive and close the door. Make sure the label on the diskette is facing to the right and the notch is facing upwards. Press any key and then you should have the following menu:





TEXAS INSTRUMENTS
HOME COMPUTER

PRESS

- 1 FOR TI BASIC
- 2 FOR "DISK MANAGER"
- 3 FOR "DISKETTEN-MANAGER"
- 4 FOR "GESTION DE DISQUES"

Press 2 <ENTER> and you will be presented with the following menu:

DISK MANAGER

- 1 FILE COMMANDS
- 2 DISK COMMANDS
- 3 DISK TESTS
- 4 SET ALL COMMANDS FOR SINGLE DISK PROCESSING

Press 2 <ENTER> and a third menu will appear:

DISK COMMANDS

- 1 CATALOG DISK
- 2 BACKUP DISK
- 3 MODIFY DISK NAME
- 4 INITIALIZE NEW DISK

Press 4 <ENTER> and your screen will show:

INITIALIZE NEW DISK

MASTER DISK (1-3)? 1

Press 1 <ENTER> and a note will appear:

DISK NOT INITIALIZED

NEW DISKNAME? _ _ _ _ _

Now key in the name PRACTICE-1 for the name and <ENTER>. Now your screen will show:

TRACKS PER SIDE? 40 (Press <ENTER>)
SINGLE SIDE [Y/N]? Y (Press <ENTER>)
SINGLE DENSITY [Y/N]? Y (Press <ENTER>)

Now your disk will start spinning (the red light will come on) and the following messages will appear:

INITIALIZE NEW DISK
WORKING PLEASE WAIT

After spinning for a while and flashing some numbers on the screen, your TV will say:

COMMAND COMPLETED
PRESS: PROC'D, REDO,
BEGIN, OR BACK

These commands are on that strip above your keyboard. To get them we need to use the FCTN key and the key below the label on the strip. Press the FCTN and 5 keys for BEGIN. That will take you back to the DISK MANAGER menu. Now press 2 <ENTER> to get to the DISK COMMANDS menu, and press 1 <ENTER>. Now you are at the CATALOG DISK section: press <ENTER>. Your screen will show

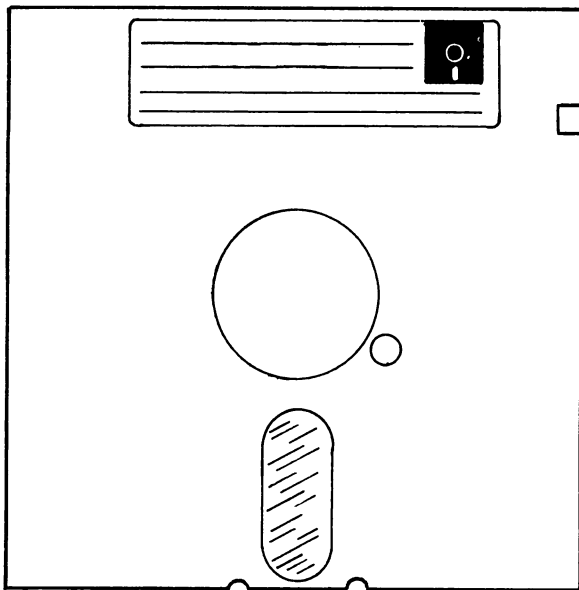
WHERE DO YOU WANT LISTING?
1 SCREEN
2 SOLID STATE PRINTER
3 RS232 INTERFACE
4 OTHER
YOUR CHOICE?

Press 1 <ENTER> and your disk drive will spin and you will see the following at the top of your screen:

CATALOG DISK

DSK1 - DISKNAME= PRACTICE-1
AVAILABLE= 358 USED= 0

At the bottom of your screen are the **COMMAND COMPLETED** choices, and this time choose **BEGIN** and then **BACK**. You will now be at the "Introductory Screen." You have successfully initialized a diskette.



Once you have initialized your diskette, you **NEVER** have to initialize it again. If you do, you will destroy any files you have saved on your diskette. Of course you might want to remove all the files from your diskette, and initializing it is one way to do it. However, for the most part, once you initialize a diskette, you simply use it until it is filled up with files, and then initialize a new diskette for additional files. Don't worry though, there's plenty of room on your diskette, and it will be a while before you fill it up. In Chapter 2 we will discuss saving files to your diskette.

WHAT DISKETTES TO BUY

When you purchase diskettes, all you need are Single Density, Single Sided, Soft Sector diskettes. These are the least expensive 5 1/4 diskettes, and it is not advisable to buy double density diskettes. They will work fine, but they are more expensive and they will be formatted as single density diskettes anyway, so don't spend the extra money. In my experiences, cheap diskettes work as well as expensive ones, and I have not found more errors on the less expensive ones. However, the more expensive ones tend to be checked more thoroughly than the cheap ones; so I will leave the decision up to you. The best thing to do is to check with other people who use the TI-99/4A with a disk system and see what their experiences have been.

(See Chapter 9 for more details on using your disk system.)

LOADing and RUNning From Tape

The procedure for loading and running programs from tape is quite simple. The following steps show you how:

STEP 1 Make sure your tape recorder is connected and rewind it to the beginning. Set your tape counter to 000. If you have a tape with programs on it, use it to test loading. (A game cassette, *not* cartridge, will work fine.) If you do not have a tape with a program on it, enter the following program: (To get the quotation marks, press the FCTN (Function) and P keys *simultaneously*.)

```
NEW <ENTER>
TI BASIC READY (Appears on screen)
10 PRINT "<YOUR NAME>" <ENTER>
20 END <ENTER>
SAVE CS1 <ENTER>
```


At this point your computer will prompt you through the SAVE process. Do what it says. *EXCEPTION: When it says PRESS CASSETTE RECORD, press BOTH the PLAY and RECORD keys on your recorder.*

* REWIND CASSETTE TAPE	CS1
THEN PRESS ENTER	
* PRESS CASSETTE RECORD	CS1
THEN PRESS ENTER	
* RECORDING	
* PRESS CASSETTE STOP	CS1
THEN PRESS ENTER	
* CHECK TAPE (Y OR N)? (Choose Y)	
* REWIND CASSETTE TAPE	CS1
THEN PRESS ENTER	
* PRESS CASSETTE PLAY	CS1
THEN PRESS ENTER	
* CHECKING	
* DATA OK (If everything is OK)	
* PRESS CASSETTE STOP	CS1
THEN PRESS ENTER	

STEP 2 To make certain everything is OK, turn your computer off, and then turn it on again. This will make double sure your program is saved on tape. Get TI BASIC up and do the following:

OLD CS1 "<name>" <ENTER>

The command OLD loads your program from tape. At this point you will be prompted through the loading process. Do as prompted.

* REWIND CASSETTE TAPE	CS1
THEN PRESS ENTER	
* PRESS CASSETTE PLAY	CS1
THEN PRESS ENTER	
* READING	
* DATA OK	
* PRESS CASSETTE STOP	CS1
THEN PRESS ENTER	

If you did not successfully save your program to tape, you will get the following:

*ERROR - NO DATA FOUND	
PRESS R TO READ	CS1
PRESS C TO CHECK	
PRESS E TO EXIT	

Or

* ERROR IN DATA DETECTED	
PRESS R TO READ	CS1
PRESS C TO CHECK	
PRESS E TO EXIT	

The first error means your recorder simply did not get the information on tape, and the second means that some error was in part of the program SAVED. If one of these errors occurs, try loading the program again with OLD. Make sure your tape has been rewound this time.

If you keep getting errors, one of the following gremlins may have crept in:

1. Your sound or tone level on your recorder needs adjusting.
2. You used a recorder that is not compatible with your computer.
3. Your cassette tape is bad. Make sure the write protect notches on your tape are in place.
4. Your connections are bad. Check to see that everything is connected properly.
5. Your recorder is too close to your TV set or on a metal surface. The TV or metal surface acted like a big magnet and wiped out everything on your tape.

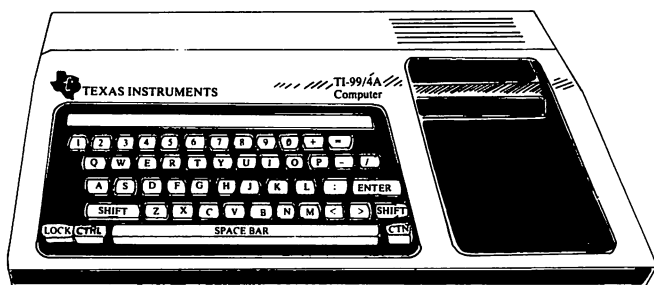
Before you go running back to the store where you bought your computer, check out these items thoroughly. If you still cannot save a program and recover it, then go back to your TI dealer and get help. (Phone first, since you might be able to solve the problem that way.)

Tape To Disk Transfer

If you have both a tape and a disk system and you don't want to wait for the longer loading time of tapes every time you run it (especially when you start accumulating several programs on tape), why not transfer your tape files to disk? Just boot your DOS, put a formatted disk into the drive, initialize it and then load your program on tape. Once your tape program is loaded, simply write in `SAVE Disk1."<name of file>"` and now your tape program is on disk! Makes life simpler.

Cartridge Programs

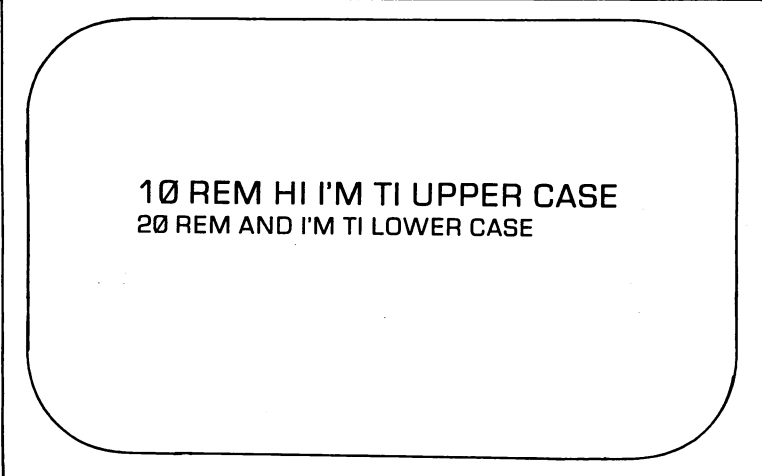
When you purchase cartridge programs for your computer, just insert the cartridge into the cartridge port and turn on your computer. It will automatically run the program for you.



The TI-99/4A Keyboard

Almost Like A Typewriter: The Familiar Keys

If you are familiar with a typewriter keyboard, you will see most of the same keys on your TI-99/4A. For the most part, they do almost the same thing as your typewriter keys. If you type in the word COMPUTER, hitting the same keys you would on a typewriter, the word COMPUTER appears on the screen just as it would appear on paper from a typewriter; however, the upper-case (capital letters) and lower-case letters do not work exactly the same as a typewriter's. On the TI-99/4A, your upper/lower-case characters are simply large and small upper case letters. When the ALPHA LOCK key is pressed, all letters are upper case, but the SHIFT key is still used to get the characters printed on top of the keys. For example, the "7" and "&" characters work the same with or without ALPHA LOCK on. You will notice that the screen has only 28 columns instead of 80 like most typewriters. Also you cannot type just anything on the screen. If you start typing away, you'll get an error message every time you press ENTER unless you put in the proper commands (e.g., *BAD NAME, *CAN'T DO THAT.) Otherwise, though, think of your keyboard as you would a typewriter keyboard. *NOTE: In most of the programming examples, we will be using upper-case only, so press ALPHA LOCK and leave it in upper-case.*



```
10 REM HI I'M TI UPPER CASE  
20 REM AND I'M TI LOWER CASE
```

Keys You Won't See on a Typewriter

While most of the keys on your TI-99/4A look like those on a typewriter, many do not, and they are important to know about. The following keys are peculiar to your computer; you will soon get used to them even though they will be a bit mysterious at first:

FCTN (Function key) This key, located in the lower right hand corner of your keyboard, is used for accessing the characters printed on the side of the keys. The strip along the top of your keyboard indicates uses of the FCTN key along with the other keys for editing and other special functions. Press FCTN and = *simultaneously* to see what happens.

CTRL (Control) In the lower left hand corner of your keyboard is the CTRL key, called the "control key." By pressing the CTRL key and one of the other keys you can get different effects. We will not be using control characters too much at this stage of the game. Try holding the CTRL key down and pressing the G key. This will give you a graphic character. The others are used for more advanced applications and will be introduced when needed. For the time being, don't worry about using it.

ENTER The ENTER key is something like the carriage return on a typewriter. In fact, you may see it referred to as a Carriage Return or CR in computer articles. It works in an analogous manner to a typewriter's carriage return because the cursor bounces back to the left hand side of the display screen after you press it. There are many uses for the ENTER key which will be discovered as you get into programming.

Arrow keys On the E, S, D and X keys are vertical and horizontal arrows. By pressing one of those keys and the FCTN key, you can move the cursor without affecting the text on the screen. These keys are used extensively in editing. For example, if you key in PRUNT instead of PRINT, you can back over the word and make the correction without having to start over again. Go ahead and try it. In the next chapter we will discuss in more detail how these and other keys are used in editing.

Some New Meanings for Old Keys

Some of the familiar keys have different meanings when used on the computer. Many are math symbols you may or may not recognize. In the next chapter we will illustrate how these keys can be operated and discuss them in detail. For now let's just take a quick look at the math symbols.

Symbol	Meaning
+	Add
-	Subtract
*	Multiply (different from conventional)
/	Divide (different from conventional)
	Exponentiation

In addition to some of the new representations for math symbols, other keys will be used in a manner which may be unfamiliar to you. As we go on, we will explain the meanings of these keys, but just to get used to the idea that your TI-99/4A has some special meanings for keys, we'll provide an example.

Symbol	Meaning
\$	Used to indicate a string variable and hexadecimal value.

For the time being, don't worry about understanding what all of these symbols do; simply be prepared to think about these symbols in "computer talk." As you become familiar with the keyboard and the uses and meanings of these symbols, you will be able to handle them easily, but the first step is to be aware that different meanings do exist.

Changing Keys

You may have wondered what the plastic strips that came with your computer are for. One of them is labelled:

DEL INS ERASE CLEAR BEGIN PROC'D AID
REDO BACK QUIT

The others are blank. Take the labelled strip and place it above the keyboard in the tray above the keys with numbers on them. The bottom row has a gray dot corresponding to the gray dot on the FCTN key. If you press the FCTN key and the key right below the label, under certain conditions your computer will do what the label on the strip suggests. We saw that if you press the FCTN and = keys together, your computer will QUIT, just as the label says. You will receive additional strips with different commands on the strips for different commercial programs. This is because the meaning of the keys can be changed and, depending on program requirements, some functions will be substituted for others. Thus, if you decide you want to change the function of the keys, you will need a different strip to label. In Chapter 6, we will explain how this is done; in the meantime, we will just use the strip with the labels.

SUMMARY

This first chapter has been an overview of your new machine. You should now know how to hook up the different parts of your TI-99/4A and get it running. You should also be able to format a diskette, list the contents of a disk, and load and run a program from disk or from tape. Finally, you should be familiar with the keyboard and know what the cursor means. At this point there is still much to learn, so don't feel badly if you don't understand everything. As we go along, you will pick up more and more; what may be confusing now will become clear later. Have faith in yourself and in no time you will be able to do things you never thought possible.

The next chapter will get you started in learning how to program your TI-99/4A. It is vitally important that you key in and run the sample programs. It is recommended that you make changes in the sample programs after you have first tried them out to see if you can make them do slightly different things. Both practical and fun (and crazy!) programs are included so that you can see the purpose behind what you will be doing and enjoy it at the same time.

CHAPTER 2

Ladies and Gentlemen, Start Your Engines

Introduction

This chapter will introduce you to writing programs in the language known as BASIC. TI-99/4A BASIC is different from some other versions of the language, and if you are already familiar with BASIC, you will spot these differences. However, if you are new to the language then you will find programming in BASIC very simple. To get ready, turn on your computer, and when the TI BASIC READY and cursor come up on your TV, you are all set to begin programming. If something else is on your screen, key in the word NEW to clear memory.

Your Very First Command! PRINT

Probably the most often used command in BASIC is PRINT. Words enclosed in quotation marks following the PRINT command will be printed to your screen, and numbers and variables will be printed if they are preceded by a PRINT command. It is used to command your computer to print output to the screen or the printer from within a program or in the Immediate mode. You may well ask what the difference is between the Immediate and the Program mode. Let's take a look.

Immediate Mode The Immediate mode executes a command as soon as you press ENTER. For example, try the following: (The notation <ENTER> means to press the key marked ENTER.)

```
PRINT "THIS IS THE IMMEDIATE MODE" <ENTER>
```

If everything is working correctly, your screen should look like this:



PRINT "THIS IS THE IMMEDIA
TE MODE"
THIS IS THE IMMEDIATE MODE
(cursor)

See how easy that was? Now try PRINTing some numbers, but don't put in the quote marks. Try the following:

PRINT 6 <ENTER>
PRINT 54321 <ENTER>

As you can see, numbers can be entered without having to use quote marks, but as we will see later, the actual value of the number is placed in memory rather than a "picture" of it.

Program Mode This mode delays the execution of the commands until your program is RUN. All commands which begin with numbers on the left side will be treated as part of a program. Try the following:

10 PRINT "THIS IS THE PROGRAM MODE" <ENTER>

nothing happens, right?

Enter the RUN command and your screen should look like this:

```
10 PRINT "THIS IS THE PROGRA  
M MODE"  
RUN  
THIS IS THE PROGRAM MODE  
** DONE **
```

Your Very First Program!

Clearing the Screen and Writing Your Name

Let's write a program and learn two new commands. First, the new commands are CALL CLEAR and END. The CALL CLEAR command clears the screen and places the cursor in the lower left hand corner. The END command tells the computer to stop executing commands. From the Immediate mode write in the CALL CLEAR command to see what happens. Now, let's write a program using CALL CLEAR, END and PRINT. From now on, press the ENTER key at the end of each line. Throughout the rest of the book, I will no longer be putting in <ENTER> except in reference to entries in the Immediate mode.

```
10 CALL CLEAR  
20 PRINT "<YOUR NAME>"  
30 END  
RUN <ENTER>
```

All you should see on the screen is your name, ** DONE ** and the blinking cursor. Now, as a rule of thumb, *always* begin your programs with CALL CLEAR. This will help you get into a habit which will pay off later when you're running all kinds of different programs. There will be exceptions to the rule, but for the most part, by beginning your programs with CALL CLEAR, you will start off with a nice clear screen rather than a cluttered one. Also, we want to make liberal use of the REM statement. After the computer sees a REM statement in a line, it goes on to the next line number, executing nothing until it comes to a command which can be executed. The REM statement works as a REMark in your program lines so that others will know what you are doing and as a reminder to yourself what you have done. Just to see how it works, let's put it into our little program.

```
10 CALL CLEAR
20 REM THIS CLEARS THE SCREEN
30 PRINT "<YOUR NAME>"
40 END
50 REM THIS MAGNIFICENT PROGRAM WAS
   CREATED BY <YOUR NAME>
```

Now RUN the program and you will see that the REM statements did not affect it at all! However, it is much clearer as to what your program is doing since you can read what the commands do in the program listing.

Setting Up a Program

Using Line Numbers

Now that we've written a little program let's take a look at using line numbers. In your first program we used the line numbers 10, 20 and 30. We could have used line numbers 1, 2 and 3 or 5, 6 and 7 or even 1000, 2000 and 3000. In fact, there is no need at all to have regular intervals between numbers, and line numbers 1, 32 and 1543 would have worked just fine.

However, we usually want to number our programs by 10's, starting at 10. You may well ask, "Wouldn't it be easier to number them 1, 2, 3, 4, 5, etc.?" In some ways maybe it would, but overall, it definitely would not! Here's why. Type in the word LIST <ENTER>, and if your program is still in memory it will appear on the screen. Suppose you want to insert a line between lines 20 and 30 which prints your home address. Rather than re-writing the entire program, all you have to do is to enter a line number with a value between 20 and 30 (such as 25) and enter the line. Let's try it, but first remove the END command in line 20. To do so simply enter the line number and <ENTER>. (i.e., 20 <ENTER>).

```
25 PRINT "<YOUR ADDRESS>"
RUN <ENTER>
```

Aha! You now have your name and address printed on the screen, and you simply wrote in one line instead of retyping the whole program. Now, if we had numbered the program by 1's instead of 10's, you would not have been able to do that since there would be no room between the lines numbered 2 and 3 like there was between the lines numbered 20 and 30. You would have had to rewrite the whole program. With a small program this would not be much of a problem, but when you start getting into 100 and 1000 line programs, you'll be glad you have space between line numbers!

Listing Your Program

As we just saw, using the word LIST gives us a listing of our program. To make it neat, type in (SHIFT) CALL CLEAR and LIST <ENTER>, and you'll get a listing on a clear screen. Once you start writing longer programs, you won't want to list everything, only portions. Let's examine the options available with the LIST command

What you Write What you Get

LIST	Lists entire program.
LIST 20	Only line 20 is listed (or any line number you choose).
LIST 20-30	All lines from 20 to 30 inclusive are listed (or any other range of lines you choose).
LIST -40	Lists from the beginning of the program to line 40 (or any other line number chosen).
LIST 40-	Lists from line 40 (or any other line number chosen) to the end of the program.

Try listing different portions of your program with the options available to see what you get. The following commands will give you some examples of the different options:

```
LIST 25
LIST 20
LIST -20
LIST 25-30
```

Renumbering Lines

Suppose you number your lines by 10, and then after working on your program, you find that you have to fill in all the spaces between lines 20 and 30. Then you find that you have to add still more between lines 20 and 30, but there is no more room. (This will happen if you program - even when a program is well planned.) With TI BASIC this is not a problem. All you have to do is use the RESEQUENCE command. This command will renumber your program for you. To use it employ the following format:

RESEQUENCE (First line number), (Increment between lines)

For example enter the following program:

```
10 REM THIS DOES NOTHING
11 REM EXCEPT SHOW YOU HOW
12 REM TO USE THE RESEQUENCE
13 REM COMMAND
```

Now enter

```
RESEQUENCE 10, 10 <ENTER>
```

Now enter

```
LIST <ENTER>
```

Your program is now numbered by increments of 10.

Automatic Line Numbering

To save programming time, you can have the computer automatically enter the line numbers for you. Using the NUMBER function, you can specify the beginning line number and the increments, exactly in the same format as used with RESEQUENCE.

NUMBER (First line number), (Increment between lines)

To see how this works, enter

```
NEW <ENTER>  
NUMBER 10, 10 <ENTER>  
10 (Appears on screen)
```

When the 10 appears enter

```
REM <ENTER>  
20 (Appears on screen)
```

You can now program without having to worry about line numbers. Every time you enter program statements and press ENTER, the next line number will pop up. When you are finished, just press ENTER when the next line number appears and you will jump back into the Immediate Mode.

Saving Your Program

Suppose you write a program, get it working perfectly and then turn off your computer. Since the program is stored in the RAM memory, it will go to Never-Never Land, and you will have to write it in again if you want to use it. Fortunately, it is a simple matter to SAVE a program to your diskette. Let's use our program for an example of SAVEing a program to disk. Make sure your program is still in memory by LISTing it, and if it is not, re-write it. Make sure a initialized disk is in the drive and write in the following:

```
SAVE DSK1.MYPROGRAM
```

(If you are not certain about disk initialization, review the section covering those items in Chapter 1.)

The disk will start whirling and the red light will glow on the disk drive. This means the disk drive is writing your program to disk. When the red light goes out, your program should be SAVED on disk.

Saving Programs on Tape

To save a program to tape, put a blank cassette into your tape recorder and rewind it. Press the RECORD button and the PLAY button together on your tape recorder and write in SAVE CS1. The tape recorder will start spinning and will be prompted through the SAVE sequence as described in Chapter 1. As you SAVE more and more programs, they will become difficult to find unless you keep some record of what is on the tape. The best way to do this is to keep a log of the starting position and ending position of the TAPE COUNTER. Enter a descriptive name of the program you have SAVED corresponding to the tape counter values. Also, as you write more and more programs, you will want to label your cassettes as well. The following shows you an example of a tape log:

CASSETTE NUMBER BEGINNING END DESCRIPTION SIDE

1	a	0	8	Variables
1	a	8	10	String program
1	a	10	15	Subroutines
1	b	0	20	Checkbook
2	a	0	5	Input
2	a	5	30	Output

Retrieving Your Programs

The best way to make sure you have SAVED a program to disk or to tape is to completely turn off your TI-99/4A, and then turn it on again. Go ahead and do it. Now you know there is nothing in memory. Enter



Saving your Program

OLD DSK1.MYPROGRAM

and your disk drive will whirl for a while and stop. Now enter LIST <ENTER> and if all went well your program will be LISTed to the screen. If you key in the name wrong or there is some other error, the screen will show something like the following:

* WARNING:
CHECK PROGRAM IN MEMORY

*I/O ERROR 50

To see if your program is in the disk CATALOG, QUIT (FCTN=), go to the DISK MANAGER and choose DISK COMMANDS and CATALOG DISK to SCREEN. Your program should be listed under FILENAMEs along with a SIZE and TYPE designation. It will say MYPROGRAM under FILENAME. If it is there, you know for certain everything has worked.

If you have a tape cassette, key in OLD CS1 <ENTER> and follow the prompts through the loading process. Since there is no file name designation for programs stored on tape, you have to use the FFWD (Fast Forward) key on your recorder to move the tape up to the location where the desired program begins. This is where your tape log becomes crucial!

A SAFETY NET

As you begin writing longer programs, every so many lines you should SAVE your program to disk or tape. In this way, if your dog accidentally trips over your cord and turns off your computer, you won't lose your program and have to shoot the offending pooch. Saves both programs and dogs.

Now that you have SAVED and loaded programs, let's look at another neat trick. Remembering you SAVED your file under the name MYPROGRAM, let's change the contents of that file. First, add the following line and then LIST your program:

```
27 PRINT "<YOUR CITY, STATE & ZIP>"
```


Your program is now different from the program you **SAVED** in the file **MY PROGRAM** since you have added line 27. Now write in

SAVE DISK1.MYPROGRAM

Clear memory with **NEW**, **OLD** the file **MYPROGRAM** and **LIST** it. As you can see, line 27 is now part of **MYPROGRAM**. All you have to do to update a program is to **OLD** it, make any changes you want, and then **SAVE** it under the same file name; however *BE CAREFUL*. No matter what program is in memory, that program will be **SAVED** when you enter the **SAVE** command; therefore, if your disk has **PROGRAM A** and you write **PROGRAM B**, and then **SAVE** it under the title **PROGRAM A**, it will destroy **PROGRAM A** and the **SAVED** program will actually be **PROGRAM B**. Also, if you have a really important program, it is a good idea to make a back-up file. For example, if you saved your current program under the file names, **MYPROGRAM** and **MYPROGRAM2** it would have two files with exactly the same program. To really play it safe, save the program on two different diskettes.

I TOLD YOU SO DEPT.

Sooner or later the following will happen to you: You will have several disks or tapes, one of which you want to initialize and one on which to save programs. You will pick up the wrong diskette or cassette, one with valuable programs on it. There will be no write protect tab on the diskette or cassette, and after you initialize it or overwrite programs on it and blow away everything you wanted to keep, you will realize your mistake and say, “!&\$#”!%&”, and kick your dog. You cannot prevent that from happening at least once, believe me. Therefore, to insure that such a mistake is not irreversible, do the following: **MAKE BACK-UPS**. Take your **ORIGINAL** and put it somewhere out of reach, and when you accidentally erase a disk or tape, you can make another copy. Remember, if you fail to follow this advice, your dog will have sore ribs. Be kind to your dog.

*It's always good to have a **COPY** around in case something happens to the **Original**...*



Using Your Editor: Fixing Mistakes on the Run

Error Messages and Repairing Them

By now you probably entered something and got a * INCORRECT STATEMENT, * INCORRECT STATEMENT IN 30 (referring to line 30 or any other line where an error is detected) or some other kind of error message, such as REDO FROM START, which told you something was amiss. This occurs in the Immediate mode as soon as you hit ENTER and in the Program mode as soon as you RUN your program. Depending on the error, you will get a different type of message. As we go along, we will see different messages depending on the operation. For now, we will concentrate on how to fix errors in program lines rather than the nature of the errors themselves. This process is referred to as editing programs. (See III-8 to III-12 of your *User's Reference Guide* for a complete list of error messages.)

Deleting Lines

The simplest type of editing involves inserting and deleting lines. Let's write a program with an error in it and fix it up.

```
NEW <ENTER>
10 CALL CLEAR
20 PRINT "AS LONG AS SOMETHING CAN"
30 PRINT "GO WRONG"
40 PRINT "IT WILL"
50 REM LINE 40 HAS AN ERROR
60 END
RUN <ENTER>
```

If the program is written exactly as depicted above, the program will BOMB. On your screen you will see the message

```
*INCORRECT STATEMENT (or BAD NAME)
  IN 40
```

There's an
Editor
in your
Computer!



Now key in

```
40 <ENTER>  
LIST <ENTER>
```

What happened to line 40?! You just learned about deleting a line. Whenever you enter a line number and nothing else, you delete the line. We already learned how to insert a line, so all you have to do to fix the program is enter the following:

```
40 PRINT "IT WILL"
```

Now run the program. It should work fine. The error was in the misspelling of PRINT. Another way you could have fixed the program was simply to re-enter line 40 correctly without first deleting it, but I wanted to show you how to delete a line by entering the line number. When you make most other kinds of errors, your TI-99/4A will let you know immediately. If this occurs, the line you attempted to enter will be deleted automatically.

Using the TI-99/4A Editor

Within your TI-99/4A is a trusty editor. To see how to work with your editor, we'll write another bad program and fix it. OK, write the following program and RUN it.

```
NEW <ENTER>  
10 CALL CLEAR  
20 PRINT "IF I CAN GOOF UP A PROGRAM "  
30 PRINT "I CAN FOX IT"  
40 REM LINE 30 ISN'T QUITE RIGHT  
50 END  
RUN <ENTER>
```

All right, you want to FIX your FOX in line 30. To repair it, instead of rewriting line 30 do the following:

- STEP 1. LIST your program
- STEP 2. Key in EDIT 30 <ENTER>
- STEP 3. Now using the right ARROW key (FCTN-D) "walk" the cursor to the right until it is over the "O" in "FOX".
- STEP 4. Key in an "I" and press {ENTER}.
- STEP 5. LIST your program to make sure the correction has been made.

RUN the program, and you should see the statement, IF I CAN GOOF UP A PROGRAM I CAN FIX IT. Let's learn more about the editor. Put in the following program:

```
10 CALL CLEAR
20 PRINT "SOMETIMES I LIKE TO WRITE LONG
LONG LINES"
30 WHEW
40 PRINT "AND SHORT ONES TOO"
50 END
LIST <ENTER>
RUN <ENTER>
```

OK, after you ran the program it went "El Bombo." The problem was that we stuck in that WHEW in line 30 without a REM statement. To repair it, LIST the program, and EDIT 30 <ENTER>. The cursor will be right over the "W" in WHEW. Press the FCTN and 2 keys *simultaneously*. *NOTE: On the strip above the keyboard there is the label "INS" right above the "2" key. That stands for INSERT.* Key in REM (SPACE). You have just used the insert function of your editor! See how easy that was. Press ENTER. Now RUN the program. Everything

Now let's take a look at a feature of the TI-99/4A editor that will help you fix programs. Key in NEW <ENTER> and we'll start a new program.

```
10 CALL CLEAR
20 PRINT "I LIKE TO COMPUUUUUUTE"
```

Whoops! There's a mistake, but no sweat. Just EDIT 20 <ENTER>, and using the RIGHT ARROW (FCTN-D) "walk" the cursor over to the first of the multiple "U's." Next, press the FCTN and 1 keys *simultaneously* until there is only a single "U" in COMPUTE. *Note: The DEL label above the 1 key stands for DELETE, but I bet you figured that out for yourself.*

More Editing

Let's do a few more things with your editor before going on. We'll practice some more with inserting characters and numbers, but we will also see how to edit groups of characters. So, let's see how we can use the editor to do more with "insertions." Try the following little program:

```
10 CALL CLEAR
20 PRINT "NOW IS THE TIME FOR ALL GOOD"
30 PRINT "MEN TO COME TO"
40 PRINT "THE AID OF THEIR COUNTRY!"
50 PRINT "AND HAVE A GOOD TIME"
```

So far so good, but you meant to include women as well as men in line 30. You could retype the entire line, but all you really need to add is AND WOMEN after MEN. Also, it's really boring to have everything in upper case. Let's change the line to include women and make it both upper and lower case. Finally, we want line 50 to be END instead of that other stuff.

- STEP 1. Press the ALPHA LOCK key so that it is in the "up" position.
- STEP 2. EDIT 20 <ENTER>. Walk the cursor to the O in NOW and key in everything in lower case over the original text.
- STEP 3. EDIT 30 and insert 'and women'. Make sure that 'and women' is inside the original pair of quotation marks. Press ENTER when the changes in line 30 are complete.

STEP 4. EDIT 40 <ENTER>. Press the FCTN and 3 keys *simultaneously*. Everything but the line number disappeared! *NOTE: That's what the ERASE label above the 3 key means! Now key in END.*

After these repairs, you now have upper and lower case in line 20, and when you RUN your program it should read:

Now is the time for all good men and women
to come to the aid of their country.
DONE

You will save yourself a great deal of time if you use the editor rather than retyping every mistake you make. Therefore, to practice with it, there are a several pairs of lines below to repair. The first line shows the wrong way and the second line in the pair shows the correct way. Since "little" things can make a big difference, there are a number of changes to be made. However, as you will soon see, those little mistakes are the ones we are most likely to get snagged on. Practice on these examples until you feel comfortable with the editor - time spent now will save you efforts later.

Editor Practice

```
50 PRINT "I LICK MY TI"  
50 PRINT "I LIKE MY TI"
```

```
10 PRINT CLEAR  
10 CALL CLEAR
```

```
80 PRINT "A GOOD MAN IS HARD TO FIND"  
80 PRINT "A GOOD PERSON IS HARD TO FIND"
```

```
40 PRINT CALL CLEAR  
40 CALL CLEAR
```

```
50 "WE'RE OFF!"  
50 PRINT "WE'RE OFF!"
```

If you fixed all of those lines, you can repair just about anything. Once you get the hang of it, it's quite simple.

ELEMENTARY MATH OPERATIONS

So far all we've done is to PRINT out a lot of text, but that isn't too different from having a fancy typewriter. Now, let's do some simple math operations to show you your computer can compute! Enter the following:

```
CALL CLEAR  
PRINT 2 + 2
```

This is what your screen should look like now:

```
PRINT 2 + 2  
4
```

Big deal, so the computer can add - so can my \$5 calculator and my 8 year old kid. Who said computers are smart? The programmer (you) is who is smart. OK, so let's give it a little tougher problem.

```
CALL CLEAR  
PRINT 7.87 * 123.65
```

Still nothing your calculator can't do, but it'd be a little rough on the 8 year old.

As we progress, we can include more and more aspects of mathematical problems. In the next chapter, we will see how we can store values in variables and a lot of things that would choke your calculator. For now, though, all we'll do is to introduce the format of mathematical manipulations. The "+" and "-" signs work just as they do in regular math, and the "x" is replaced by the "*" (asterisk) for multiplication and the "+" is replaced by the "/" (slash) for division.

As we begin dealing with more and more complex math, we will need to observe a certain order in which problems are executed. This is called precedence. Depending on the operations we use, and the results we are attempting to obtain, we will use one order or another. For example, let's suppose we want to multiply the sum of two numbers by a third number - say the sum of 15 and 20 multiplied by 3. If you entered

```
PRINT 3 * 15 + 20
```

you would get 3 multiplied by 15 with 20 added on ($3 \times 15 = 45$ and $45 + 20 = 65$). That's not what you wanted. You wanted to get 3 times 15 plus 20 ($3 \times 35 = 105$). The reason for that is precedence - multiplication precedes addition. To help you remember the precedence, let's write a little program you can run and then play with some math problems in the Immediate mode to see the results and refer to your "Precedence Chart" on the screen. (This little program is quite handy; so save it to disk or tape to be used later.)

```
10 CALL CLEAR
20 PRINT "1. - (MINUS SIGNS FOR NEGATIVE
NUMBERS"
30 PRINT " - NOT SUBTRACTION)"
40 PRINT "2.    (EXPONENTIATION)"
50 PRINT "3. * / (MULTIPLICATION AND DIVISION)"
60 PRINT "4. + - (ADDITIONS AND SUBTRACTIONS)"
70 PRINT "NOTE: ALL OTHER PRECEDENCE"
74 PRINT "BEING EQUAL, PRECEDENCE"
78 PRINT "IS FROM LEFT TO RIGHT"
80 PRINT "YOUR COMPUTER FIRST EXECUTES
THE NUMBERS IN PARENTHESES,"
90 PRINT "WORKING ITS WAY FROM THE INSIDE
OUT IN MULTIPLE PARENTHESES."
```

Try some different problems and see if you can get what you want.

Re-ordering Precedence

Once you get the knack of the order in which math operations work, there is a way to simplify the organization of math problems. By placing two or more numbers in PARENTHESES, it is possible to move them up in priority. Let's go back to our example of adding 15 and 20 and then multiplying by 3, but this time we will use parentheses.

PRINT 3 * (15 + 20)

Now since the multiplication sign has precedence over the addition sign, without the parentheses, we would have gotten 3 times 15 plus 20. However, since all operations inside parentheses are executed first, your computer FIRST added 15 and 20 and then multiplied the sum by 3. If more than a single set of parentheses is used in an equation, then the innermost is executed first, working its way out.

THE PARENTHESES DUNGEON

To help you remember the order in which math operations are executed within parentheses, think of the operations as being locked up in a multi-layer dungeon. Each cell represents the innermost operation, and the cells are lined up from left to right. Each "prisoner" is an operation surrounded by walls of parentheses. To escape the dungeon, the prisoner must first get out of the innermost cell, then go to his right and release any other prisoners in their cells. Then they break out of the "cell-block" and finally out into the open. Unfortunately, since operations are "executed," this is a lethal analogy for our poor escaping "prisoners." Do some of the examples and see if you can come up with a better analogy.

The following examples show you some operations with parentheses.

```
PRINT 20 + 10 * (8 - 4)
PRINT (12.43 + 92) / 3    (11 - 3)
PRINT (22 - 3.1415) * (22 + 3.1415)
PRINT ((16 + 4) - (3 + 5)) / 18
PRINT 19 + 2 * (51 / 3) - (100 / 14)
```

Now try some of these problems in the proper format expected by your computer:

Multiply the sum of 4, 9 and 20 by 15.
Multiply 35 by 35 and the result by pi (3.14159265).
(You realize that this will compute the area of a circle with a radius of 35; to find the area of any other circle, just change 35 to another value.) Pretty neat, huh?
Add up the charges on your long distance calls and divide the sum by the number of calls you made. This will give you the average expense of your calls. Remember, though, you have to do this in one set of statements in a single line. Do the same thing with your checkbook for a month to see the average (mean) amount for your check.
Add up the total amount you spent on your computer and peripherals and subtract from that sum the amount you would have spent at video arcades. (If your results are negative, you can claim that amount saved by buying a computer!)

SUMMARY

This chapter has covered the most basic aspects of programming. At this point you should be able to use the editor in your TI-99/4A and write commands in the Immediate and Program (deferred) modes. Also, you should be able to manipulate basic math operations. However, we have only just begun to uncover the power of your computer, and at this stage, we are treating it more as a glorified calculator than a computer. Nevertheless, what we have covered in this chapter is ex-

tremely important to understand, for it is the foundation upon which your understanding of programming is to be built. If there are parts you do not understand, review them before continuing. If you still do not understand certain operations after a review, don't worry. You will be able to pick them up later, but it is still important that you try to get everything to do what it is supposed to do and what you want it to do.

The next chapter will take us into the realm of computer programming and increase your understanding of your TI-99/4A considerably. If you take it one step at a time, you will be amazed at the power you have at your fingertips and how easy it is to program. Also, we will be leaving the realm of calculator-like commands and getting down to some honest-to-goodness computer work. This is where the fun really begins.

CHAPTER 3

Moving Along

Introduction

In the last chapter, we saw how to get started in executing commands in both the Immediate and Program modes. From now on we will concentrate our efforts on building from the foundation set in Chapter 2 in the Program mode, tying various commands together in a program. We will, however, use the Immediate mode to provide simple examples and to give you an idea of how a certain command works. As we learn more and more commands, it would be a good idea if you started saving the example programs on your disk or cassette so that they can be used for review and a quick “look-up” of examples. Use file names that you can recognize, such as VARIABLE EXAMPLE or HOW TO SUBROUTINES, and *remember* each file has to have a different name; so be sure to number example file names (e.g., ARRAYS 1, ARRAYS 2, etc.). In your cassette log, you can have more descriptive names and even comments about the programs.

VARIABLES

Perhaps the single most important computer function is in variable commands. Basically, a variable is a symbol that can have more than a single value. If we say, for example, $X = 10$, we assign the value of 10 to the variable we call “X”. Try the following:

```
X = 10 <ENTER>
```

```
PRINT X <ENTER>
```

Your computer responded

```
10
```

Now type in

```
X=55.7 <ENTER>
PRINT X <ENTER>
```

This time you got

55.7

Each time you assign a value to a variable, it will respond with the last assigned value when you PRINT that variable. Now try the following:

```
X = 10 <ENTER>
Y = 15 <ENTER>
PRINT X + Y <ENTER>
```

And your TI-99/4A responded with

25

As you can see, variables with numbers can be treated in the same way as math problems. However, instead of the numbers, you use the variables. Now let's try a little program using variables to calculate the area of a circle.

```
10 CALL CLEAR
20 PI = 3.14159265
30 REM THE VALUE OF PI RECALLED
  FROM GEOMETRY
40 R = 15
50 REM 'R' IS THE RADIUS OF OUR CIRCLE
60 PRINT PI * (R * R)
70 REM THIS GIVES US PI TIMES THE SQUARE OF
  THE RADIUS
80 END
```

When you RUN the program, you will get the area of a circle with a radius of 15. If you change the value of R in line 30, it is a simple matter to quickly calculate the area of any circle you want! Since our example "squares" a number, why don't we use our exponential sign " $\wedge$ ". Change line 60 to read:

```
60 PRINT PI * (R ^ 2)
```

RUN the program again and see if you get the same results. You should. Also, change the value of R to see the areas of different circles.

Variable Names

At this point you might wonder why not use variables. First, in programs where a value will change, it is very difficult to keep entering new numbers. Secondly, as we saw above, we can use descriptive names for variables so that we know what to expect. (PI in our program.) For example, the following program uses MEAN as a descriptive variable name:

```
10 CALL CLEAR
20 A = 15
30 B = 23
40 C = 38
50 MEAN = (A + B + C) / 3
60 PRINT MEAN
70 END
```

If the above program were a hundred or more lines long, you would know what the variable MEAN does - it calculates a "mean."

Other considerations in naming variables include reserved words. These are words set aside for programming commands, functions and statements. Let's look at some examples of what is and what is not a valid variable name:

```
PRINT = 987 (Invalid name since PRINT is a
reserved word.)
R1 = 321 (Valid name since first character is a
letter.)
1R = 55 (Invalid since first character is not a letter.)
PR = 99 (Valid name, even though reserved word
PRINT begins with PR, because only part of the re-
served word is used in variable name.)
TO = 983 (Invalid name since TO is a reserved two-
character word.)
ADFETDCVRRWRDAAF = 10 (Valid name, but
really dumb.)
```

It is also possible to give values to variables with other variables or a combination of variables and numbers. In our example with the variable MEAN we defined it with other variables. Here are some more examples:

$$T = A * (B + C)$$
$$N = N + 1$$
$$SUM = X + Y + Z$$

Types of Variables

Real Variables

So far we've used only "real" or "floating point" variables in our examples. Any variable which begins with a capital letter and does not end with a dollar sign (\$) is a real variable. The value for a real variable can be from + or -9.999999999999999E127. The "E" is the scientific notation for very big numbers. For the time being, don't worry about it, but if you get a result with such a letter in a numeric result, get in touch with a math instructor. At this juncture, figure you can enter numbers in their standard format from 0.01 to 999,999,999. (If your check-book debit or income tax payments have a scientific notation in them, leave the country.) Think of real variables as being able to hold just about any number you would need along with the decimal fractions.

String Variables

String variables are extremely useful in formatting what you will see on the screen, and like real variables, they are sent to the screen by the PRINT statement. However, rather than printing only numbers, string variables send all kinds of characters, called "strings", to the screen. String variables are indicated by a dollar sign (\$) on the end of a variable. For example, A\$, BAD\$, G\$, and PULL\$ are all legitimate string variables. (In computer parlance, we use the term "string" for the dollar sign. Thus, our examples would be called "A string", "BAD string", etc.) String variables are defined by placing the "string" in quotation marks, just as we did with other messages we PRINTed out.

Let's try out a few examples from the Immediate mode:

```
ABC$ = "ABC"  
PRINT ABC$ <ENTER>
```

```
G$ = "BURLESQUE"  
PRINT G$ <ENTER>
```

```
KAT$ = "CAT"  
PRINT KAT$ <ENTER>
```

```
NUMBER$ = "123456789"  
PRINT NUMBER$ <ENTER>
```

```
B1$ = "5 + 10 + 20"  
PRINT B1$ <ENTER>
```

```
PRINT$ = "PRINT"  
PRINT PRINT$ <ENTER>
```

Like real variables, string variables must begin with a letter; however, string variables can use reserved words. More importantly, you probably noticed in our examples that numbers in string variables are not treated as numbers, but rather as "words" or "messages." For example, you may have noticed that when you PRINTed B1\$, instead of printing out 35 (the sum of 5, 10 and 20), B1\$ printed out exactly what you put in quotes, 5 + 10 + 20. Do not attempt to do math with string variables. (In later chapters, we'll see some tricks to convert string variables to numeric ones, but for now just treat them as messages.)

Now let's put all of our accumulated knowledge together and write a program that uses variables. We will start a little program which will allow you to subtract a check from your checkbook and print the amount. This program will be the beginning of something we will later develop to give you a handy little program to do checkbook balancing.

```

10 CALL CLEAR
20 BALANCE = 571.88
30 REM ANY FIGURE WILL DO.
40 REM BALANCE IS A REAL VARIABLE
50 CHECK = 29.95
60 REM WHAT YOU LAST SPENT IN THE
  COMPUTER STORE.
70 REM CHECK IS A REAL VARIABLE.
80 B$ = "BEGINNING BALANCE=$"
90 C$ = "YOUR CHECK IS $"
100 NB$ = "NEW BALANCE IS $"
110 REM B$, C$ AND NB$ ARE STRING VARIABLES
120 PRINT B$;BALANCE
130 PRINT C$;CHECK
140 N = BALANCE - CHECK
150 PRINT NB$; N
160 END

```

Since this is a fairly long program for this stage of the game, make sure you put in everything correctly. For the computer, it is critical that you distinguish between commas, semi-colons, periods, etc. Also, save it to disk. To play with it, change the values in lines 20 and 30.

Let's quickly review what we have done.

- STEP 1. First we defined the real variables BALANCE and CHECK.
- STEP 2. Then we defined string variables B\$, C\$ and NB\$ to use as labels in screen formatting.
- STEP 3. Finally, we printed out all of our information using our variables, with one new variable, N, defined as the difference between BALANCE and CHECK.

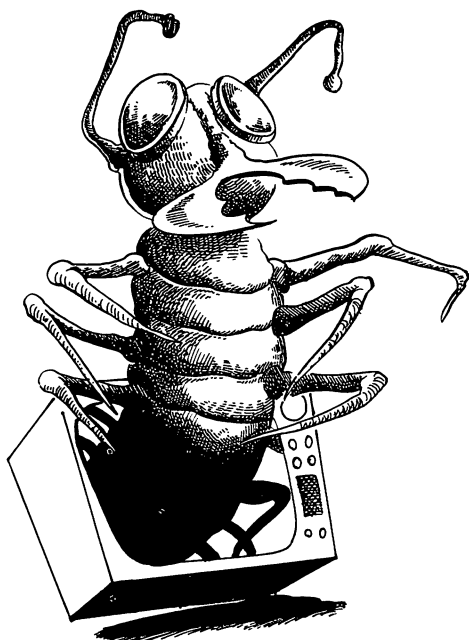
Note how we formatted the OUTPUT (what you see on your screen) of our PRINT statements. The semi-colon ";" between the variables accomplished two things: (1) it told the computer where one variable ended and the next began, and (2) it told the computer to PRINT the second variable right after the first one. Thus, it took the string variable NB\$

NEW BALANCE IS \$#

and stuck the value of the real variable N right after the dollar sign (exactly where we placed the hatch #). Later we will go more into the formatting of OUTPUT, but for now let's take a quick look at using punctuation in formatting text. We will use the comma “,” and semi-colon “;” and “new line” to illustrate basic formatting. Put in the following little program:

```
NEW <ENTER>
10 CALL CLEAR
20 A$ = "HERE"
30 B$ = "THERE"
40 C$ = "WHERE"
50 PRINT A$;
60 PRINT B$;
70 PRINT C$;
80 REM SEMI COLONS
90 PRINT
100 PRINT A$,
110 PRINT B$,
120 REM COMMAS
130 PRINT
140 REM A 'PRINT' BY ITSELF GIVES A
150 REM VERTICAL 'SPACE' IN FORMATTING
160 PRINT A$
170 PRINT B$
180 PRINT C$
190 REM 'NEW LINES'
200 END
```

Now RUN the program. As you should see, the little differences in lines 30, 40 and 50 made big differences on the screen. The first set is all crammed together, the second set is spaced evenly across the screen in two columns and the third set is stacked one on top of the other. As we saw in the previous program, semi-colons put numbers and strings right next to one another. However, using commas after a PRINTed variable will space output in groups of two across the screen, and using new lines in the form of colons or new line numbers will make the output start on a new line. A PRINT statement all by itself will put a vertical linefeed between statements. Try the following little program to see how PRINT statements all by themselves can be used.



your program may have
BUGS

```
NEW <ENTER>
10 CALL CLEAR
20 PRINT "WHENEVER YOU PUT IN A
PRINT STATEMENT";
30 REM NOTE PLACEMENT OF SEMI-COLON
40 PRINT " ALL BY ITSELF,"
50 PRINT "IT GIVES A 'LINEFEED'."
60 PRINT
70 PRINT "SEE WHAT I MEAN?"
80 END
```

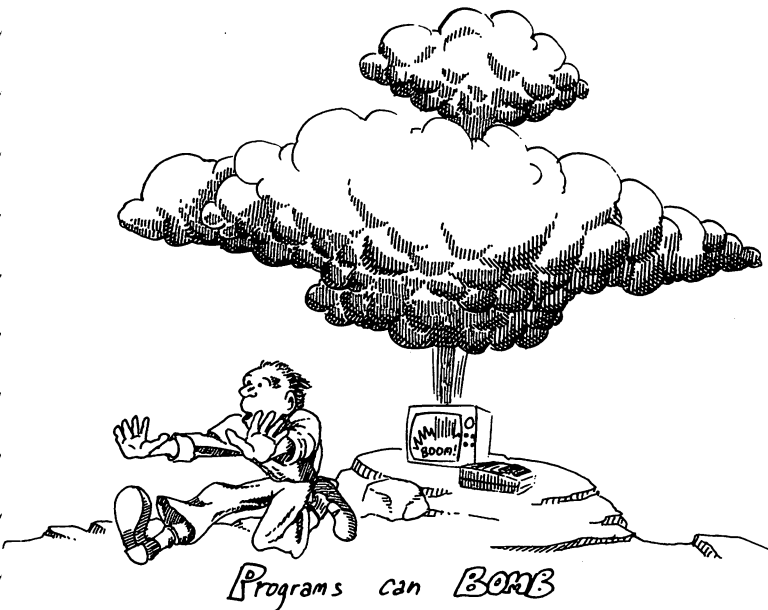
Play with commas, semi-colons and new lines with variables and string variables until you get the hang of it. They are very important and are the source of program "bugs." If your line is too long after a semi-colon, instead of having the next line of printed text where you expect it, it will "linefeed." To see this effect, combine lines 40 and 50 into a single line with one PRINT statement.

BUGS and BOMBS

We've mentioned "bugs" and "bombs" in programs but never really explained what they meant. "Bugs" are simply errors in programs that either create SYNTAX ERRORS or prevent your program from doing what you want it to do. "Debugging" is the process of removing "bugs." "Bombing" is what your program does when it encounters a "bug." This is all computer lingo; if you use it in your conversations, people will think you really know a lot about computers or have a bug in your personality.

INPUT and OUTPUT (I/O)

Input and output, often referred to as I/O, are ways of putting things into your computer and getting them out. Usually we put IN information from the keyboard, save it to disk or tape, and then later put it in from the disk drive or cassette recorder. When we want information OUT of the computer, we want it to go to our screen or printer. This is what I/O means. So far, we



have entered information IN the computer from the keyboard either in the Program or in the Immediate mode. Using the PRINT statement, we have sent information OUT to the screen. However, there are other ways we can INPUT information with a combination of programming and keyboard commands. Let's look at some of these ways and make our CHECKBOOK program a lot simpler to use.

INPUT

The INPUT command is placed in a program and expects some kind of response from the keyboard and then an ENTER. (An ENTER alone will also work, but the response is read as ""). It must be part of a program and cannot be used from the Immediate mode. (If attempted from the Immediate mode, there will be a *CAN'T DO THAT message.) Let's look at a simple example:

```
NEW <ENTER>
10 CALL CLEAR
20 INPUT X
30 REM 'X' IS A NUMERIC VARIABLE SO ENTER
  A NUMBER
40 PRINT X
50 END
```

RUN the program and your screen will go blank and a ? along with a blinking cursor will sit there until you enter a number and then the computer will PRINT the number you just entered. Really interesting, huh?

Let's try INPUTting the same information but using a slightly different format. The nice thing about INPUT statements is that they have some of the same features as PRINT statements for getting messages on the screen. Look at the following program:

```
NEW <ENTER>
10 CALL CLEAR
20 PRINT "WHAT IS YOUR AGE";
30 INPUT X
40 CALL CLEAR
50 PRINT "YOUR AGE IS "; X
```

Now RUN the program; you will see that the presentation is a little more interesting. Also notice we did not put an END command at the end of the program. In TI-99/4A it is not necessary to enter an END command, but it is usually a good idea to do so. As we get into more advanced topics, we will see that our program can jump around, and the place we want it to END will be in the middle. We will need an END statement so that it will not crash into an area we don't want it to go. So, while an END command really has not been necessary up to now, it is nevertheless a good habit to develop.

Let's soup up our program a little more with the INPUT statement. *NOTE: To make things simple, enter NUMBER 10, 10 <ENTER> before you begin this program where indicated after NEW <ENTER>. When you have finished entering your program, just hit <ENTER> when the next line number appears.*

```
NEW <ENTER>
NUMBER 10,10 <ENTER>
10 CALL CLEAR
20 PRINT "ENTER YOUR NAME ->";
30 INPUT NAMES$
40 PRINT "ENTER YOUR AGE ->";
50 INPUT AGE
60 PRINT "<ENTER> TO CONTINUE";
70 INPUT ENTER$
80 CALL CLEAR
90 PRINT NAMES$; " IS"; AGE ; "YEARS OLD."
100 REM BE CAREFUL WHERE YOU PUT YOUR
QUOTE MARKS AND SEMI-COLONS IN THIS LINE
110 END
```

Now we're getting somewhere. You can enter information as numeric or string variables and the OUTPUT is formatted so you know what's going on. As your programs become larger and more complicated, it is very important to connect your string variables and numeric variables in such a way that it is easy to see what the numbers on the screen mean. Let's face it, a computer wouldn't be very helpful if it filled the screen with numbers and you did not know what they meant! Lines 60 and 70 contain the format for a pause in your program. ENTER\$ doesn't hold any information, but since INPUT statements expect something from the keyboard and a variable, ENTER\$ (for ENTER) is as good as any.

READING In DATA

A second way to enter data into a program is with READ and DATA statements. However, instead of entering the data through the keyboard, DATA in one part of the program is READ in from another part. Each READ statement looks at elements in DATA statements sequentially. The READ command is associated with a variable which looks at the next DATA statement and places the numeric value or string in the variable. Let's look at the following example: *NOTE: I'm not going to remind you to use NUMBER 10,10 <ENTER> this time!*

```
NEW <ENTER>
10 CALL CLEAR
20 READ NAMES$
30 REM READS NAME
40 READ JOBS$
50 REM READS OCCUPATION
60 READ ADDRESS
70 REM READS STREET NUMBER
80 READ STREET$
90 REM READS STREET NAME
100 READ CITY$
110 REM READS CITY
120 READ STATES$
130 REM READS STATE
140 READ ZIP
150 REM READS ZIP CODE
160 PRINT
170 PRINT
180 PRINT
190 REM BEGIN PRINTING OUT WHAT 'READ'
    READ IN.
200 REM (BE CAREFUL TO PUT IN EVERYTHING
210 REM EXACTLY AS IT IS LISTED.)
220 PRINT NAMES$
230 PRINT JOBS$
240 PRINT ADDRESS; STREET$
250 PRINT CITY$; ","; STATES$ ; ZIP
260 END
```


1000 DATA DAVID GORDON, PUBLISHING TYCOON,
8943, FULLBRIGHT AVE
1010 DATA CHATSWORTH, CALIFORNIA, 91311

In the DATA statements there is a comma separating the various elements, unless the DATA statement is at the end of a line. If you have one of the elements out of place or omit a comma, strange things can happen. For example if the READ statement is expecting a numeric variable (such as the street address) and runs into a string (such as the street name) you will get an error message. Think of the DATA statements as a stack of strings and numbers. Each time a READ statement is encountered in the program the first element of the DATA is removed from the stack. The next READ statement looks at the element on top of the stack, moving from left to right. Go ahead and SAVE this program and let's put an error in it. (SAVE it first, though, so you will have a correct listing of how READ and DATA statements work.)



LIST the program to make sure you have it in memory and enter the following line:

```
145 READ EX$
```

Now RUN the program and you should get a * DATA ERROR IN 145. The error occurred because you have a READ statement without enough DATA statements (or elements); so, be sure that 1) there are enough elements in your DATA statements to take care of your READ statements, and 2) the variables in your READ statements are compatible with the elements of the DATA statements. (i.e., Your numeric variables read numbers and string variables read strings.) To repair your program, simply type in

```
1020 DATA WORD
```

This will give it something to READ. (Of course you could have DELETED line 145).

If an element is a DATA statement (and is enclosed in quotation marks), all the characters inside the quotes are considered to be a single string element. For example, make the following changes in your program and RUN it.

```
255 PRINT EX$  
1020 DATA "10 DOWNING ST,  
LONDON, 45, ENGLAND"
```

Both numbers and commas were happily accepted by a READ statement with a string variable since they were all enclosed in quotation marks. Now remove the quote marks and RUN it again. This time it printed only up to the first comma, '10 DOWNING ST' but the string variable EX\$ had no problem accepting a numeric character! (However, since it read the '10' as a string, it cannot be used in a mathematical operation.) Experiment with different elements in the DATA statements to see what happens. Also, just for fun, put the DATA statements at different places in the program. You will quickly find that they can go anywhere and are READ in the order of placement in the program.

```

130 REM VARIABLE FOR CHECK
140 BALANCE = BALANCE - CHECK
150 REM KEEPS A RUNNING BALANCE
160 NEXT I
170 REM TOP OF LOOP
180 CALL CLEAR
190 REM CLEAR SCREEN WHEN ALL CHECKS
ARE ENTERED
200 PRINT "YOU NOW HAVE $"; BALANCE ; "IN
YOUR ACCOUNT"
210 PRINT
220 PRINT "THANK YOU AND COME AGAIN"
230 END

```

Our checkbook program is coming along, making it easier to use, and that is the purpose of computers. Notice what we did with formatting in line 30. To get the space between CHECKBOOK and the rest of the program we put in four commas. This worked like entering an extra line and a PRINT statement. It saved some programming and did what we wanted. Now let's look at something else with loops.

NESTED LOOPS

With certain applications, it is going to be necessary to have one or more FOR/NEXT loops working inside one another. Let's look at a simple application. Suppose you had two teams with 10 members on each team. You want to make a team roster indicating the team number (#1 or #2) and member number (#1 through #10). Using a nested loop, we can do this in the following program:

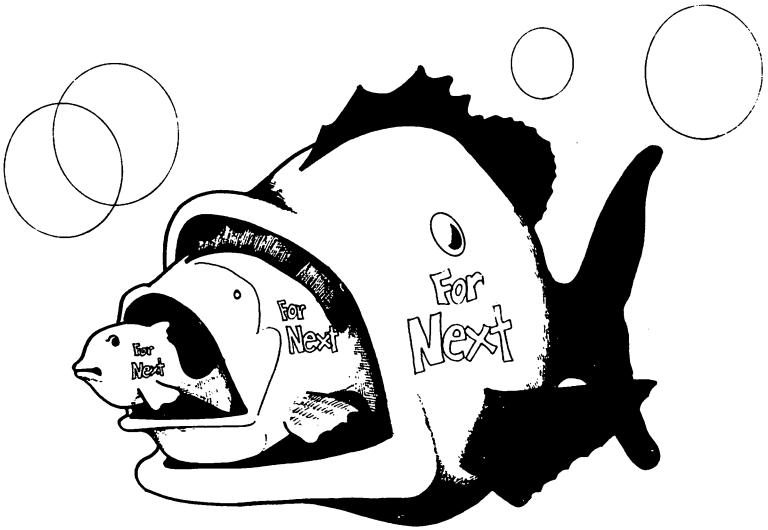
```

NEW <ENTER>
10 CALL CLEAR
20 FOR TEAM = 1 TO 2
30 REM TEAM FOR TEAM #
40 FOR PLAYER = 1 TO 10
50 REM PLAYER FOR MEMBER #
60 PRINT "TEAM #" ; TEAM ; "PLAYER #" ; PLAYER

```

70 NEXT PLAYER
80 PRINT
90 NEXT TEAM
100 END

In using nested loops, it is important to keep the loops straight. The innermost loop (the **PLAYER** loop in our example) must not have any other **FOR** or **NEXT** statement inside of it. Think of nested loops as a series of fish eating one another, the largest fish's mouth encompassing the next largest and so forth on down to the smallest fish.



Look at the following structure of nested loops:

```
FOR A = 1 TO N
  FOR B = 1 TO N
    FOR C = 1 TO N
      FOR D = 1 TO N
        NEXT D
      NEXT C
    NEXT B
  NEXT A
```

Looping With FOR/NEXT

The FOR/NEXT loop is one of the most useful operations in BASIC programming. It allows the user to instruct the computer to go through a determined number of steps, at variable increments if desired, and execute them until the total number of steps is completed. Let's look at a simple example to get started.

```
NEW <ENTER>
10 CALL CLEAR
20 NAMES$ = "<YOUR NAME>"
30 FOR I = 1 TO 10
40 REM BEGINNING OF LOOP
50 PRINT NAMES$
60 NEXT I
70 REM LOOP TERMINAL
80 END
```

Now RUN the program and you will see your name printed 10 times along the left side of the screen. That's nice, but so what? OK, not too impressive, but we will see how useful this can be in a bit. First let's look at another simple illustration to show what's happening to "I" as the loop is being executed.

```
NEW <ENTER>
10 CALL CLEAR
20 FOR I = 1 TO 10
30 PRINT I
40 NEXT I
```

As we can see when the program is RUN, the value of "I" changes each time the program proceeds through the loop. Think of a loop as a child on a merry-go-round. Each time the merry-go-round completes a revolution, the child gets a gold ring, beginning with one and ending, in our example, with 10.

TRIVIA

As you begin looking at more and more programs, you will see that the variable `I` is used in `FOR/NEXT` loops a lot. Actually, you can use any variable you want, but the `I` keeps cropping up. Like yourself, I was most curious as to why programmers kept using the letter `I`, and after several moments of exhaustive research I found out. The `I` was the "integer" variable in `FORTRAN` (an early computer language), and it was used in "DO loops" since it was faster. The `I` also can be interpreted to stand for "increment." I told you it was trivia.

Having seen how loops function, let's do something practical with a loop. We'll fix up our `CHECKBOOK` program we've been playing with.

In our souped up `CHECKBOOK` program, we are going to use variables in many ways. First, our `FOR/NEXT` loop will use a variable. We'll stick with tradition and use `I`. Second, we will use a variable to indicate the number of loops to be executed. We will use `N` as our "counter" variable. Third, we will use variables for the balance, the amount of the check and the new balance. This program is going to be a little longer; so be sure to `SAVE` it to disk every five lines or so. For cassette, `SAVE` it about every 10 lines.

```
NEW <ENTER>
10 CALL CLEAR
20 CB$ = "CHECKBOOK"
30 PRINT CB$, , ,
40 PRINT "HOW MANY CHECKS ->";
50 INPUT N
60 PRINT "YOUR CURRENT BALANCE ->";
70 INPUT BALANCE
80 REM BEGIN LOOP
90 FOR I = 1 TO N
100 PRINT "BALANCE NOW=$";BALANCE
110 PRINT "AMOUNT OF CHECK #";I; "->";
120 INPUT CHECK
```

Note how each loop begins (a FOR statement is executed) and is terminated (encounters a NEXT statement) in a “nested” sequence. If you have ever stacked a set of different sized cooking bowls, each one fits inside the other; that is because the outer edge of one is larger than the next one. Likewise, in nested loops, the “edge” of each loop is “larger” than the one inside it and “smaller” than the one it is inside.

Stepping Forward and Backwards

Loops can go one step at a time, as we have been using, or they can step at different increments. For example the following program “steps” by 10.

```
NEW <ENTER>
10 CALL CLEAR
20 FOR I = 10 TO 100 STEP 10
30 PRINT I
40 NEXT I
```

This allows you to increment your count by whatever you want. You can even use variables or anything else that has a numeric value. For example

```
NEW <ENTER>
10 CALL CLEAR
20 K = 5
30 N = 25
40 FOR I = K TO N STEP K
50 PRINT I
60 NEXT I
```

Go ahead and RUN the program.

It is also possible to go backwards. Try this program:

```
NEW <ENTER>
10 FOR I = 4 TO 1 STEP -1
20 PRINT I
30 NEXT I
```



As we get into more and more sophisticated (and useful) programs, we will begin to see how all of these different features of TI-99/4A BASIC are very useful. Often, you may not see the practicality of a command initially, but when you need it later on, you will wonder how you could program without it!

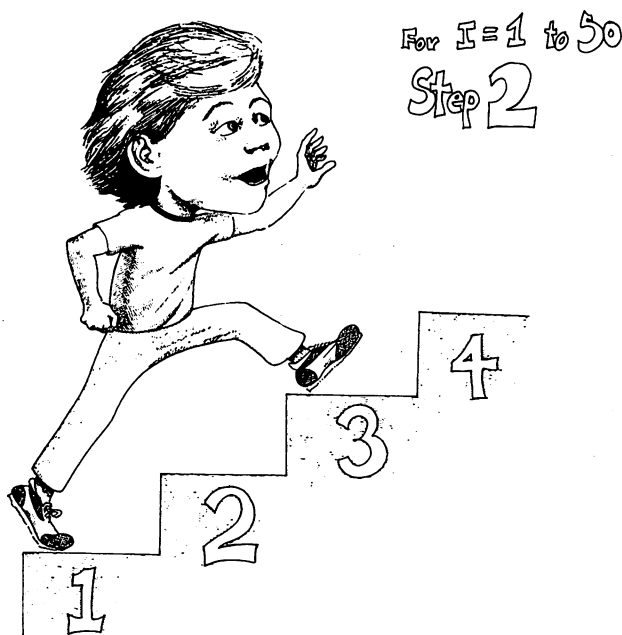
IN CASE YOU WONDERED

You may have noticed that the lines inside the loops were indented. If you tried that on your TI-99/4A you probably found that as soon as you LISTed your program, all the indentations were gone. Unfortunately, that will happen, and without special utilities, there's nothing you can do about it. However, don't worry about it. It is a programming convention for clarity to indent or tab loops to make it easier to understand what the program is doing. It does not affect your program at all.

Counters

Often you will want to count the number of times a loop is executed and keep a record of it in your program for later use. For example, if you run a program that loops with a STEP of 3, you may not know exactly how many times the loop will execute. To find out, programmers use "counters", variables which are incremented, usually by +1, each time a loop is executed. The following program illustrates the use of a counter:

```
NEW <ENTER>
10 CALL CLEAR
20 FOR I = 3 TO 99 STEP 3
30 PRINT I
40 N = N + 1
50 REM LINE 40 IS THE COUNTER
60 NEXT I
70 PRINT
80 PRINT "LOOP EXECUTED"; N; "TIMES."
90 END
```



The first time the loop was entered, the value of N was 0, but when the program got to line 40, the value of 1 was added to N to make it 1 (i.e., $0 + 1 = 1$). The second time through the loop, the value of N began at 1, then 1 was added, and at the top of the loop, line 50, the value of N was 2. This went on until the program exited the loop. Then, after all the looping was finished, presto! Your N told you how many times the loop was executed. Of course, counters are not restricted to counting loops, and they can be incremented by any value you need, including other variables. For example, change line 40 to read:

```
40 N = N + (I * 2)
```

RUN your program again and your “counter total” will be a good deal higher.

SUMMARY

This chapter has begun to show you the power of your computer, and we have really begun programming. One of the most important concepts we have covered is that of the “variable.” The significant feature of variables is that they *vary* (change depending on what your program does). This is true not only with numeric variables, but also with string variables. The various input commands show how we enter values or strings into variables depending on what we want the computer to compute for us. Finally, we have learned how to loop. This allows us, with a minimal amount of effort, to tell the computer to go through a process several times with a single set of instructions. With loops, we can set the parameters of an operation at any increment we want and then sit back and let our TI-99A/4A’s go to work for us.

However, our programming has just begun! In the next chapter we will begin getting into more commands and operations which allow us to delve deeper into the TI-99A/4A’s capabilities and make our programming jobs easier. The more commands we know the less work it is to write a program.

CHAPTER 4

Branching Out

Introduction

In this chapter we will begin exploring new programming constructs that will geometrically increase your programming ability. We will be examining some more sophisticated techniques but, by taking each one step at a time, you will begin using them with ease. Later, when you are developing your own programs, be bold and try out new commands. One problem new programmers have is a tendency to stick with the simple commands they have learned to get a job done. After all, why use “complicated” commands to do what simpler ones can do. Well, the answer to that has to do with simplicity. If one “complicated” command can do the work of 10 “simple” commands, which one is actually simpler? As you get into more and more sophisticated programming applications, your programs become longer and subject to more bugs. The more commands you have to sift through, the more difficult it is to find the bugs; therefore, while it is perfectly OK to write a long program using a lot of simple commands while you’re learning, begin thinking about short-cuts through the use of the more advanced commands.

Related to this issue of maximizing your knowledge of different commands is that of letting the computer perform the computing. This may sound strange at first, but often novices will figure everything out for the computer and use it as a glorified calculator. In the last chapter you may remember we set up a counter to count the times a loop was executed when we used a STEP 3 loop. We could have figured out how many loops were executed instead of letting the computer do it with the counter, but that would have defeated the purpose of programming! So, as you learn new commands, see how they can be used to perform the calculations you had to work out yourself.

BRANCHING

So far all of our programs have gone straight from the top to the bottom with the exception of loops. However, if our TI-99A/4A is to do some real decision making, we must have some way of giving it options. When a program leaves a straight path, it is referred to as either "looping" or "branching." We already know the purpose of a loop, but what is a branch? Well, using the IF/THEN commands, we will see. Consider the following program: *NOTE: By now you should know enough to clear memory with a NEW command, so I won't keep on insulting your intelligence by putting them at the beginning of each program.*

```
10 CALL CLEAR
20 PRINT "CHOOSE ONE OF THE"
30 PRINT "FOLLOWING BY NUMBER:"
40 PRINT
50 PRINT "1. BANANAS"
60 PRINT "2. ORANGES"
70 PRINT "3. PEACHES"
80 PRINT "4. WATERMELONS"
90 PRINT ", , , , "WHICH";
100 INPUT X
110 CALL CLEAR
120 IF X = 1 THEN 200
130 IF X = 2 THEN 300
140 IF X = 3 THEN 400
150 IF X = 4 THEN 500
160 GOTO 10
170 REM LINE 160 IS A 'TRAP' TO MAKE SURE THE
    USER CHOOSES 1, 2, 3, OR 4
200 PRINT "BANANAS"
210 END
300 PRINT "ORANGES"
310 END
400 PRINT "PEACHES"
410 END
500 PRINT "WATERMELONS"
510 END
```

As you can see, your computer “branched” to the appropriate place, did what it was told and ENDED. Not very inspiring I admit, but it is a clear example. Now let’s try something a little more practical for your kids to play with in their math homework.

```
10 CALL CLEAR
20 AG$="ADDITION GAME"
30 PRINT AG$
40 PRINT
50 PRINT
60 PRINT "FIRST NUMBER -->" ;
70 INPUT A
80 PRINT "SECOND NUMBER-->" ;
90 INPUT B
100 PRINT "WHAT IS"; A ; "+" ; B ;
110 INPUT C
120 IF C = A + B THEN 200
130 PRINT
140 PRINT "THAT'S NOT QUITE IT."
150 PRINT "TRY AGAIN."
160 PRINT
170 GOTO 100
200 PRINT "THAT'S RIGHT!"
210 PRINT "VERY GOOD"
220 PRINT
230 PRINT "MORE (Y/N)";
240 INPUT AN$
250 IF AN$="N" THEN 300
260 IF AN$ = "Y" THEN 270
300 CALL CLEAR
310 PRINT , , , ,
320 PRINT "HOPE TO SEE YOU AGAIN SOON"
330 END
```

As you can see, the more commands we learn, the more fun we can have. Just for fun, change the program so that it will handle multiplication, division, and subtraction.

WHAT'S IN A NAME?

Kids (of all ages) like to have their names displayed. See if you can change the above program so that it asks the child's name; then when the program responds with either a correction or affirmation command, it mentions the child's name. (e.g. THAT'S RIGHT! VERY GOOD, SAM). Use NA\$ as the name variable.

IF/THEN/ELSE

Another aspect of TI BASIC is in choosing between two branches. This can be done by adding the ELSE statement to our IF/THEN statements. For example, let's look at the following simple program to see how this works:

```
10 CALL CLEAR
20 PRINT "PRESS <ENTER> TO CONTINUE"
30 INPUT "OR 'Q' TO QUIT" : AN$
40 IF AN$ = "Q" THEN 100 ELSE 200
50 PRINT "YOU CAN'T GET HERE!!!"
100 REM ****
110 REM QUIT
120 REM ****
130 PRINT "YOU CHOSE TO END IT ALL"
140 END
200 REM *****
210 REM CONTINUE
220 REM *****
230 PRINT "YOU CHOSE TO CONTINUE"
240 PRINT
250 GOTO 20
```

Obviously there are easier ways to do that, but it is important that you see how IF/THEN/ELSE works. You might note that no matter what you do, you will not get to Line 50. (Well, you can change the program, but that's not cricket.)



RELATIONALS

So far we have used only “=” to determine whether or not our program should branch. However, there are other states, referred to as “relationals,” that we can also query. The following is a complete list of the relationals we can employ:

SYMBOL	MEANING
=	Equal to
<	Less than
>	Greater than
<>	Not equal to
>=	Greater than or equal to
<=	Less than or equal to

Now let's play with some of these, and then we'll examine them for their full power. Here are some quickie programs:

```
10 CALL CLEAR
20 PRINT "NUMBER 1-->";
30 INPUT A
40 PRINT "NUMBER 2-->";
50 INPUT B
60 IF A > B THEN 100
70 IF A < B THEN 200
80 IF A = B THEN 300
100 PRINT "NO. 1 GREATER THAN NO. 2"
110 END
200 PRINT "NO. 1 LESS THAN NO. 2"
210 END
300 PRINT "NO. 1 EQUAL TO NO. 2"
```

```
10 CALL CLEAR
20 PRINT "CONTINUE (Y/N)";
30 INPUT AN$
40 IF AN$ <> "Y" THEN 60
50 GOTO 10
60 END
```




```
10 CALL CLEAR
20 PRINT "HOW OLD ARE YOU";
30 INPUT AGE
40 IF AGE >= 21 THEN 200
50 CALL CLEAR
60 PRINT
70 PRINT "SORRY, YOU'VE GOT"
80 PRINT "TO BE 21 OR OLDER"
90 PRINT "TO COME IN HERE!"
100 END
200 CALL CLEAR
210 PRINT
220 PRINT "WHAT WOULD YOU LIKE?"
```

OK, you have the idea how relationals can be used with IF/THEN statements; note they work with strings as well as numeric variables. However, there is another way to use relationals. Try the following from the Immediate mode:

```
A = 10
B = 20
PRINT A = B
```

Your computer responded with a 0, right? This is a logical operation. If a condition is false, your TI-99A/4A responds with a 0, but if it is true, it responds with a -1. Now try the following little program.

```
10 CALL CLEAR
20 A = 10
30 B = 20
40 C = A > B
50 PRINT C
```

When you RUN the program, you again get a 0. This is because the variable C was defined as A being greater than B. Since A was less than B, the variable C was 0 or "false." Now, let's take it a step further:

```
10 CALL CLEAR
20 A = 10
30 B = 20
40 C = A > B
50 IF C = 0 THEN 100
50 PRINT "A IS GREATER THAN B"
60 END
100 PRINT "A IS LESS THAN B"
```

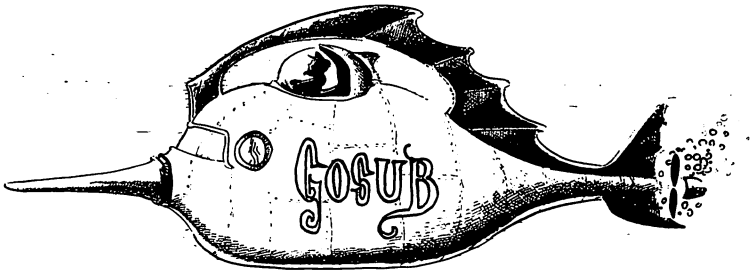
Later, we will see further applications of these logical operations of the TI-99A/4A. For now though, it is important to understand that a true condition is represented by a -1 and a false condition by a 0.

Subroutines

Often in programming there is some operation you will want your computer to perform at several different places in the program. You can either repeat the instructions again and again or use GOTOs all over the place to return to your original spot after branching to the operation. On the other hand, you can set up "subroutines" and jump to them using GOSUB and get back to your starting point using the RETURN statement. Up to a point the GOSUB statement works pretty much like the GOTO statement since it sends your program

bouncing off to a line out of sequence. Also, the RETURN statement is something like GOTO since it also sends your program to an out-of-sequence line. However, the GOSUB/RETURN pair is unique in what it does. Let's take a look at a simple example to see how it works:

```
10 CALL CLEAR
20 A$ = "HELLO"
30 GOSUB 100
40 A$ = "HOW ARE YOU TODAY?"
50 GOSUB 100
60 A$ = "I'M FINE"
70 GOSUB 100
80 END
100 PRINT A$
110 RETURN
```



Our example shows that a GOSUB statement works exactly like a statement on the line itself except that it is executed elsewhere in the program. The RETURN statement brings it back to the next statement after the GOSUB statement. Using the GOSUB/RETURN pair, it is much easier to weave in and out of a program than using GOTO since the RETURN automatically takes you back to the jump-off point.

To better illustrate the usefulness of GOSUB, let's change line 100 to something more elaborate. Try the following. *NOTE: We will be getting ahead of ourselves a bit with this example, but the following is meant to illustrate something very useful in GOSUBs.*

```
100 L = LEN (A$)/2
110 PRINT TAB(11 - L);A$
120 RETURN
```

Now when you RUN the program, all of your strings are centered. As you can see, a single routine handled all of the centering and, instead of having to rewrite the routine every time you want a string centered, you just used a GOSUB to line 100.

NEATNESS COUNTS

We really have not discussed the structure of programs too much up to this point. In part, this is because we have not really had the need to do so. As our instruction set grows, so too does the possibility for errors, and by now if you haven't made an error you haven't been keying in these programs! One way to minimize errors, especially using GOSUBs, is to organize them into coherent blocks. Basically, a "block" is a subroutine within a range of lines. For example, you might block your subroutines by 100s or 1000s, depending on how long the subroutines are; thus, you might have subroutines beginning at lines 500, 600 and 700. It doesn't matter if the subroutine is 1 line or 10 lines; as long as it is confined to the block, it is easier to debug, easier for others and easier for you to understand what is happening in the program. In general it is just a good programming practice.

Computed GOTO and GOSUB

Now we're going to get a little fancier, but in the long run, it will result in clearer and simpler programming. As we have seen, we can branch on a "conditional" (e.g., IF A = 1 THEN 200). The easier way to make a conditional jump is to use "computed" branches using the ON statement. While we're at it, why not save some time INPUT'ing values. We can have our prompt on the same line as our INPUT statement. Look at lines 20 and 60 in the next program. The INPUT variable is separated from the prompt message by a colon. Using this format, we can save the extra line every time we use INPUT. Now let's look at an example using both computed GOSUBs and our new INPUT format.

```
10 CALL CLEAR
20 INPUT "A NO. FROM 1-5":A
30 IF A < 1 THEN 20
40 IF A > 5 THEN 20
50 ON A GOSUB 100,200,300,400,500
60 INPUT "CONTINUE? (Y/N)" : AN$
70 IF AN$ = "" THEN 60
80 IF AN$ < > "Y" THEN 1000
90 GOTO 10
100 PRINT "ONE"
110 PRINT
120 RETURN
200 PRINT "TWO"
210 PRINT
220 RETURN
300 PRINT "THREE"
310 PRINT
320 RETURN
400 PRINT "FOUR"
410 PRINT
420 RETURN
500 PRINT "FIVE"
510 PRINT
520 RETURN
1000 END
```

The format for a computed GOSUB/GOTO is to enter a variable following the ON command. The program will then jump the number of commas to the appropriate line number. If a 1 is entered, it takes the first line number, a 2, the second, and so on. It's a lot easier than entering

```
70 IF A = 1 THEN 100
80 IF A = 2 THEN 200
etc.
```

However, it is necessary to use relatively small numbers in the ON variable since there is a limited number of subroutines. If your program is computing larger numbers, convert the larger numbers into smaller ones by changing the variables. For example:

```
10 CALL CLEAR
20 INPUT "ANY NUMBER-->":A
30 IF A < 100 THEN 100
40 IF A >= 100 THEN 200
50 ON B GOSUB 1000, 2000, 3000
60 REM COMPUTED GOSUB ON 'B' VARIABLE IN
  LINE 50
70 INPUT "PRESS <ENTER> TO CONTINUE":
  ENTER$
80 IF ENTER$ = "QUIT" THEN 5000
90 GOTO 10
100 B = 1
110 GOTO 50THAN 200 "
200 IF A >= 200 THEN 300
210 B = 2
220 GOTO 50
300 B = 3
310 GOTO 50
1000 PRINT "LITTLE"
1010 RETURN
2000 PRINT "MEDIUM"
2010 RETURN
3000 PRINT "BIG"
3010 RETURN
5000 END
```

RUN the program and enter any number you want. Since the program is branching on the variable B and not on A (the INPUT variable), you will not get an error since the greatest value of B can only be 3.

Now let's get back to relationals and see how they can be used with computed GOSUBs. Remember, in using relationals, the only numbers we get are 0's and 1's for false and true respectively. However, we can use these 0's and 1's just like regular numbers. Try the following:

```
10 CALL CLEAR
20 X = 1
30 Y = 2
40 Z = 3
50 A = X < Z
60 B = Y > Z
70 C = Z > X
80 PRINT "A + A="; A + A
90 PRINT
100 PRINT "A + B="; A + B
110 PRINT
120 PRINT "A + B + C="; A + B + C
130 END
```

Now, before you RUN the program, see if you can determine what will be printed by lines 60, 70 and 80. Once you have made a determination, RUN the program and see what happens. Go ahead and do it. How'd you do? Let's go over it step by step.

1. Since X is less than Z, A will be "true" with a value of one (-1). Therefore A + A (-1 + -1) will equal -2.
2. Since Y is not less than Z, (Y = 2 and Z = 3, remember) B will be "false" with a value of 0. Therefore, A + B (-1 + 0) will total -1.
3. Since Z is greater than X, C will be "true" with a value of -1. Therefore A + B + C (-1 + 0 + -1) will equal -2. If you got it right, congratulations! If not, go over it again. Remember, very simple things are happening, and so don't look for a complicated explanation!

Now that we see how we can get numbers by manipulating relationals, let's use them in computed GOSUBs. The following program shows how:

```
10 CALL CLEAR
20 INPUT "HOW BIG WAS THE CROWD":SIZE
30 R = 1 + (SIZE >= 500) + (SIZE >= 1000)
40 IF $=0 THEN 1000
50 IF R = -1 THEN 2000
60 ON R GOSUB 100,200,300
70 INPUT "PRESS <ENTER> OR 'Q' ":ANS
80 IF ANS <> "Q" THEN 20
90 END
100 PRINT "SMALL"
110 RETURN
200 PRINT "MEDIUM"
210 RETURN
300 PRINT "HUGE"
310 RETURN
1000 R = 2
1010 GOTO 60
2000 R = 3
2010 GOTO 60
```

This program is hinged on line 30's formula or algorithm. Let's see how it works:

1. There are three conditions:
 - a. SIZE is less than 500
 - b. SIZE is 500 or more but less than 1000
 - c. SIZE is 1000 or greater.
2. If the first condition exists, both $\text{SIZE} \geq 500$ and $\text{SIZE} \geq 1000$ would be false. Thus $1 + 0 + 0 = 1$. Therefore $R = 1$.
3. If SIZE is ≥ 500 but less than 1000 then $\text{SIZE} \geq 500$ would be true but $\text{SIZE} \geq 1000$ would be false. Thus we would have $1 + (-1) + 0 = 0$. Convert the value of R to 2.

4. Finally if SIZE is both ≥ 500 and ≥ 1000 then our formula would result in $1 + (-1) + (-1) = -1$. Convert the value of R to 3.

REST AREA

At this point let's take a little rest and reflection. In programming, there is no such thing as *the right way* and *the wrong way*. Certain programs are more efficient, faster or take less code and memory than others, but the computer makes no moral judgments. If a program does what you want it to do, no matter how slowly it does it or how long it took you to write it, it is *right*. In the above example we used an algorithm with relationals to do something we could have done with more code. Don't expect to use such formulas right off the bat unless you have a strong background in math. If you're not used to using algorithms, don't expect to understand their full potential right away. The one we used is relatively simple, and you will find far more elaborate ones as you begin looking at more programs. The main point is to keep plugging ahead. With practice you will learn all kinds of little shortcuts and formulas, but if you get stuck along the way, just keep on going. Remember, as long as you can get your program running the way you want it to, you're doing the *right* thing.

Strings and Relationals

Before we leave our discussion of computed GOTOs and GOSUBs with relationals, let's take a look at how relationals handle strings. Try the following :

```
A$ = "A"  
B$ = "B"  
PRINT B$ > A$ <ENTER>
```

Surprised? In addition to comparing numeric variables, relationals can compare alphabetic string variables with "A" being the lowest and "Z" the highest. (Actually, any string variables can be compared, but we will look at just the alphabetic ones here.) So if we ask is B\$ greater than A\$, we get a "-1" (true) since B\$ was a B and A\$ was an A. Now you might be wondering what on earth you could possibly want to do with this knowledge. Well, in sorting strings (like putting names in alphabetical order) such an operation is crucial. Later on we will show you a routine for sorting strings, but for now let's make a simple string sorter for sorting two strings.

```
10 CALL CLEAR
20 INPUT "WORD #1 --> " : A$
30 INPUT "WORD #2 --> " : B$
40 PRINT , , ,
50 IF A$ < B$ THEN 100
60 IF A$ > B$ THEN 200
100 PRINT A$ , , B$
110 END
200 PRINT B$ , , A$
```

Just what you needed! A program that will arrange two words into alphabetical order!

ARRAYS

The best way to think about arrays is as a kind of variable. As we have seen, we can name variables A, D\$, KK, X1\$ and so forth. An array uses a single name with a number to differentiate different variables. Consider the following two lists, one using regular string variables and the other using a string array:

STRING VARIABLES	STRING ARRAY
P\$ = "PIG"	AM\$(1) = "PIG"
C\$ = "CHICKEN"	AM\$(2) = "CHICKEN"
D\$ = "DOG"	AM\$(3) = "DOG"
H\$ = "HORSE"	AM\$(4) = "HORSE"

Now if we PRINT H\$ we'd get HORSE and if we PRINT AM\$(4) we'd also get HORSE. Likewise, we could use arrays for numeric variables such as:

```
A(1) = 1
A(2) = 2
A(3) = 3
A(4) = 4 etc.
```

Again, you may well ask, "So what? Why not use just regular numeric or string variables instead of arrays?" Well, for one thing it can be a lot easier to keep track of what you're doing in a program using arrays, and for another, it can save a lot of time. Consider the following program for INPUTting a list of 10 names using a string array.

```
10 CALL CLEAR
20 FOR I = 1 TO 10
30 PRINT "NAME #"; I ;
40 INPUT NA$(I)
50 NEXT I
60 FOR I = 1 TO 10
70 PRINT NA$(I)
80 NEXT I
```

Now write a program that does the same thing using non-array variables. It would take a lot more code to do so, but go ahead and try it. Use the variables N0\$ through N9\$ for the names just to see what it would take.

If you re-wrote the program, you would see how much time you saved using arrays, but before going on let's take a closer look at how the program worked with the FOR/NEXT loop and array variable:

1. The FOR/NEXT loop generated the numbers sequentially so that the array would be the following:

```
FOR I = 1 TO 10
  NAS(1) <--First time through loop
  NAS(2) <--Second time through loop
  NAS(3) <--Third time through loop
  NAS(4) etc.
  NAS(5)
  NAS(6)
  NAS(7)
  NAS(8)
  NAS(9)
  NAS(10)
NEXT I
```

2. Each string INPUT by the user was stored in a sequentially numbered array variable.
3. Output, using the PRINT statement, was generated by the FOR/NEXT loop sequentially supplying numbers to be entered into array variables.

Now, to get used to the idea that an array variable is a variable, enter the following:

```
A(10) = 432
PRINT A(10) <ENTER>
XYZ(9) = 2.432
PRINT XYZ(9) <ENTER>
R2D2$(1) = "BEEP!"
PRINT R2D2$(1) <ENTER>
J%(5) = 321
PRINT J%(5) <ENTER>
```

OK, maybe it didn't take all that to convince you that an array is a variable with a number in parentheses after it, but it's easy to forget and think of arrays as something more exotic than they are.

The DIMension of an ARRAY

If you've been very observant, you may have noticed we haven't gone over the number 10 in our array examples. The reason behind that is because once our array is larger than 10 we have to use the DIM (dimension) statement to reserve space for our array. (Actually 11 array elements are automatically dimensioned - 0 to 10.) The following is an example of the format for DIMensioning an array.

```
10 CALL CLEAR
20 DIM AB(150)
30 REM DIMENSION OF ARRAY VARIABLE 'AB' IN
  LINE 20
40 FOR I = 1 TO 150
50 AB(I) = I
60 NEXT I
70 FOR I = 1 TO 150
80 PRINT AB(I),
90 NEXT I
```

RUN the program as it is written. It should work fine. Now delete line 20 by simply entering 20. (Remember how we learned to delete single line numbers by entering that number?) Now RUN the program and you will get an error for not DIMming the ARRAY. (* BAD SUBSCRIPT IN 50 - that's because there was no DIM statement in line 20.) So, whenever your arrays are going to have more than 11 values from 0 to 10, be sure to DIM them.

BETTER SAFE THAN SORRY DEPT.

Many programmers always DIM arrays, regardless of the number in the array. It is perfectly all right to do so, and statements such as DIM X\$(3) or DIM N% (5) are valid. Often, when copying programs from books or magazines, you may run across these lower level DIM statements because the programmer thinks it's a good idea to DIM all arrays as part of programming style and clarity. Furthermore, you can save memory space by using the minimal amount of DIMension space; if the program is large enough, it may be necessary to DIM an array at less than 11. Finally, some versions of BASIC require all arrays to be DIMensioned.

Multi-dimensional Arrays

So far, all we have examined are single dimension arrays. However, it is possible to have arrays with two or more dimensions. Let's begin with two dimensional arrays and examine how to use arrays with more than a single dimension.

The best way to think of a two-dimensional array is as a matrix. For example, if our array ranged from 1 to 3 on two dimensions the entire set would include: A(1,1) A(1,2) A(1,3) A(2,1) A(2,2) A(2,3) A(3,1) A(3,2) and A(3,3). By laying it out on a matrix we can think of the first number as a row and the second as a column. This makes it much clearer:

	COLUMN #1	COLUMN #2	COLUMN #3
ROW #1	A(1,1)	A(1,2)	A(1,3)
ROW #2	A(2,1)	A(2,2)	A(2,3)
ROW #3	A(3,1)	A(3,2)	A(3,3)

Again, it is important to remember that each element in the array is simply a type of variable. To drum that into your head do the following:

```
XV$(3,1) = "I'M A VARIABLE"  
PRINT XV$(3,1) <ENTER>  
JK(2,2) = 21  
PRINT JK(2,2) <ENTER>  
MM (1,1) = 3.212  
PRINT MM(1,1) <ENTER>
```

Now let's use a two-dimensional array in a program. Our program will be to line up people in a four member marching band. (This band is from a very small town.)

```
10 CALL CLEAR  
20 DIM BA$(2,2)  
30 REM MAKE 2 'ROWS' AND 2 'COLUMNS'  
40 FOR I = 1 TO 2  
50 REM ROWS  
60 FOR J = 1 TO 2
```

```

70 REM COLUMNS
80 READ BA$(I,J)
90 NEXT J
100 NEXT I
110 DATA MARY, TOM, SUE, PETE
120 REM OUTPUT BLOCK
130 FOR I = 1 TO 2
140 REM ROWS
150 FOR J = 1 TO 2
160 REM COLUMNS
170 PRINT BA$(I,J),
180 REM COMMA WILL FORMAT
  OUTPUT 2 ACROSS
190 NEXT J
200 NEXT I

```

When you RUN this program, all of your band members will be lined up. However, you could have done the same thing with a single dimension array since all that “lines them up” is the use of the comma to format the PRINT statement in line 170. So, what’s the big deal about a two-dimensional array? Well, to see, let’s add some lines to our program:

```

300 INPUT "PRESS <ENTER>" : ENTER$
310 CALL CLEAR
320 PRINT "WHAT ROW & COLUMN"
330 PRINT "WOULD YOU LIKE TO SEE?"
340 INPUT "ROW #->":R
350 INPUT "COL #->":C
360 PRINT
370 PRINT BA$(R,C); "IS IN ROW"; R;
  "COLUMN";C
380 PRINT
390 INPUT "MORE?(Y/N)":M$
400 IF M$ = "Y" THEN 300

```

Now you can locate the value or contents of a specific array on two dimensions. In our example, if you know the row number and column number, you can find the band member in that position. The use of two-dimensional arrays in problems dealing with matrixes is an important addition to your programming commands.

It is also possible to have several more dimensions in an array variable. As you add more and more dimensions, you have to be careful not to confuse the different aspects of a single array. Sometimes, when a multi-dimensional array becomes difficult to manage (or use), it is better to break it down into several one- or two-dimensional arrays. But just for fun, let's see what we might want to do with a three-dimensional array with the following program: (By the way, this program is based on an actual application!)

```
10 CALL CLEAR
20 PRINT "WINECELLAR ORGANIZER",,,
30 DIM W$(5,5,5)
40 INPUT "NO. BOTTLES TO STORE?":NB
50 PRINT
60 FOR I = 1 TO NB
70 INPUT "RACK #->":RA
80 INPUT "ROW #->":RO
90 INPUT "COL #->":CO
100 INPUT "NAME OF WINE:":WINE$
110 W$(RA,RO,CO) = WINE$
120 NEXT I
200 REM *** ROUTINE FOR CHECKING CONTENTS
    OF WINE CELLAR ***
210 CALL CLEAR
220 INPUT "WHICH RACK # TO CHECK?":RR
230 FOR I = 1 TO 5
240 FOR J = 1 TO 5
250 IF W$(RR,I,J) = "" THEN 400
260 PRINT "RACK #";RR;"ROW #";I;"COLUMN #";J
270 PRINT "CONTAINS";W$(RR,I,J)
280 NEXT J
290 NEXT I
300 END
400 REM *** EMPTY SUBROUTINE ***
410 W$(RR,I,J) = "EMPTY"
420 GOTO 260
```


Now that was a pretty long program, but go over it carefully to make sure you understand what it is doing. Again, let me remind you that all a three-dimensional array is, is a variable with a lot of numbers in parentheses.

SUMMARY

We covered a good deal in this chapter; if you understood everything, excellent! If you did not, don't worry; for with practice it will all become very clear. Whatever your understanding of the material, though, experiment with all the statements. Be *bold* and daring with your computer's commands. As long as you have a disk or cassette on which you can practice your skills, the worst that can happen is that you will erase a few programs!

We learned that your TI-99/4A computer can compute! Using the IF/THEN commands and relationals we can give the computer the power of "decision making." Using subroutines it is possible to branch at decision points to anywhere we want in our program. Computed GOTOs and GOSUBs allow the execution to move appropriately with a minimal amount of programming.

Finally, we examined array variables. Arrays allow us to enter values into sequentially arranged variables (or elements). Using FOR/NEXT loops it is possible to quickly program multiple variables up to the limits of our DIMensions. Not only do arrays assist us in keeping variables orderly, they save a good deal of work as well.

In the next chapter we will begin working with commands that help arrange everything for us. As our programs become more and more sophisticated, we will need to keep better track of what we're doing. By organizing our programs into small, manageable chunks, we can create clear, useful programs.

CHAPTER 5

Organizing the Parts

Introduction

Unless we organize, as we accumulate more and more information, work, or just about anything else, things get confusing. Good organization allows us to do more and to handle more complex and larger problems. These principles hold with programming. As we learn more commands, we can do more things; but the more we do, the more likely we are to get tangled up and lost.

One of the areas that is likely to be the first to suffer from overflow is that of formatting output. Variables get mixed up, arrays are misnumbered and the screen is a mess. In order to handle this kind of problem, we will deal extensively with text and string formatting. Not only will we be able to put things where we want them, but we will do it with style!

The second major area of disorganization is I/O (INPUT/OUTPUT). Part of the problem has to do with formatting, but even more elementary is the problem of organizing the input and output so that data is properly analyzed. Data has to be connected to the proper variables and be subject to the correct computations. Thus, in addition to examining string formatting, we will also look at organizing data manipulation.

FORMATTING TEXT

In Chapter 1 we said that the TI-99/4A keyboard works in many ways like a typewriter. One feature of a typewriter is its ability to set tabs so that the user can automatically place text a given number of spaces from the left margin. With your TI-99/4A, you can TAB almost like a typewriter. Before examining the TAB statement, let's look at your screen. The following program uses every vertical and horizontal position available:

```

10 CALL CLEAR
20 H$= "1"
30 FOR I = 1 TO 28
40 PRINT H$;
50 NEXT I
60 FOR I = 2 to 23
70 PRINT I
80 NEXT I
90 PRINT 24;
100 FOR I = 1 TO 2000
110 NEXT I

```

There are 28 horizontal positions and 24 vertical positions where you can place your text. Everything begins at the bottom of your screen and moves upward. Examine the program carefully to see what has been done to place the numbers where we did. First we used a string variable, H\$, to lay out our horizontal positions. Why did we have to use a string? Why not just have PRINT 1; in line 40? Well, change the program so that line 40 is PRINT 1; and see what happens. Okay, if you did so, you found that the number 1 takes up three positions, since all numbers and numeric variables are preceded and followed by a space. Strings, however, have no leading or following spaces, so they can be positioned directly adjacent to one another.

A second item to note in the program is in line 90. Instead of having our loop in line 60 run up to 24, we ran it only to 23 and then added the 24 in line 90. This was done so that there was not a "linefeed" after 24. We could not put a semi-colon after the PRINT I in line 70 or we would have horizontal placement of our text. Thus we ran our loop up to 23, PRINTed 24 with a semi-colon and avoided a line feed. Finally, to hold everything on the screen we had a "pause loop" in line 100. (This avoids the ** DONE ** message.)

TAB (N) is used within a PRINT statement to place the next character N spaces from left margin. We are able to produce a vertical tab by using empty PRINT statements in loops. To see how this works, the following program will put an "X" right smack dab in the middle of your screen:

```

10 CALL CLEAR
20 INPUT "ENTER MESSAGE": MS$
30 PRINT
40 PRINT "HORIZONTAL POS.": H
50 PRINT
60 INPUT "VERTICAL POS.": V
70 NEXT PAUSE

```

Now let's have a some fun with our commands. Here's a little program that will give you an idea of how to place text within your program.

```

10 CALL CLEAR
20 INPUT "ENTER MESSAGE ": MS$
30 PRINT
40 INPUT "HORIZONTAL POS. ": H
50 PRINT
60 INPUT "VERTICAL POS. ": V
70 CALL CLEAR
80 PRINT TAB(H); MS$;
90 FOR VER = 1 TO V
100 PRINT
110 NEXT VER
120 PRINT "<ENTER> TO CONTINUE"
130 INPUT " ": AS$
140 IF AS$=" " THEN 10
150 IF AS$<>"Q" THEN 120
160 END

```

As you can see, variables can be used with formatting statements. Thus, TAB (H) is read in the same way as TAB(10) or TAB(15) or any other number between 1 and 28. (TAB (0) is the same as TAB(1)). Using the above program, what do you think would happen if you entered "THIS IS A LONG STRING", a HORIZONTAL placement of 27 and a VERTICAL placement of 23? Since the maximum TAB is 28 and the maximum vertical placement is 24, the string (MS\$) will go over the boundaries. Go ahead and try it to see what happens. In fact, it would be a good idea to test the limits of TAB and vertical placement with this program to get a clear understanding of their parameters.

Unraveling Strings

Our discussion of strings up to this point has involved “whole” strings. That is, whatever we define a string to be, no matter how long or short, can be considered a “whole” string. For example, if we define R\$ as WALK then we can consider WALK to be the whole of R\$. Likewise, if we defined R\$ as A VERY LONG AND WORDY MESSAGE , then A VERY LONG AND WORDY MESSAGE would be the whole string of R\$. There will be certain occasions when we want to use only part of a string or tie several strings together. (When we get into data base programs, we will find this to be very important.) Also, there are applications where we will need to know the length of strings, find the numeric values of strings and even change strings into numeric variables and back again.

TRUST ME!

I hate to admit it, but when I first learned about the sub-string commands we are about to discuss I thought, “Boy, what a waste of time!” It was enough to get the simple material straight, but why in the world would anyone want to chop up strings and put them back together again? If you want only a certain segment of a string, why not simply define it in terms of that segment? And if you want a longer string, then just define it to be longer! Those were my thoughts on the matter of string formatting. However, I have now come to the point where I find it very difficult to even conceive of programming without these powerful commands. So, trust me! String formatting commands are terrific little devices to have. If you do not see their applicability right away, you will as you begin writing more programs.

String Formatting

We will divide our discussion of string formatting into four parts: 1) Calculating the length of a string, 2) Locating parts of strings, 3) Changing strings to numeric variables and back again and 4) Tying strings together (concatenation).

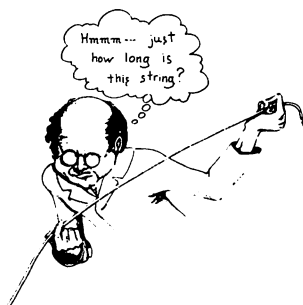
Calculating the LENGTH of Strings

Sometimes it is necessary to calculate the length of a string for formatting output. Happily, your TI-99/4A is very good at telling you the length of a particular string. By the command `PRINT LEN (A$)`, you will be given the number of characters, including spaces, your string has. Try the following little program to see how this works:

```
10 CALL CLEAR
20 INPUT "NAME OF STRING " :A$
30 PRINT A$; " HAS "; LEN(A$); "CHARACTERS"
40 PRINT
50 INPUT "MORE?(Y/N)"; AN$
60 IF AN$ = "" THEN 50
70 IF AN$ = "Y" THEN 20
```

Now to see a more practical application, we will look at a modified version of the centering routine we used in the last chapter.

```
10 CALL CLEAR
20 PRINT "ENTER A STRING LESS
  THAN 28 CHARACTERS"
30 INPUT "->": S$
40 CALL CLEAR
50 L = 14 - LEN(S$)/2
60 PRINT TAB(L); S$
70 FOR I = 1 TO 18
80 PRINT
90 NEXT I
100 PRINT "PRESS <ENTER> TO CONTINUE"
110 INPUT "OR 'Q' TO QUIT": A$
120 IF A$=" " THEN 10
130 IF A$ <> "Q" THEN 100
```



Now that we can see how to compute the LENgth of a string and then use that LENgth to compute our tabbing, let's see how we can control the input with the LEN command. Suppose you want to write a program that will print out mailing labels but your labels will hold only 15 characters. You want to make sure all of your entries are 15 or fewer characters long, including spaces. To do this we will write a program that checks the LENgth of a string before it is accepted.

```
10 CALL CLEAR
20 PRINT "ENTER A NAME LESS THAN 15"
30 PRINT "CHARACTERS INCLUDING SPACES"
40 INPUT "DO NOT USE COMMAS ": NAMES$
50 REM THE FOLLOWING LINE IS A TRAP
60 IF LEN (NAMES$) > 15 THEN 200
70 PRINT
80 INPUT NAMES$ , , ,
90 INPUT "ANOTHER NAME?(Y/N)": AN$
100 IF AN$ = "" THEN 90
110 IF AN$ < > "Y" THEN 130
120 GOTO 10
130 END
200 REM
210 REM **ERROR ROUTINE**
220 REM
230 CALL CLEAR
240 PRINT "PLEASE USE 15"
250 PRINT "CHARACTERS OR LESS" , , ,
260 GOTO 20
```

Break the rule!!! Go ahead and enter a string of more than 15 characters to see what happens. (If your computer gets snotty with you, you can always re-program it. It helps to remind it of that fact periodically.) If the program was entered properly, it is impossible to enter a string of more than 15 characters.

From the above examples, you can begin to see how the LEN command can be useful in several ways. There are many other ways that such commands can be employed to reduce programming time, clarify output and compute information. The key to understanding its usefulness is to experiment with it and see how other programmers use the same command.

Finding the SEG\$ments of a String

Suppose you want to use a single string variable to describe three different conditions, such as POOR FAIR GOOD, but you want to use only part of that string to describe an outcome. Using SEG\$, it is possible to PRINT only that part of the string you want. For example, the following program lets you use a single string to describe three different conditions:

```
10 CALL CLEAR
20 X$="POOR FAIR GOOD"
30 PRINT "HOW DO YOU FEEL?"
40 INPUT "<P>OOR <F>AIR <G>OOD" : F$
50 IF F$="" THEN 40
60 IF F$="P" THEN 100
70 IF F$="F" THEN 200
80 IF F$="G" THEN 300
100 PRINT SEG$(X$,1,4)
110 GOTO 500
200 PRINT SEG$(X$,6,4)
210 GOTO 500
300 PRINT SEG$(X$,11,4)
500 REM
510 REM ** CHOICE SUBROUTINE **
520 REM
530 PRINT , , ,
540 INPUT "ANOTHER GO?(Y/N)" : CHOICES$
550 IF CHOICES$="Y" THEN 10
60 IF CHOICES$ <> "N" THEN 530
```

NOTE: You may have noticed that in the last several example programs there have been different ways to choose to continue. This has been done to give you an idea of various ways to "trap" branches.

Let's face it, it would have been easier and no less efficient to simply branch to a PRINT 'GOOD' 'FAIR' or 'POOR'. But no matter, it was for purposes of illustration and not optimizing program organization. Let's see what the new commands do.

To give you some immediate experience with these commands, try the following:


```

W$ = "WHAT A MESS"
PRINT SEG$(W$,8,4) <ENTER>
G$ = "BURLESQUE"
PRINT SEG$(G$,4,3) <ENTER>
X$="A PLACE IN SPACE"
PRINT SEG$(X$,12,5); " "; SEG$(X$,5,3)<ENTER>

```

Another trick with partial strings is to assign parts of one string to another string. For example:

```

10 CALL CLEAR
20 BIG$ = "LONG LONG AGO AND FAR FAR AWAY"
30 LITTLE$ = SEG$(BIG$,11,3)
40 AWY$ = SEG$(BIG$,27,4)
50 LG$ = SEG$(BIG$,1,4)
60 PRINT ""
70 PRINT AWY$;" ";LG$;" ";LITTLE$
80 REM BEFORE YOU RUN IT, SEE IF YOU
   CAN GUESS THE MESSAGE.

```

For an interesting effect, try the following little program:

```

10 CALL CLEAR
20 INPUT "YOUR NAME--> ": N$
30 CALL CLEAR
40 FOR I = LEN(N$) TO 1 STEP -1
50 PRINT SEG$(N$,I,1);
60 NEXT I
70 REM DELAY LOOP IN LINES 70-80
80 FOR I = 1 TO 1000
90 NEXT I
100 FOR V = 1 TO 11
110 PRINT
120 NEXT V
200 REM ** IN LINES 230-250 THE 'K LOOP'
   SLOWS IT DOWN FOR SLOW MOTION EFFECT **
210 FOR I = 1 TO LEN(N$)
220 PRINT SEG$(N$, I,1);

```

```

230 FOR K = 1 TO 50
240 NEXT K
250 NEXT I
300 FOR VT = 1 TO 5
310 PRINT
320 NEXT VT
330 PRINT TAB(5); "AGAIN?(Y/N)";
340 INPUT AN$
350 IF AN$ = "" THEN 330
360 IF AN$ = "Y" THEN 10

```

Now you have probably been wondering ever since you got your computer how to make it print your name backwards. Well, now you know! (If your name is BOB you probably didn't notice it was printed backwards - try ROBERT.) Actually, the above exercise did a couple of things besides goofing off. First, it is a demonstration of how loops and partial strings (or substrings) can be used together for formatting output. Second, we showed how output could be slowed down for either an interesting effect or simply to give the user time to see what's happening.

Changing Strings to Numbers and Back Again

Now we're going to learn about changing strings to numbers and numbers to strings. If you're like me, when I first found out about these statements, I thought they were pretty useless. After all, if you want a string use a string variable, and if you want a number use a numeric variable. Simple enough, but again, once you understand their value, you wonder how you could do without them. To get started let's RUN the following program:

```

10 CALL CLEAR
20 FOR I = 1 TO 5
30 READ NAS(I)
40 NEXT I
50 FOR I = 1 TO 5
60 L = LEN (NAS(I))
70 X(I) = VAL(SEG$(NAS(I),L,1))
80 NEXT I

```

```

90 FOR I = 1 TO 5
100 PRINT "OVERTIME PAY= $"; X(I) * (1.5 * 7)
110 NEXT I
200 DATA SMITH 7, JONES 8, MCKNAP 6,
JOHNSON 2, KELLY 3

```

Using DATA which were originally in a string format, we were able to change a portion of that string array to a numeric array. By making such a conversion, we were able to use our mathematical operations on line 100 to figure out the overtime pay for someone receiving time and a half at seven dollars (\$7) an hour. Well, that's pretty interesting, but we don't have a list of who got what and the total overtime paid! Why don't you try it yourself. Change the program so that everyone's name appears with the amount of overtime each received and a total overtime paid. (Hint: You are looking for the substring `SEG$ (NA$(I), LEN (NA$(I)-2))` since you want to drop the number and space after each name.) When you get it, write me a letter to show me how you figured it out.

It always helps to do a few immediate exercises with a new command to get the right feel, so try these:

```

A$ = "123"
PRINT VAL(A$) + 11 <ENTER>
Q$ = "99.5"
PRINT VAL(Q$) * 7 <ENTER>
SALE$ = "44.95"
PRINT "ON SALE AT HALF PRICE ->$";
VAL(SALE$) / 2 <ENTER>
DO$ = "$103.88"
DN$ = "$18.34"
PRINT VAL (SEG$(DO$,2,4)) + VAL
(SEG$(DN$,2,3)) <ENTER>

```

Note: Since you may want to SAVE the above examples on tape or disk, all you have to do is to add line numbers and SAVE them as little programs.

From Numbers to Strings

All right, now let's go the other way. We saw why we might want to change strings to numbers, but we may also want to change numbers to strings. To make the conversion we use the `STR$` command. For example, look at the following program:

```
10 CALL CLEAR
20 PRINT "ENTER A NUMBER"
30 PRINT "WITH 5 DIGITS AFTER"
40 INPUT "THE DECIMAL POINT": A
50 A$=STR$(A)
60 PRINT
70 L = LEN(A$)
80 PRINT SEG$(A$,1,L-3)
```

As you can see, you have truncated the number to two decimal points. Using substrings we can vary the size of strings, and by converting numbers into string variables we can effectively use the same commands on numbers and numeric variables. Now let's do some in the Immediate mode to get the idea firmly into your mind. A little later we will do something very practical with these commands.

```
A = 5.00
A$ = STR$(A)
PRINT A$ <ENTER>
V = 2345
V$ = STR$(V)
PRINT V$ <ENTER>
BUCKS = 22.36
BUCKS$ = STR$(BUCKS)
PRINT "$"; SEG$(BUCKS$,4,2) <ENTER>
```

(Now the last example is a way to increase your bucks!) Remember these commands, and when you are dealing with decimal points you will often find them handy.

Tying Strings Together : Concatenation

We have seen how we can take a portion of a string and PRINT it to the screen. Now we will tie strings together. This is called **CONCATENATION** and is accomplished by using the "&" sign with strings. For example:

```
10 CALL CLEAR
20 INPUT "FIRST NAME ->": NF$
30 INPUT "LAST NAME ->": NL$
40 NA$ = NF$ & NL$
50 PRINT NA$
```

A little messy, huh? However, you can see how NF\$ and NL\$ were tied together into a single larger string. Now change line 40 to read

```
40 NA$=NF$ & " " & NL$
```

This time when you RUN the program, your name will turn out fine. Not only did we concatenate string variables, we also concatenated strings themselves. For example, it is perfectly all right to do the following:

```
PRINT "ONE" & "ONE" <ENTER>
```

Now there isn't much you can do with ONEONE, but we can see the principle of operation with concatenating strings.

One of the problems with the way your TI-99/4A formats numbers is that it drops 0's off the end. For example, try the following:

```
PRINT 19.84
PRINT 5.00
```

In dealing with dollars and cents, this can be a real pain in the neck, and it doesn't look very good. So, using concatenation and our VAL and STR\$ commands, let's see if we can fix that.

```

10 CALL CLEAR
20 PRINT "BE SURE TO INCLUDE ALL CENTS"
30 PRINT
40 INPUT "AMOUNT SPENT?-> $" :S
50 T = T + S
60 T$ = STR$(T)
70 T$ = "000" & T$
80 REM THIS IS TO INSURE THAT LEN(T$) IS
LONG ENOUGH
90 L = LEN(T$)
100 IF SEG$(T$, L - 1, 1) = "." THEN 300
110 IF SEG$(T$, L - 2, 1) < > "." THEN 400
120 PRINT
130 PRINT "YOU HAVE SPENT $"; SEG$(T$, 4, L)
140 PRINT "<ENTER> TO CONTINUE"
150 INPUT "OR 'Q' TO QUIT" : R$
160 IF R$ = "" THEN 10
170 IF R$ = "Q" THEN 190
180 GOTO 140
190 END
300 REM *****
310 REM ADD A ZERO
320 REM *****
330 T$ = T$ & "0"
340 GOTO 130
400 REM *****
410 REM ADD DECIMAL AND 2 ZEROS
420 REM *****
430 T$ = T$ & ".00"
440 GOTO 130

```

This may look pretty complicated, but let's break it down to see what has been done.

1. We entered numeric variables in line 40 and computed their sum in line 50.
2. The sum represented by T was then converted to a string variable T\$ in line 60.
3. In line 70 we "padded" T\$ with three 0's to give it a minimum length we will need in lines 100 and 110.

4. Line 100 computes the second from the last character in T\$. If that character is a decimal point (.) then we know it must be a figure that dropped off the last cent column (e.g., 5.4, 19.5, etc.). So we tack on a 0 in the subroutine in 300.
5. Line 110 computes the third from the last character and if it is not a decimal point (.), then we know it must have dropped all the cents completely — an even dollar number. So we tack on the decimal point and two 0's (.00) in the subroutine at 400.
6. Finally, in line 130 we print out our results but first drop the “padding” we added in line 70 using SEG\$. The statement SEG\$(T\$,4,L) computes the length of T\$ and subtracts three, the unwanted three 0's. The variable L was defined as the LENGTH of T\$.

All of this may seem a bit complicated just to get our 0's back, but actually the entire process was done in five lines (60 through 130 and the subroutines at 300 and 400). SAVE the program, and when you need those 0's in your output, just include those lines! (Be careful, though, this will not work with subtraction when you get below \$1! A better formula will be shown later on.)

Setting Up Data Entry

Now that we have a firm grip on numerous commands, it is time we begin thinking seriously about organizing our programs. The first thing we must do is to arrange our data entry in a manner that we ourselves and others can understand. This involves blocking elements of our program and deciding what variables and arrays we will be using. Also, when we enter data we want to make sure that we are entering the correct type of data. We have to set “traps” so that any input which is over a certain length or amount can be checked against our parameters. Let's look at a way to make our strings a certain length (no shorter or longer than a length we want). We've already discussed how to keep strings to a maximum length, so let's see how to keep them to a minimum as well. This latter process is referred to as padding.



```
10 CALL CLEAR
20 INPUT "YOUR COMPANY-->" : CM$
30 IF LEN(CM$) = 10 THEN 60
40 IF LEN(CM$) > 10 THEN 200
50 IF LEN(CM$) < 10 THEN 300
60 CALL CLEAR
70 PRINT "THE COMPUTER HAS DECIDED"
80 PRINT CM$; " SHOULD GIVE"
90 PRINT "YOU A RAISE!" , , ,
100 PRINT "<ENTER> TO CONTINUE OR"
110 INPUT "'Q' TO QUIT" : AN$
120 IF AN$ = "" THEN 10
130 IF AN$ < > "Q" THEN 100
140 END
200 REM *****
210 REM TOO LONG A STRING
220 REM *****
230 PRINT "ONLY USE 10 CHARACTERS"
240 PRINT "OR LESS PLEASE!"
250 GOTO 20
300 REM *****
310 REM PADDING
320 REM *****
330 CM$ = CM$ & "X"
340 GOTO 30
```


If YOUR COMPANY <CM\$> is less than 10 characters, you will see some Xs stuck on the end of the company name. These were put there to show you how padding works. Now change the X to " " (a space) in line 330 and see what happens. Go ahead. The second time you ran the program, if your company's name was less than 10 characters, there were a number of blank spaces after the company name. To remove the spaces, we would enter

```
53 L = LEN(CM$)
55 IF SEG$(CM$,L,1)=" " THEN 400
400 REM *****
410 REM REMOVE PADDING
420 REM *****
430 CM$ = SEG$(CM$,1,L-1)
440 GOTO 53
```

In addition we would change line 300 to read:

```
300 IF LEN(CM$) = 10 THEN 53
```

You're probably wondering, why bother putting the padding in if you have to remove it? Wouldn't it be easier simply to remove the subroutines at 300 and 400? It would be, but there are applications where you will want all strings in a given field to be a certain length. However, later on after the program has used the standardized length, you will want to remove the padding for printing it to the screen or printer. The above program simply shows how to do that - not optimal programming!

Setting Up Data Manipulation

Once you have organized your input, the next major step is performing computations with your data. There are essentially two kinds of data manipulation you will deal with:

1. **NUMERIC** - Manipulating numeric data with mathematical operations.
2. **STRING** - Manipulating strings with concatenation and substring commands.

Most of the string manipulations are for setting up input or output, so we will concentrate on manipulating numeric data. We will use a simple example that keeps track of three manipulations: (1) additions, (2) subtractions and (3) running balance. This will be our checkbook program we started earlier.

```
10 CALL CLEAR
20 REM #####
30 REM HEADER & INPUT BLOCK
40 REM #####
50 CB$ = "COMPUTER CHECKBOOK="
60 L = 14 - LEN (CB$) / 2
70 PRINT TAB(L); CB$
80 FOR V = 1 TO 4
90 PRINT
100 NEXT V
110 INPUT "CURRENT BALANCE=> $":BA
120 PRINT ,, "1. ENTER DEPOSITS" ,,,
130 PRINT "2. DEDUCT CHECKS" ,,,
140 PRINT "3. EXIT"
150 FOR V = 1 TO 7
160 PRINT
170 NEXT V
180 INPUT "CHOOSE BY NUMBER":A
190 REM ## TRAP IN LINES 200-210 ##
200 IF A > 3 THEN 180
210 IF A < 1 THEN 180
220 ON A GOTO 300,500,700
300 REM #####
310 REM DEPOSITS
320 REM #####
330 CALL CLEAR
340 INPUT "AMOUNT OF DEPOSIT $":DP
350 REM RUNNING BALANCE IN 360
360 BA = BA + DP
370 PRINT ,,
380 PRINT "YOU NOW HAVE $":BA ,,,
390 INPUT "MORE DEPOSITS? (Y/N)": AN$
400 IF AN$="Y" THEN 340
410 PRINT ,,
420 INPUT "DEDUCT CHECKS? (Y/N)": AN$
```

```

430 IF AN$ = "N" THEN 700
440 IF AN$ = "Y" THEN 500
450 CALL CLEAR
460 GOTO 390
500 REM #####
510 REM CHECKS
520 REM #####
530 CALL CLEAR
540 INPUT "AMOUNT OF CHECK $":CK
550 REM ## RUNNING BALANCE IN 560 ##
560 BA = BA - CK
570 PRINT , , ,
580 PRINT "YOU NOW HAVE $";BA
590 PRINT
600 PRINT "MORE CHECKS? (Y/N)"
610 INPUT "'Q' TO QUIT": AN$
620 IF AN$ = "Y" THEN 540
630 IF AN$ = "Q" THEN 700
640 PRINT
650 INPUT "ANY DEPOSITS(Y/N)": AD$
660 IF AD$ = "Y" THEN 300
670 GOTO 600
700 REM #####
710 REM TERMINATION BLOCK
720 REM #####
730 CALL CLEAR
740 FOR T = 1 TO (10 * 28)
750 PRINT "$";
760 NEXT T
770 PRINT "YOU NOW HAVE A"
780 PRINT "BALANCE OF $";BA

```

This program is designed to provide a simple illustration of how to block data manipulation. There are some problems with it in the output; we are not getting the 0's on the end of our balance! This is an output problem we will discuss in the following section; but before we continue, make sure you understand how we blocked the data manipulation. We used only three variables:

BA = BALANCE

CK = CHECK

DP = DEPOSIT

When we subtracted a check, we simply subtracted CK from BA, and when we entered a deposit, we added DP to BA. In this way we were able to keep a running balance and at the very end BA was the total of all deposits and checks. By keeping it simple and in blocks we were able to jump around and still keep everything straight.

Organizing Output

Let's go back to our program and repair it so that our balance will have the Ø's where they belong. This is essentially a problem of output because all of the computations have been done and they correctly tell us our balance, but it doesn't look right with the missing Ø's. However, we don't want to have to enter the lines for converting our balance into a string variable every time the running balance is printed. Therefore, we will put the subroutine for our conversion into a block. We can add a subroutine after the TERMINATION BLOCK starting at 800. We'll use that block to format our output.

```
800 REM #####
810 REM FORMAT OUTPUT
820 REM #####
830 BA = BA + .001
840 PLACE = 1
850 BA$ = STR$ (BA)
860 IF BA < .01 THEN 920
870 IF SEG$ (BA$,PLACE,1) < > "." THEN 900
880 BA$ = SEG$ (BA$,1,PLACE + 2)
890 RETURN
900 PLACE = PLACE + 1
910 GOTO 870
920 BA$ = "0.00"
930 GOTO 890
```

Now we'll change a few lines in our program so that when there is an output of our balance, it will jump to the subroutine between lines 800 and 930 and then RETURN to output BA\$. The following lines in our COMPUTER CHECKBOOK program should be changed and/or added:

```
375 GOSUB 800
380 PRINT "YOU NOW HAVE $";BA$ , ,
390 INPUT "MORE DEPOSITS? (Y/N)": AN$
580 PRINT "YOU NOW HAVE $"; BA$
780 PRINT "BALANCE OF $"; BA$
790 END
```

Now, if you put everything together properly, you should have a handy little program for working with your checkbook. Just to make sure you got everything, here's the complete program with all the subroutines and changes we made:

```
10 CALL CLEAR
20 REM #####
30 REM HEADER & INPUT BLOCK
40 REM #####
50 CB$ = "COMPUTER CHECKBOOK="
60 L = 14 - LEN (CB$) / 2
70 PRINT TAB(L); CB$
80 FOR V = 1 TO 4
90 PRINT
100 NEXT V
110 INPUT "CURRENT BALANCE=> $":BA
120 PRINT "1. ENTER DEPOSITS"
130 PRINT "2. DEDUCT CHECKS"
140 PRINT "3. EXIT"
150 FOR V = 1 TO 7
160 PRINT
170 NEXT V
180 INPUT " CHOOSE BY NUMBER :A
190 REM ## TRAP IN LINES 200-210 ##
200 IF A > 3 THEN 180
210 IF A < 1 THEN 180
220 ON A GOTO 300,500,700
300 REM #####
```

```

310 REM DEPOSITS
320 REM #####
330 CALL CLEAR
340 INPUT "AMOUNT OF DEPOSIT $":DP
350 REM RUNNING BALANCE IN 360
360 BA = BA + DP
370 PRINT
375 GOSUB 800
380 PRINT "YOU NOW HAVE $"; BA$
390 INPUT "MORE DEPOSITS? (Y/N) : AN$
400 IF AN$ = "Y" THEN 340
410 PRINT
420 INPUT "DEDUCT CHECKS? (Y/N) : AN$
430 IF AN$ = "N" THEN 700
440 IF AN$ = "Y" THEN 500
450 CALL CLEAR
460 GOTO 390
500 REM #####
510 REM CHECKS
520 REM #####
530 CALL CLEAR
540 INPUT "AMOUNT OF CHECK $":CK
550 REM ## RUNNING BALANCE IN 560 ##
560 BA = BA - CK
570 PRINT , , ,
575 GOSUB 800
580 PRINT "YOU NOW HAVE $"; BA$
590 PRINT
600 PRINT "MORE CHECKS? (Y/N)"
610 INPUT "'Q' TO QUIT": AN$
620 IF AN$ = "Y" THEN 540
630 IF AN$ = "Q" THEN 700
640 PRINT
650 INPUT "ANY DEPOSITS? (Y/N)": AD$
660 IF AD$ = "Y" THEN 300
670 GOTO 600
700 REM #####
710 REM TERMINATION BLOCK
720 REM #####
730 CALL CLEAR
740 FOR T = 1 TO (10 * 28)

```

```

750 PRINT "$";
760 NEXT T
770 PRINT "YOU NOW HAVE A"
780 PRINT "BALANCE OF $"; BA$
790 END
800 REM #####
810 REM FORMAT OUTPUT
820 REM #####
830 BA = BA + .001
840 PLACE = 1
850 BA$ = STR$ (BA)
860 IF BA < .01 THEN 920
870 IF SEG$ (BA$,PLACE,1) < > "." THEN 900
880 BA$ = SEG$ (BA$,1,PLACE + 2)
890 RETURN
900 PLACE = PLACE + 1
910 GOTO 870
920 BA$ = "0.00"
930 GOTO 890

```

Scroll Control!

One of the big problems in output occurs when you have long lists that will scroll right off the screen. For example, the output of the following program will kick the output right out the top of the screen:

```

10 CALL CLEAR
20 FOR I = 1 TO 100
30 PRINT I
40 NEXT I

```

Instead of numbers, suppose you have a list of names you have sorted or some other output you wanted to see before it zipped off the top of the screen. Depending on the desired output, screen format and so forth there are several different ways to control the scroll. Consider the following:

```

10 CALL CLEAR
20 FOR S = 1 TO 100
30 IF S = 21 THEN 100
40 IF S = 41 THEN 100
50 IF S = 61 THEN 100
60 IF S = 81 THEN 100
70 PRINT S
80 NEXT S
90 END
100 PRINT ,,
110 INPUT "<ENTER> TO CONTINUE": AN$
120 CALL CLEAR
130 GOTO 70

```



REMEMBER!! You, not the computer, are in **CONTROL!** You can have your output any way you want it. To use more of the screen, you could have the output sectioned to different parts of the screen. For example:

```

10 CALL CLEAR
20 FOR I = 1 TO 20
30 PRINT I; TAB(5); I+20; TAB(10); I+40; TAB(15);
  I+60; TAB(20); I+80
40 NEXT I

```


You get the idea. Format your output in a manner that best uses the screen and your needs and get that scroll under control!

More PRINT Formatting

Up to now, we've used the comma (,), semi-colon (;), TAB and PRINT statement in formatting out PRINTed strings and variables. Now we will see a very handy TI BASIC way of formatting PRINT output with a lot less effort. We will use the colon (:). Basically, in a line with a PRINT statement, the colon serves as a "linefeed." For example, compare the following two programs:

```
10 REM *****
20 REM METHOD ONE
30 REM *****
```

```
10 REM *****
20 REM METHOD TWO
30 REM *****
40 CALL CLEAR
50 PRINT "ONE" : "TWO"
```

Both programs did exactly the same thing, except the second method took only one line (line 50) while the first method took two (lines 50-60). Whenever the colon is encountered, the computer simply linefeeds. The colon can also be used in vertical scrolling following strings and numbers. For example, the following will put HERE in the middle of your screen:

```
10 CALL CLEAR
20 PRINT "HERE" :::::
```

Now, just for fun, let's write a program that uses colons and scrolling to make a "Computer Commercial."

```
10 REM *****
20 REM COMPUTER COMMERCIAL
30 REM *****
```

```

40 CALL CLEAR
50 A$ = "EAT AT JOE'S CAFE"
60 GOSUB 500
70 A$ = "THE FOOD IS ALMOST"
80 GOSUB 500
90 A$ = "GOOD ENOUGH TO EAT"
100 GOSUB 500
110 FOR PAUSE = 1 TO 1000
120 NEXT PAUSE
130 GOTO 50
500 REM *****
510 REM FORMAT OUTPUT
520 REM *****
530 L = LEN(A$)
540 PRINT TAB(14-L/2); A$ :;
550 FOR HOLD = 1 TO 200
560 NEXT HOLD
570 RETURN

```

Now that will keep on running until you press FCTN-4 [CLEAR]. Whenever you want to stop a program the CLEAR function key will do it. You will get a message that says

* BREAKPOINT AT 120

or whatever line number you "broke into" the program. You can also "break" a program with the QUIT function, but that erases your program from memory. (In desperation, you can turn your computer off!)

SUMMARY

The formatting of programs makes the difference between a useful and a not-so-useful application of your computer. The extent to which your program is well organized and clear, the better the chances are for simple, yet effective, programming. Formatting is more than an exercise in making your input/output fancy or interesting. It is a matter of communication between your TI-99/4A and you! After all, if you can't make heads or tails of what your computer has computed, the best calculations in the world are worthless

In the same way that it is important to have your computer tell you what you want, it is also important to write your programs so that you and others can understand what is happening. By using "blocks" it is easier to organize and later understand exactly what each part of your program does. Obviously it is possible to write programs sequentially so that each command and subroutine is in an ascending order of line numbers, but to do so means that you will have to repeat simple and/or complex operations which could be better handled as subroutines. It will also be considerably more difficult to locate bugs and make the appropriate changes. In other words, by using a structured approach to programming you make it simpler, not more difficult.

Finally, you should begin to see why there are commands for substrings and all the fuss about TABs. These are handy tools for organizing the various parts in a manner which gives you complete control over your computer's output. What may at first seem like a petty, even silly command in TI-99/4A BASIC, will, after a useful application, be appreciated as an excellent tool. Therefore, as we delve deeper into your computer, look at the variety of commands as mechanisms of more efficient and ultimately simpler control, and not a complex "gobbleygook" of "computerese" for geniuses. After all, if you have come this far you should realize that what you know now looked like the work of "computer whizzes" when you first began.

CHAPTER 6

Some Advanced Topics (But Not Too Difficult Once You Get To Know Them)

Introduction

The topics of this chapter are more “code like” and contain the kinds of commands that look frightening. At least that’s how I interpreted them when I first saw them. Many of the functions can be done with commands we already know, but others cannot. Still others, as we will see, can be accomplished better using these new commands. Like so much else you have seen in this book, what at first may appear to be *impossible* is really quite simple once you get the idea. More importantly, by playing with the commands, you can quickly learn their uses.

The first thing we will learn about is the ASCII code. ASCII (pronounced ASS-KEY) stands for the AMERICAN STANDARD CODE for INFORMATION INTERCHANGE. Essentially, this is a set of numbers that have been standardized to represent certain characters. In TI-99/4A BASIC the CHR\$ (character string) command ties into ASCII and can be used to directly output ASCII. As we will see, the CHR\$ command is very useful for outputting special characters; however, there are six “keyboards” on your TI-99/4A (Ø-5) that can be linked through the CALL KEY command. We will look at the different keyboards in a separate section of this chapter.

The next commands have to do with accessing subroutines in your computer’s memory. These use CALL. We have already been using CALL CLEAR to call up the subroutine in your computer that clears the screen. Some of the ones we will discuss will allow you to do a lot more with screen formatting and other tricks you cannot do using standard BASIC program

commands. A number of the CALL commands will be left until the next chapter when we discuss computer graphics, but by then you will be an old hand with CALL.

The ASCII Code and CHR\$ Functions

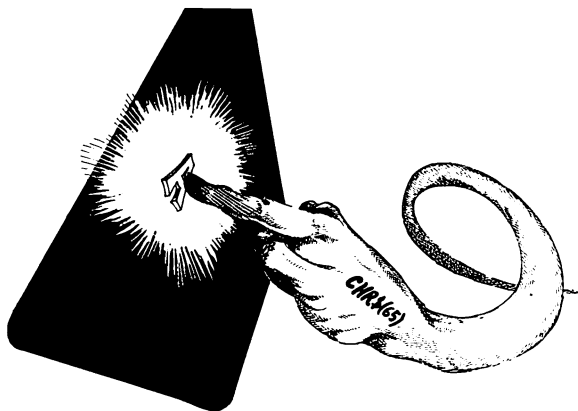
A way to access any characters we want, including control characters, is to use CHR\$ commands and the ASCII code. Whenever we want to access a character, we simply enter the CHR\$ and the decimal value of the character we want. For example enter the following:

```
PRINT CHR$(65) <ENTER>
```

You got an A. That's simple enough and not too interesting. On the other hand, try the following little program, and I'll bet you couldn't do it without using the CHR\$ function:

```
10 CALL CLEAR
20 REM 34 IS THE ASCII VALUE FOR QUOTE MARKS
30 QU$ = CHR$(34)
40 PRINT "HIT <ENTER> TO CONTINUE OR"
50 PRINT "PRESS "; QU$ ; "Q" ; QU$ ;
  " TO QUIT " : : : : :
60 INPUT "=CHOOSE=" : AN$
70 IF AN$ = "" THEN 10
80 IF AN$ = "Q" THEN 100
90 GOTO 60
100 END
```

RUN the program and look carefully. Note the quotes around the Q. If we tried to PRINT a quote mark, the computer would think it got a command to begin printing a string. However, by defining QU\$ as CHR\$(34) we were able to slip in the quote marks and not confuse the output! (Just for fun, see if you can do that without using the CHR\$ command.) To see what different characters you have available, RUN the following program:



```
10 CALL CLEAR
20 FOR I = 32 TO 127
30 PRINT CHR$(I); CHR$(32);
40 NEXT I
```

Voila! There you have all of your symbols for the standard keyboard. We used `CHR$(32)` - a SPACE - to separate our different characters rather than a pair of quotation marks (" "). Also, the first character we printed was a space, so the first character to appear was an explanation point (!) that seemed to be indented. We also got lower case characters, even if we had the ALPHA LOCK key pressed down. Depending on whether the lower case letters are "on" or "off," `CHR$(I)`'s will output different symbols. Now, to watch funny things happen to your screen, RUN the following program.

```
10 CALL CLEAR
20 FOR I = 0 TO 31
30 PRINT CHR$(I) ; CHR$(32)
40 NEXT I
```

Not much happened since in that range of ASCII (from 0 to 31) you ran through the function codes. On your TI-99/4A standard ASCII codes are only in the range from 32 to 127. However, `CHR$(30)` printed your cursor. Let's see what we can do with that. Try the following program:

```
10 CALL CLEAR
20 FOR I = 1 TO 28
30 PRINT CHR$(30);
```

```

40 NEXT I
50 FOR I = 1 TO 20
60 PRINT CHR$(30); TAB(28);CHR$(30);
70 NEXT I
80 FOR I = 1 TO 28
90 PRINT CHR$(30);
100 NEXT I
1000 FOR PAUSE = 1 TO 1000
1010 NEXT PAUSE

```

On the last program, you will get an idea of the use of CHR\$ commands with graphics. The box was created using CHR\$(30) , a block. If we had finer resolution, we could do more with graphics. Well, as we will see in the next chapter, we do have a lot more control with graphics. How do you think they made the TI logo that appears on your screen when you start up? It sure wasn't done with CHR\$(30)!

The following program is a handy little device for printing out all of the CHR\$ values to screen. Save it to tape or disk to use as a handy reference guide to look up CHR\$ values and symbols.

CHR\$ MAP

```

10 CALL CLEAR
20 FOR CH = 32 TO 127
30 CH$ = STR$(CH)
40 PRINT "CHR$(";CH$;")=";CHR$(CH),
50 N=N+1
60 IF N = 34 THEN 80
70 NEXT CH
80 PRINT :::
90 INPUT "PRESS <ENTER> TO CONTINUE "
: CONTINUE$
100 PRINT :::
110 N=0
120 IF CH > 126 THEN 140
130 GOTO 70
140 PRINT "PRESS <ENTER> TO CONTINUE"
150 PRINT "OR";CHR$(34);"Q";CHR$(34);
"TO QUIT";
160 INPUT AN$

```

```

170 IF AN$ = "Q" THEN 200
180 IF AN$ = "" THEN 10 ELSE 140
200 END

```

The program, CHR\$ MAP, can be used as a handy reference for you to look up the CHR\$ values of different symbols. However, you may not want to run through a lot of screens to look up a CHR\$ value. You may simply want to find a single one quickly. To do that, we will need a new BASIC statement - ASC. This command does the *opposite* of CHR\$. If you enter a string, it returns the CHR\$ value of that string. For example, if you entered

```
PRINT ASC("A")
```

You'd get

```
65
```

The following program will find any CHR\$ value for you:

```

10 CALL CLEAR
20 INPUT "CHARACTER=> " : C$
30 PRINT C$;" IS CHR$(";" ; ASC(C$); ")"
40 PRINT :::
50 INPUT "ANOTHER (Y/N) ": CHOICES$
60 IF CHOICES$ = "Y" THEN 10
70 IF CHOICES$ = "N" THEN 100 ELSE 50
100 END

```

With the above programs, you ought to be able to find just about any CHR\$ you want. But aren't there six keyboards on the TI-99/4A? What about the other five? Well, for those, we need new commands, so read on.

For a Good Time — CALL

In this section, we will begin looking at the various CALL sub-routines available on your TI. We'll examine those CALLs used for the keyboard, character set and sound. In the next chapter, we'll look at the CALLs for graphics and the joystick.

CALL KEY

Before we look at all the keyboards we can access with CALL KEY, let's see how we can use it in a program. Up to now, whenever we have come to a place in the program where we want to give the user an option, we have used INPUT. The user then presses a key and then presses <ENTER> or simply <ENTER>. In some cases it would be nice if we offered the user a choice and as soon as a key was pressed, the program would branch off in the desired direction without having to press <ENTER>. Using CALL KEY, we can do this. The following little program shows how:

```
10 CALL CLEAR
20 RESTORE
30 FOR I = 1 TO 3
40 READ A$
50 PRINT A$::
60 NEXT I
100 REM *****
110 REM GET A KEYPRESS
120 REM *****
130 PRINT ::: "CHOOSE BY NUMBER"
140 CALL KEY (0,K,C)
150 IF K = 49 THEN 200
160 IF K = 50 THEN 300
170 IF K = 51 THEN 400 ELSE 130
200 REM *****
210 REM CHOICE ONE
220 REM *****
230 PRINT ::: "THE FIRST CHOICE"
240 PRINT ::: "HIT ANY KEY"
250 CALL KEY (0,K,C)
260 IF C = 0 THEN 250
270 GOTO 10
300 REM *****
310 REM CHOICE TWO
320 REM *****
330 PRINT ::: "THE SECOND CHOICE"
340 PRINT ::: "HIT ANY KEY"
350 CALL KEY(0,K,C)
```

```

360 IF C=0 THEN 350 ELSE 10
400 REM *****
410 REM EXIT CHOICE
420 REM *****
430 PRINT ::: "GOODBYE"
440 END
500 REM *****
510 REM DATA
520 REM *****
530 DATA "1. CHOOSE ME", "2. NO,
CHOOSE ME", "3. EXIT"

```

In the above program, we used CALL KEY in three slightly different manners, but they all resulted in our having to press only one key to make the desired branch. First, let's look at the CALL KEY format. There are three variables in CALL KEY.

**CALL KEY(KEYBOARD NUMBER, ASCII VALUE
OF KEY PRESSED, CONDITION)**

Let's take it apart piece by piece:

1. **KEYBOARD NUMBER.** Remember, you have six keyboards numbered from 0 to 5. The default keyboard is 0, but it must be specified when using CALL KEY. You may also specify 1-5 as this first value.
2. **ASCII VALUE OF KEY PRESSED.** In this variable, K, the ASCII value of the last key pressed is stored. If, for example, you press "A" then K=65. (Remember CHR\$(65) equals A.)
3. **CONDITION.** The condition or status of the keyboard is one of the following:
 - a. 0 = No key has been pressed.
 - b. 1 = A new key was pressed since the last CALL KEY command.
 - c. -1=The same key was pressed as the last time the CALL KEY command was accessed. The value of C is always 0, 1 or -1.

OK, now we can see how our program worked. In lines 130-160, the ASCII values of 1, 2 and 3 (49, 50 and 51) were examined in the variable K. If none of those values were in K, then the program branched back to the CALL KEY statement in line 130. In line 250, the program examined the status of the C variable to see whether any key had been pressed. As soon as a key was pressed, the program branched back to the beginning. The same thing was done in lines 340-360 using a slightly different format. The CALL KEY is a very versatile command, and when you want a single keypress for a program branch, be sure to use it. (In fact, you can go over some of the programs we've already written and substitute CALL KEY for INPUT if you want.)

Now we can look at the six different keyboards in your TI. We will not be using the different keyboards here except to show you how they encode different keys differently. We will continue to use keyboard 0 (or KEY UNIT 0). To see what we're doing, go to the Appendix of your User's Reference Manual that comes with your computer. There you will see different keyboards or "Keyboard Maps." These maps show what codes are returned depending on what keyboard you're using. More advanced applications than those covered in this book use the different keyboards, but since you should know about them, the following little program will take you on a tour of your different keyboards. You will be able to see that, depending on the keyboard you select, different CHR\$ values will be returned.

KEYBOARD

```
10 CALL CLEAR
20 PRINT "WHICH KEYBOARD{0-5}"
30 INPUT "ENTER 6 TO END" : KU
40 IF KU = 6 THEN 110
50 CALL KEY [KU,K,C]
60 N=K
70 IF C = 0 THEN 50
80 PRINT "CHR$[\";N;\"]"
90 IF N = 1 THEN 10
100 GOTO 50
110 END
```

Now go ahead and RUN the program. The first time through, choose 0, the keyboard you are familiar with. When you press A you get CHR\$(65). Now press FCTN-7 (AID) to get CHR\$(1) and a return to the beginning of the program. This time choose 1 but do not press A right away. This time the keys produce different CHR\$ values! The value of "A" is 1 instead of 65; so when you press A you will be sent back to the beginning. Before you do that, press some keys on the right side of the keyboard. Nothing happens! That's because Keyboard 1 doesn't access the right side. Now try the rest of the keyboards to see what you get. Remember, if you get locked up, press FCTN-4 (CLEAR) to get out.

Calls With Text Formatting

Imagine your screen as a large checkerboard and each text character as a single checker. This checkerboard has 32 columns and 24 rows and you can place your checkers anywhere you want simply by stating 1) the row number and 2) the column number. The upper lefthand corner of your screen is Row 1, Column 1 and the bottom right is Row 24, Column 32. The middle of the screen would be Row 12, Column 16. OK, now that would make formatting a lot easier, but there's one catch: All the numbers have to be entered as ASCII values. But that's not too difficult since we can either look up the ASCII code or have our ASC statement find it for us.

CALL HCHAR and CALL VCHAR

To place text on our screen as described above, we can use either the CALL HCHAR or CALL VCHAR. Using these commands we can also repeat the horizontal or vertical placement of each character. The format for each is:

CALL VCHAR (Row, Column, Code, Number of Vertical Repetitions)

CALL HCHAR (Row, Column, Code, Number of Horizontal Repetitions)

The number of repetitions is optional, and for the most part we will not use it; however, there may be times when it will come in handy so we'll provide an example. First, let's just place a character on the screen. Enter the following: (Before you press <ENTER> see if you can guess what will appear on the screen.)

```
CALL HCHAR (12,16,65) <ENTER>
CALL VCHAR (12,16,65) <ENTER>
```

They both did the same thing - printed the letter "A" right in the middle of the screen. Now, let's take a look at the repetition:

```
10 CALL CLEAR
20 CALL HCHAR (1,1,72,32)
30 CALL VCHAR (2,3,86,23)
40 CALL KEY (0,K,C)
50 IF C = 0 THEN 40 ELSE 60
60 END
```

As you will see when you run the program, you get a horizontal row of Hs and a vertical column of Vs. The CALL KEY held everything in place while we had a chance to look at it.

Now let's do something a little more useful with our programming skills so that the computer can do all the figuring instead of having us do it! I can't remember all the ASCII values, and it's a pain in the neck looking them up every time I want to use HCHAR and VCHAR. So, let's write a program that will stick a letter anywhere we want on the screen, and we simply enter the Row and Column where we want it.

```
10 CALL CLEAR
20 INPUT "WHICH CHARACTER DO YOU
WANT?" :C$
30 ASCII = ASC(N$)
40 INPUT "WHICH ROW WOULD YOU LIKE?" : ROW
50 INPUT "WHICH COLUMN?" : COLUMN
60 CALL HCHAR (ROW,COLUMN,ASCII)
70 PRINT :: "CONTINUE(Y/N)"
```

```

80 CALL KEY(0,K,C)
90 IF C <> 0 THEN 100 ELSE 80
100 G$ = CHR$(K)
110 IF G$ = "Y" THEN 10
120 IF G$ = "N" THEN 130 ELSE 80
130 END

```

Now look at the above program carefully. We did not put in a single ASCII code and we did not have to enter any ASCII code when we ran the program. We let the computer figure it out for us. All ASCII was stored in variables and converted either from or to ASCII by our program statements. This represents ideal programming in that 1) the programmer doesn't have to look up the code and 2) the user doesn't have to look up the code. The computer does all the work!

The next step is to enter multiple character strings. After all, a program wouldn't be very useful if all we could enter were single digit strings. To enter multiple character strings, we will have to have our program examine each character in our strings, convert the character to ASCII code and print it where we want it to go. This may sound complicated, but using our LEN and SEG\$ statements in a loop, it is not too difficult at all. Here's how:

```

10 CALL CLEAR
20 INPUT "ENTER MESSAGE" : M$
100 REM *****
110 REM TRANSLATE MESSAGE INTO ASCII
120 REM *****
130 CALL CLEAR
140 FOR I = 1 TO LEN(M$)
150 A$ = SEG$(M$,I,1)
160 ASCII = ASC(A$)
200 REM *****
210 REM PRINT OUT CHARACTERS
220 REM *****
230 CALL HCHAR (10,I+5,ASCII)
240 NEXT I

```

Let's take the program one step at a time to make sure you understand how it works.

Step 1. We enter our message into the variable M\$.

Step 2. We set up a loop the LENgth of M\$ so that we can examine every character in the string.

Step 3. Each character, one by one, is entered into the variable A\$ using SEG\$ to pick up single characters in M\$.

Step 4. We translate A\$ into ASCII code in the variable (what else?) ASCII.

Step 5. In line 230 we print out our characters one by one in Row 10 and a column offset by our loop variable "I" plus 5.

The program not only prints out visible characters, but it includes spaces as well since a space is simply read as ASCII value 32. Using our new information, let's make a slick menu program. We'll remake the "box" with CHR\$(30) as our border, but we'll do it in a different way.

```
10 CALL CLEAR
20 REM *****
30 REM MAKE A BORDER
40 REM *****
50 CALL VCHAR(1,2,30,31)
70 CALL VCHAR(2,32,30,22)
80 CALL HCHAR(23,2,30,31)
100 REM *****
110 REM DEFINE STRINGS
120 REM *****
130 C$(1) = "1. CHOICE ONE"
140 C$(2) = "2. CHOICE TWO"
150 C$(3) = "3. EXIT"
160 C$(4) = "CHOOSE BY NUMBER"
200 REM *****
210 REM EXAMINE ALL STRINGS
```

```

220 REM *****
230 FOR X = 1 TO 4
240 FOR I = 1 TO LEN(C$(X))
250 GOSUB 500
260 R = (X^2) + 3
270 CALL HCHAR(R,I+5,ASCII)
280 NEXT I
290 NEXT X
300 REM *****
310 REM EVALUATE CHOICE
320 REM *****
330 CALL KEY(0,K,C)
340 IF C=0 THEN 330
350 V$ = CHR$(K)
360 IF V$ = "1" THEN 600
370 IF V$ = "2" THEN 700
380 IF V$ = "3" THEN 800 ELSE 330
500 REM *****
510 REM TRANSLATION SUBROUTINE
520 REM *****
530 A$ = SEG$(C$(X),I,1)
540 ASCII = ASC(A$)
550 RETURN
600 REM *****
610 REM CHOICE ONE
620 REM *****
630 CALL CLEAR
640 PRINT "YOU CHOSE THE FIRST!"
650 PRINT :: "[HIT ANY KEY]"
660 CALL KEY(0,K,C)
670 IF C<> 0 THEN 10 ELSE 660
700 REM *****
710 REM CHOICE TWO
720 REM *****
730 CALL CLEAR
740 PRINT "YOU CHOSE THE SECOND"
750 PRINT :: "[HIT ANY KEY]"
760 CALL KEY(0,K,C)
770 IF C<> 0 THEN 10 ELSE 760
800 REM ****
810 REM EXIT

```



```

820 REM ****
830 CALL CLEAR
840 PRINT :::"MENU EXIT"
850 END

```

Now that will look professional! You're on your way to making the ultimate, easy-to-use menu program. Next we'll add some sound and you can have fanfare with your menu!

Missiles and Music : CALL SOUND

If you've ever wondered how arcade games make the sound of missile firing and exploding or how music is played on a computer, you are about to find out. Each SOUND has three values:

Variables	Values
1. Duration	1 to 4250 -1 to -4250
2. Frequency	(Tone) 110 to 44733 (Noise) -1 to -8
3. Volume	0 (loudest) to 30

The basic format for CALL SOUND is:

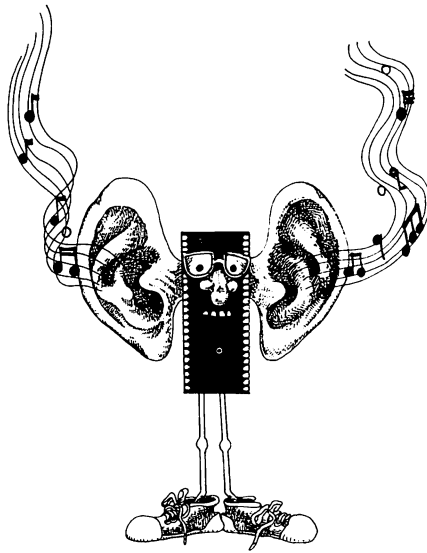
CALL SOUND (DURATION, FREQUENCY,
VOLUME)

Okay, for some practice try the following, but *FIRST* turn up the volume on your TV or monitor:

```

CALL SOUND (100,110,3)
CALL SOUND (200,220,0)
CALL SOUND (250,1000,2)
CALL SOUND (1000,2000,4)
CALL SOUND (50,300,9)
CALL SOUND (100,-5,1)
CALL SOUND (500,-2,2)

```



Did you notice the difference between the first five and the last two sounds? When negative numbers from -8 to -1 are used as the Frequency value, the computer produces “noise” instead of “tones.” These noises are good for game sounds. To find out more about the sounds we can make, let’s write a program that will tell us which frequencies produce which tones. (We’ll make a different one for producing noises.)

```
10 REM *****
20 REM SOUND FINDER 1
30 REM *****
40 CALL CLEAR
50 CALL KEY(0,K,C)
60 IF C=0 THEN 50
70 FREQUENCY = K + 110
80 PRINT "FREQUENCY="; FREQUENCY
90 CALL SOUND(150,FREQUENCY,2)
100 GOTO 50
```

When you RUN this program, note the different values of FREQUENCY that produce different sounds. Using this program, you can jot down the tones you want for later use. Now let’s do something slightly different for making noise.

```

10 REM *****
20 REM NOISE MAKER 1
30 REM *****
40 CALL CLEAR
50 INPUT "NOISE (-1 TO -8) " : NOISE
60 INPUT "DURATION (1-4250) " : DURATION
70 INPUT "VOLUME (0 TO 30) " : VOLUME
80 CALL SOUND(DURATION,NOISE,VOLUME)
90 PRINT :: "THAT RACKET WAS MADE BY: "
100 PRINT "DURATION "; DURATION
110 PRINT "FREQUENCY "; NOISE
120 PRINT "VOLUME "; VOLUME
130 PRINT :: "{HIT ANY KEY}"
140 CALL KEY(0,K,C)
150 IF C = 0 THEN 140 ELSE 10

```

Using the NOISE MAKER 1 program you can find, by trial and error, the various sounds for a game program, or some similar program requiring specific sounds. You will soon find that you really can't quite get that atomic explosion sound you've been wanting. (The sound needed to indicate your checking account is overdrawn.) Well, let's go back to the drawing board and see what else we can get out of CALL SOUND.

With the TI sound routine it is possible to have up to four sounds simultaneously. The duration value remains constant, but it is possible to have a maximum of three tones and one noise all at the same time. The following format is used:

```
CALL SOUND (DURATION, FREQ1, VOL1,
            FREQ2, VOL2, FREQ3, VOL3, FREQ4, VOL4)
```

That gives us a lot more possibilities, but even if we only use two different frequencies, only one can be noise. Therefore, we have to make addition noise by combining tones with noise. The following program allows you to explore the maximum possibilities with your TI sound generations system:

```

10 REM *****
20 REM SUPER SOUND

```

```

30 REM *****
40 CALL CLEAR
50 INPUT "DURATION": D
60 INPUT "NOISE [-8 TO -1]": N
70 INPUT "NOISE VOLUME [0-30]": NV
80 INPUT "TONE1 [110-44733]": T1
90 INPUT "TONE1 VOLUME [0-30]": T1V
100 INPUT "TONE2 [110-44733]": T2
110 INPUT "TONE2 VOLUME [0-30]": T2V
120 INPUT "TONE3 [110-44733]": T3
130 INPUT "TONE3 VOLUME [0-30]": T3V
140 CALL SOUND (D, N, NV, T1, T1V, T2, T2V, T3, T3V)
200 REM *****
210 REM SHOW VALUES
220 REM *****
230 PRINT :: "DURATION"; D

```



```

240 PRINT : "NOISE"; N; "NOISE VOLUME"; NV
250 PRINT : "TONE1"; T1; "VOLUME";T1V
260 PRINT : "TONE2"; T2; "VOLUME";T2V
270 PRINT : "TONE3"; T3; "VOLUME";T3V
300 REM *****
310 REM CONTINUE OR EXIT
320 REM *****
330 PRINT :: "[HIT ANY KEY - OR Q TO QUIT]"
340 CALL KEY (0,K,C)
350 IF C = 0 THEN 340
360 H$ = CHR$(K)
370 IF H$ < > "Q" THEN 10
380 END

```

Now crank the program up and drive your neighbors nuts!

So far all we've seen is how to INPUT values and determine what values produce various sounds. Such programs are useful in that we can determine sound values, but we need a way to make sounds using several values quickly. To see one way of doing this we will first introduce a new function to randomly generate values, RANDOMIZE and RND. The RANDOMIZE function "seeds" the random number generator, and the RND function generates random numbers. However, we need another new statement to make it useful, the INT statement. All INT does is to transform floating point numbers into integer (whole) numbers. For example, INT(123.45) will change 123.45 to the integer number 123. To generate random numbers, we use the format

$$\text{INT}(N * \text{RND}) + 1$$

The variable N is equal to the maximum number we can randomly generate. To generate a range of random numbers, use the format

$$\text{INT}[(\text{NN} - N + 1) * \text{RND}] + N$$

The variable NN is the high number in the range and N is the low number. The following program generates random numbers between 220 and 440.

```

10 CALL CLEAR
20 RANDOMIZE 222
30 N = 220
40 NN = 440
50 FOR I = 1 TO 40
60 R = INT[(NN - N + 1) * RND] + N
70 PRINT R,
80 NEXT I

```

Now that shows we can generate different values without having to INPUT anything. The next step is to use those values in a program that will generate tones. The following program does that, and it just so happens that the values from 220 to 440 will generate tones above and below middle C. (See the Musical Tone Frequencies in the appendix of your *TI User's Reference Guide*.) With this information we will make a SPACE ALIEN BAND program.

```

10 REM *****
20 REM SPACE ALIEN BAND
30 REM *****
40 CALL CLEAR
50 RANDOMIZE 222
60 N = 220
70 NN = 440
80 FOR I = 1 TO 100
90 T = INT[(NN-N+1) * RND] + N
100 CALL SOUND (100,T,2)
110 NEXT I

```

Depending on your luck, that generated sounds anywhere from Venus to Pluto. However, we would like to compose our own music that leaves nothing to chance. To do this, we will use number arrays and DATA statements. First we will load our array with the DATA element and then we will run the array values through our CALL SOUND statements to produce our tune.

```

10 REM *****
20 REM MUSIC MAKER
30 REM *****

```

```

40 CALL CLEAR
50 DIM T(46)
60 FOR I = 1 TO 46
70 READ N
80 T(I) = N
90 NEXT I
100 REM *****
110 REM PLAY THE SONG
120 REM *****
130 FOR J = 1 TO 2
140 FOR I = 1 TO 46
150 CALL SOUND(230, T(I),2)
160 NEXT I
170 NEXT J
200 REM *****
210 REM VALUES FOR NOTES
220 REM *****
230 DATA 392, 233, 294, 311, 294, 233, 392, 233,
294, 311, 392
240 DATA 311, 294, 233
250 DATA 262, 311, 392, 440, 466, 440, 392,
311, 392, 233
260 DATA 294, 311, 392, 311, 294, 233
270 DATA 220, 233, 262, 277, 294, 262, 233, 220,
392, 233, 294
280 DATA 311, 392, 311, 294, 233

```

SUMMARY

This chapter covered some advanced topics, but as we saw (I hope) they really were not too difficult. You should now have a good deal more control over your computer's input and output with the use of CHR\$. You should also be able to translate characters to and from code with ASC. The ASCII code is not difficult to handle once you're used to it, and it certainly is not mysterious.

With the CALL functions we examined, you can now deal more effectively with the keyboard and the positioning of characters. The CALL KEY function allows you to get a single key value and branch your programs more quickly than with INPUT. You also found that there are six keyboards available to you if you need them with CALL KEY. Likewise, the CALL HCHAR and CALL VCHAR commands allow you to place characters anywhere you want on the screen without having to scroll up from the bottom for more professional screen presentations.

Finally, we saw that with CALL SOUND we can add both musical notes and noise to our programs. On the one hand we can make special sound effects for our programs. Knowing this, we can use all kinds of noises to simulate arcade sounds. On the other hand, we can make music with CALL SOUND to play tunes or prompt choices or whatever we feel like. In the next chapter, dealing with graphics, we will see how to mix SOUND and animation to produce some very exciting programs.



CHAPTER 7

Using Graphics

Introduction

One of the nicest features of the TI-99/4A is its graphics capability. Basically, there are two kinds of graphics: (1) Screen Graphics and (2) Bit Graphics. Screen graphics are something like text except that we use a lot more color. We will use text as symbols for other than conventional meanings. This will allow us to make "text graphics." While we're at it, we will also see how to use the joysticks within a program and some basic animation.

Bit graphics are wholly different from screen graphics and they are a good deal more difficult to use; however, bit graphics give you an incredible amount of flexibility and power in creating figures in fine detail. Once you become adept at using bit graphics, there is far more you can do to create graphics on your TI. To make bit graphics simpler, there is a program for translating your drawings into the correct hexadecimal code.

SCREEN GRAPHICS

Coloring Your Graphics

If all of the graphics we did were in the shades we've seen so far, it would be pretty dull. We will now begin using the full range of colors on the TI-99/4A. If you do not have a color TV or monitor, the colors will appear as different shades of black and white or green (if you have a green screen monitor). The different color patterns will create different density in the lines and figures you create. If you have something other than a color TV or monitor, it is best to experiment with white until you get used to the commands. Later, when you become accustomed to the line patterns created on a non-color screen, you can mix them for different effects.



Assuming you have a color screen, it might be necessary to adjust your TV/monitor to get the proper colors. The color chart that appears on your screen when your turn on you computer is a good one to use.

Making Color : CALL COLOR and CALL SCREEN

To get colors we use CALL COLOR and CALL SCREEN. The first is used for coloring text and text background and the second for the color of the screen. That gives us three colors we can use together: The color of the text, the background of the text and the surrounding screen color. The following little program shows us a blue green screen, with a dark yellow letter "L" against a red background.

```
10 CALL CLEAR
20 CALL SCREEN (4)
30 CALL COLOR (6,11,9)
40 CALL HCHAR (10,14,76)
50 CALL KEY (0,K,C)
60 IF C = 0 THEN 50
```

In order to see how this was done, we will start with the color codes. In line 20 we used Code 4 for our screen color. Looking at the chart below, we see that Color Code 4 is light green.

COLOR	COLOR CODE
Black	2
Blue (Dark)	5
Blue (Light)	6
Cyan	8
Gray	15
Green (Dark)	13
Green (Light)	4
Green (Medium)	3
Magenta	14
Red (Dark)	7
Red (Light)	10
Red (Medium)	9
Transparent	1
White	16
Yellow (Dark)	11
Yellow (Light)	12

Getting screen colors is really easy. Just CALL SCREEN and enter the color code in parentheses.

Getting letter colors in foreground and background shades is a little trickier. The format for CALL COLOR is the following:

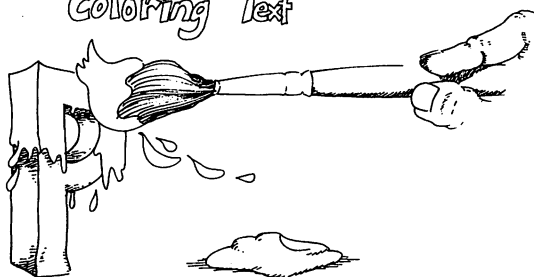
CALL COLOR (CHARACTER SET, FOREGROUND,
BACKGROUND)

The FOREGROUND and BACKGROUND codes are simply the codes for the desired colors. However, the CHARACTER SET code requires another chart:

SET	CODES	CHARACTER RANGES
1	32-39	(SPACE) - '(apostrophe)
2	40-47	(- /
3	48-55	Ø - 7
4	56-63	8 - ?
5	64-71	@ - G
6	72-79	H - O
7	80-87	P - W
8	88-95	X -
9	96-103	(Right Bracket) - g
10	104-111	h - o
11	112-119	p - w
12	120-127	x - DEL (blank)



Coloring Text



Now, although not as simple as coloring the screen, it is not too difficult to determine which character set is needed for the various text characters. To get several different letters from different sets requires planning, *BUT* there's a shortcut so that you do not have to worry about figuring out a lot of code. All of the capital letters are in Sets 5-8 inclusive. If you CALL COLOR to this set, then your output will be in the desired colors. Try the following program to see how this works:

```
10 CALL CLEAR
20 CALL COLOR (5,16,5)
30 CALL COLOR (6,16,5)
40 CALL COLOR (7,16,5)
50 CALL COLOR (8,16,5)
60 PRINT "THIS IS WHITE ON BLUE"
70 CALL KEY (0,K,C)
80 IF C=0 THEN 70
```

That was easier than using several CALL HCHAR or CALL VCHAR commands, and if you want you can have each letter in a different foreground/background scheme depending on the CALL COLOR commands for the different character sets. Experiment with different color combinations to find ones that are useful. Try fixing up your menu programs from the last chapter.

Since you may want only certain parts of your program in dazzling color and the rest in the default mode, you will need a way to get everything back to "normal." To do this requires CALL COLOR to black letters and light green background. Add the following lines to the previous program to see how this is done:

```

90 CALL COLOR (5,2,4)
100 CALL COLOR (6,2,4)
110 CALL COLOR (7,2,4)
120 CALL COLOR (8,2,4)
130 PRINT ::: "HOW ABOUT THIS?"
140 CALL KEY(0,K,C)
150 IF C=0 THEN 140

```

At this point you should be able to handle colors and text easily. But we really have not done much with graphics; so let's continue with some more tricks.

If the foreground and background color of a letter are the same, you will get a block of color. Your letter is there but, since the background and foreground are the same, it is invisible. A line of blocks would produce a bar. Try the following program to see the effect:

```

10 CALL CLEAR
20 REM ** RED SCREEN **
30 CALL SCREEN(11)
40 REM ** COLOR SET #5 **
50 REM ** YELLOW FOREGROUND &
  BACKGROUND **
60 CALL COLOR (5,7,7)
70 REM ** VERTICAL REPEAT 'A' **
80 REM ** THE 'A' IS INVISIBLE **
90 CALL VCHAR (5,10,65,15)
100 CALL KEY(0,K,C)
110 IF C=0 THEN 100

```

Now, if we can make a single bar, I'll bet we can make a bar graph. Also, we will want to label our graph, so we will use something other than the letter "A" to make our bars. In fact, we will make our label outside the character code range of the alphabet. We will use the =, equal sign, in Set 4 with an ASCII code of 61.

```

10 REM *****
20 REM BAR GRAPH 1
30 REM *****
40 CALL CLEAR

```

```

50 INPUT "TITLE OF GRAPH" : TITLE$
60 INPUT "HOW MANY PLOTS (1-5)" : PLOTS
70 IF PLOTS > 5 THEN 60
80 FOR X = 1 TO PLOTS
90 INPUT "VALUE (1-20)" : PV(X)
100 NEXT X
110 CALL CLEAR
120 REM *** END OF INPUT BLOCK ***
200 REM *****
210 REM MAKE THE GRAPH
220 REM *****
230 CALL SCREEN(11)
240 CALL COLOR(4,7,7)
250 FOR X = 1 TO PLOTS
260 ROW = 24 - PV(X)
270 COLUMN = X * 5
280 SCALE = PV(X)
290 CALL VCHAR(ROW,COLUMN,61,SCALE)
300 NEXT X
400 REM *****
410 REM LABEL GRAPH
420 REM *****
430 FOR I = 1 TO 28
440 PRINT "___";
450 NEXT I
460 L = LEN(TITLE$)/2
470 PRINT TAB(14-L); TITLE$
480 CALL KEY(0,K,C)
490 IF C = 0 THEN 480

```

RUN the program and see how nicely you can present data graphically. The program is severely limited in that it only does a maximum of five plots and values from 0 to 20. It is simple to change the number of plots above five. Just change the trap value to a higher number and change the offset in line 270 to less than 5 (e.g., $X * 2$) to set the bars closer together. Changing the values to above 20 requires more sophisticated manipulations, however. This is because 20 represents the maximum length of a vertical plot and still puts in the material at the bottom of the screen. Using our editor and RESEQUENCE command, let's fix up our graph program. *First* make sure your graph program is in memory and enter RESE-

QUENCE <ENTER>. This should renumber your lines by 10 beginning at line 100. Now enter the following lines:

```
10 CALL CLEAR
20 INPUT "MAX VALUE->": MV
30 N = 1
40 NN = MV
50 IF NN <= 20 THEN 100
60 N = N + 1
70 NN = MV / N
80 GOTO 50
```

Now, change/insert the following lines.

```
145 PRINT "MAXIMUM VALUE="; MV
180 INPUT "VALUE ": PV[X]
185 PV[X] = INT (PV[X] / N)
```

In order to understand what happened, we will go over the significant lines and explain each.

1. In line 40 the variable NN was defined to equal the maximum value (MV) entered in line 20.
2. In line 50, the crucial line for creating a proportional scale, NN is compared with 20 to find if the maximum value is equal to or less than 20. If it is greater, then the counter variable N is incremented by 1 and NN is re-defined to be the value of MV divided by N and looped back to line 50 for another comparison. As soon as the value of N increases to a point where the maximum value, MV, divided by N is not greater than 20, the loop exits to the main program. Whatever the value of N is at that time will be used in the rest of the program to divide any value entered.

FOR EXAMPLE:

The value of MV is established to be 100. Since 100 is greater than 20, 1 is added to N and 100 is divided by 2 resulting in the value of NN equaling 50. Since 50 is still larger than 20, N is incre-

mented to 3. When MV is divided by 3, the result is 33.33. Again it is larger than 20, so there is another loop. The program loops two more times. When N is equal to 5, MV divided by N equals 20. This time, when the comparison to 20 is made, it is found that NN is not larger than 20 and so the line is exited and the value of N is established at 5. No matter what value is entered, as long as it does not exceed the maximum value, there will be no errors since all plot values PV (1), etc., will be divided by 5. Since 100 is the maximum value to be entered, 20 is the maximum value which will be charted.

3. Two values for PV (X) are entered in lines 180 and 185. First, the raw value is entered in line 180. Then in line 185 PV(X) is changed to be an integer value using the formula, INT(PV(X)/N). The INT command is introduced to provide an integer (whole) number for charting.
4. The remaining program is the same as it was before.

Just to make sure you have all the correct changes, here is the complete program. (The RESEQUENCE messed up our "blocking", but it's better doing that than having to start over from scratch!)

```

10 CALL CLEAR
20 INPUT "MAX VALUE->" : MV
30 N = 1
40 NN = MV
50 IF NN <= 20 THEN 100
60 N = N + 1
70 NN = MV / N
80 GOTO 50
100 REM *****
110 REM BAR GRAPH 1
120 REM *****
130 CALL CLEAR

```

```

140 INPUT "TITLE OF GRAPH" : TITLE$
145 PRINT "MAXIMUM VALUE="; MV
150 INPUT "HOW MANY PLOTS (1-5) " : PLOTS
160 IF PLOTS > 5 THEN 150
170 FOR X = 1 TO PLOTS
180 INPUT "VALUE " : PV(X)
185 PV(X) = INT(PV(X)/N)
190 NEXT X
200 CALL CLEAR
210 REM *** END OF INPUT BLOCK ***
220 REM *****
230 REM MAKE THE GRAPH
240 REM *****
250 CALL SCREEN(11)
260 CALL COLOR(4,7,7)
270 FOR X = 1 TO PLOTS
280 ROW = 24 - PV(X)
290 COLUMN = X * 5
300 SCALE = PV(X)
310 CALL VCHAR(ROW,COLUMN,61,SCALE)
320 NEXT X
330 REM *****
340 REM LABEL GRAPH
250 REM *****
360 FOR I = 1 TO 28
370 PRINT "—";
380 NEXT I
390 L = LEN(TITLE$)/2
400 PRINT TAB(14-L); TITLE$
410 CALL KEY(0,K,C)
420 IF C = 0 THEN 410

```

FOR THE PERFECTIONIST WITH SOME TIME

We incremented N by 1 each time we passed through our test loop in line 50. If we wanted to get a finer value, we could have incremented N by .1 or .01 or even .00001! This would give us a nearer minimum value by which to divide PV(X) and still keep it proportional; however, it would take longer for the loop to find the minimum value of N. Change the program to see the different results in the charts. The smaller the increment, the closer to the top of the chart the maximum value will appear, but the longer the program will take to execute.

We have spent a good deal of time working on charts in screen graphics, but it is important to see the practical applications of such graphics. Often users see screen graphics simply as something to draw mosaic pictures on and nothing else; but, as we have seen, it is possible to make very good practical use of them as well. Now let's have a little fun with animation before going on to bit graphics.

Animation in screen graphics can be used in games and for special effects. We will only touch upon some elementary examples to provide you with the concepts of how animation works. Basically, by placing a figure on the screen, covering it up and then putting it in a new position; you can create the illusion of moving figures. It works in exactly the same way as animated cartoons. A series of frames are flashed on the screen sequentially. Even though each individual frame has a stationary figure, by rapidly flashing a series of such frames, the figures appear to move. Your computer does the same thing. For example, the following little program appears to bounce a ball in the upper left hand corner:

```
10 REM *****
20 REM ANIMATION 1
30 REM *****
40 CALL CLEAR
50 CALL VCHAR(2,3,79)
60 FOR PAUSE = 1 TO 20
70 NEXT PAUSE
```

```

80 CALL VCHAR(2,3,32)
90 CALL VCHAR(3,3,79)
100 FOR PAUSE = 1 TO 20
110 NEXT PAUSE
120 CALL VCHAR(3,3,32)
130 GOTO 50

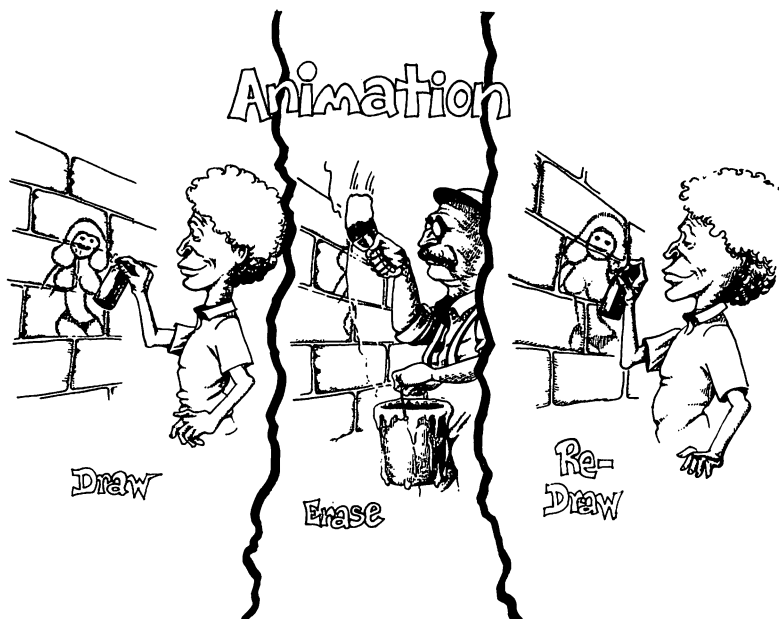
```

What appeared to be a moving “ball,” was actually a figure being placed on the screen, erased and then replaced in a different location. Now let’s do the same thing using the entire screen, and, just for fun, let’s add some sound and special effects. (Remember to turn up your sound for this one.)

```

10 REM *****
20 REM ANIMATION 2
30 REM *****
40 CALL CLEAR
50 FOR FALL = 1 TO 23
60 CALL VCHAR(FALL,16,79)
70 CALL SOUND(1,800,2)
80 CALL VCHAR(FALL,16,32)
90 NEXT FALL
100 REM *****
110 REM HIT THE GROUND
120 REM *****
130 CALL SOUND(70,-4,0,120,1)
140 REM *****
150 REM AND SPLATTERS
160 REM *****
170 FOR SPLAT = 23 TO 20 STEP -1
180 FLY = 24-SPLAT
190 CALL VCHAR(SPLAT,16,42)
200 CALL VCHAR(SPLAT,16+FLY,39)
210 CALL VCHAR(SPLAT,16-FLY,39)
220 CALL VCHAR(SPLAT,16,32)
230 NEXT SPLAT
240 CALL VCHAR(SPLAT,16,42)
250 CALL KEY(0,K,C)
260 IF C = 0 THEN 250

```



By experimenting with different algorithms you can create a wide range of effects. If you have played arcade games with movement and sound, you now have an idea of how they were created. Now go ahead and start working on that SUPER SPACE BLASTER ALIEN EATER game.

BIT GRAPHICS

All right, gang, we've seen just about all there is to see with screen graphics, and let's face it, most of what we did was not "graphic" but rather color and text manipulation. That's all right, though, for the same principles apply to the next step, Bit Graphics. In order to use Bit Graphics it is necessary to understand something about binary and hexadecimal numbers. There is nothing difficult or unusual about these number systems, but since we're used to the decimal system, these new systems may appear strange at first. To get started, let's take a look at how numbers are ordered in decimal, binary and hexadecimal:

THREE NUMBER SYSTEMS

DECIMAL	BINARY	HEXADECIMAL
0	0000	\$0
1	0001	\$1
2	0010	\$2
3	0011	\$3
4	0100	\$4
5	0101	\$5
6	0110	\$6
7	0111	\$7
8	1000	\$8
9	1001	\$9
10	1010	\$A
11	1011	\$B
12	1100	\$C
13	1101	\$D
14	1110	\$E
15	1111	\$F

(Hex is conventionally prefaced by a dollar sign.)

Above we have three different counting systems. The first is base 10 (decimal), the second, base 2 (binary) and the third, base 16 (hexadecimal). Each system is similar in that all follow the same counting rules. In decimal, we count from 0 to 9, run out of unique characters, add on another digit and start all over again. In the binary system, where there are only 2 digits (0 and 1) we run out of unique digits much sooner than in the decimal system. With the hexadecimal system, with 16



unique characters, it is possible to count farther than decimal before having to repeat digits. Let's take a look:

Binary	Decimal
0	0
1	1
Add digit and start over.	Ran out of unique digits
10	2
11	3
Add digit and start over.	Ran out of unique digits
100	4
101	5
etc.	

Hexadecimal	Decimal
\$9	9
	Ran out of unique digits
\$A	10 Add digit and start over.
\$B	11
\$C	12
etc.	

Decimal	Hexadecimal
14	\$E
15	\$F
	Ran out of unique digits.
16	\$10 Add digit and start over.
17	\$11

You may well be wondering why in the world even bother with binary and hexadecimal numbers. Well, to make a long story short, it has to do with the structure of microprocessors. Basically, the computer reads a bit of information in terms of its being ON (1) or OFF (0), and the binary system can "read" the state of ONs and OFFs with zeros and ones better than decimal. That's somewhat of an oversimplification, but essentially that is why we bother with binary. Since binary translates into hexadecimal in 8 and 16 bit chunks, and the microprocessors are also in similar chunks, hexadecimal is more quickly translated than decimal.

However, let us not spend all our time trying to understand the design of computers. Rather, let's see how we can do something with graphics! To begin, it is important to understand that all of the text characters you see on your screen are made up of dots or pixels of light on your screen. All of the characters are arranged in 8 by 8 matrixes giving 64 spots to shoot light to make a character. The basic unit of each matrix is a four cell block that has 16 different combinations of filled cells. The following shows a four-cell block:

Basic Graphic Block

(1)	(2)	(3)	(4)
-----	-----	-----	-----

Suppose we wanted to fill in Cells 1 and 3 and leave Cells 2 and 4 empty. Our block would appear as follows:

Basic Graphic Block

(X)	(2)	(X)	(4)
-----	-----	-----	-----

With paper and pencil that would be a simple enough matter, but how do we do that with the computer? Instead of filling in the cells with a pencil, we would do it by *turning on* a dot or pixel. To do that, we could use a "1" to indicate the light is on and a "0" to indicate the light is off. Now our block would look like this:

Basic Graphic Block

(1)	(0)	(1)	(0)
-----	-----	-----	-----

So far so good. We now have a way of representing a pattern on our computer with zeros and ones, but how do we translate that so that we can get it on our screen? Okay, go back to the chart that shows the decimal, binary and hexadecimal counting systems. Look at the binary system for the number "1010". In decimal the value is 10 and in hexadecimal it is \$A or simply A. Your TI-99/4A uses the hexadecimal code, in capital letters and numbers, to represent different patterns. By entering the hexadecimal value A, it is possible to get the pattern in a basic block we designed above. However, to get that we need to

enter it as part of an 8 by 8 matrix, and all we have so far is a 1 by 4 matrix. Let's put the rest of the matrix together:

Full Graphics Block

ODD				EVEN				
(1)	(2)	(3)	(4)	(1)	(2)	(3)	(4)	Blocks 1 and 2
(1)	(2)	(3)	(4)	(1)	(2)	(3)	(4)	Blocks 3 and 4
(1)	(2)	(3)	(4)	(1)	(2)	(3)	(4)	Blocks 5 and 6
(1)	(2)	(3)	(4)	(1)	(2)	(3)	(4)	Blocks 7 and 8
(1)	(2)	(3)	(4)	(1)	(2)	(3)	(4)	Blocks 9 and 10
(1)	(2)	(3)	(4)	(1)	(2)	(3)	(4)	Blocks 11 and 12
(1)	(2)	(3)	(4)	(1)	(2)	(3)	(4)	Blocks 13 and 14
(1)	(2)	(3)	(4)	(1)	(2)	(3)	(4)	Blocks 15 and 16

Now we're just about ready to make our graphics! First, though, we have to have a new command that will allow us to make our own characters. That command is `CALL CHAR` with the format

`CALL CHAR(ASCII,"HEX-PATTERN")`

The value for `ASCII` is the ASCII character we choose to replace with our custom designed character. The `HEX-PATTERN` is the hexadecimal value that makes up our 64 cell block. Each single hexadecimal value is based on the four-cell basic block. On the Full Graphic Block, there are 16 Basic Blocks numbered from 1 to 16. The left side has odd numbered blocks (1-15), and the right side has even numbered blocks (2-16). By assigning a hexadecimal value, beginning with Basic Block 1 and working our way sequentially to Basic Block 16, we can fill our Full Graphic Block with a replacement character.

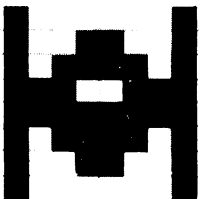
To get started you will need:

1. A pencil
2. Graph paper (or a hand-drawn 8 by 8 matrix.)
3. An eraser
4. Lots of creativity

(Go get those things, and I'll wait here.)

Now we're all set to create a replacement character. On the graph paper, block off an 8 by 8 area and draw a vertical line down the middle of it. On the left hand side write ODD and on the right hand side, EVEN. Now all you have to do is to fill in the little squares to make a graphic character. Once you are finished, indicate with 0's and 1's which squares are filled in and which are blank. Once that is done, translate each Basic Block into a hexadecimal number. When you have 16 hexadecimal values, you are all finished! The following shows how this can be arranged on graph paper:

TI SPACE FIGHTER

Block	ODD	EVEN	Block
(1) 1000		0001	(2)
(3) 1001		1001	(4)
(5) 1011		1101	(6)
(7) 1110		0111	(8)
(9) 1111		1111	(10)
(11) 1011		1101	(12)
(13) 1001		1001	(14)
(15) 1000		0001	(16)

Assuming everything went according to plan, you should have come up with the following set of hexadecimal numbers:

8199BDE7FFBD9981

So now let's see if everything worked. Enter the following program. *Note: We will be replacing the letter "A" with our new character.*

```

10 REM *****
20 REM TI SPACE FIGHTER
30 REM *****
40 CALL CLEAR
50 CALL CHAR(65,"8199BDE7FFBD9981")
60 CALL VHAR(12,12,65)
70 CALL KEY(0,K,C)
80 IF C=0 THEN 70

```

We did it! Instead of an "A", our little "Space Fighter" was in the middle of the screen. As soon as you hit a key, the "Space Fighter" went back to the letter "A." Using CALL HCHAR and CALL VCHAR, we can position our replacement characters anywhere we want on the screen. Since it is really a pain in the neck to make all of those binary to hexadecimal translations, let's write a program that will do it for us.

```

10 REM *****
20 REM BINARY-HEX
30 REM *****
40 CALL CLEAR
50 FOR X = 1 TO 16
60 PRINT "BLOCK";X;
70 INPUT " " : BL$
80 FOR Y = 1 TO 4
90 B$ = SEG$(BL$,Y,1)
100 L(Y) = VAL(B$)
110 NEXT Y
120 TL = (L(1)*8) + (L(2)*4) + (L(3)*2) + L(4)
130 IF TL>9 THEN 200
140 T$ = STR$(TL)
150 CALL VCHAR(23,20,ASC(T$))
160 NEXT X
170 CALL KEY(0,K,C)
180 IF C=0 THEN 170
190 END
200 REM *****
210 REM TRANSLATE 10-15
220 REM *****
230 IF TL= 10 THEN 290
240 IF TL= 11 THEN 310

```

```

250 IF TL= 12 THEN 330
260 IF TL= 13 THEN 350
270 IF TL= 14 THEN 370
280 IF TL= 15 THEN 390
290 T$="A"
300 GOTO 150
310 T$="B"
320 GOTO 150
330 T$="C"
340 GOTO 150
350 T$="D"
360 GOTO 150
370 T$="E"
380 GOTO 150
390 T$="F"
400 GOTO 150

```

At this point you should be able to create anything that will fit into the 8 by 8 matrix. Using the Binary-Hex conversion program, you can quickly convert your 4-digit binary numbers into single-digit hexadecimal numbers, block by block. The final 16-digit hexadecimal number is then entered into the CALL CHAR command. Before going on to making multiple character graphics, let's take a quick look at animation with our replacement character. We'll make a galaxy of stars for our "Space Fighter" and fly through the stars.

```

10 REM *****
20 REM MAKE A GALAXY
30 REM *****
40 CALL CLEAR
50 CALL SCREEN(11)
60 CALL COLOR (2,2,11)
70 RANDOMIZE 12
80 FOR I = 1 TO 20
90 R = INT[(24-1+1) * RND] +1
100 C = INT[(32-1+1) * RND] +1
110 CALL VCHAR(R,C,42)
120 REM ## 42 = ASTERISK ##
130 NEXT I
200 REM *****

```

```

210 REM MOVE SPACE FIGHTER
220 REM *****
230 CALL COLOR(5,2,11)
240 CALL CHAR(65,"8199BDE7FFBD9981")
250 FOR M = 1 TO 24
260 CALL VCHAR(M,M,65)
270 CALL VCHAR(M,M,32)
280 NEXT M
290 CALL KEY(0,K,C)
300 IF C < > 0 THEN 310 ELSE 250
310 END

```

Let's look at the program step by step:

STEP 1. Using the random number generator we created random (R)ow and (C)olumn values in lines 90 and 100.

STEP 2. Using the asterisk (*) character as stars, we plotted them on the screen using CALL VCHAR with the random R and C variables.
NOTE: We did not allow our random values to exceed 24 Rows or 32 Columns.

STEP 3. Using our TISPACEFIGHTER character, we plotted a (M)ove loop from 1 to 24.

STEP 4. We alternatively plotted our replacement character (65) with a space (32).

STEP 5. The movement continues until a key is pressed.

MULTIPLE CHARACTER GRAPHICS

Multiple character graphics is simply a matter of positioning characters next to one another so that a larger graphic can be made from two or more single characters. For example, if we use an 8 by 16 matrix, we can have half the image on the left half of the matrix and the other half on the right side.

Double Character Matrix

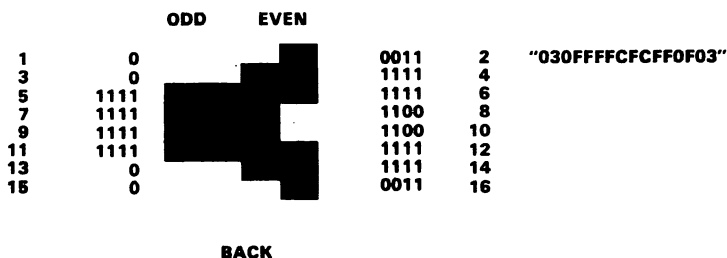
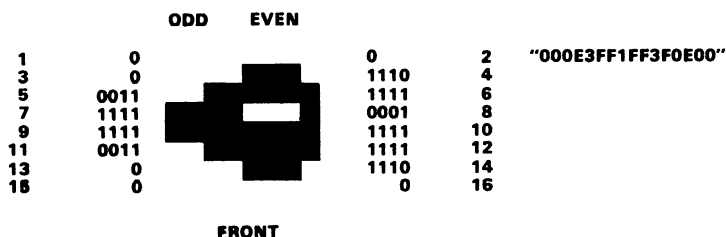
1.....8.....16

. First Second
. Half Half

. Row 8

If we want larger images, it is simply a matter of adding more blocks. Let's make another space rocket to go with our first one. First we'll make the right half and then we'll make the left half.

DOUBLE TI SPACE ROCKET



Using our Binary-Hex conversion program we generate the hexadecimal code and use the following program to put it on the screen. *NOTE: What do you think the label "DOUBLE CHARACTER" in line 110 is going to look like on the screen? We are replacing the characters B (Code 66) and C (Code 67) with our graphics.*

```
10 REM *****
20 REM DOUBLE CHARACTER
30 REM *****
40 CALL CLEAR
50 CALL COLOR(5,2,11)
60 CALL SCREEN(11)
70 CALL CHAR(66,"000E3FF1FF3F0E00")
80 CALL CHAR(67,"030FFFFCFCFF0F03")
90 CALL HCHAR(12,14,66)
100 CALL HCHAR(12,15,66)
110 PRINT "DOUBLE CHARACTER"
120 CALL KEY(0,K,C)
130 IF C = 0 THEN 120
```

To move a multiple character, we do the same thing we did with a single character except we have to be more careful. In horizontal movement, all we have to do is erase the trailing half of the image since the second half will replace it. Add/change the following lines to make your double character image move:

Move Double Character

```
85 FOR M= 31 TO 2 STEP -1
90 CALL HCHAR(12,M,66)
100 CALL HCHAR(12,M+1,67)
105 CALL HCHAR(12,M+1,32)
107 NEXT M
```

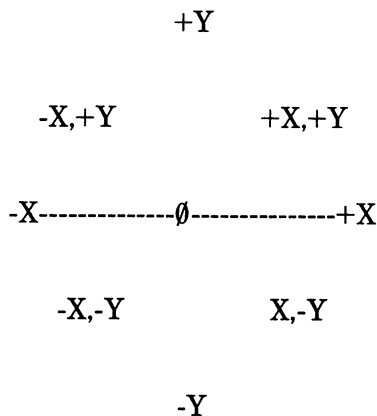
You will notice the movement is not as smooth as our single character graphic, and as we add more blocks to our graphics, movement is very rough looking. However, you can plan your programs so that the larger multiple character graphics are stationary and the moving ones are single character. *NOTE: There are enhancement packages for the TI-99/4A you can get that will you the ability to do more with moving and creating graphics. See Chapter 10.*

Joystick Control

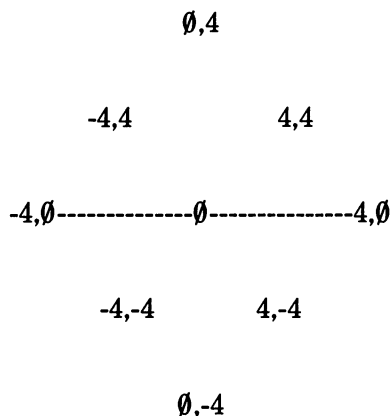
If you do not have joysticks, skip to the next section, but if you do then we will examine how to use them to move graphics. Turn off the power and plug the joysticks into the *LEFT* side of the computer. To get started let's look at the CALL JOYST format.

CALL JOYST (STICK#, HORIZONTAL, VERTICAL)

The STICK# is either 1 or 2. The horizontal axis is commonly called the "X-axis" and the vertical, the "Y-axis". In general terms an X,Y Axis can be seen as follows:



The center value on an X,Y Axis is always zero (0). As you move away from the center, horizontally, the X value increases or decreases. Up or down movement causes the Y value to increase or decrease. With joysticks, *any* vertical or horizontal movement of the stick causes a value of +/- 4, and substituting the joystick values for our X,Y Axis, we get the following:



The value is stored in the CALL JOYST variable set up for horizontal and vertical value. Since we are using the concept of the X,Y Axis, we might as well use the value X and Y to represent horizontal and vertical positions of the joystick. Therefore, we will define:

CALL JOYST{1 or 2,X,Y}

If the first value is "1" then it will affect Joystick # 1 and if "2" then Joystick # 2.

Finally, at the top of your joysticks is a *fire button*. This button is read with CALL KEY. It is the same format as we have been using to read the keyboard, but the first value is either "1" or "2" indicating the fire button of Joystick # 1 or Joystick # 2. If the fire button is pressed, then the (K)ey value is 18, and the (C)ondition value is non-zero. Okay, we're all set to see how joysticks work. The following program reads the X and Y

values of both joysticks and prints them to the screen. While the joystick is in the neutral (center) position, the values will be "0", and as you move them, the values will change to +/- 4. When you press the fire button on Joystick #1, the program will stop. Until then, it will scroll the values up the screen.

NOTE: Before you RUN this program, make sure the ALPHA LOCK key is in the UP or OFF position.

```
10 REM *****
20 REM JOYSTICK
30 REM *****
40 CALL CLEAR
50 CALL JOYST(1,X,Y)
60 CALL JOYST(2,X2,Y2)
70 PRINT "JOY X1=";X,"JOY Y1=";Y
80 PRINT "JOY X2=";X2,"JOY Y2=";Y2
90 CALL KEY(1,K,C)
100 IF K=18 THEN 110 ELSE 50
110 END
```

The program is somewhat klunky, but it is a simple one to show you how the values of joysticks are generated. Now let's see about moving graphics with the joystick. We will only use a single joystick. To begin, we'll simply move our TI SPACE FIGHTER and then progress to where we will blast something by pressing the fire button.

```
10 REM *****
20 REM JOYSTICK MOVEMENT
30 REM *****
40 H=12
50 V=12
60 OH=H
70 OV=V
80 CALL CLEAR
90 CALL SCREEN(11)
100 CALL CHAR(65,"8199BDE7FFBD9981")
110 CALL COLOR(5,2,11)
120 CALL JOYST(1,X,Y)
130 IF X=4 THEN 260
140 IF X=-4 THEN 280
```

```
150 IF Y=4 THEN 300
160 IF Y=-4 THEN 320
170 IF H>32 THEN 340
180 IF H<1 THEN 360
190 IF V>23 THEN 380
200 IF V<1 THEN 400
210 CALL VCHAR{OV,OH,32}
220 CALL VCHAR{V,H,65}
230 OH=H
240 OV=V
250 GOTO 120
260 H=H+1
270 GOTO 150
280 H=H-1
290 GOTO 150
300 V=V-1
310 GOTO 170
320 V=V+1
330 GOTO 170
340 H=32
350 GOTO 190
360 H=1
370 GOTO 190
380 V=23
390 GOTO 210
400 V=1
410 GOTO 210
```

WATCH OUT FOR THE ALPHA LOCK WITH JOYSTICKS!!

If your movement does not seem to be working correctly, it is probably the ALPHA LOCK key. When the key is DOWN or ON, your joysticks do not respond correctly. Therefore, before you run any program using the joysticks, make sure your ALPHA LOCK key is in the UP position.

The above program does several things, so let's go over it step-by-step.

STEP 1. First, the variables H and V are defined as 12 to start to image more or less in the center of the screen. (H indicates “horizontal” and V “vertical” — very clever programming.)

STEP 2. OH and OV (for “Old Horizontal and Old Vertical”) are defined to equal H and V. These variables are needed to keep track of the part of the screen we need to erase to simulate movement.

STEP 3. In Line 120 we check Joystick #1 for changes in the values X and Y.

STEP 4. In Lines 130-160 we see if the values of X and Y are +/-4, and if they are, we adjust the values of H and V in Lines 25/-320.

STEP 5. In Lines 170-200 we check the values of H and V to see if they are within the parameters of the screen. If they are not, the program goes to 340-410 to adjust them to maximum and minimum values.

STEP 6. Once all the adjustments are made, we go to Lines 210-220 to erase the old image and display the new one. *THEN* we redefine OH and OV so that the next time through the loop, they will erase the current image. Line 250 loops back to line 120 to check the values of the joystick.

Now that took a pretty big program to do all that, and we might as well have some fun with it. Let's make it into a simple game. By adding “stars”, we can create a STAR MAZE. We'll start the TISPACE FIGHTER in the upper left hand corner of the screen where it will be invisible, and see if you can guide it through the star maze without hitting any stars. Alternatively, you could make a game by seeing how quickly you could erase all the stars. Just add/change the following lines:

*** STAR MAZE ***

```
40 H=0
50 V=0
91 RANDOMIZE 31<-Change this value for
different mazes
92 FOR I = 1 TO 50 <-Change this value
for more stars
93 R=INT((24-1+1) *RND)+1
94 C=INT((32-1+1) *RND)+1
95 CALL VCHAR(R,C,42)
96 NEXT I
```

Finally, we come to shooting the space ship's Vaporizer Ray! It will vaporize stars, income tax and homework! Add the following lines to your program:

```
222 CALL KEY(1,K,C) {- Fire button on Joystick #1
224 IF K=18 THEN 500 {- See if fire button is pressed
500 REM *****
510 REM FIRE VAPORIZER RAY
520 REM *****
530 CALL HCHAR(V,H+1,45,32-H)
540 CALL SOUND(100,-4,0,910,2,110,1)
550 CALL HCHAR(V,H+1,32,32-H)
560 GOTO 230
```

That ought to cream a few stars. If you want, experiment with the sound in line 540 to see if you can get a more fearsome sound for the Vaporizer Ray. (The last time I heard one, that's what it sounded like. Honest!)

CALL GCHAR

The last thing to examine in this chapter, especially in relation to programming arcade type graphics, is CALL GCHAR. This command returns the ASCII value of a character from a given position on the screen. It can be used as a "collision check" in that it can see if a given character occupies a target position on the screen. For example, in our example of the Vaporizer Ray,

we fired little dashes - Code 45. By checking a target position against that code, we could find whether the Ray hit the target position. Of course, CALL GCHAR can check a screen position for anything else we may be interested in, from a menu choice to a graphic limit. For now though, we will see if we can make a little game using our Vaporizer Ray and CALL GCHAR. First let's take a look at the format of the statement and how it works.

CALL GCHAR(ROW,COLUMN,VARIABLE)

For example, enter the following little program:

```
10 CALL CLEAR
20 CALL VCHAR(10,20,65)
30 CALL GCHAR(10,20,ASCII)
40 PRINT ASCII
```

As you can see, the CALL GCHAR returned the value 65 that was put into Row 10, Column 20 of the screen. Now let's see how that could be incorporated into our VAPORIZER RAY program. First, delete all the lines that create the stars, then add/change the following lines:

```
95 CALL VCHAR(12,30,42) {- A single star.
542 GCHAR(12,30,HIT)
543 IF HIT = 45 THEN 600
600 *****
610 VAPORIZE STAR
620 *****
630 RANDOMIZE 15
640 FOR BANG = 1 TO 20
650 R = INT([(19-9+1)*RND]+9
660 C = INT([(32-28+1)*RND]+28
670 CALL VCHAR(R,C,46)
680 CALL SOUND(5,-2,0)
690 NEXT BANG
700 GOTO 550
```

Now that you know how, you can do the animation for your own arcade games. Use both joysticks, set up a two-player game and experiment with sound values to add drama.

SUMMARY

By combining the different tricks we've learned in this chapter, you should be able to make everything from business graphs to animated games. Graphics can be used for entertainment, education and business. To be sure, most of what we saw with graphics was for entertainment, but the same techniques can be used for non-game applications as well. As computers are becoming more "user friendly," so too are programs. With graphics, you can make very "user friendly" software yourself.

It is important to understand graphics as a form of programming. That is, to best use graphics you must plan them and then construct a program that will execute the plan. This may seem self-obvious after what we covered in this chapter, but often users consider graphics to be a separate kind of non-programming feature of the computer. As we have seen, using graphic commands takes as much programming skill as other programming aspects of the computer. Thus, rather than being a "toy" added to your computer, treat graphics as an additional tool to use and enjoy.

CHAPTER 8

Data and Text Files with the Tape and Disk System

Introduction

In this chapter we are going to learn more about some advanced applications with the tape and disk system. We will be covering two types of files: (1) Tape Files and (2) Sequential Disk Files. There are many similarities between tape and sequential disk files. Your disk system's data files are a type of sequential file, and we might even consider the way in which your cassette stores data to be a form of sequential disk file. However, for the sake of clarity we will discuss each separately.

Before beginning, I want to point out that the TI floppy disk system is a very sophisticated and smart device. For beginners, it can be difficult to understand some of its more advanced applications, and there is a very real risk of destroying programs and data on your disk. Therefore, in this section we will take each step slowly and, even at the risk of redundancy, explain the various functions of commands dealing with your disk system. Also, we will not be dealing with the most advanced features of the disk operating system, for they are beyond the scope of this book; however, we will be going to a middle range of sophistication. It is strongly advised for those of you with a disk system to use a blank initialized disk on which you have *not* accumulated programs. By doing so you will not inadvertently destroy valuable data and programs. (This comes from the voice of experience, having clobbered numerous disks myself!)

Data Files and Your Cassette

Wouldn't it be nice if, after keying in a lot of data, you could save it to your tape? For example, let's say you have created a long list of names and phone numbers or several checks in a checkbook program. Instead of having to re-enter that data,

or use READ and DATA statements, wouldn't it be nice if you could just store the data on tape and read the data with a small program? Well, using tape files, you can do that and a lot more. You can save any kind of numeric or string data to tape and then, using a special set of commands we will learn, load that data directly into your program. You can create a check-book program which saves all of your check entries and balances to tape, make a mailing list which creates, saves and retrieves names, addresses and telephone numbers, or even a list of your favorite recipes.

In Chapters 1 and 2 we discussed how to SAVE a program and retrieve it with OLD on your TI-99/4A using a computer cassette tape recorder. Both of these commands are executed in the Immediate mode. The commands we will now discuss are executed from the Program mode, but they too function to load and save information to your tape. They simply do it in a different format. To begin we will review the different commands for working with tape files, and then we will work with some programs employing these commands.



OPEN, INPUT#, PRINT# and CLOSE

In order to prepare your cassette for reading or writing information from within a program, the tape file must first be prepared with an OPEN statement. The format is as follows:

OPEN FILE#: "CS1/2", FILE ORG, FILE TYPE, MODE,
RECORD TYPE

Any integer from 1 to 255 can be used to reference the file number. For example, you might want to reference your file with number 21 (but any number between 1 and 255 would do just as well); so you would write:

OPEN #21:etc.

Second, since the device is the cassette recorder, the second entry would be "CS1" or "CS2". (We will assume you only have a single cassette recorder, so we will be using "CS1" in all of our examples.

OPEN #21:"CS1",etc.

Third, your tape always uses SEQUENTIAL files, and since the default is SEQUENTIAL, we do not have to specify FILE ORGANIZATION. However, for purposes of illustration, we will here.

OPEN #21:"CS1",SEQUENTIAL, etc.

Fourth, provide a FILE TYPE of either, INTERNAL or DISPLAY. For the most part we will be using INTERNAL since it is far more efficient for storing data. Since DISPLAY is the default type, it is important to include the FILE TYPE as INTERNAL.

OPEN #21:"CS1",SEQUENTIAL,INTERNAL, etc.

Next, you must indicate whether your program is reading data from the tape or writing to the tape. This may be a little confusing, but think of the tape as you would your keyboard.

When your INPUT from your keyboard, your computer *reads* information from your keyboard. Likewise, if you indicate the mode to be INPUT, your program will *read* data from your tape. On the other hand, if you OUTPUT data, you normally do it to your screen. With the tape, however, when you indicate OUTPUT, it means you send information to be *written* on the tape.

```
OPEN #21:"CS1",SEQUENTIAL,INTERNAL,INPUT,  
etc.
```

or

```
OPEN #21:"CS1",SEQUENTIAL,INTERNAL,OUTPUT,  
etc.
```

Finally, when you OPEN a file, you can optionally enter the record type as FIXED or RELATIVE. The tape always uses FIXED, defaulting with a 64-position record. If your files are longer, you can specify their length up to 192. You must include FIXED with tape files.

```
OPEN #21:"CS1",SEQUENTIAL,INTERNAL,INPUT,  
FIXED
```

or

```
OPEN #21:"CS1",SEQUENTIAL,INTERNAL, INPUT,  
FIXED 128
```

For the most part, all you will have to concern yourself with is whether the file is to be OPENed for INPUT or OUTPUT and enter:

```
OPEN #21:"CS1", INTERNAL, INPUT (or  
OUTPUT),FIXED
```

Everything else defaults to what you will need in most cases.

The procedure may appear to be somewhat involved, but is very simple once you get used to it. At the same time, it is quite flexible as well, since you can open a number of different files

simply by giving them different names. Usually you will want to CLOSE a file before OPENing another. To close a file, enter CLOSE and the file number. In our example, we would enter

```
CLOSE #21
```

So while there is a lot to remember in OPENing a file, there is not much when it comes to CLOSEing one.

The next command involves writing data to tape. Using the PRINT# command we can do this. The format for PRINT# is

```
PRINT #N:D1,D2,D3,etc.
```

where N is the file number and D1-3 is the data. For instance, sticking with our example, to print a number or string to tape we would enter

```
PRINT #21:etc.
```

If our data were strings, we would enter

```
PRINT #21:A$
```

or if numeric

```
PRINT #21:A
```

or a combination

```
PRINT #21:A,A$,B,B$
```

It is important to remember that PRINT# is not the same as PRINT, and with INTERNAL type storage, the commas do not act as they would when PRINTing to the screen. With DISPLAY data, you can format the PRINT# data with the same punctuation that is used with PRINT on the screen *except* that the commas serve solely as separators.

In the same way that PRINT# prints data to your tape, INPUT# inputs information into your computer from the tape. It has the same format as PRINT# using the OPENed file's number and reads in numeric or string variables.

```
INPUT #21:A<- Numbers
INPUT #21:A$<- Strings
INPUT #21:A$,A,B$,A<- Combination
```

Once the data are entered into the computer with INPUT #, you can then use PRINT (not PRINT #) to PRINT the information to your screen. This is especially important for data stored as INTERNAL data since it has to be transformed into a readable format.

Now that we have seen all of the commands for reading and writing files from and to tape, let's take a look at an application. We might as well use a practical application, so we will make a list of our friends' phone numbers. Whenever we want to call a friend, all we have to do is read the list from tape. First we must create a list to enter names and save them to tape. After we have done that, we will write a program to retrieve the names and numbers.

CREATE A FILE

```
10 CALL CLEAR
20 REM *** ENTER DATA ***
30 INPUT "NO. OF NAMES TO ENTER": N
40 DIM NA$(50), PH$(50)
50 FOR X = 1 TO N
60 PRINT "NAME#"; X ;
70 INPUT # "[(FIRST LAST)]: NA$(X)
80 INPUT "PHONE[#####]": PH$(X)
90 NEXT X
100 REM *** SAVE DATA TO TAPE ***
110 OPEN #1:"CS1", INTERNAL, OUTPUT, FIXED
120 PRINT #1: N
130 FOR X = 1 TO N
140 PRINT #1:NA$(X),PH$(X)
150 NEXT X
160 CLOSE #1
```

To use this program, get a blank tape and rewind your cassette. RUN the program and after you have entered all the names and phone numbers, you will be prompted to

```
* REWIND CASSETTE TAPE    CS1
  THEN PRESS ENTER
* PRESS CASSETTE RECORD    CS1
  THEN PRESS ENTER
```

As soon as you press the play and record buttons, your tape recorder spindles will begin turning. When all the information is saved, your screen will prompt you to

```
* PRESS CASSETTE STOP    CS1
  THEN PRESS ENTER
```

When you do that, the message

```
** DONE **
```

will appear, indicating that all your data has been saved. (Tape storage is relatively slow compared to disks, so to save time it is suggested to use just a few names (three or four) at first.)

Now let's see if everything worked out according to plan. To do that we need a program to read our data. We will use INPUT# to read the names and numbers. Since both the names and phone numbers were saved as strings, we have to read them back as strings. Since we are PRINTing to the screen as soon as we read them in, we do not have to worry about where they are in an array so we will simply use NA\$ and PH\$. (Remember to rewind your tape before RUNNING this program!). The first character we read from tape is the number of entries we have in our file. Therefore, to set up our loop to INPUT # our strings into memory, we will first INPUT #: N, the number of string groups we stored.

```
10 CALL CLEAR
20 OPEN #1:"CS1", INTERNAL, INPUT, FIXED
30 INPUT #1:N
40 FOR I = 1 TO N
50 INPUT #1: NA$,PH$
60 PRINT NA$,PH$
70 NEXT I
80 CLOSE #1
```

When you RUN this program, you will be prompted to

```
*REWIND CASSETTE TAPE    CS1  
THEN PRESS ENTER  
*PRESS CASSETTE PLAY    CS1  
THEN PRESS ENTER
```

When you do so, the recorder will begin spinning and soon the names and phone numbers you entered will begin appearing at the bottom of your screen. You may say, "Now just a minute here! I entered that data as two different string arrays, and this program read only two string variables! What happened to the arrays and how was it possible to get all that information back without the arrays?"

The answer to that question can be seen in how the data is stored and what our program did. While the file was OPENed, we INPUT# whatever data came along. As soon as it was in memory, we PRINTed it with our BASIC PRINT statement, not the PRINT# statement we use to print information to tape. The computer did not care whether the data entered into memory was a name or a phone number, only a string, and as soon as that string was in memory it PRINTed to the screen. The loop beginning in line 40 simply read the information in the file, picked it up and printed it to the screen, regardless of whether it was a name or phone number. To test this, simply enter PRINT PH\$ from the Immediate mode, and the last entry will be printed to the screen.

Now let's make our program a little fancier and more useful. If you use this program to store friends' phone numbers, the list will eventually cover more than a single screen. Therefore, you will be able to see only the last screenful of names and phone numbers. What we need is a program to search for and find a specific name and then close the file and print the name and number to the screen as soon as it has been located.

```

10 CALL CLEAR
20 INPUT "NAME TO LOCATE":NA$
30 OPEN #1:"CS1", INTERNAL, INPUT, FIXED
40 INPUT #1:N
50 FOR I = 1 TO N
60 INPUT #1:DA$,PH$
70 IF DA$ = NA$ THEN 200
80 NEXT I
90 CLOSE #1
100 CALL CLEAR
110 PRINT "NAME NOT FOUND"
120 END
200 REM ** PRINT OUT NAME AND NUMBER ***
210 CLOSE #1
220 CALL CLEAR
230 PRINT DA$,PH$
240 END

```

Now you have a handy program for storing names and numbers to tape and retrieving a single name and number you want to call. The next problem is updating your file without having to re-enter all of the names. That is, once you have made your phone list, you may want to add new names, but you do not want to key in all the names you already have on your list. Can this be done? Yes, but we have to first read all the names into memory from tape and then write them back to tape. There are several ways this can be done; our example is simply one way. We will do the following:

1. Load all the names and numbers on the tape into an array.
2. Input the new names and numbers on the end of the array.
3. Rewind the tape and resave the old and new data to tape.

REVISED TAPE PHONE LIST

```

10 CALL CLEAR
20 DIM NA$(50), PH$(50)
30 OPEN #1:"CS1", INTERNAL, INPUT, FIXED

```



```

40 INPUT #1 : N
50 FOR I = 1 TO N
60 INPUT #1:NA$(I),PH$(I)
70 NEXT I
80 CLOSE #1
100 REM *** NEW DATA ENTRY ***
110 CALL CLEAR
120 INPUT "NO. OF NEW NAMES":NN
130 FOR I = (N+1) TO (N+NN)
140 INPUT "NAME":NA$(I)
150 INPUT "PHONE":PH$(I)
160 NEXT I
200 REM *** COMBINE OLD AND NEW DATA AND
PUT IT ON TAPE ***
210 CALL CLEAR
220 NP = N + NN
230 REM COMBINED TOTAL OF ALL NAMES
240 OPEN #1:"CS1", INTERNAL, OUTPUT, FIXED
250 PRINT #1:NP
260 FOR I = 1 TO NP
270 PRINT #1:NA$(I),PH$(I)
280 NEXT I
290 CLOSE #1
300 END

```

Make sure to follow all the prompts, especially rewinding your tape. Test your revised list to be certain all the new names and numbers are saved. Simply use the same program we used to read the data off the first phone list.

Sequential Files and the Disk System

If you do not have a disk system you can skip this section and go on to the next chapter; however, if you are considering purchasing a disk drive for your TI-99/4A, the following material will be of interest. In many respects storing data on disks is similar to storing it on tape except the storage and retrieval process is much quicker. In fact, all of our examples in the previous section can be operated with the disk system by making only a few minor changes in the format. To get started

we will see how we can store data to disks using a slightly different format than we did with tape. To do this we will examine the OPEN, APPEND and UPDATE commands for disk. The other commands, PRINT #, INPUT # and CLOSE are used in the same way as they are on tape.

OPEN To open a file on disk, we do the same as on tape, except we must include a filename. On tape, we used OPEN #21, "CS1", etc., but we did not use a file name. With a disk system we would use the following format:

OPEN #21:"DSK1.PHONELIST", etc.

Note that the only difference is that instead of referencing "CS1" the reference was to "DSK1" and a FILENAME.

Fortunately, INPUT#, PRINT# and CLOSE use the same format as we did with tape. The number following each command is the number of the OPENed file. So, if we wanted to PRINT# in our example, we would write

PRINT #21

The same is true with INPUT # and CLOSE.

For a general format for OPENing files, we use a slightly different one than for tape, assuming the default conditions of SEQUENTIAL files.

OPEN #21: "DSK1.FILENAME", INTERNAL, INPUT
(or OUTPUT)

Now to see how all of this goes together, we will re-do our original PHONELIST program we created for tape. The data entry block is identical, so we will do only the block which saves the information to disk.

```

100 REM *****
110 REM WRITE DATA TO DISK
120 REM *****
130 CALL CLEAR
140 DIM NA$(50),PH$(50)
150 INPUT "HOW MANY NAMES: " :N
160 FOR I = 1 TO N
170 INPUT "NAME =>": NA$(I)
180 INPUT "PHONE=>": PH$(I)
190 NEXT I
200 OPEN #21:"DSK1.PHONELIST",
INTERNAL, OUTPUT
210 FOR I = 1 TO N
220 PRINT #21:NA$(I),PH$(I)
230 NEXT I
240 CLOSE #21

```

As can be seen, the main difference between tape and disk is in the format in line 190; otherwise, the disk and tape writing format are very similar. Likewise in retrieving information from disk, there are more similarities than differences between tape and disk.

```

10 REM *****
20 REM READ A FILE ON DISK
30 REM *****
40 CALL CLEAR
50 OPEN #21: "DSK1.PHONELIST",
INTERNAL, INPUT
60 IF EOF(21) THEN 100
70 INPUT #21:NA$,PH$
80 PRINT NA$,PH$
90 GOTO 60
100 CLOSE #21

```

Now look carefully at line 60. We introduced a new function that can be used with the disk system not available with the cassette. The EOF function is to check to see if there is an (E)nd (O)f (F)ile. If there is more data in the file, EOF(fn) returns a "0." If the end of file is reached, EOF returns a +1,

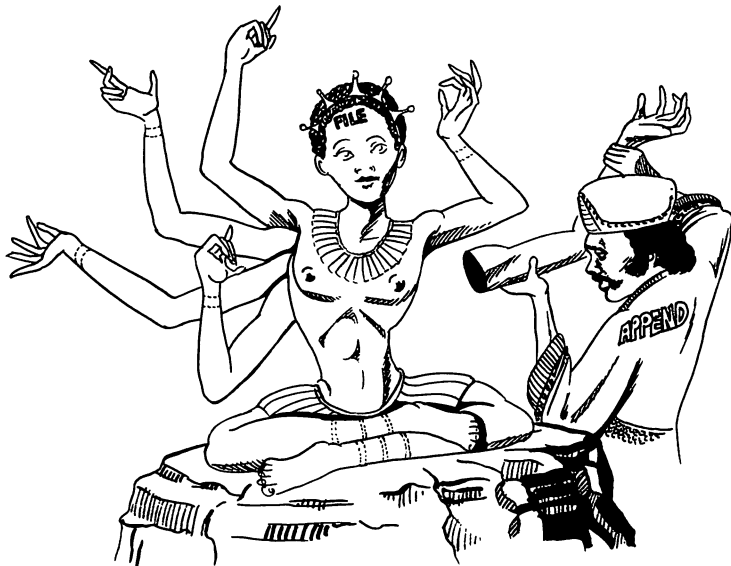
and if the physical end of file is reached, a -1 is returned. *NOTE the format in line 60 also. It simply reads IF EOF(21) THEN ... There is no relational or number before THEN. This format can be used if the value is anything but zero.*

Before going on to some more techniques using the disk system, there is a different technique for updating files than that used with tape. As you remember from our tape program, we first read in all the data from our old file, added new data, rewound the tape and simply wrote over the old material. With a disk we use the APPEND command to add data to the end of the file. When we OPEN the file, we use APPEND instead of OUTPUT.

```
200 REM *****
210 REM APPEND DATA TO FILE
220 REM *****
230 CALL CLEAR
240 INPUT "NO. NAMES TO APPEND=>" : N
250 DIM NA$(50),PH$(50)
260 FOR I = 1 TO N
270 INPUT "NAME: " : NA$(I)
280 INPUT "PHONE: " : PH$(I)
290 NEXT I
300 OPEN #21: "DSK1.PHONELIST",
INTERNAL,APPEND
310 FOR I = 1 TO N
320 PRINT #21: NA$(I),PH$(I)
330 NEXT I
340 CLOSE #21
```

Using the READ FILE ON DISK program, you will now get the original list of names and phone numbers, plus the new ones you added. Since we are using the EOF function, it is unnecessary to keep updating the number of items in our file as we did with tape.

Now that we have seen how to do a number of programs individually, let's make a single program which will 1) Write files, 2) Read a single file or all the files and 3) Add to a file. Instead of using names and phone numbers, let's use names and addresses.



```

10 REM *****
20 REM FILE MASTER
30 REM *****
40 DIM NA$(50), AD$(50), CITY$(50), STATE$(50),
ZIP$(50)
50 RESTORE
60 CALL CLEAR
70 TITLE$= "FILE MASTER"
80 CALL SCREEN(11)
90 CALL COLOR (9,7,7)
100 CALL VCHAR(3,3,97,20)
110 CALL VCHAR(3,30,97,20)
120 CALL HCHAR(3,3,97,28)
130 CALL HCHAR(23,3,97,28)
140 FOR W = 1 TO LEN(TITLE$)
150 C=W+10
160 AS$ = SEG$(TITLE$,W,1)
170 ASCII = ASC(AS$)
180 CALL HCHAR(2,C,ASCII)
190 NEXT W
200 FOR M=1 TO 7
210 IF M>5 THEN 1090
220 C1=5

```

```

230 NU$=STR$(M)
240 CALL HCHAR[(M+2) * 2,C1,ASC(NU$)
250 READ MENU$
260 FOR LABEL = 1 TO LEN(MENU$)
270 C=LABEL+7
280 AS$=SEG$(MENU$,LABEL,1)
290 ASCII=ASC(AS$)
300 CALL HCHAR[(M+2) * 2,C,ASCII]
310 NEXT LABEL
320 NEXT M
330 CALL KEY(0,K,C)
340 IF C=0 THEN 330
350 K$=CHR$(K)
360 PICK=VAL(K$)
370 IF PICK= THEN 960
380 GOSUB 1120
390 ON PICK GOSUB 410,650,680,820
400 GOTO 10
410 REM *****
420 REM CREATE/APPEND FILE
430 REM *****
440 CALL CLEAR
450 INPUT "HOW MANY NAMES TO ENTER=> ": N
460 FOR X=1 TO N
470 INPUT "NAME ": NAME$(X)
480 INPUT "ADDRESS ": AD$(X)
490 INPUT "CITY " : CITY$(X)
500 INPUT "STATE ": STATE$(X)
510 INPUT "ZIP " : ZIP$(X)
520 NEXT X
530 IF MODE$ = "APPEND" THEN 560
540 OPEN #7: D$&FILE$,INTERNAL,OUTPUT
550 GOTO 570
560 OPEN #7: D$&FILE$,INTERNAL,APPEND
570 FOR X=1 TO N
580 PRINT #7:NAME$(X),AD$(X),CITY$(X),
STATE$(X),ZIP$(X)
590 NEXT X
600 CLOSE #7
610 PRINT ::: "<HIT ANY KEY> 2
620 CALL KEY(0,K,C)

```

```

630 IF C=0 THEN 620
640 RETURN
650 MODE$="APPEND"
660 GOTO 410
670 REM *****
680 REM READ ENTIRE FILE
690 REM *****
700 OPEN #7: D$&FILE$, INTERNAL, INPUT
710 IF EOF(7) THEN 760
720 INPUT #7: N$,A$,C$,S$,Z$
730 PRINT N$:A$:C$;" ";S$;" ";Z$
740 PRINT
750 GOTO 710
760 PRINT ::"<HIT ANY KEY>"
770 CALL KEY(0,K,C)
780 IF C=0 THEN 770
790 CLOSE #7
800 RETURN
810 REM *****
820 REM FIND SINGLE NAME
830 REM *****
840 INPUT "NAME TO FIND=> ": NTF$
850 OPEN #7:D$&FILE$, INTERNAL, INPUT
860 IF EOF(7) THEN 910
870 INPUT #7: N$,A$,C$,S$,Z$
880 IF N$=NTF$ THEN 900
890 GOTO 860
900 PRINT NTF$:A$:C$;" ";S$;" "; Z$
910 CLOSE #7
920 PRINT ::"<PRESS ANY KEY>"
930 CALL KEY(0,K,C)
940 IF C=0 THEN 930
950 RETURN
960 REM ****
970 REM EXIT
980 REM ****
990 END
1000 REM *****
1010 REM DATA FOR MENU
1020 REM *****
1030 DATA CREATE NEW FILE,APPEND FILE,READ

```

```

ENTIRE FILE,FIND SINGLE NAME
1040 DATA EXIT, *****
=CHOOSE BY NUMBER=
1050 END
1060 REM *****
1070 REM ADJUST MENU
1080 REM *****
1090 NU="*"
1100 C1=7
1110 GOTO 240
1120 REM *****
1130 REM GET FILE NAME
1140 REM *****
1150 CALL CLEAR
1160 INPUT "ENTER FILE NAME=> ": FILE$
1170 D$="DSK1."
1180 RETURN

```

Now that was a long program! When writing such a program, it is a good idea to save your file about every 10-15 lines so that if you accidentally lose it, you can retrieve most of your work. It is important to note several aspects of this program so that you can understand how to work with longer programs. The first important aspect to note is how the program is blocked into subroutines. Not only does this make it easier to read, but

SUMMARY

In this chapter we learned how to save a lot of time by saving files to tape and disk. Data can be saved to your cassette tape for use later within a program. This is handy since it allows you to enter data at one time and then use it later without having to key in the data all over again. Of course this can be done within a single program using READ and DATA statements, but the user is stuck with that program for using the data. By storing it on tape, it is possible to use it in many different programs. This is especially handy with information you may want to store, retrieve and change. Using a disk system, it is possible to store data in sequential files much like saving data

to tape. However, disks access the data much faster than tapes, and it is possible to have a single program do several different things with data files on disks. The “FILE MASTER” program showed how a single program could be used to create, append, and read a single or multiple files. Care has to be taken to keep everything straight with such a program, but using sequential files increases the power of your computer a great deal. The practical applications of such programs are immense.

CHAPTER 9

You and Your Printer

Introduction

By now you should be used to outputting information to your screen, cassette or disk. When you write in `PRINT "HELLO"` you output to your screen. When you `SAVE` or `PRINT#` something, you output to your tape or disk. In the same way that you access your screen, tape or disk, you can access your printer; it is simply another output target. However, you cannot `LOAD`, `INPUT` or in some other way get anything from your printer as you can from your keyboard, tape or disk. (How are you going to get the ink off the paper and back into memory?)

The procedures for getting material out to your printer and using your printer's special capabilities requires certain procedures not yet discussed. Therefore, while much of what we will examine in this chapter will not be new in terms of the language of commands, it will be new in terms of how to arrange those commands. Also, we will see how we can use the printer in ways which have been done poorly using the screen. For example, no matter how long a program listing is, it can be printed out to the printer, while long listings on the screen scroll right off the top into Never-Never land. Likewise, in Chapter 8 we made a handy little program for storing friends' phone numbers and another one for storing names and addresses. With a printer we can print out our phone numbers or run off mailing labels with commands that output information to the printer.

There are a lot of printers on the market for computers; however, to keep things simple and to show the maximum use of your TI-99/4A with a printer, all examples will be with the TI-99/4 printer (Model No. PHP2500). This printer will provide all graphic and text features you will need and is easily

interfaced with the TI-99/4A system; besides, it is a very inexpensive printer. If you have another printer and an interface for the TI-99/4A, then you will have to rely heavily on your printer's manual. Unfortunately, many printer's manuals are not very good for beginners since they tend to use highly technical descriptions of how to interface and operate printers. Pay special attention to the codes used to turn on or off special features of your printer. This is usually done with a CHR\$ command from BASIC, so typically all you will have to do is to follow the instructions in this book using the appropriate code from your printer's manual.

BEFORE YOU BUY A PRINTER!!

The most important aspect in purchasing a printer is making certain it will interface with your TI-99/4A. Many times, over-enthusiastic salespersons will tell buyers all the qualities of a printer and naively believe it can be used on any computer. This is simply not true! In order for a printer to work with a computer, it must have the proper interface; the best printer in the world will not work with your TI-99/4A without such an interface. Therefore, when you buy a printer other than one made specifically for your TI-99/4A, make sure to buy the proper interface for it. The only certain way to insure the printer works with a TI-99/4A is to have it demonstrated with your computer. The TI-99/4 printer will work with the TI-99/4A, but otherwise you should have the printer's ability to work with your computer shown to you. (Any printer you plan to hook up to your RS232 Interface must have a serial interface port OR a special cable for parallel Interfacing. See Chapter 10 for some good deals on printers.)



Printing Text on Your Printer

The first thing you will want to do with your printer is to print some text in "hardcopy." (Hardcopy is a really impressive term computer people use to talk about printouts on paper. Use the term and your friends will be amazed.) Load any program you would like listed to your printer and enter

LIST "RS232"

Instead of listing to your screen, your listing was to your printer.

Like using your cassette tape and disk drive, it is necessary to first go through a number of steps to channel information to your printer. Let's review those steps now.

OPEN First, you OPEN a channel to your printer. Since your printer is connected through your RS232 module, all references are to the RS232 when accessing the printer. (If using a parallel printer, the references are to "PIO" for "Parallel Input/Output".) To OPEN a channel to your printer, you would enter

OPEN #5 : "RS232"

OPEN #5 : "PIO"

Now your printer is ready to receive instructions from PRINT #5, just as your disk or tape received PRINT # statements to the OPENed channel; however, it is a lot easier to direct information to your printer since you only have a single parameter - "RS232" - in most cases. Later we will examine the "softswitch" options, but for now we will stick with just "RS232".

PRINT# You will remember that we use PRINT# in programs where we want to print our information to our tape or disk. Well, with your printer the same principle also applies. Let's say that you want to print out only a few things in a program and you do not want everything going to the printer. Using PRINT#, only the information following the PRINT# would be printed. For example, suppose you want to have your screen prompt you with "Name?" and as soon as you enter the name, it is printed to your printer; you would want to use PRINT#. The format is

PRINT #5: NA\$

or

PRINT #5: "CHARLIE TUNA"

or

PRINT #5: 12345

Let's try a little program to print names to the printer to show how PRINT# can be used in programs where you want to use both the screen and printer.

```
10 CALL CLEAR
20 PRINT "TURN ON PRINTER" ::
30 PRINT "<HIT ANY KEY>"
40 CALL KEY [0,K,C]
50 IF C=0 THEN 40
60 CALL CLEAR
```

```

70 OPEN #5 : "RS232"
80 INPUT "NAME TO PRINT ":NAS$
90 PRINT #5:NAS$
100 INPUT "ANOTHER(Y/N) ":ANS$
110 IF ANS$="Y" THEN 80
120 CLOSE #5
130 END

```

CLOSE The final command in accessing your printer is **CLOSE**. As we can see in the above program, it closes the channel to the printer and turns it off. **CLOSE** works much the same way as it does with the tape and disk systems; however, instead of closing a channel to the tape or disk, you **CLOSE** it to the printer via the RS232 module. In the above example in line 120, we used **CLOSE #5** to turn off access to the printer after we had finished entering our names.

CHR\$ To The Rescue

The secret to using printers is in understanding what their control codes mean and how to use those codes. For example, the following is a partial list of codes provided with a CEN-TRONICS 737 printer:

Mnemonic	Decimal	Octal	Hex	Function
ESC,SO	27,14	033,016	1B,0E	Elongated Print
ESC,DC4	27,20	033,034	1B,13	Select 16.7 cpi
ESC,DC1	27,17	033,021	1B,11	Proportional Print

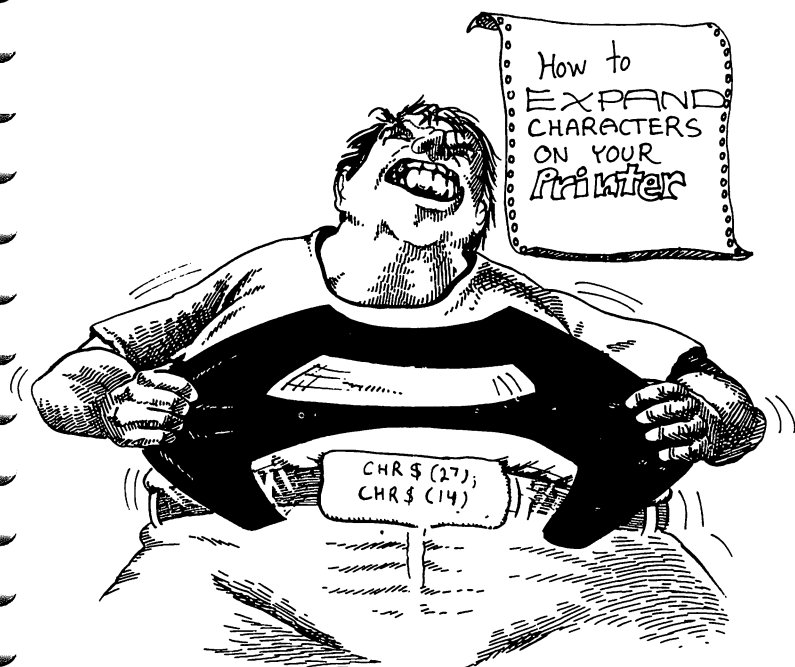
Now, for most first-time computer owners, that could have been written by a visitor from another planet for all the good it does. However, there is important information in those codes and, once you get to know how, it is relatively easy to use them.

To get started, forget everything except the "Decimal" and "Function" columns. Now, taking the first row, we have decimal codes 27 and 14 to get elongated print. To tell your

printer you want elongated print you would use `CHR$(27); CHR$(14)`. To kick that into your printer you would do the following:

1. `OPEN #5: "RS232" (or "PIO" if parallel)`
2. `PRINT #5: CHR$(27); CHR$(14); "MESSAGE"`

If you have a Centronics 737 or 739 printer, it would have printed the string MESSAGE in an elongated print. Likewise, for the condensed printing 16.7 cpi (characters per inch), you would have entered `CHR$(27); CHR$(20)` and for the proportional type face, `CHR$(27); CHR$(17)`. Once you get the decimal code, enter that code to the printer and it will do anything from changing the type-face to performing a back-space function.



With other printers the same is true, but let's get back to the TI-99/4 printer we have been examining since it was designed with TI computers in mind. As we will see, like the Centronics printers or any other, the TI-99/4 also uses CHR\$ commands to access the printer's different abilities. Let's look at the various CHR\$ commands associated with the TI-99/4 printer:

CHR\$	FUNCTION
10	Line feed
12	Form feed
13	Carriage Return
8	Back Space
14	Double width
20	Turn off double width
15	Condensed
18	Turn off condensed
27	Escape (used in conjunction with the following characters:)
"E"	Emphasized printing
"F"	Turn off emphasized
"G"	Double printing
"H"	Turn off double printing
"K"	Normal density printing
"L"	Dual density printing
"Q"	Set column width

To see how the CHR\$ functions work we will use a simple program that will print out your name. Since we already know how to print out normal text, we will begin with expanded text. Looking at our chart, we see that CHR\$(14) will expand our printout, so we will use it in our program.

```
10 CALL CLEAR
20 OPEN #5 : "RS232"
30 INPUT "YOUR NAME": NA$
40 PRINT #5: CHR$(14);NA$
50 CLOSE #5
```

RUN the program and print out some names and note the expanded characters. (Try that on your typewriter!)

We have not done very much with upper and lower case so far, but in printing text to your printer there are many times you will want to have upper and lower case characters. For example, in printing out names you may want your printer to print out

Captain John W. Smith

instead of

CAPTAIN JOHN W. SMITH

Now press the ALPHA LOCK key so that it is in the UP or OFF position. Your printout now shows upper and lower case. BUT there is a big difference between the lower case characters on your printer and the lower case characters on your screen. The printer lower case characters are "true" lower case as opposed to the small upper case characters you get on your screen. On some printers, such as the EPSON MX-80FT with GRAFTRAX PLUS and GEMINI 10, it is possible to have not only expanded print but also italicized, condensed, double strike, emphasized and super/subscript type faces and any combination of them together. Using CHR\$, all of the different type styles can be used separately or in combination with one another.

Now that we have seen different ways to operate the type faces on the printer, let's do something practical. We will make a mailing label program for the TI-99/4 printer. Various label manufacturers make adhesive labels with tractor-feed margins so that you can put them into your printer just like your paper. Our program will make labels that will print the addressee's name in expanded and everything else in the emphasized mode. (Keep the ALPHA LOCK KEY OFF!)

```
10 CALL CLEAR
20 OPEN #5: "RS232"
30 INPUT "NAME ": NA$
40 INPUT "ADDRESS ": AD$
50 INPUT "CITY ": CT$
60 INPUT "STATE ": SA$
```

```

70 INPUT "ZIP CODE " : ZIP$
100 REM *****
110 REM PRINT LABELS
120 REM *****
130 PRINT #5: CHR$(14); NAS
140 PRINT #5: CHR$(27); "E"; AD$
150 PRINT #5: CT$; ", "; SA$; ", "; ZIP$
160 PRINT #5: CHR$(27); "F"
170 CLOSE #5

```

As you will see when you RUN this program, the label you printed looks very clear and professional. In the program, we used CHR\$(14) to get the double width, but we did not turn it off. After the printer printed the name in double width, nothing else was printed. This is because with double width only, after there is a carriage return, the double width is cancelled. With the emphasized mode we turned it on only once in line 110, yet everything following it was emphasized. That is because with the other modes, they stay there until turned off. In line 130 we turned off the emphasized mode so that the next thing printed would be normal. If you RUN the program twice without line 30, the second and subsequent RUNs will make the name both double width and emphasized.

In order for the program to be more practical we will need a few line feeds at the end of the printing so that your labels can be properly aligned. Depending on the size of your mailing labels, you will need a greater or fewer number of line feeds. Insert the following line into your program and adjust the size of the loop to align your labels properly.

```

152 FOR I = 1 TO 3 (Loop may be changed)
154 PRINT #5; CHR$(10)
156 NEXT
158 REM CHANGE "3" TO THE CORRECT
NUMBER OF LINE FEEDS FOR YOUR LABELS

```

In Chapter 8 we promised to insert a subroutine in the FILE MASTER program to print out the names and addresses to your printer. Well, that's just what we're going to do. To make the changes, load your FILE MASTER program into memory

and make the following additions or changes in the program.
(Good grief! Don't rewrite the whole thing!)

```
695 INPUT "SEND TO PRINTER{Y/N}? " : PRINTER$
725 IF PRINTER$ = "N" THEN 730
727 GOSUB 2000
2000 REM *****
2010 REM PRINTER SUBROUTINE
2020 REM *****
2030 EOPN #5:"RS232"
2040 PRINT #5: N$: A$: C$: " ", S$: " "; Z$
2050 PRINT #5 : CHR$(10)
2060 CLOSE #5
2070 RETURN
```

Tab Stops on Your Printer

Sometimes you do not want your printout to begin at the left hand side of your paper or label. To position the starting point of your text, you use `CHR$(9)` in conjunction with `CHR$(27)` and `CHR$(68)`. The format is fairly convoluted, but once you get used to it, it isn't too difficult. Try to think of the sequence as first setting the tab stops and then tabbing to the next tab position whenever `CHR$(9)` is encountered. When the tab sequence ends, it is delineated with `CHR$(10)`. The general format is as follows:

```
PRINT #5: CHR$(27); CHR$(68); CHR$(TAB 1);
CHR$(TAB 2); ... CHR$(TAB N); CHR$(0)
CHR$(tab-n); CHR$(0)
```

To tab to a given column after the tabs have been established simply insert `CHR$(9)` before the string to be printed.

```
PRINT #5: CHR$(9); "STRING-A"; CHR$(9);
"STRING-B"; CHR$(9); "STRING-C"
```

It is important to remember how many tabs you have since each `CHR$(9)` jumps one tab regardless of whether or not you print a string. For example, if you printed

```
PRINT #5: CHR$(9);CHR$(9); "STRING"
```

the string would be printed at the second tab stop. For example, try the following:

```
10 REM *****
20 REM HORIZONTAL TAB
30 REM *****
40 T1$ = "TAB 1"
50 T2$ = "TAB 2"
60 T3$ = "TAB 3"
70 OPEN #5: "RS232"
80 PRINT #5 :CHR$(27);CHR$(68); CHR$(10);
CHR$(20); CHR$(30); CHR$(0)
90 REM TABS OF 10, 20 AND 30
100 PRINT #5: CHR$(9); T1$; CHR$(9);
T2$; CHR$(9); T3$
110 CLOSE #5
```

In the above example, your printer will print your output evenly across the page; however, if you change line 100 to read

```
100 PRINT #5: CHR$(9); CHR$(9); T1$, CHR$(9);
T2$; CHR$(9); T3$
```

the first string, TAB 1 will be at the second tab stop, and TAB 2 and TAB 3 will be jammed up against one another since all three tab stops were used before the third string was printed.

Before going on to printer graphics we will examine how to use positioning in a program. This is useful in making lists where columns are important. For example, we can make a list of items for a garage sale. The first column will be the item for sale, the second column the asking price for the item and the third column the actual price for which the item was sold. We will use INPUT statements so that all items can be entered from the keyboard and used with an actual garage sale. (Who knows when you will want to use it? So why not make it useful!)

```

10 CALL CLEAR
20 INPUT "HOW MANY ITEMS TO SELL ": N
30 DIM IT$(50), AP(50), SP(50)
40 PRINT :
50 FOR I = 1 TO N
60 PRINT "ITEM #"; I;
70 INPUT IT$(I)
80 INPUT "ASKING PRICE $": AP(I)
90 INPUT "SELLING PRICE $": SP(I)
100 PRINT
110 NEXT I
200 REM *****
210 REM PRINTER FORMAT ROUTINE
220 REM *****
230 OPEN #5: "RS232"
240 ITEM$ = "ITEM"
250 ASK$ = "ASKING PRICE"
260 SELL$ = "SELLING PRICE"
270 PRINT #5: CHR$(27); CHR$(68); CHR$(10);
CHR$(30); CHR$(50); CHR$(0)
280 PRINT #5: CHR$(9); ITEM$; CHR$(9); ASK$;
CHR$(9); SELL$
290 REM ** PRINT A LINE **
300 FOR LINE = 1 TO 65
310 PRINT #5: "-";
320 NEXT LINE
330 PRINT #5: CHR$(10)
340 FOR I = 1 TO N
350 PRINT #5: CHR$(9); IT$(I); CHR$(9); AS$(I);
CHR$(9); SP(I); CHR$(10)
360 NEXT I
370 CLOSE #5

```

There are a couple of things to note in this program. First of all, notice how we employed CHR\$ code to set up our tab positions in line 270. The tabs were set for 10, 30 and 50. Then in line 280 we printed the heading using the tab stops we created. In lines 340 to 360 we read in our arrays and instead of having the output printed to the screen, we printed it to the printer. Using those tab stops, we could not have done a very

good job of printing the output to our screen since it used only 28 columns. Our second tab stop was beyond the parameters of the screen.

To improve the program, figure out how to have the program compute the totals of the asking price and selling price of the items. It might be an interesting addition to have a fourth column which keeps a tally of the differences between the asking and selling prices. This is something that you should be able to work out on your own! (Hint: Create a fourth array and tab stop.)

Printing Graphics

Now that we have seen how to print text, we will look at graphics printing.

Making Your Own Graphic Characters on the Printer

In Chapter 7 we showed how to create graphic characters using a binary coding translation to hexadecimal. Now we will do the same thing with printer graphics except we will translate binary to decimal. First of all, we will be using a 7 by 7 matrix instead of an 8 by 8 matrix. (With dual density graphics, we can use an 8 by 8 matrix, but to use the dual density graphics we have to change one of the dip switches in the printer. Your printer manual tells you how to do this, but we will stick with the 7 by 7 matrix to keep it simple. We could have up to a 7 by 480 matrix!) To get started, instead of sending you off for some graph paper we will make our own graph for our matrix on the printer, explaining the process as we go along.

To begin, we use the following format to initiate the graphics mode.

```
OPEN #5: "RS232.CR.DA=7"
```

This format is different from our regular text format. The CR turns off the carriage return/linefeed, and the DA tells the printer to expect 7 Data bits. Since the carriage return is turned off, we have to insert a CR with CHR\$(10) when we want a linefeed. Depending on what we are printing, we may or may not want CR, and since DA defaults to 7, we usually do not need it either.

Once we OPEN the printer channel for graphics, we must then set up the normal density Graphics Mode with the following:

```
PRINT #5: CHR$(27);"K"; CHR$(LON); CHR$(HON)
```

The CHR\$(27);"K" tells the computer to turn on normal density graphics. That's simple enough. The next part might be a bit strange, though. LON stands for "Low Order Number" and HON for "High Order Number." As long as your number of graphic points is below 128, you simply enter that number in LON and a value of "0" for HON. However, with normal density graphics, you can have up to 480 dot positions; so you may need numbers greater than 127. To make this conversion, you use the "modulo" of your data number divided by 128 for LON and the INTeger of your data number divided by 128 for HON. (Use 256 if you use dual density graphics.) Getting the HON number is really easy since all we have to do is to PRINT INT(N/128) with N being the number of Graphic Mode data. Getting the modulo (the remainder after division) of a number takes either some pencil and paper work or a program. Since we've got a computer in front of us, let's do it with a program that will tell us the values of LON and HON.

```
10 REM *****
20 REM GR. NO. CONVERTER
30 REM *****
40 CALL CLEAR
50 INPUT "GRAPHIC DATA NUMBER " :X
60 Y= 128
70 Z=INT(X/Y)
80 M1=Z*Y
90 MOD=X-M1
100 PRINT "LOW ORDER NUMBER=" ;MOD
110 PRINT "HI ORDER NUMBER=" ;Z
```

So far so good, but what the heck is the graphic data number? To understand that, let's examine how the "dots" of graphics are set up. The following matrix shows the work area we are using — a 7 x matrix.

128	-----	{- For 8 bits
64	_____	
32	_____	
16	_____	
8	_____	
4	_____	
2	_____	
1	_____	

By inserting "dots" into the blanks, we can create a figure and this is translated to a way in which the TI-99/4A can understand by a vertical total of the positions containing dots. For example, if we draw a square, we would have the first and last columns filled and the top and bottom rows filled. Beginning with the first column, the value would be $64 + 32 + 16 + 8 + 4 + 2 + 1$ equaling 127. The next five columns would have a dot at the top and bottom. A dot in the top row would be 64, a dot in the bottom row would be 1, and adding them together we get 65. The last column would be the same as the first, 127. Therefore, we would want to create a CHR\$ with the following values:

127 65 65 65 65 65 127

for our box figure. To do this we could have a line which reads as follows:

```
PRINT #5: CHR$(127) ; CHR$(65) ; CHR$(65) ;
CHR$(65) ; CHR$(65) ; CHR$(65) ; CHR$(127)
```

but that (whew!) would take a lot of time. Instead it would be a lot simpler to READ in the values as DATA statements and PRINT # the CHR\$ we need for our figure, such as,


```

10 FOR I = 1 TO 7
20 READ GRAPHICS
30 PRINT #5: CHR$(GRAPHICS);
40 NEXT
50 DATA 127, 65, 65, 65, 65, 65, 127

```

Now let's put it all together into a program.

```

10 REM *****
20 REM GRAPHIC BOX
30 REM *****
40 CALL CLEAR
50 OPEN #5: "RS232"
60 PRINT #5: CHR$(27);"K"; CHR$(7); CHR$(0)
50 FOR G = 1 TO 7
80 READ A
90 PRINT #5:CHR$(A);
100 NEXT G
110 CLOSE #5
200 REM *****
210 REM GRAPHIC DATA
220 REM *****
230 REM DATA 127, 65, 65, 65, 65, 65, 127

```

When you RUN this program, a little box will be printed. Nothing very exciting, I admit, but now let's see how we can use that little box to make a matrix to create new characters. The following program will make a 7 by 7 matrix for you and requires making only a few changes in the above program:

```

10 REM *****
20 REM BIT MATRIX
30 REM *****
40 CALL CLEAR
50 FOR K = 1 TO 7
60 OPEN #5:"RS232"
70 FOR J=1 TO 7
80 PRINT #5: CHR$(27);"K"; CHR$(7); CHR$(0);
90 RESTORE
100 FOR I = 1 TO 7
110 READ A

```

```

120 PRINT #5: CHR$(A);
130 NEXT I
140 NEXT J
200 REM *****
210 REM GRAPHIC DATA
220 REM *****
230 DATA 127,65,65,65,65,127
300 REM *****
310 REM END OF ROW
320 REM *****
330 CLOSE #5
340 NEXT K
350 END

```

Now that you have a better idea of what can be created, print up a batch of matrixes and design some original printer graphics! You always wanted your own logo; now you can do it!

Printer Graphic Utilities

Since it is not much fun figuring out the LON and HON for our printer graphics and converting binary numbers to decimal, let's write a program that will do it for us. The following two utilities will automatically figure out 1) The Low Order Number and High Order Number for you if you supply the Graphic Data Number and 2) convert binary to decimal for you. The graphic data number is determined simply by counting the number of DATA entries you have to make up a graphic figure. For converting binary to decimal the program uses eight binary numbers so that you can use both normal and dual density graphics if you wish. Since we have been using normal density graphics, always enter "0" for the first value when converting binary to decimal with 7 bit graphics.

GRAPHIC NUMBER CONVERTER

```

10 REM *****
20 REM GR. NO. CONVERTER
30 REM *****

```

```

40 CALL CLEAR
50 INPUT "GRAPHIC DATA NUMBER " : X
60 Y = 128
70 REM CHANGE THE VALUE OF Y TO 256
  FOR DUAL DENSITY GRAPHICS
80 Z=INT(X/Y)
90 M1=Z*Y
100 MOD=X-M1
110 PRINT "LOW ORDER NUMBER=";MOD
120 PRINT "HI ORDER NUMBER=";Z

```

EIGHT BIT BINARY-DECIMAL CONVERTER

```

10 REM *****
20 REM BINARY-DECIMAL
30 REM *****
40 CALL CLEAR
50 INPUT "BINARY VALUE (8 DIGITS)": BIN$
60 IF LEN(BIN$) <> 8 THEN 50
70 FOR X=1 TO 8
80 Y$=SEG$(BIN$,X,1)
90 P(X)=VAL(Y$)
100 NEXT X
200 REM *****
210 REM CONVERT
220 REM *****
230 TOP=128
240 FOR C=1 TO 8
250 DEC=P(C)*TOP
260 DTOTAL=DTOTAL+DEC
270 TOP=TOP/2
280 NEXT C
290 PRINT "DECIMAL=";DTOTAL
300 PRINT "::"ANOTHER(Y/N)? ";
310 CALL KEY(0,K,C)
320 IF C=0 THEN 310
330 DTOTAL=0
340 IF CHR$(K)="Y" THEN 10

```

Now let's see if we can make our TI SPACE FIGHTER into a printer character. Since we're using a 7 by 7 matrix, it will be a

little different than the one we made for the screen. Using our two printer utility program and our program to print out a matrix to create our own graphic, it should be easy.

TI Space Fighter Graphic

<u>Ø</u>	<u>Ø</u>	<u>Ø</u>	<u>Ø</u>	<u>Ø</u>	<u>Ø</u>	<u>Ø</u>	8 <- Unused
<u>x</u>	<u>Ø</u>	<u>Ø</u>	<u>Ø</u>	<u>Ø</u>	<u>Ø</u>	<u>x</u>	7
<u>x</u>	<u>Ø</u>	<u>Ø</u>	<u>Ø</u>	<u>Ø</u>	<u>Ø</u>	<u>x</u>	6
<u>x</u>	<u>Ø</u>	<u>x</u>	<u>x</u>	<u>x</u>	<u>Ø</u>	<u>x</u>	5
<u>x</u>	<u>x</u>	<u>x</u>	<u>x</u>	<u>x</u>	<u>x</u>	<u>x</u>	4
<u>x</u>	<u>Ø</u>	<u>x</u>	<u>x</u>	<u>x</u>	<u>Ø</u>	<u>x</u>	3
<u>x</u>	<u>Ø</u>	<u>Ø</u>	<u>Ø</u>	<u>Ø</u>	<u>Ø</u>	<u>x</u>	2
<u>x</u>	<u>Ø</u>	<u>Ø</u>	<u>Ø</u>	<u>Ø</u>	<u>Ø</u>	<u>x</u>	1
<u>Ø</u>	<u>Ø</u>	<u>Ø</u>	<u>Ø</u>	<u>Ø</u>	<u>Ø</u>	<u>Ø</u>	8 <- Unused
<u>1</u>	<u>Ø</u>	<u>Ø</u>	<u>Ø</u>	<u>Ø</u>	<u>Ø</u>	<u>1</u>	7
<u>1</u>	<u>Ø</u>	<u>Ø</u>	<u>Ø</u>	<u>Ø</u>	<u>Ø</u>	<u>1</u>	6
<u>1</u>	<u>Ø</u>	<u>1</u>	<u>1</u>	<u>1</u>	<u>Ø</u>	<u>1</u>	5
<u>1</u>	<u>1</u>	<u>1</u>	<u>1</u>	<u>1</u>	<u>1</u>	<u>1</u>	4
<u>1</u>	<u>Ø</u>	<u>1</u>	<u>1</u>	<u>1</u>	<u>Ø</u>	<u>1</u>	3
<u>1</u>	<u>Ø</u>	<u>Ø</u>	<u>Ø</u>	<u>Ø</u>	<u>Ø</u>	<u>1</u>	2
<u>1</u>	<u>Ø</u>	<u>Ø</u>	<u>Ø</u>	<u>Ø</u>	<u>Ø</u>	<u>1</u>	1
1	2	3	4	5	6	7	

The first diagram shows where we put our figure, and the second shows our conversion to binary. Using our BINARY-DECIMAL utility, we enter the column binary numbers from left to right. Our DATA values will be

127,8,28,28,28,8,127

Our LON value is 7 and HON is Ø. Now we're all set for the program:

```

1Ø REM *****
2Ø REM PR SPACE FIGHTER
3Ø REM *****

```

```

40 CALL CLEAR
50 OPEN #5: "RS232"
60 PRINT #5: CHR$(27);"K"; CHR$(7); CHR$(0);
70 FOR G=1 TO 7
80 READ GRAPHIC
90 PRINT #5: CHR$(GRAPHIC);
100 NEXT G
110 CLOSE #5
200 REM *****
210 REM FIGHTER DATA
220 REM *****
230 DATA 127,8,28,28,28,8,127

```

The key to working with printer graphics is to experiment! Try designs with different sized matrixes, make your own type faces or whatever you want. You are now in control!

SUMMARY

When you got your printer, you may have thought the only thing you could print was text in the same way a typewriter does; however, as we saw, that was just the beginning. Besides printing text it is possible to generate different style type faces, position the text wherever you want and even print graphics. Not only can you print the graphics from the keyboard, you can also create your own printer graphics. Typewriters just cannot do that!

The secret to using printers with your TI-99/4A is the CHR\$ function. In some ways CHR\$s are used as ASCII code in exactly the same way as they are when output is to the screen, but in other ways they are used either as special printer functions or within certain sequences to produce printouts. Unfortunately it is not possible to simply access your printer and have it automatically put what's on the screen onto paper. By planning your program around output to the printer, just about anything printed to the screen can be printed to your printer.

CHAPTER 10

Program Hints and Help

Introduction

Well, here we are at the last chapter. We've covered most of the commands used for programming in BASIC on the TI-99/4A as well as many tricks of the trade. However, if you are seriously interested in learning more about your computer and using it to its full capacity, there is more to learn. In fact, this last chapter is intended to give you some direction beyond the scope of this book.

First, we will introduce you to the best thing since silicon — TI-99/4A User Groups. These are groups who have interests in maximizing their computer's use. Second, I would like to suggest some periodicals with which you can learn more about your TI-99/4A computer. Third, we will examine some languages other than BASIC that you can use on your TI-99/4A. BASIC has many advantages, but like all computer languages it has its limitations and you should know what else is available.

Next, we will examine some more programs. There will be listings of programs that you may find useful, fun or both. The ones included were chosen to show you some applications of what we have learned in the previous nine chapters, to enhance what you already know. Then we will look at different types of programs you can purchase. These are programs written by professional programmers to do everything from making your own programming simpler to keeping track of your taxes. Finally, we will examine some hardware peripherals to enhance your TI-99/4A.

TI-99/4A User Groups

Of all of the things you can do when you get your TI-99/4A, the most helpful, economical and useful is joining a TI-99/4A User Group. Not only will you meet a great group of people with TI-99/4A computers, but you will learn how to program and generally what to do and what not to do with your computer. The club in your area may either be one dedicated exclusively to TI computers, or it may be a general one with lots of different computers.

Usually the best way to contact your TI-99/4A User Group is through local computer or software stores. Often stores selling TI-99/4A computers and/or software will have application forms and some even serve as the meeting site for the clubs. Other microcomputer clubs in your area may also have TI-99/4A users in them, so if there is not a TEXAS INSTRUMENTS club, join a general computer group. The help you will get will be worth it.

To start your own TI-99/4A User group, post a notice and meeting time and site in your local computer store. Write to:

Users Group Editor
99'er Home Computer Magazine
1500 Valley River Drive - Suite 250
Eugene, OR 97401

and ask them to publish a notice that you want to start a TI-99/4A club in your area. Your club will then be listed in the 99'er Home Computer magazine in their "Group Grapevine" column and TI users in your area will soon join up.

Another way to get in touch with fellow TI-99/4A users is via a TI MODEM. Dial up the computer bulletin boards in your area and look for messages pertaining to TI-99/4As. Usually you can get in contact with other users very quickly this way. (Ask for the PMS (Public Message System) numbers at your local computer store). If you don't see any references to the TI-99/4A, leave a message for people to get in touch with you.

Related to local user groups is the 99/4A Program Exchange, P.O. Box 3242, Torrance, CA 90510. This international user group will give members five programs in exchange for one. Life-time membership is \$10, and the group has a library of over 600 programs available either for trade or sale. Similarly, Luv-Tronics User Group, 1111 Park Ave., Suite 303, Baltimore, MD 21201 (301) 523-5820, has a similar organization and discounts on TI commercial software as well.

TI-99/4A Magazines

There are several periodicals with information about the TI-99/4A. Some microcomputer magazines are general and others are for the TI-99/4A only. When you're first starting, it is a good idea to stick with the ones dedicated to the TI-99/4A since there are different versions of BASIC for non-TI-99/4A computers. When you become more experienced, you can choose your own, but to get started there are several good ones with articles exclusively on the TI-99/4A. These are as follows:

99'er Home Computer Magazine
1500 Valley River Drive - Suite 250
Eugene, OR 97401

The *99'er* is the main periodical dedicated exclusively to the TI. Its articles and programs provide a wealth of information about your computer and each issue has several excellent programs you can key copy. In addition, there are ways to enhance your computer's performance, ways to save money and generally get the most out of your TI-99/4A. It contains programs for both the standard BASIC as well as the Extended BASIC. You will also be able to find a wide range of programs, peripherals and services in the reviews and advertisements. Subscriptions are \$25 for one year.

COMPUTE!
P.O. Box 5406
Greensboro, NC 27403

COMPUTE! is not dedicated to Texas Instruments computers, let alone the TI-99/4A, but it generally has one or more articles on the TI-99/4A in each issue. More than most other general computer magazines, **COMPUTE!** will provide you with programs and programming techniques that can be applied to your computer. Additionally, it has several general articles on programming, hardware and software which you will find useful. Finally, there are a good deal of bargains on software and peripherals to be found in the magazine. For beginners, there is an excellent tutorial series for the TI. Subscriptions are \$20 for 12 issues.



Other Useful Publications

In addition to the above three magazines, there are several others that you may find useful. Publications such as *Creative Computing*, *Byte*, *Interface Age*, *Popular Computing* and *Personal Computing* all have had articles about the TI-99/4A. The best thing to do is go through the table of contents in the various computer magazines in your local computer store. This will tell you at a glance if there are any articles or programs for the TI-99/4A. As more and more clubs begin springing up, club newsletters can often be an invaluable source of good tips and

programs for your computer, and they are a resource that should not be overlooked.

TI-99/4A Speaks Many Languages

Besides BASIC, your computer can be programmed and can run programs in several other languages. In some cases, special hardware devices are required to run the languages, and there is special software required as well. We'll look at some of these other languages.

ASSEMBLY LANGUAGE

Assembly language is a low level language, close to the heart of your computer. It is quite a bit faster than BASIC and virtually every other language we will discuss. To write in assembly language, it is necessary to have a monitor or assembler to enter code. This language gives you far more control over your TI-99/4A than BASIC, but it is more difficult to learn and a program takes more instructions to operate than BASIC. (The object code is more compact, taking up fewer sectors on your disk.) For the TI-99/4A, Texas Instruments makes an Editor-Assembler, requiring 32K and a disk drive. Also, M.K Eckhaus, P.O. Box 1079, Elgin, IL 60120 has the MAXimum Assembler for the Mini-Memory Module on cassette for only \$25. A third assembler, also on cassette, is the DOW EDITOR/ASSEMBLER, working with the TI Mini-Memory Module. Available from John T. Dow, 6360 Caton, Pittsburgh, PA 15217.

There is not much available for the TI to teach you how to program in assembly language. At this time, the Texas Instrument's *EDITOR/ASSEMBLER MANUAL*) is about the only publication you can get that will work with the TI opcodes. *NOTE: There are several books available on assembly language programming, but they are not for the type of microprocessor used in the TI.* In some issues of the 99'er there are assembly language tutorials (April, 1983 issue for example), and if you look around you may be able to find more. User groups can be a big help when it comes to finding this kind of information.



HIGH AND LOW LEVEL LANGUAGES

When computer people talk of high and low level languages, think of high level as being close to talking in normal English and low level in terms of machine language, e.g., binary and hexadecimal. Assembly language is a low level language, one notch above machine level. The other languages we will discuss are high level.

PASCAL

Pascal is a high level language originally developed for teaching students structured programming. It is faster than BASIC, but is not as difficult to master as assembly language. It is probably the most popular high level language next to BASIC.

You will find different versions of Pascal, but the language is fairly well standardized so that whatever version of Pascal you purchase will work with just about any Pascal program. To learn how to program in Pascal, there are several books available, the following having been found to be among the best:

ELEMENTARY PASCAL: LEARNING TO PROGRAM YOUR COMPUTER IN PASCAL WITH SHERLOCK HOLMES. By Henry Ledgard and Andrew Singer. (New York: Vintage Books.) This is a fun way to learn Pascal since the authors use Sherlock Holmes type mysteries to be solved with Pascal. It is based on the draft standard version for Pascal called X3J9/81-003 and may be slightly different from the version you have, but only slightly so.

PASCAL FROM BASIC. By Peter Brown. (Reading, MA: Addison-Wesley, 1982). If you understand BASIC, this book will help you make the transition from BASIC to Pascal. It is written with the Pascal novice in mind but assumes the reader understands BASIC.

FORTH

FORTH is a very fast high level language, developed to create programs which are almost as fast as assembly language but take less time to program. Faster than Pascal, Basic, Fortran, Cobol, and virtually every other high-level language, FORTH is programmed by defining "words" which execute routines. New words incorporate previously defined words into FORTH programs. The best part of FORTH is that several versions are public domain. The Fig (FORTH Interest Group) FORTH version is in the public domain, and if you are handy with assembly programming, you might even be able to install your own. There are FORTH vendors who have FORTH for the TI-99/4A. One version recommended is:

FORTH
Wycove Systems Limited
P.O. Box 1105
Dartmouth, Nova Scotia
B2Y-4B8 CANADA

This FORTH requires one of the following modules: Editor/Assembler, Minimemory or Extended Basic, available on disk or cassette. \$40 and an additional \$10 for source code. It is fast, compact and can be used for professional program development.

The best source to learn about what is available is through the publication, *FORTH Dimensions* (see below) and your magazines where TI-99/4A products are advertised.

Good books on learning FORTH are only just now becoming available. For learning FORTH, the following are recommended:

FORTH PROGRAMMING by Leo J. Scanlon (Indianapolis: Howard S. Sams & Co., 1982). This book uses the FORTH-79 and fig-FORTH models as standards, thereby providing the user with the most widely distributed versions of FORTH. This a well organized and clear presentation of FORTH.

STARTING FORTH by Leo Brodie (Englewood Cliffs: Prentice-Hall). Well written and illustrated work on FORTH for beginners. Uses a combination of words from Fig, 79-Standard and polyFORTH.

POCKET GUIDE TO FORTH by Linda Baker and Mitch Derick. (Reading, Mass.: Addison-Wesley, 1983). This is a handy alphabetical reference to the FORTH vocabulary and a good explanation of the structure of FORTH. It is good for beginners since each FORTH instruction is explained clearly and is easy to find; however, it should be considered a supplement to one of the above books.

FORTH Dimensions. Journal of FORTH INTEREST GROUP. P.O. Box 1105, San Carlos, CA 94070. This periodical has numerous articles on FORTH and tutorial columns for persons seriously interested in learning the language.

LOGO

This language is for children. It was developed primarily as a teaching tool and it is very simple to use, especially with graphics. One version of this language available for the TI-99/4A is, TI LOGO from Texas Instruments. For a first programming language for children, LOGO is highly recommended. 99'er magazine runs an excellent column, "LOGO TIMES", to help you get acquainted with the language.

EXTENDED BASIC

Finally, if you find that programming in BASIC is most suitable for you, but you would like to do more with it, you will be definitely interested in Extended BASIC for the TI-99/4A. The language allows you to access the various memory locations directly and give you more program control. With music and graphic "sprites", you will be able to enhance your BASIC programs in ways that cannot be done with the standard BASIC that comes with your computer. At the same time, all the programs you have for your standard BASIC will work with Extended BASIC. It is available from TI.

Sort Routines

These programs will sort strings for you. The first uses the "Bubble Sort" algorithm which is good for short and partially sorted lists. It is simple, since all it does is to compare two strings (or numbers) and swap them if the first is larger than the second. It "bubbles up" the first word in the list from the bottom; however, it is relatively slow. The second sort, known as the "Shell" or "Shell-Metzer" sort uses a more efficient

algorithm and is a great deal faster. Compare the speeds of the two sorts and you will see the importance of good algorithms in your programs.

Bubble Sort

```
10 REM *****
20 REM BUBBLE SORT
30 REM *****
40 CALL CLEAR
50 DIM W$(50)
60 INPUT "NUMBER OF WORDS ":N
70 FOR X=1 TO N
80 INPUT "WORD=> ": W$(X)
90 NEXT X
100 REM *****
110 REM SORT STRINGS
120 REM *****
130 TOP = N-1
140 FLIP=0
150 FOR X=1 TO TOP
160 IF W$(X)<=W$(X+1) THEN 220
170 WW$=W$(X)
180 W$(X)=W$(X+1)
190 W$(X+1)=WW$
200 FLIP=1
210 TOP=X
220 NEXT X
230 IF FLIP=1 THEN 140
300 REM *****
310 REM OUTPUT SORTED LIST
320 REM *****
330 CALL CLEAR
340 FOR X=1 TO N
350 PRINT W$(X)
360 NEXT X
```

Shell Sort

```
10 REM *****
20 REM SHELL SORT
30 REM *****
40 CALL CLEAR
50 DIM W$(50)
60 INPUT "HOW MANY WORDS ":N
70 FOR X=1 TO N
80 INPUT "WORD=> ":W$(X)
90 NEXT X
100 REM *****
110 REM SORT LIST
120 REM *****
130 Y=1
140 Y=2*Y
150 IF Y<=N THEN 140
160 Y=INT(Y/2)
170 IF Y=0 THEN 300
180 FOR X = 1 TO N-Y
190 Z=X
200 K=Y+Z
210 IF W$(Z)<=W$(K) THEN 270
220 WW$=W$(Z)
230 W$(Z)=W$(K)
240 W$(K)=WW$
250 Z=Z-Y
260 IF Z>0 THEN 200
270 NEXT X
280 GOTO 160
300 REM *****
310 REM OUTPUT SORTED LIST
320 REM *****
330 CALL CLEAR
340 FOR X=1 TO N
350 PRINT W$(X)
360 NEXT X
```


Utility Programs

What's A Utility?

Utility programs are those which help you program or access different parts of your computer. To a large extent, many utilities that you have to buy for other computers are built in your TI. For example, the automatic line numbering and re-numbering commands (NUMBER and RESEQUENCE) are built-in utilities. Likewise, Extended BASIC has several built-in utilities for helping you develop programs. One utility we have not yet examined is TRACE and UNTRACE. Load the BUBBLE SORT program into memory, and before entering RUN, enter the command TRACE. Now enter RUN. When you do, your screen will show:

```
<50><60> NUMBER OF WORDS
```

The numbers represent the executed line numbers. After you enter your list of words, the screen will fill with numbers as your program is executed, showing the lines through which the program moves. The BUBBLE SORT program is a good one to see how TRACE works since it goes through an elaborate loop. Try it also with SHELL SORT to see the differences. TRACE is a handy debugging utility, and if you cannot figure out why a certain bug is in your program, TRACE will help you find it. To turn off TRACE, enter UNTRACE.

Usually TI user groups have public domain (FREE!) utility programs available. Check with your local user group to find out what utilities they have in their library and which ones are the most useful.

Word Processors

Your TI-99/4A computer can be turned into a first class word processor with a word processing program. Word processors turn your computer into a super typewriter. They can do everything from moving blocks of text to finding spelling mistakes. Editing and making changes is a snap; once you get

used to writing with a word processor, you'll never go back to a typewriter again. This book was written with a word processor and it took a fraction of the time a typewriter would have taken. (Believe me, I've written 10 books with a typewriter!)

There are some limitations with word processors. First, the TI-99/4A screen displays only 28 columns. Since the standard page size is 80 columns, this bothers some people since what appears on the written page is different from what appears on the screen. However, since I write material which will be printed out in everything from 20 to 132 columns, the 28 columns do not bother me. To give you some help in making up your mind about what word processor you need, the following are some features you might want to look for:

1. Find/Replace.

Will find any string in your text and/or find and replace any one string with another string. Good for correcting spelling errors and locating sections of text to be repaired.

2. Block Moves.

Will move blocks of text from one place to another (e.g., move a paragraph from the middle to end of document). Extremely valuable editing tool.

3. Link Files.

Automatically links files on disks. Very important for longer documents and for linking shorter standardized documents.

4. Line/Screen Oriented Editing.

Line oriented editing requires locating the beginning of a line of text and then editing from that point. Screen oriented editing allows editing to begin from anywhere on the screen. The latter form of editing is important for long documents and where a good deal of editing of large files is normally required.



5. Automatic Page Numbering.

Pages are automatically numbered without having to determine page breaks in writing text.

6. Embedded Code.

In word processors this enables the user to send special instructions directly to the printer for changing tabs, printing special characters on the printer and doing other things to the printed text without having to set the parameters beforehand and/or having the ability to override set parameters.

These are just a few of the things to look for in word processors. As a rule of thumb, the more a word processor can do, the more it costs. If you only want to write letters and short documents, there is little need to buy an expensive word processor. However, if you are writing longer, more complex and a wider variety of documents, the investment in a more sophisticated word processor is well worth the added cost. If you have specialized needs (e.g., producing billings forms), you will want to look for those features in a word processor. Therefore, while a word processor may not do certain things, it may be

just what you want for your special applications. As with other software, get a thorough demonstration of any word processor on a TI-99/4A *before* laying out your hard earned cash. The TI WRITER from Texas Instruments was made for the TI-99/4A. It is a cartridge and disk combination requiring 32K expanded RAM along with a disk drive, RS232 module and printer. Compare it with others your dealer may have available for the TI-99/4A and then make your choice on the basis of what you like best. The following are some other word processors you should consider:

TEXTIGER \$59.95
24433 Hawthorne Blvd.
Torrance, CA 90505
(213) 378-9286

Requires Extended BASIC, printer and disk drive or cassette unit. There are different versions for the TI-99/4 and TI-99/4A.

LETTER WRITER \$39.95
Memory Devices
5014 Hwy. 29
Lilburn, GA 30247

Requires disk drive. Recommended for short documents.

TYPWRITER \$32.00 (cassette) \$35.00 (diskette)
Extended Software Company
11987 Cedar creek Drive
Cincinnati, OH 45240

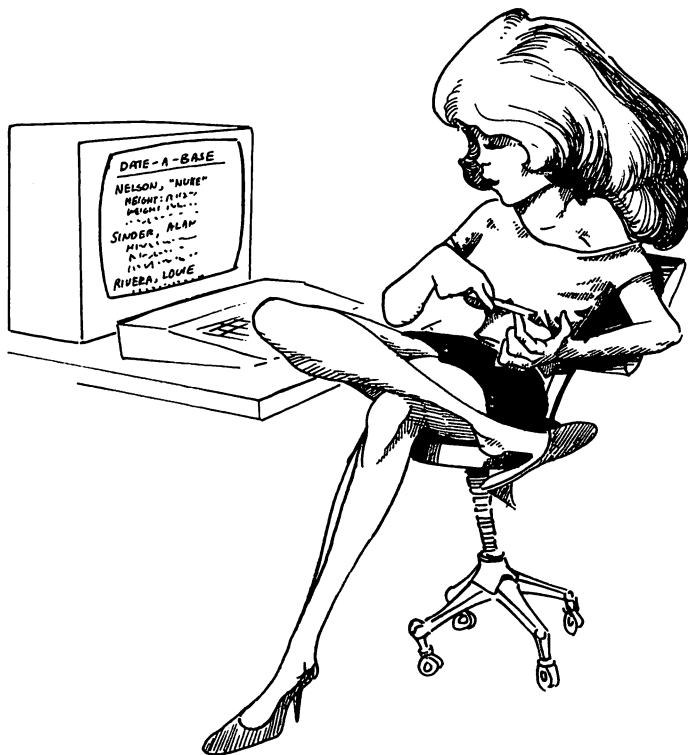
This word processor requires no special equipment other than a cassette unit or disk drive.

As a cautionary note, word processors do take a bit of time to learn to use effectively. With most word processors it is possible to start writing text immediately, but in order to use all the features effectively, some practice is required. One of the strange outcomes of this is that once a user learns all of the techniques of a certain word processor, he or she will swear it is the best there is! Therefore, avoid arguments about the best word processor — it's like arguing politics and religion.

Data Base Programs

When you need a program for creating and storing information, a "data base" program is required. Essentially, data base programs are either sequential or random access files. When you use one, all you have to do is to use the pre-defined fields provided or create fields. For example, a user may want to keep a data base of customers. In addition to having fields for name and address, the user may want fields for the specific type of product the customer buys, dates of last purchase, how much money is owed, date of last payment, etc.

Probably more than most other packages, data base programs should be examined carefully before being purchased. Some of the more expensive data bases can be used with virtually any kind of application but, for example, if you're going



to be using your data base only to keep a list of names and addresses to print out mailing labels, a data base program designed to do that one thing will usually do it better and for a lot less money than a more complex one. On the other hand, if your needs are varied and involve sophisticated report generation and changing record fields, then do not expect a simple, specialized program to do the job. TI's PERSONAL RECORD KEEPING is a data base program in a module pack for keeping track of names and addresses, phone numbers, inventories and other general purpose lists. It works with both cassette tape and diskettes storage systems; however, for really serious data base work, a disk drive is crucial. Another general data base program, on disk and requiring Extended BASIC, is EASYDATA from Ayers Computer Products, 1619 Geyser Circle, Antioch, CA. 94509 (415) 757-1124. It is relatively inexpensive at \$29.95, but serves as a good general data base program. For more specific database packages, you might want to look at MAIL-OUT and INVENTORY CONTROL from Memory Devices, 5014 Hwy. 29, Liburn, GA 30247. Both programs require a disk drive and are limited to specific functions, but if those are the functions you need, then they may be more appropriate for you than the more general programs. Finally, check with your club's library of public domain software. They may have excellent data base programs available and a lot less expensive!

Business Programs

Business programs have such a wide variety of functions that it is best to start with a specific business need and see if there is a program which will meet that need. On the other hand there are general business programs which are applicable to many different businesses. Specific business programs include ones that deal only with single areas such as real estate, stock transactions and hospital nutritional planning. More general programs include "General Ledger," "Financial Planning," and, as discussed above, data base programs.

Unfortunately, business people often spend far too much money for systems which do not work. They believe that if software and hardware costs a lot of money then it must be

better than a less expensive simpler system. This thinking is based upon a "You Get What You Pay For" mentality and it leads to systems which are not used at all. Here is where a good dealer or consultant comes in handy. First, since computers are getting more sophisticated and less expensive, often you do not "Get What You Pay For" when purchasing a big expensive one. Often all the business person ends up with is a dinosaur system which is outmoded, too big and too expensive for the needs. Some computer dealers specialize in helping the business person. They will help set up the needed system in your place of business, help train office personnel and provide ongoing support. These dealers will charge top dollar for your system and supporting software, as opposed to the discount dealers and mail order firms; however, if you have any problems you will have someone who will come and help you out. Since the TI-99/4A is so inexpensive to begin with, the extra money spent on buying from a business supportive dealer is well worth the little extra cost. Alternatively, there are several consultants for setting up your system. If you use a consultant, get one who is an independent without any connection to a vested interest in selling computers. Contact one through your phone book and tell him you want to set up a TI-99/4A system in your office and let him know exactly what your needs are. If he is familiar with your system, he will know the available software and peripherals you need. If the TI-99/4A simply will not do what you want or will not do it in an optimal manner, he may recommend another system. If that occurs, first check with another consultant to see if the first one knows what he/she is doing before re-investing.

I do not mean to sound cynical, but I have encountered too many unhappy business people who bought the wrong system for their needs. One businessman said he paid \$14,000 for a computer system that never did work for his requirements and finally bought a microcomputer system for about a tenth of the price and everything worked out fine. This does not mean that a business may not require an expensive computer to handle certain business functions and the TI-99/4A certainly has limitations; however, before you buy any system, make sure it does what you want and have it shown to you working in the manner that suits your needs. Often you will find that

the less expensive new micros like the TI-99/4A will actually work better than big costly machines. (TI does make another microcomputer designed specifically for business, and it will do a better overall job than your TI-99/4A, but if you already have a TI-99/4A, see what it can do first!)

The following are some different business programs available for your computer. Check with other TI-99/4A users who have the programs in their businesses if possible to make sure they will meet your business requirements.

Futura Software
Ehniger Associates, Inc.
P.O. Box 5581
Fort Worth, TX 76108
(817) 246-6536

1. Accounts Payable \$149.95*
2. Accounts Payable \$149.95*
3. Billing \$149.95*
4. General Ledger \$149.95*
5. Amorization Schedule \$49.95 (C) \$59.95 (D)
6. Non-Profit Organization Income and Expense Report \$49.95 (C) \$59.95 (D)
7. Personal Income and Expense Record-Keeping \$49.95 (C) \$59.95 (D)

* Requires Extended Basic, 32K memory, RS232 interfaced printer.

SA2 Software
P.O. Box 2465
Naperville, IL 60565

1. Monthly Budget\$ Master (C) \$12 (D) \$14
2. Income Tax Planner (C) \$12 (D) \$14

* Both for \$18/\$22

Memory Devices
5014 Hwy. 29

Liburn, GA 30247

1. Accounting Ledger \$39.95 (D)

The above sampling should give you a general idea of what is available for business in different price ranges; however, there is another alternative - write your own business program! For a novice programmer it may seem like a waste of time to write a program for business when one already exists; however, since necessity is the mother of invention, if your *exact* needs are not met by professionally produced software, give it a try yourself. All you have to lose is the enjoyable time spent with your computer and you can gain a valuable business tool.

Graphics Packages

In our chapter on graphics we discussed some of the TI-99/4A's capabilities with graphics. Certain uses require either highly advanced programming skills or a good graphics package. For example, it is possible to draw on the screen in hi-resolution graphics, just as you would with a pallet. The pictures produced can then be saved to disk or tape or printed out to your printer. Also, sprite developers, for producing different sprite characters are available. These programs allow you to concentrate on the graphics themselves rather than the programming techniques necessary to produce them.

An inexpensive program for creating graphics is **COMPUTERIZED CRAYOLA** from Fox Valley Software, 4954 Lori Land, Elgin, IL. 60120. With this program you can easily create graphics that would otherwise require a lot of programming. It costs only \$14.95 and requires a cassette unit to run. It's a good starting point. Similarly, **GRAPHICS PACKAGE** from Norton Software, P.O. Box 575, Picton, Ontario, K0K 2T0, Canada, will allow you to easily draw high-resolution graphics on your TI-99/4A. For enhancing your own programming skills with graphics, if you decide to go to Extended Basic, there is the "Smart Programming Guide For Sprites." This manual explains how to program sprites clearly using **CALL PEEK** and other Extended Basic commands to get the most out of your graphics. The best part is the price - \$5.95. It's available from Millers Graphics, 1475 W. Cypress, San Dimas, CA 91773. Related to the general work with graphics, an

assembler along with Extended Basic will allow you to do things with your joysticks and graphics that are impossible from the standard BASIC. For professional game development, Extended BASIC, an assembler and a more powerful language, such as FORTH, can lead to a new career!

Hardware

The TI-99/4A is "expandable." That means you can add various attachments to it to make it do more than it does normally. The easiest way to do this is with the TI Expansion System since all you have to do is insert the added hardware into the Expansion System. Only a single connection goes from the Expansion System to the computer, making for a simple and neat interface. In some cases all you may need is a single connection to a single peripheral and purchasing the Expansion System would not be worth the cost. For example, all you might need is a printer. Since there is a wealth of inexpensive parallel printers on the market, you can save on both the printer and RS232 card required with the TI-99/4A printer. Doryt Systems, Inc., (14 Glen Street, Glen Cove, N.Y. 11542 (516) 676-7950), sells a PARAPRINT 18A that will interface any parallel printer to the TI-99/4A for \$105. So for about \$350, you can get the printer and the interface for your computer. For even less than that, Alphacom, Inc., 2323 South Bascom Ave., Campbell, CA 95008, (408) 559-8000, has a printer for \$179.95 and cable interfaces for as little as \$29.95. If you have a serial printer, Model Masters (2512B E. Fender Ave., Fullerton, CA 92631) sells a product called JOYPRINT for \$59.95. It interfaces with any serial printer. That is a considerable savings over buying the Expansion System, RS232 and the printer.

Probably the most important addition to your TI-99/4A will be memory expansion modules. With 16K of RAM there is a surprising amount you can do, but with certain applications, such as data base programs and word processing, you will need the added memory. TI has 32K MEMORY EXPANSION modules that slip into the Expansion System. If you do not have the Expansion System, Intellitec Computer Systems

(2337 Bonanza Court, Riverton, Utah 84065 (801) 254-2333) have 32K memory add-ons that plug into the right side of your computer. The same company makes a combination RS232/32K module that also plugs into the right side port. If you really want a lot of memory, Foundation (74 Claire Way, Tiburon, CA 94920) makes a 128K memory card that plugs into the Expansion System.

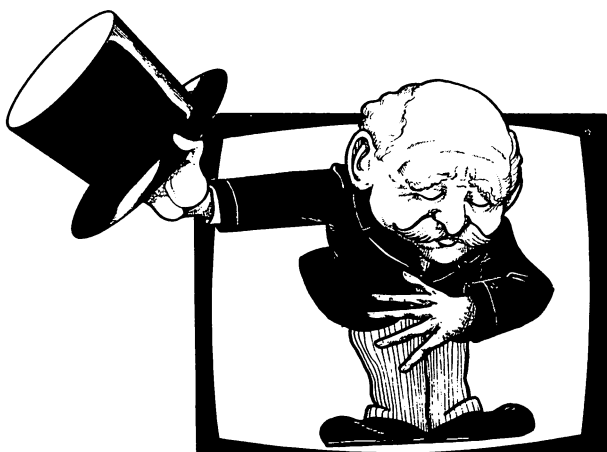
A final product you may be interested in purchasing is the TEX-SETTE Adapter. It allows you to use any compatible cassette tape recorder with your TI. It is available for \$5.95 from 99'er-WARE, P.O. Box 5537, Eugene, Oregon 97405 (503) 485-8796. So if you already have a cassette recorder and you don't want to have to buy another one for your computer, you can save a bundle and still have cassette storage for your TI.

Like software, before you purchase an interface or peripheral, make sure it works with your computer! Unfortunately, many hardware attachments come with such poor documentation that without someone to show you how to work it, it is almost impossible to get them to operate properly.

SUMMARY

The most important thing to understand from this last chapter is that we have only scratched the surface of what is available for the TI-99/4A computer. There is far too much information to cover than what we could squeeze into one chapter and, as you come to know your TI-99/4A, you will find that the choice of software and peripherals is limited only by the confusion in making up your mind. There were other items for the TI-99/4A that came to mind, but this chapter and book would have never ended were I to indulge myself and keep prattling on. The software and hardware I suggested were based on personal preferences; I would suggest that you choose on the basis of your own needs and preferences, not mine. Think of the items mentioned as a random sampling of what one user found to be useful and then after your own sampling, examination and testing, get exactly what you need.

As you end this book, you should have a beginning level understanding of your computer's ability. Whether you use it for a single function or are a dedicated hacker, it is important that you understand the scope of its capacity to help you in your work, education and play. It is not a monstrous electronic mystery, but rather a tool to help you in various ways. You may not understand exactly how it operates, but you probably do not understand everything about how your TV set operates either, yet that never prevented you from watching the evening news. With your computer you make the "news" on your TV.



Farewell
and Good Luck !!

TI-99/4A COMMAND EXAMPLES

This glossary is arranged in alphabetical order. The examples are set up to show you how to use the commands and their proper syntax. In some cases when a command has different contexts of usage, more than a single example will be used. Some examples are given in the Immediate mode and some in the Program mode <those with line numbers> and some with both. For clarification, results are given in some examples to show what a particular configuration would create. Some commands of specialized use that were not covered in the text have been included here for a more complete glossary.

ABS() Gives the absolute value of a number or variable.

```
PRINT ABS(-123.45)  
(Result) 123.45
```

ASC() Returns ASCII value of first character in string.

```
PRINT ASC ("W")
```

or

```
A$ = "TI-99/4A"  
PRINT ASC(A$)
```

ATN() Returns arctangent of number or variable.

```
PRINT ATN (123)  
(Result) 1.562666425
```

CALL CHAR (C,"HEX") Replaces the ASCII value C with the character represented by the hexadecimal numbers "HEX".

```
100 CALL CHAR (65, "8199BDE7FFBD9981")
```

CALL CLEAR Clears screen and places cursor in lower left hand corner of screen. It does not clear memory or variables.

```
10 CALL CLEAR
```

CALL COLOR (CG,F,B) Establishes color for characters in character group CG, in a foreground color of F and background color of B.

80 CALL COLOR (4,2,11)

CALL GCHAR (R,C,V) Reads row R, column C of ASCII value into variable V.

50 CALL GCHAR (20,30,X)

60 PRINT X

(Result) Whatever character was in Row 20, Column 30, its ASCII value will be printed to the screen.

CALL HCHAR (R,C,A,(X)) Puts the character for ASCII value A, in row R, column C, repeated (optionally) X number of times horizontally to the right.

10 CALL HCHAR (12,16,65)

or

10 CALL HCHAR (1,1,77,30)

CALL JOYST (J,X,Y) Joystick number J (1 or 2) or key unit J, value is stored as 0, 4 or -4 in variables X, and Y depending on position of stick or key pressed.

40 CALL JOYST (1,X,Y)

50 IF X=4 THEN 200

60 IF X=-4 THEN 300

CALL KEY (N,K,C) Checks to see what key in key unit N has been pressed. ASCII values is stored in K and key status in variable C.

60 CALL KEY (0,K,C)

CALL SCREEN (C) Screen color is changed to color C.

50 CALL SCREEN (10)

CALL SOUND (D,F1,V1,F2,V2,F3,V3,F4,V4) Creates sound of duration D, frequency F and volume V.

90 CALL SOUND (100,150,3)
100 CALL SOUND (50,120,0,145,1,156,2,181,2)

CALL VCHAR (R,C,A,(X)) Puts the character for ASCII value A, in row R, column C, repeated (optionally) X number of times vertically downwards.

230 CALL VCHAR (10,20,65)
240 CALL VCHAR (1,30,76,20)

CHR\$() Returns the character with a given decimal value.

PRINT CHR\$(65)
(Result) A

CLOSE # Closes channel to device or file.

210 CLOSE #7
220 REM 7 IS FILE NUMBER OF DEVICE OR FILE
BEING CLOSED.

CONTINUE Continue program after a BREAK line.

BREAK 100
80 FOR X = 1 TO 4
90 PRINT X
100 NEXT X
(Result)
RUN
1
* BREAKPOINT AT 100
CONTINUE
2
3
4

COS() Returns to cosine of variable or number.

PRINT COS(123)
(Result) -.8879689067

DATA Strings or numbers to be read with READ statement.

1000 DATA 2, 345, HELLO, "WALK"

DEF Defines a substitute function for real variable.

```
40 DEF SX = 10 * 10
50 PRINT 100/SX
(Result) 1 when RUN
```

DIM Allocates maximum range of array.

```
130 DIM A$ (100)
```

DISPLAY Works exactly like PRINT when output is to screen. It will not work with other devices for output.

```
10 A$="SHOW ME"
20 DISPLAY A$
```

EDIT L Line number L is brought to the screen.

```
EDIT 20
```

END Terminates running of program.

```
200 END
```

EXP(P) Returns $e=2.718281828$ to indicated power, P.

```
PRINT EXP (5)
(Result) 148.4131591
```

EOF(F) Used with disk files only. Check for End Of File in in file F from within program. 0 = not end of file, +1 = logical end of file, -1 = physical end of file.

```
100 OPEN #5: "PHONES",SEQUENTIAL,INTERNAL,
INPUT,FIXED
110 IF EOF(5) THEN 200
....
200 CLOSE #5
```


FOR Sets up beginning of FOR/NEXT loop and top limit of loop.

```
40 FOR I = 1 TO 100
```

GOSUB Branches to subroutine at given line number.

```
100 GOSUB 200
```

GOTO (or **GO TO**) Branches to given line number.

```
100 GOTO 200
```

IF/THEN/ELSE Sets up conditional logic for execution to line number only.

```
60 IF A$ = "Q" THEN 100 ELSE 200
```

INPUT Halts program execution until string or numbers entered and RETURN key is pressed. May enter message within INPUT statement.

```
90 INPUT "ENTER WORD-> ": W$(I)
100 INPUT "ENTER NUMBER -> ": A
110 PRINT "HIT 'RETURN' TO CONTINUE ";
120 INPUT R$
```

INPUT# Takes data from a previously OPENed file or device.

```
200 INPUT #1, R$(I)
```

INT() Returns the integer value of real variable or number.

```
PRINT INT (123.45)
(Result) 123
```

LEN Returns the length in terms of number of characters of a specified string.

```
A$ = "COMPUTER AWAY"
PRINT LEN(A$)
(Result) 12
```

LIST Lists program currently in memory.

LIST (Entire program)

LIST 100-140 (Range)

LIST 200- (From line to end of program)

LOG() Returns logarithm of specified number or variable.

PRINT LOG (123)

(Return) 4.81218455

or

20 G = 123

30 PRINT LOG (G)

NEW Clears program in memory.

NEW

NEXT Sets the top of the loop begun with FOR statement.

10 FOR I = 1 TO 100

20 PRINT "THIS".

30 NEXT I

NUMBER B,I Sets up automatic numbering beginning at B with increments of I. (Default to B=100, I=10)

NUMBER 10,10

OLD DEVICE.(NAME) Loads program from DEVICE. If loaded from disk, program name must be included.

OLD CS1 (Cassette load)

OLD DSK1.WARPPAR (Disk load)

ON Sets up computed GOTO and GOSUB.

190 ON A GOSUB 1000,2000,3000

OPEN #: "FN",FO,FT,M,RT Opens channel to device or file with device or file name FN, file organization FO, file type FT, mode M and record type RT.

500 OPEN #1:"CS1",SEQUENTIAL,INTERNAL,
INPUT,FIXED
600 OPEN #5: "RS232" (Opens channel to printer
via RS232.)
700 OPEN #15 : "NAMES",SEQUENTIAL,INTERNAL,
OUTPUT,FIXED
(Opens disk file named "NAMES" for writing to disk
in sequential files.)

POS (S1\$,S2\$,N) Returns the position of S2\$ in S1\$ begin-
ning at position N.

10 A\$ = "WHAT'S UP SPOCK?"
20 U\$ = "UP"
30 PRINT POS(A\$,U\$,1)
(Result) 8

PRINT Outputs string, number or variable to screen or printer.

PRINT 1;2;3; "GO"; F\$, A

PRINT# Sends output to specified OPENed device or file.
(The question mark (?) cannot be substituted when using
PRINT#.)

250 PRINT #1: NA\$(I)
or
OPEN #7:"RS232"
PRINT #7: "HELLO TI-99/4A"
(Result) Prints message HELLO TI-99/4A
to printer.

RANDOMIZE Seeds random number generator.

20 RANDOMIZE
or
20 RANDOMIZE 22

READ Enters DATA contents into variable.

10 READ A
20 READ B\$
900 DATA 5, "BATS"

REM Non-executable command. Allows remarks in program lines.

```
10 DIM A$(122)
20 REM DIMENSIONS STRING ARRAY "A$" TO 122
```

RESEQUENCE B,I Renumbers program beginning with line number B with increments of I from Immediate Mode.

```
RESEQUENCE 10,10
```

RESTORE Resets position of READ to first DATA statement.

```
10 FOR I = 1 TO 5 : READ A$(I) : NEXT
20 RESTORE
```

RESTORE # Resets position of pointer to beginning of file.

```
10 OPEN #5: "DFILE",SEQUENTIAL,INTERNAL,
INPUT,FIXED
20 INPUT #2: NAME$,AD$,CITY$
.
200 RESTORE #2
210 INPUT #2: NAME$,AD$,CITY$
```

RETURN Returns program to next line after GOSUB command

```
500 RETURN
```

RND Generates a random number greater than 0 and less than 1.

```
PRINT INT(101*RND) - Prints a whole random number between 0 and 100 inclusive.
INT((N2-N1+1)*RND)+N1 - Generates whole random numbers from N1 to N2, with N2 being the upper limit of desired numbers.
```

RUN Executes program in memory.

RUN

SAVE Records program on tape or disk.

SAVE CS1 (Tape)

SAVE DSK1.GRAPH PLOT (Disk)

SIN() Returns the sine of variable or number.

PRINT SIN(123)

(Result) -.4599034907

SGN() Returns 1 for positive number, -1 for negative number and 0 for zero.

PRINT SGN(-13); SGN(13)

(Result) -1 1

SQR() Returns the square root of variable or number.

PRINT SQR(64)

STEP Used in FOR/NEXT loop to indicate loop increments and direction ("-" for negative increment).

10 FOR I = 1 TO 50 STEP 2

20 FOR J = 88 TO 44 STEP -1

STOP Halts execution and prints line number where break occurs.

100 STOP

STR\$() Converts number/variable into string variable.

20 T = 123

30 T\$ = STR\$(T)

40 TT\$ = "\$" & T\$ & ".00"

TAB() Sets horizontal tab from within a PRINT statement.

```
PRINT TAB(20);"HERE"
```

TAN() Provides the tangent of number or variable.

```
40 T = 34
50 V = 55
60 R = T + V
70 PRINT TAN(R)
(Result) 1.685825371
```

TO Sets range separator in FOR/NEXT loop.

```
40 FOR K = 0 TO 120
```

TRACE Displays line numbers of program executed to screen during program execution.

```
TRACE
```

UNTRACE Turns off TRACE function.

```
UNTRACE
```

VAL() Used to convert string to numeric value.

```
30 H$ = "123"
40 PRINT VAL(H$)
(Result) 123
```

INDEX

A

- arrays 96
- arrow keys 35
- ASCII 130, 131
- assembly language 224
- automatic line numbering 43

B

- BASIC 38
- backups 47
- binary 164, 165
- bit graphics 163
- black and white monitor 19
- booting disks 26
- branching 82-83
- Bubble sort 230
- business programs 236
- buying diskettes 30

C

- CALL 134
- CALL CLEAR command 40
- CALL color 152, 244
- CALL GCHAR 179, 244
- CALL HCHAR 138, 139, 244
- CALL key 135
- CALL screen 152, 244
- CALL sound 143, 244
- CALL VCHAR 138, 139, 245
- CALLs with text formatting 138
- cartridge programs 33
- changing keys 36
- changing numbers to string 114
- changing strings to numbers 112
- CHR\$ function 131-134, 204-209, 245
- clearing the screen 40
- CLOSE 184, 204, 245
- color 151, 152
- color codes 153
- color monitor 19
- concatenation 115
- counters 79
- command examples 243
- computed GOSUB 91-95
- computed GOTO 91-95
- creating 187

- CTRL (control) key 35

D

- data base programs 235
- data entry 117
- data files 182
 - with cassettes 182
- data manipulation 119
- DATA statement 70
- decimal 164, 165
- deleting lines 49
- DIM statement 99, 246
- DIMension of an array 99, 246
- disk drive 16
 - controller 16
 - hook-up 16
- disk system 191
- dot matrix printer 20

E

- editor 48, 50-54
- eight-bit binary-decimal converter 217
- END command 40
- ENTER key 35
- eproms 13
- error messages 48
- Expansion system 16
- Extended BASIC 228

F

- FCTN (function) key 35
- firmware 13
- formatting text 104
- FOR/NEXT 75-77
- FORTH 226

G

- GOSUB 91-95, 247
- GOTO 91-95, 247
- graphic characters 212
- graphic number converter 216
- graphic utility 216
- graphics 151-181
 - bit graphics 163
 - CALL color 152
 - CALL screen 152

- color 151
- color codes 153
- multi-character 171
- screen graphics 151
- graphics packages 239
- green screen monitor 18

H

- hardware 13, 240
- hexadecimal 164, 165

I

- IF/THEN/ELSE 84, 247
- Immediate Mode 38
- initializing a diskette 27-29
- input 68
- input and output (I/O) 67
- INPUT# 184, 247

J

- joystick control 174-177

K

- keyboard 34, 104
 - changing keys 36
 - new meanings for old keys 36
 - special keys 35
 - arrow keys 35
 - CTRL 35
 - ENTER 35
 - FCTN 35

L

- languages 323
 - Assembly language 224
 - Extended BASIC 228
 - FORTH 226
 - LOGO 228
 - Pascal 225
- LEN command 108, 247
- LENGth of strings 108
- letter-quality printers 20
- line numbers 41
- LIST command 42
- listing a program 42
- LOADing from tape 30
- LOGO 228
- looping with FOR/NEXT 75

M

- magazines 222-223
- math operations (+, -, /, *) 54
- missiles 143-149
- MODEM 23
- monitor 17
 - hook-up 17-18
 - types of 18-19
- multi-dimensional arrays 100
- multiple character graphics 171
- music 143-149

N

- nested loops 73
- numbering systems 163-165
 - binary 164, 165
 - decimal 164, 165
 - hexadecimal 164, 165

O

- OPEN 184, 202, 248
- organizing output 122

P

- parentheses 56-67
- Pascal 225
- peripheral equipment 15
- PRINT command 38
- PRINT formatting 127
- PRINT# 184, 203, 249
- printer graphic utility 216
- printers 17, 200-219
 - before you buy 201
 - check-out 25
 - hard copies 202
 - hook-up 17-18
 - graphic utilities 216
 - graphics 212
 - purchase of 21
 - tab stops 209
 - types of 19-21
- Program Mode 39
- proms 39

R

- RAM 14
- READ statement 70

REAding in DATA 70
real variables 62
relationals 86, 95
renumbering lines 43
re-ordering precedence 56-57
RESEQUENCE command 43, 250
retrieving programs 45
ROM 14
RUNning from tape 30

S

saving 31, 44, 251
 on tape 45
screen graphics 151
scroll control 125
SEG\$ments of a string 110
sequential files 191
setting up a program 41
Shell sort 230
software 14
sort routines 228
 Bubble sort 229
 Shell sort 230
stepping 77, 251
string array 96
string formatting 107
string variables 62, 96

strings 95, 107-117
subroutines 88

T

tab stops 209
tape recorder hook-up 16
tape to disk transfer 33
text files 182
thermal printers 20-21
TV 17, 18
types of variables 62

U

unraveling strings 107
utility programs 231
user groups 221

V

variables 59
 array 59
 names of 61
 real 62
 string 63
 types of 62

W

word processors 231-235

THE ELEMENTARY TI-99/4A

YOU KNOW YOUR NEW TI-99/4A COMPUTER DOES MORE THAN PLAY GAMES BUT... HOW ARE YOU GOING TO REALIZE THE GREAT POTENTIAL OF THIS MARVELOUS MACHINE?

THIS BOOK WILL TEACH YOU TO PROGRAM YOUR COMPUTER!

Written by William B. Sanders, THE ELEMENTARY TI-99/4A is like having a friendly, cheerful, easy-going teacher at your side — gently and clearly explaining everything you want to know. Carefully leading you from point to point, this book will help you understand and program the TI-99/4A. Just open it up to any page and read a paragraph or two. Once you do, you are sure to agree this book is as fantastic and user-friendly as we say.

Ten chapters lead you step-by-step through the process of hooking up the computer, loading and saving programs, creating graphics, music, and all kinds of handy utilities. Everything is made simple! By the time you're finished, you'll be writing and using programs! Even if you're already programming, this book has lots of helpful information and will satisfy the entire family's desire to participate in the computer revolution!

Published by DATAMOST Inc., THE ELEMENTARY TI-99/4A is another in the highly successful Elementary Series.

ISBN 0-88190-247-0



DATAMOST INC.™

8943 Fullbright Ave., Chatsworth, CA 91311-2750
(213) 709-1202