A New Apple for Everyone

The Apple IIGS computer, which just became available in the fall of 1986, is a welcome addition to the Apple II series. Its color graphics and sound capabilities, faster processing time, greater memory, and wide availability of quality software make it the latest success in a new generation of home computers.

The Elementary Apple IIGS is all you'll need to become familiar and comfortable with this exciting new machine. You'll learn in easy, logical steps how to set up the computer, load software, and write your own programs. Abundantly illustrated with examples, *The Elementary Apple IIGS* contains the necessary information that will have you up and running in no time.

And before you know it, you'll progress from learning to enter short programs to using variables, strings, DATA statements, and loops within your own programs.

Here's just some of what you'll find in *The Elementary Apple IIGS*:

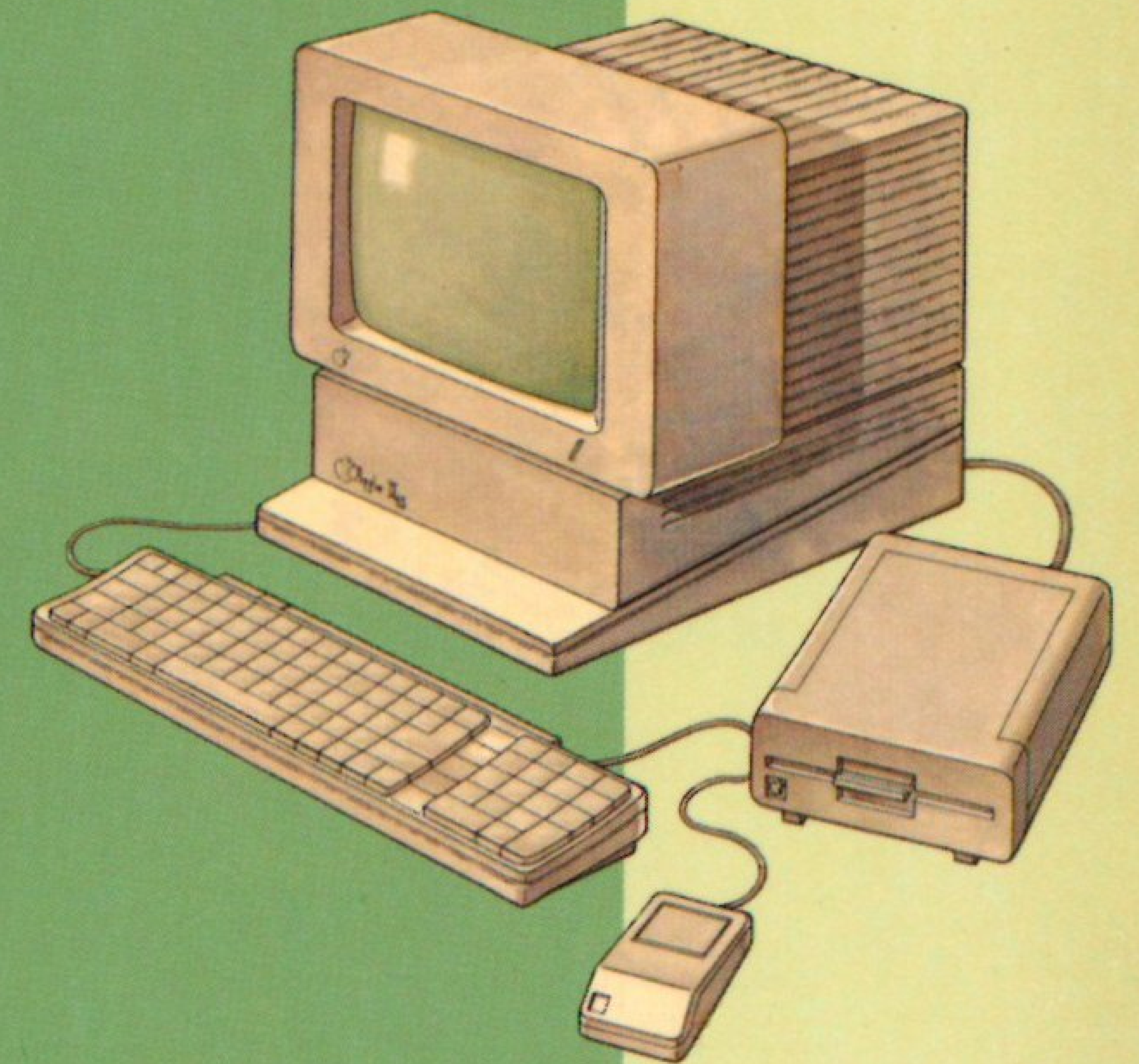
- Detailed instructions on hooking up and operating your IIGS and its peripherals
- A step-by-step guide to creating and using your own programs
- Numerous examples and clearly written tutorials for hands-on guidance in developing application programs
- Information about various types of printers, plus how to interpret printer manuals and use printer control codes
- An introduction to the built-in graphics and sound toolboxes
- Utility programs, and hints and tips for expanding your knowledge of software available for the IIGS
- Handy quick-reference appendices and much, much more

The Elementary Apple IIGS will ease your way toward understanding and using this powerful machine to its maximum capabilities. Written in the clear and concise style that has become a hallmark of COMPUTE! Publications, it's just what you need to make a practical start with this exciting new computer.

ISBN 0-87455-072-6

The Elementary APPLE IIGS

COMPUTE! Books



THE ELEMENTARY APPLE IIGS

William B. Sanders

A friendly beginner's guide to the new Apple IIGS. Learn how to use and program this powerful new computer—everything from setting up to creating graphics.

A **COMPUTE! Books** Publication $15.95

Scanned by cvxmelody
http://www.cvxmelody.net/AppleUsersGroupSydneyAppleIIDiskCollection.htm

The Elementary APPLE IIGS

William B. Sanders

COMPUTE!™ Publications, Inc. abc
Part of ABC Consumer Magazines, Inc.
One of the ABC Publishing Companies
Greensboro, North Carolina

Copyright 1986, William B. Sanders. All rights reserved.

Reproduction or translation of any part of this work beyond that permitted by Sections 107 and 108 of the United States Copyright Act without the permission of the copyright owner is unlawful.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-87455-072-6

The author and publisher have made every effort in the preparation of this book to insure the accuracy of the programs and information. However, the information and programs in this book are sold without warranty, either express or implied. Neither the author nor COMPUTE! Publications, Inc., will be liable for any damages caused or alleged to be caused directly, indirectly, incidentally, or consequentially by the programs or information in this book.

The opinions expressed in this book are solely those of the author and are not necessarily those of COMPUTE! Publications, Inc.

COMPUTE! Publications, Inc., Post Office Box 5406, Greensboro, NC 27403, (919) 275-9809, is part of ABC Consumer Magazines, Inc., one of the ABC Publishing Companies, and is not associated with any manufacturer of personal computers. Apple IIGS, ImageWriter, LaserWriter, and ProDOS are trademarks of Apple Computer, Inc. PostScript is a trademark of Adobe Systems Incorporated.

Contents

Foreword v

1. Introduction 1
2. Getting Started 31
3. Moving Along 51
4. Branching Out 69
5. Organizing the Parts 89
6. Some Advanced Topics 111
7. Using Graphics 129
8. Text Files and the Disk System 157
9. You and Your Printer 177
10. Super High-Resolution Graphics and Sound 193
11. Utility Programs, Hints, and Help 203

Appendices 227
A. Applesoft BASIC Token Chart 229
B. ASCII Characters 233
C. Hex-to-Decimal and Decimal-to-Hex Conversion 235
D. Error Messages 241
E. Glossary 245

Index 260

Foreword

Home computers have come a long way since Apple Computer was founded a decade ago. Now, Apple's introduction of the IIGS brings renewed excitement to the durable Apple II series of computers. Because the IIGS runs the whole gamut of software already available for Apple II computers, there's an enormous choice of ready-made software just waiting for you to run—not to mention the added graphics and sound power you have right inside your IIGS. In fact, *graphics* and *sound* are what *GS* stands for.

The Elementary Apple IIGS will lead you gently through the intricacies of using your new computer, from setting it up to writing accomplished, sophisticated programs that will make the machine do your bidding. Step-by-step instructions and numerous practical examples will show you exactly how to go about achieving what you want. You'll learn to organize your work so that it's easy and logical to follow—a very important consideration in creating workable programs.

One of the most exciting aspects of the IIGS is its graphics and sound capabilities, and you'll learn how to use these advanced features and become acquainted with the built-in toolboxes that offer access to them. You'll get an overview of working with sequential data files and random access files as well.

Newcomers to computing are often frustrated when they consult their printer manuals. Chapter 9 is devoted to information on various types of printers, and it also aids in interpreting printer codes so that you can generate the output you need.

Finally, you'll be introduced to some of the utility programs that can help you make your own programs operate more smoothly and efficiently. Chapter 11 also includes several type-in programs that consolidate some of the programming techniques you'll learn. The appendices provide a handy reference to Applesoft tokens, ASCII characters, hexadecimal/decimal conversions, error messages, and a glossary.

With *The Elementary Apple IIGS*, you'll make rapid progress toward being able to develop your own applications, whether your interests are educational, business- or home-oriented, or just plain fun.

1

Introduction

Introduction

The Elementary Apple IIGS is intended to help you operate your new computer, get started programming, and generally make life with your computer easier. It is not designed for professional programmers or for learning more about applications. It is the first step for beginners on the Apple IIGS. Material will be kept on an introductory level, but by the time you have finished reading, you should be able to write and use programs.

To use *The Elementary Apple IIGS* most effectively, start at the beginning and work your way through, step by step. The book has been arranged so that each part or 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 Chapter 10, which lists a number of utility programs that will help you write programs. Also, there are descriptions of programs that perform business applications, word processing, and so forth. When you're finished with this introductory chapter, it would be a good idea to glance through the programs described in the Chapter 10 to see if any of them will fit your needs while you're learning about your Apple IIGS. Depending on your interests and needs, you may find some of them useful.

First Things First

The first thing to learn about your computer is that it will not bite you. It does, however, require a certain amount of care. It's possible to destroy disks and information, but by following a few simple rules you should have no problems. We have all used sophisticated electronic equipment, such as stereos, televisions, and videotape recorders. They, too, require a certain amount of care; otherwise, there is no need to fear them.

Likewise, your computer is electronic. If you pour water or other liquids on the computer, you're likely to damage it. But use reasonable care, and go ahead and put it to use. Remember, it is impossible to write a program that will harm the

hardware (the electronic circuits) in your machine. The worst thing any program you write can do is to erase the information on a disk. Throughout this book you'll read tips about doing things the right way and the 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.

It's not necessary to learn a lot of computer jargon. However, some terminology will help you understand how your IIGS operates. As you progress, more new terms will be introduced, but for the most part the text will be in plain English. Here are a few terms you'll need to know just to get started.

Hardware and Software

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.

Software consists of the programs that tell a computer to do different things. Whatever goes into the computer's memory is software, which is analogous to the ideas processed by the human brain. Software is to computers as records are to stereos. Software operates either in random access memory (RAM) or read only memory (ROM).

You may hear people talk about expanding their *RAM.* This is the part of the computer's memory into which information is entered in the form of data and programs. The more memory you have, the larger the program and the more data that can be entered. Think of RAM as a warehouse. When you first turn on your computer, the warehouse is just about empty. As you run programs and enter information, the warehouse begins filling up. The larger the warehouse, the more information you can store there. When the warehouse is full, you have to stop.

The basic Apple IIGS comes with 256K of RAM. The *K* refers to kilobytes, or thousands of bytes. (Actually, a K is 1024 bytes, a bit more than a thousand.) You can expand your Apple's RAM to over a megabyte, or one million bytes. For now, you just need to know that in computer terms bytes are a measure of storage: the more bytes, the more room. Think of bytes as you would gallons, inches, or meters—simply a unit of measure.

ROM is a second type of computer memory. This type of memory is "locked" into your computer's chips. Your Apple IIGS's programming language, called BASIC, is stored in ROM. The difference between ROM and RAM is that whenever you

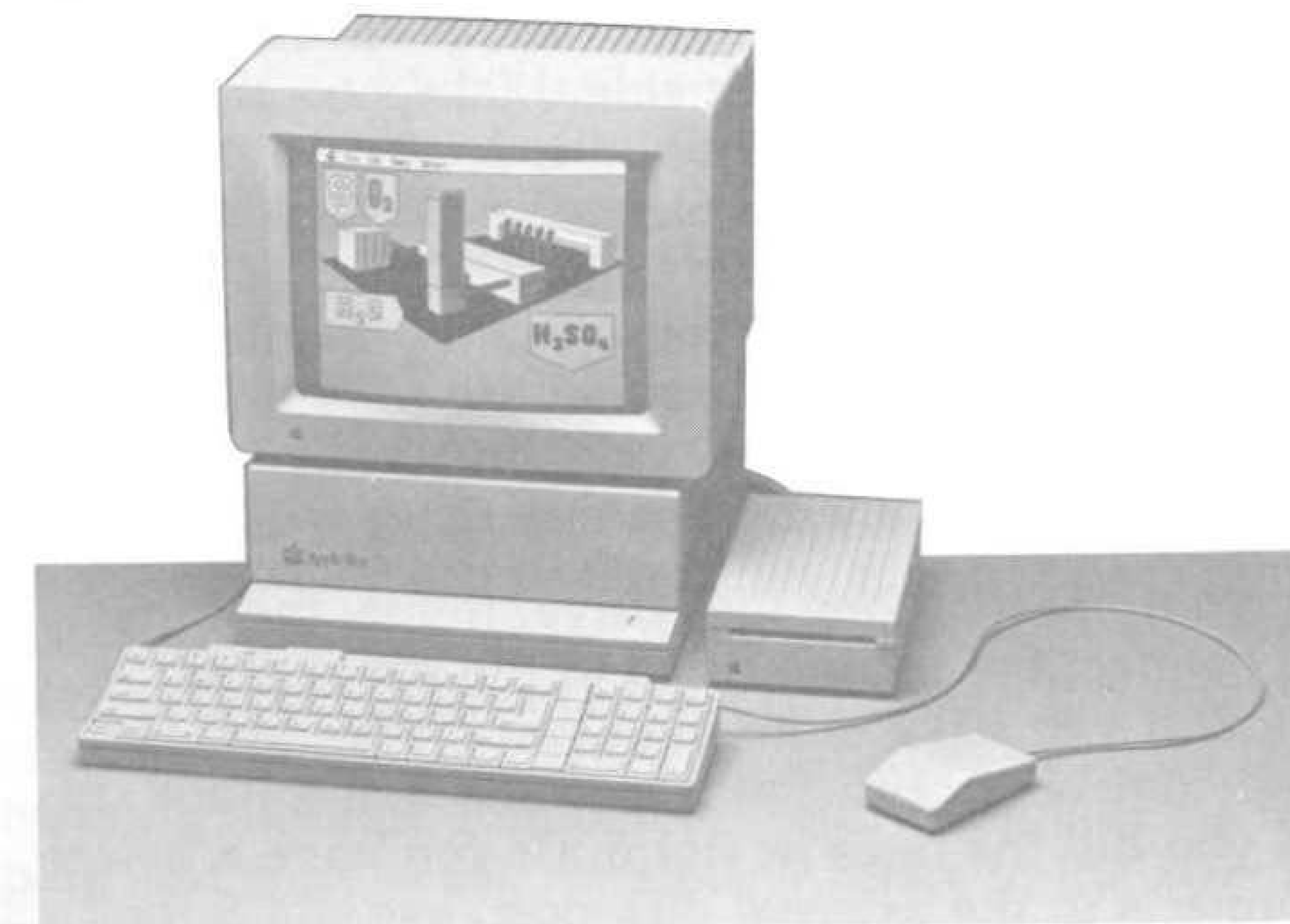
turn off your computer, all information in RAM will evaporate, but ROM will keep all of its information. Don't worry, though; you can save whatever is in RAM on disks and get it back. You'll learn how later.

Now that you know a few terms and don't fear your computer, let's get it cranked up and running. If your IIGS is already hooked up and working properly, you can skip the next section and go directly to the section "Power On."

Hooking Up Your IIGS

The *last* thing you should do after reading this section is plug in your IIGS and turn it on. Everything else should be done first. This newest Apple II gives you more than one way to connect the different parts. If this is your first Apple II, you'll probably want to get all of the latest equipment for it and plug everything into the special ports provided. On the other hand, if you have an old Apple (a IIe or older) and want to use your old interface cards to hook up your disk drive and other peripherals, you can use the slots. Of course, you may want to save money by purchasing older used equipment and interface cards, and so will use the slots instead of the external ports. Likewise, you can use a combination of slots and ports. We'll explain the simplest way to do things first.

Figure 1-1. The Apple IIGS



The Apple IIGS computer, shown here with the AppleColor RGB monitor and 5¼-inch disk drive, features 256K of RAM, high-resolution graphics, high-quality sound synthesis capabilities, and complete compatibility with existing Apple II software.

The Disk Drive

You can use either 5¼- or 3½-inch-disk drives, or even a combination of both. Having used both, I recommend getting the 3½-inch system. They're easier to handle, and the recording surface is completely enclosed so you have less chance of accidentally destroying data on the disks by touching or scraping the media. If you decide on the 5¼-inch system, get the newer kind that you can plug into the disk port.

The main advantage of the 5¼-inch system is that there are more commercial programs for the Apple II series in that format. The older versions of the 5¼-inch drives require an interface card and cable. The card goes into slot 6 inside your IIGS. You'll need to take the top off your computer, place the card in the sixth slot from the left, and run the flat ribbon cable from the card to your drive outside your Apple. (If you have both a newer drive that plugs into the disk port and an older one that goes into one of the slots, place the disk drive interface card into slot 5.)

Installing old drives. Here is a detailed explanation of how to install an older drive:

1. Find slot 6 (the second from the last slot counting from left to right as you face the keyboard). With your disk drive you should also have bought a *controller card* to which you attach the cable from your disk drive. This card goes into slot 6. Now, gently, but firmly, place the controller card into slot 6. It will fit only one way, with all of the black chips and electronic apparatus on the right side. Make sure that the card fits all the way into the slot and is level.
2. On the back of the card are two sets of pins. One set is labeled drive 1 and the other drive 2. *Carefully* place the connector on the end of the cable from your drive onto the set of pins labeled drive 1. Make sure that the pins and the holes in the connector line up. Gently, but firmly, press the connector into place. That's it. Your disk drive is hooked up.

If you have more than one disk drive, follow the same procedure to connect the second drive to the set of pins marked drive 2. If you have more than two drives, you will need a second controller card. Put it into slot 5 and hook up the drives as you did the ones in slot 6. (If you have a newer drive that connects to the external disk port, use slot 5 for your first disk drive controller card.)

Hard disks. Some of you will have hard disk systems that have multiple-megabyte capacity. Usually, you will want to have a floppy disk drive, either a 3½- or 5¼-inch system, to use in conjunction with the hard disk. It is strongly recommended that you have your Apple dealer demonstrate the best arrangement for hooking up a hard disk with a floppy system.

TV or Monitor

In order to see what's going on in your computer, you need either a TV set or a monitor. We all know what a TV set does; a monitor is essentially a TV set for your computer. Monitors are preferable to TV sets since their resolution is higher (they're clearer). However, since an outstanding feature of the IIGS is its color graphics (especially on some of the games), many people use color televisions (sometimes the family TV) with their Apples because they cost less than color monitors.

TVs come in many shapes and sizes. You can choose either a color or black-and-white set. But since not all televisions work well with your IIGS, ask first before you select one to buy. When I bought a 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 a discount house. Whatever the case, check to make sure that the TV set you purchase will work with your IIGS.

TV connection. To connect a television set to your IIGS, you will have to use an RF modulator. The modulator unit goes into the video port, secured by two screws in the unit. Attach the adapter box to the VHF antenna connectors on your television set (where the aerial normally goes). Finally, connect the cord between the modulator and the adapter box. (*Note:* Some televisions are now both televisions and monitors. If your television can be used as a monitor, connect the video cord directly between the monitor and computer. To hook it up, simply plug the monitor connector cord into the monitor jack. The other end of the connector cord goes into a similar socket in the back of the monitor. When I purchased my monitor, a connector cord did not come with it. I just went into an electronics store and asked for an RCA standard jack connector for a monitor. Remember, "RCA standard" is the kind of plug and cord to ask for since it refers to the size of jack connector you'll need.)

Types of Monitors

Apple Computer makes some perfectly good monitors that you may want to purchase for your computer. However, you may want to consider the many different types of monitors that are available.

Monochrome screen. This type of monitor gives a monochrome (single color) display with prices beginning at less than $100. The display may be green-on-black, amber-on-black, or white-on-black, depending on the monitor. The green-on-black display, usually the least expensive, is quite good for people doing a lot of word processing and noncolor graphics programming since it is easy on the eyes. Amber is supposed to be a good hue for working under fluorescent light.

Since monochrome displays show only a single color, they are not very good for color graphics. On the other hand, graphics that are going to be printed in black and white may actually be viewed better with a monochrome display; there's a better chance of seeing what you'll get.

Color/RGB. This type of monitor is the most expensive, but for people who work a lot with graphics, it is probably worth the added cost. RGB monitors connect to the video socket (the monitor socket is round, and the video socket is elongated). They provide the high resolution needed for seeing graphics in detail.

Flat screen. Apple makes a flat panel display for maximizing portability. Just about the only part of your Apple that cannot be tucked into a relatively small case is the monitor or TV. However, using the flat display screen, you can have a small portable screen. It is useful for business applications when you have to travel; otherwise, it's fairly expensive, lacks color, and is not as big as a TV monitor's screen.

Printers

This section will touch on the different kinds of printers and will briefly explain how to hook up a printer. If your printer is already hooked up and working, refer to Chapter 9 for tips on maximizing your printer's use.

Several types of printers are available for the Apple IIGS. Apple sells the ImageWriter and ImageWriter II, Scribe, Daisy Wheel Printer, LaserWriter, and Apple Color Plotter. Other companies also make excellent printers, but before you run out and buy a printer, carefully consider your needs.

Dot-matrix. The most popular kind of printer is the dot-matrix printer. It contains 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 generate both text and graphics. The improved quality of text printing with dot-matrix printers gives an almost letter-quality product, and usually you can get several different typefaces.

Much of the graphics and word processing software written for the Apple IIGS has been produced with a dot-matrix printer in mind. For example, a word processor program may have a sequence for emitting italicized letters on a dot-matrix printer that will not work on other kinds of printers. Not all dot-matrix printers are alike, and you should carefully consider the software available for a printer before buying one. Certain printers, like the ImageWriter, have more software support in both graphics and word processing software than do others. Thus, while you may get a perfectly good, inexpensive printer, if special software is required to work it, you may end up spending more time and money getting it to do what you want than if you had bought a printer that has software support.

Letter-quality. For people whose major computer use is word processing, there are letter-quality printers. Most of these are daisywheel printers, which type characters in much the same way a typewriter does. Each symbol has a molded image like those found on typewriter heads. These printers are not good for graphics, but if you want letters, manuscripts, reports, and other written documents to look top-notch , letter-quality printers are the next best thing to laser printers. They tend to be relatively expensive and slow, however, and for most written materials, dot-matrix printers are fine.

Compare before you buy. If a dot-matrix printer does not deliver the quality of print you require, and if a laser printer is more than you need, take a look at different daisywheel printers. Finally, if you decide a daisywheel printer is what you need, check the speed in terms of characters per second (cps). If you do not require great speed, you can get away with a relatively inexpensive, but slow, daisywheel printer. If you do a lot of printing—for example, in a business application—you will need a faster, heavier, and (alas) more expensive one.

Laser. This type of printer will give you the highest-quality printing available, but it is probably more than most people need. Laser printers are for *desktop publishers*, who require

near-typeset quality. On the low end, a laser printer will run about $2,000; on the high end, around $5,000. This printer is not very good for printing things like self-adhesive labels, and it cannot use carbon forms that make multiple copies with a single pass. However, for publishing newsletters, designing forms, and myriad other uses, laser printers provide top-of-the-line output. In time, if they follow the path of computers, laser printers will become much less expensive and do more.

Other printers. Besides the printers discussed above, you may want to consider a few other kinds of printers and plotters. One printer that works somewhat like a dot-matrix printer is the ink-jet printer. These printers are quiet and fast, and they give somewhat better print quality than do dot-matrix printers. The dots are formed by little dots of ink being shot from a jet instead of by pins hitting a ribbon. However, the ink cartridges are more expensive than dot-matrix ribbons, and you cannot make single-strike carbons with the ink-jet system.

You may want to consider a thermal printer, but I do not recommend it. These printers can be very inexpensive, they are quiet, and they can create either graphics or text. However, they require special, expensive thermal paper that can lose the images printed on it. If portability is important, a thermal printer might fill the bill as a portable printer, but you can get a really good ink-jet portable that uses standard paper that does not fade.

Finally, you may need a plotter. This device is good for certain kinds of design applications, but it's not recommended as a general-use printer. Basically, plotters are machines that draw with pens. They are good for designs and diagrams of everything from blueprints to circuitry design. They work on a different principle from that of standard printers, and they can do very interesting text. However, the text generated by plotters is primarily for labeling different parts of a diagram, not for general text.

A word of advice. Before you buy a printer, decide what you will need it for, and then look at the features of all the different kinds. And, by all means, ask to see a demonstration on a IIGS just like yours. For some kinds of printers, you will need an interface card and cable to hook the printer up to your computer, just as you do with some disk drives. Make certain to get the correct interface and cable that go with your printer. Some interfaces and cables are sold separately; it's possible to get the wrong interface and cable if you're not

careful. This is why it is important to see a demonstration of the printer hooked up to a IIGS. The ImageWriter and ImageWriter II are the most popular Apple II printers and are easily hooked up to your IIGS. Before you purchase a printer, it's a good idea to ask others with printing needs similar to your own for suggestions.

Connecting a Printer

To connect your printer, simply plug it into the printer port (serial 1) of your IIGS or connect it to a special interface card. For connecting a parallel (Centronics) printer interface, you will have to buy a special parallel-interface card. Follow these steps to install it:

1. Connect the cable to the printer. On one end of the cable is a connector to the interface card, and on the other is a connector to the printer. As a general rule, the connector to the printer is the larger one.
2. Connect the other end of the cable to the interface card, much in the same way you connected the disk drive cable to the disk controller card.
3. Make sure there is some slack in the cable from the printer and put the interface card into slot 1. That's it.

Caution: *Never insert interface cards or remove them from your computer while the power is on.* You could give yourself the shock of a lifetime.

Other Gadgets

Besides the disk drive, TV or monitor, and printer, most new users don't have anything else to hook up at this point, so you can skip to the next section. However, if you plan to expand your IIGS or have already bought other gadgets with your system, read the following.

Lots of slots. One of the nicest features of the Apple IIGS is its expandability and adaptability. The seven slots in the back provide you with the ability to grow as your needs and interests do, and the memory expansion slot lets your computer's RAM size grow. So there are a total of eight slots available for your use. However, you cannot use all of the slots (on the inside) and the ports (on the outside) at once. You will have to see the section on using the Control Panel and read the section on configuring the slots. It's not difficult, but it is important.

RAM cards. The firmware cards known as RAM cards are additional memory. These cards go into the memory expansion slot and are used to add more memory to the IIGS for general use. Your IIGS can handle several megabytes of RAM, and once you start using programs that take up a lot of memory, you will want to think about expansion. For example, if you do a lot of word processing, the files can get pretty big. Without added RAM memory, you'll have to break your files into smaller chunks and link them from disk files. This works fine, but it's much easier to work with a complete file in memory at one time. If you have really big files, such as a whole book, you will want to be careful about how much RAM you use. The double-sided 3½-inch disks hold 800K, and if you have more than that in RAM, you cannot save it to your disk. If you fully expand your RAM, you will probably want to get a hard disk with ten or more megabytes of storage.

RAM drive. Using your Control Panel, you can reserve a certain amount of RAM for a RAM drive. A RAM drive is like a disk drive, except it exists in RAM. The advantage of using a RAM drive is speed. It's a lot faster to load and execute programs or to save them using your RAM drive instead of the physical drive. If you use graphics a lot, you will want to load the picture files into your RAM drive, and that will make it easier to load and use them. However, be careful. When you turn off your IIGS, everything in your RAM drive will be vaporized. So before you quit, be sure that anything you have created and saved in your RAM drive has also been saved to a physical disk as well.

Modem. A modem is a device that enables your computer to communicate with other computers over telephone lines. You can use one of your serial ports, usually serial 2, to plug in your modem. Other modems plug into a slot inside your computer. The older types of modems are called *acoustic couplers,* and they have a cradle for the telephone handset. Acoustic couplers are more likely to lose data being transmitted and are generally less reliable than ones that connect directly to your computer. If you buy a modem, avoid the acoustic type, and make sure you can connect it to your IIGS, either through one of the slots or directly to one of the serial ports.

With a modem, you can call other computers and engage in the lively exchanges of information available on local computer bulletin boards. Commercial services allow you to check

stock prices, send and receive electronic mail, access your bank account, and take advantage of an increasing number of services.

Finally, modems operate at different baud rates. The baud rate refers to the speed at which a modem can transmit and receive data. Most modems are 300, 1200, or 2400 baud—price increases with speed. In general I recommend a 1200-baud modem since, at the moment, the speed/price ratio is best on these. When you make a call with your computer, the telephone company charges the same rates as for voice calls. The faster your modem sends and receives data, the less time you will have to spend on a call, and the lower your phone bill. If you plan to make heavy use of a modem in business or in transferring data long distance, the 2400-baud modem can very quickly pay for the difference in price.

Mice and things. There are numerous other cards that make the Apple into many different computers: special graphics printer drivers, Z80 cards to give you CP/M, and cards to turn your IIGS into just about anything you want. There are even multifunction cards that combine a number of the above cards into a single card using a single slot.

In addition, you can plug in a mouse, game paddles, or a joystick. These devices are used with special programs ranging from word processors, such as *Mouse Write,* to hundreds of games that depend on the joystick and game paddles. Since these are linked to certain types of software, check what software goes with the hardware.

Power On

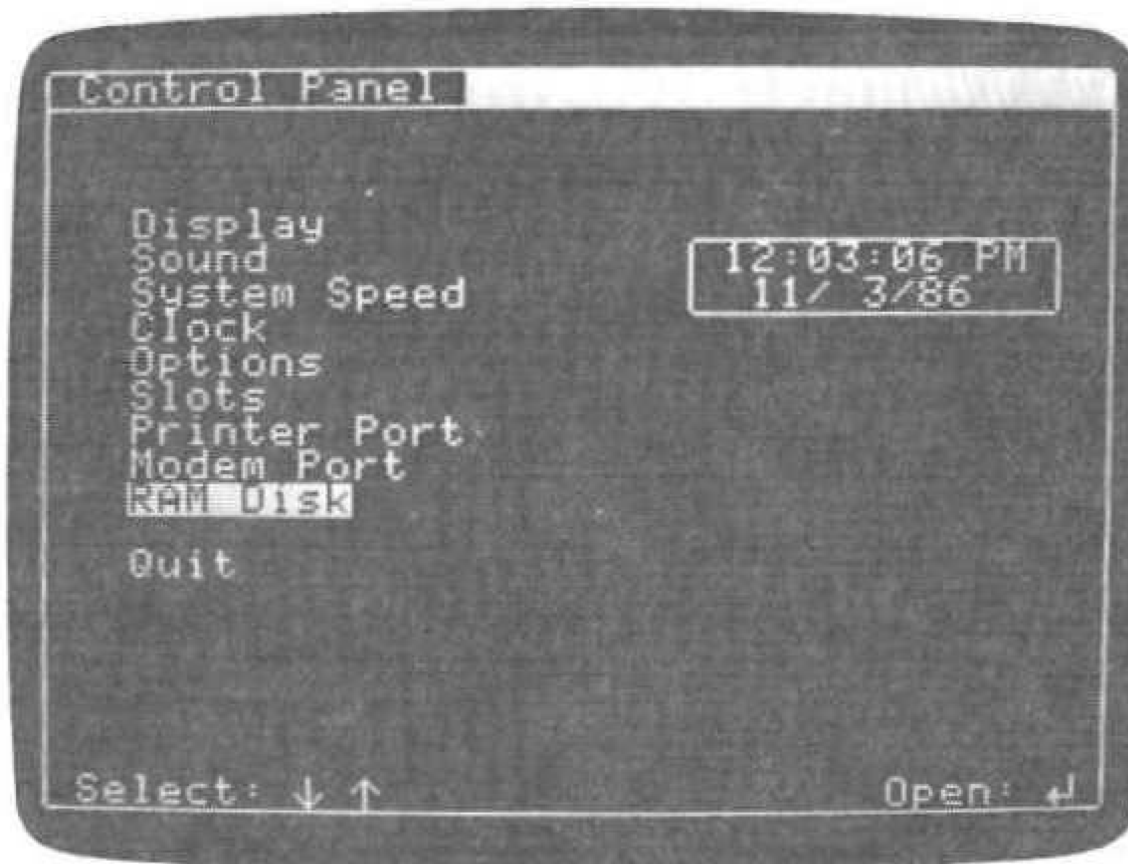
Now that your IIGS is all set to go, simply plug it in, along with your television or monitor and printer; then turn on the power and let it rip. Find the power switch and turn it to the *on* position. If everything is working correctly, you will hear a bell and your disk will make a noise. Press the key marked *control* and the one marked *reset* simultaneously. You should see a bracket and blinking cursor at the bottom of your screen. If you have an Apple ImageWriter or some other kind of printer connected to the serial 1 port, you're all set to test your printer. Skip ahead to the section "Printer Test Program."

If you have a printer connected to an interface card in slot 1, you'll have to use the Control Panel. To do that, hold down the key marked control and press the *open Apple* key (it's an outline of an apple); then press the key marked *esc.* That will give you access to the Desk Accessories. Select the Control

Panel by moving the up- and down-arrow keys, and then press the *return* key when the cursor is over the Control Panel option.

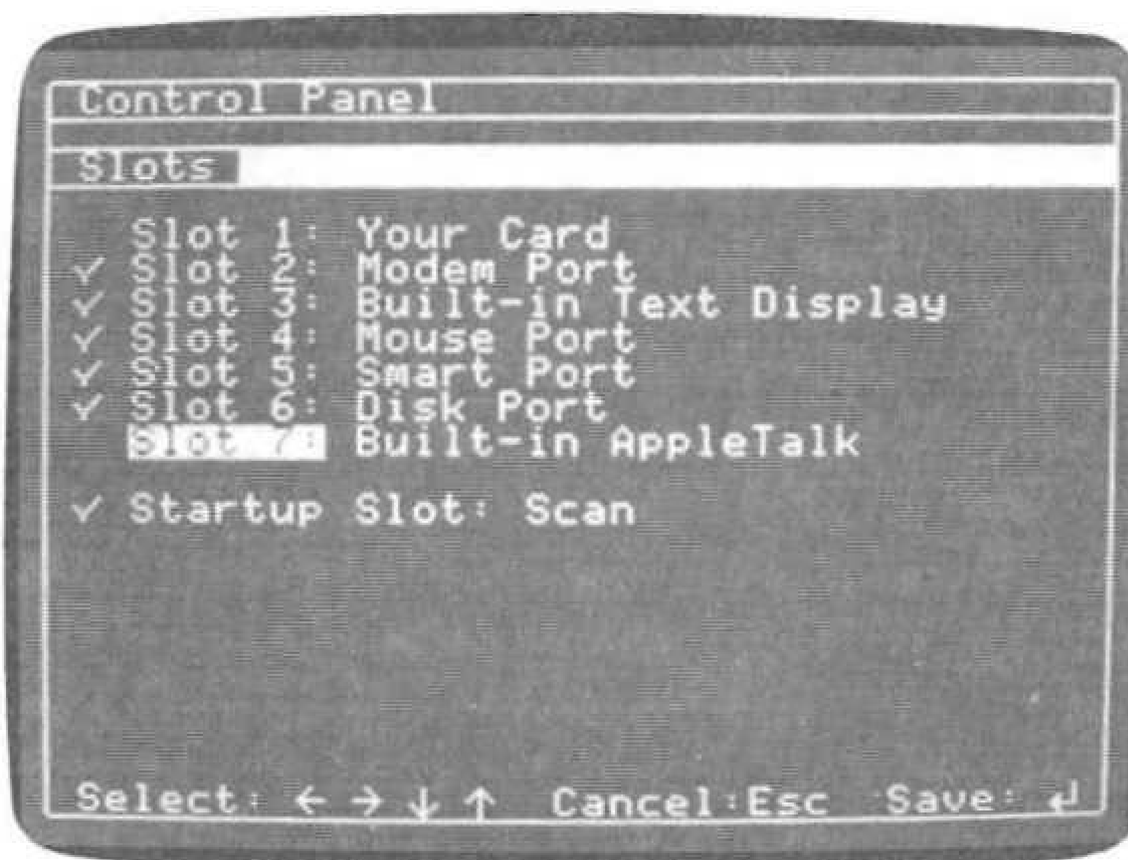
You will see the Control Panel appear on your screen.

Figure 1-2. The Control Panel



Select the *Slots* option, and you will then go to that screen.

Figure 1-3. Slots Option—Control Panel



Place the cursor over *Slot 1: Printer Port,* and press the right-arrow key. This will toggle slot 1 so that it now shows:

Slot 1: Your Card

If a checkmark is beside an option, it means that's the default option. Now, since slot 1 is your interface card and not serial 1 (the printer port), your IIGS will know to use your printer interface card and will not try to send output to the default printer port.

Printer Test Program

First, enter the word NEW; then press the return key. Press the return key at the end of each line.

```
NEW
10 D$=CHR$(4)
20 PRINT D$; "PR#1"
30 PRINT "MY PRINTER IS WORKING!"
40 PRINT D$;"PR#0"
```

Make certain that you have entered the program exactly as it appears. If there is even a minor difference, correct it so that it is the same. Put the ribbon and some paper into your printer. Now, turn on your printer and make certain that its switches and lights indicate that it is online. Type in the word RUN, and press the return key. If your printer is attached properly, it will print out the message MY PRINTER IS WORKING! If a ?SYNTAX ERROR or another error message jumps onto the screen, you wrote the test program improperly. Go back and do it again. If the system hangs up (if the screen goes blank and nothing happens), check to be sure the printer is turned on and is online. If it still doesn't work, turn off the power on the printer and computer, and review the steps for hooking up your printer.

Booting Disks

Assuming your system is working correctly, let's *boot* a disk. This will get your disk operating system (DOS—pronounced *doss)* operating. Apple's DOS is called ProDOS, and it's stored on a disk that comes with your computer. (Actually, there's a ProDOS 8 and a ProDOS 16 as well as DOS 3.3 and 3.2. Since we will be working with Applesoft BASIC, you will be in the Apple II mode and using ProDOS 8. We'll just refer to it as ProDOS or DOS throughout this book.) The disks are shipped in the 3½-inch format, so if you are using a 5¼-inch drive, go to your computer store and ask to have ProDos transferred to a 5¼-inch disk.

Booting With the DeskTop

If you're familiar with Apple's Macintosh computer, then you will have a very easy time using the DeskTop. The DeskTop is a graphic interface between you and the IIGS operating system. Rather than having to type commands like CAT and DELETE to manipulate information on disk, DeskTop lets you perform disk operations using menus and graphic figures (called *icons*) controlled by the mouse. If you are new to the DeskTop system, you'll find that using it is almost intuitive. Just stick your 3½-inch *System Utilities* disk into your drive and wait until it clicks in. Or with a 5¼-inch drive, close the drive door once the disk has been inserted. Turn on your computer and you will see two icons (small figures) on your screen that represent a disk and a trash can. Along the top of your screen you will see the following menu bar:

File Edit View Special Help

Using the mouse as an arm, you will pull down the various menus by placing the pointer on the menu you want to see and then pressing the mouse button. To see what is in a disk, either double-click on the icon or choose Open from the File menu. To get used to it, try it both ways.

To prepare a disk for use, you'll need to format it. Take a blank disk, or a disk whose information you don't mind being destroyed. Place it in a drive. Pull down the Special menu and choose Format. *Be sure you choose to format the blank disk and not your system disk.* (The best way to guarantee that you do not format the wrong disk is to eject the disk you do not want formatted. Remember, everything on a disk is destroyed when you format the disk. So watch yourself when you do this.) You may also format a disk from the *System Utilities* options and from BASIC if you have DOS 3.3 running. See below for instructions on formatting disks without the DeskTop.

To run a program from the DeskTop, just open the disk with the program on it and double-click it with the mouse pointer. Once you have your BASIC programs saved on disk, you can execute them from the DeskTop. To erase a program or file you don't want any more, *drag* (place the pointer on the icon and hold the mouse button down) it to the trash can icon. The program or file will immediately be erased from your disk. With 3½-inch disks, if you place the disk in the trash, it will just be ejected from the drive. (With 5¼-inch disks, you have to open the door and take them out.)

Booting Without the DeskTop

Skip this section if you use the Desktop.

Now let's see how to boot a ProDOS disk that does not have the DeskTop. Take the ProDOS disk and place it into the disk drive. Insert it gently and evenly until it is all the way in, and it will click into place. If you have a 5½-inch drive, close the door on the disk drive once the disk is in. Now turn on your computer and the drive will start spinning. The disk drive will spin for a while, and then a message will appear on your screen with the title of the disk. When you're presented with a menu to select an action, take the option to exit. When you have successfully done this, a cursor will appear indicating that your system is all set to go.

Reading the Contents: CAT and CATALOG

To find out what is on a disk without the DeskTop, simply type in the word CAT and press return. If your system is properly booted, you will be presented with a listing on your screen showing you the various files on your disk. If you get a ?SYNTAX ERROR after you have correctly typed in CAT, it means your system did not boot. Instead of shutting off the power, enter PR#6 and press the return key. This is another way of booting your system, except you do not have to turn off the power. (You can also reboot by pressing the control and open-Apple keys and pressing the reset key.)

Impressing your friends. Remember that the disk drive is controlled by slot 6. By typing in PR#6, you access that slot. (If you put a drive controller card in slot 5, PR#5 would boot the disk system.) To impress people, type in IN#6, and it will do the same thing as PR#6. If you really want to knock 'em dead, type in CALL −1401 . That's simply a way to show off and another way to boot your system.

Running Your First Program

When you use the DeskTop, running a program just requires that you place the pointer on the program you want and perform a double-click. (Alternatively, you can do a single-click to select a program and choose Open from the File menu.) However, if you are writing BASIC programs and you want to run a program from BASIC without reentering the DeskTop, do the following. Enter

```
RUN Program.Name
```

Your disk drive will whirl, and shortly your program will appear. That's just about all there is to it; however, there are a few more things you should know before we continue. Let's look at a few files. Enter

```
CAT
```

and press the return key.

The files on your disk will appear on the screen. Actually, since there's a hierarchical file system on your disk, only those of the "current directory" will be seen. (A hierarchical file system means that your disk is arranged like a file cabinet with a lot of drawers and file folders. What you see depends on what drawer and file folder are currently open.) The files represent the main types you will see on a disk.

In this book, we will spend most of the time writing BAS (BASIC) files, but we will also be creating TXT (text) files. BIN (binary) and VAR (variable) files will be introduced in later chapters. There are a couple of other DOS commands we'll mention now. Notice the little asterisk (*) next to the files on the disk. This means the file is LOCKed. If a file is locked, it cannot be removed from the disk with the DELETE command. If you want to delete a locked file, you must first UNLOCK it. See Table 1-1.

Table 1-1. Some DOS Commands

(Must be all uppercase)

Command	Result
RUN or –	Executes BAS files
BRUN	Executes BIN files
LOCK	Prevents a file from being overwritten or removed with a DELETE command
DELETE	Removes a file from the disk
UNLOCK	Removes the LOCKed status of a file and enables the user to remove it with a DELETE command

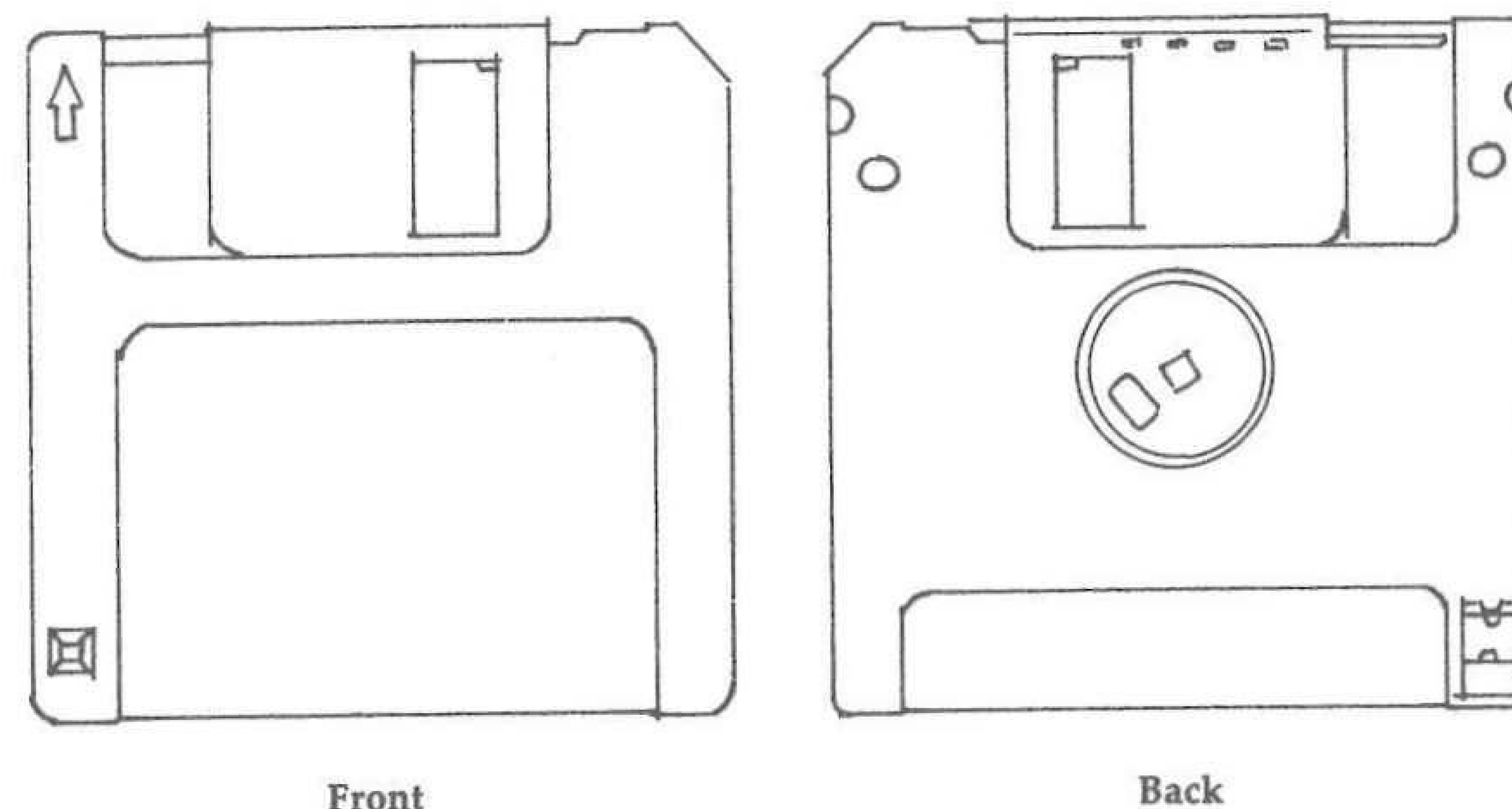
For the time being, don't change anything on the disk that came with your computer. Get a blank disk and use it for experiments with your DOS commands.

Formatting Blank Disks Without the DeskTop

First, if you do not have a blank disk, go get one. *Do not use your* System Disk *for this next procedure except to boot your*

DOS. Look at the diagram of a 3½-inch disk (Figure 1-4), or skip ahead to the discussion of 5¼-inch disks if that's what you're using.

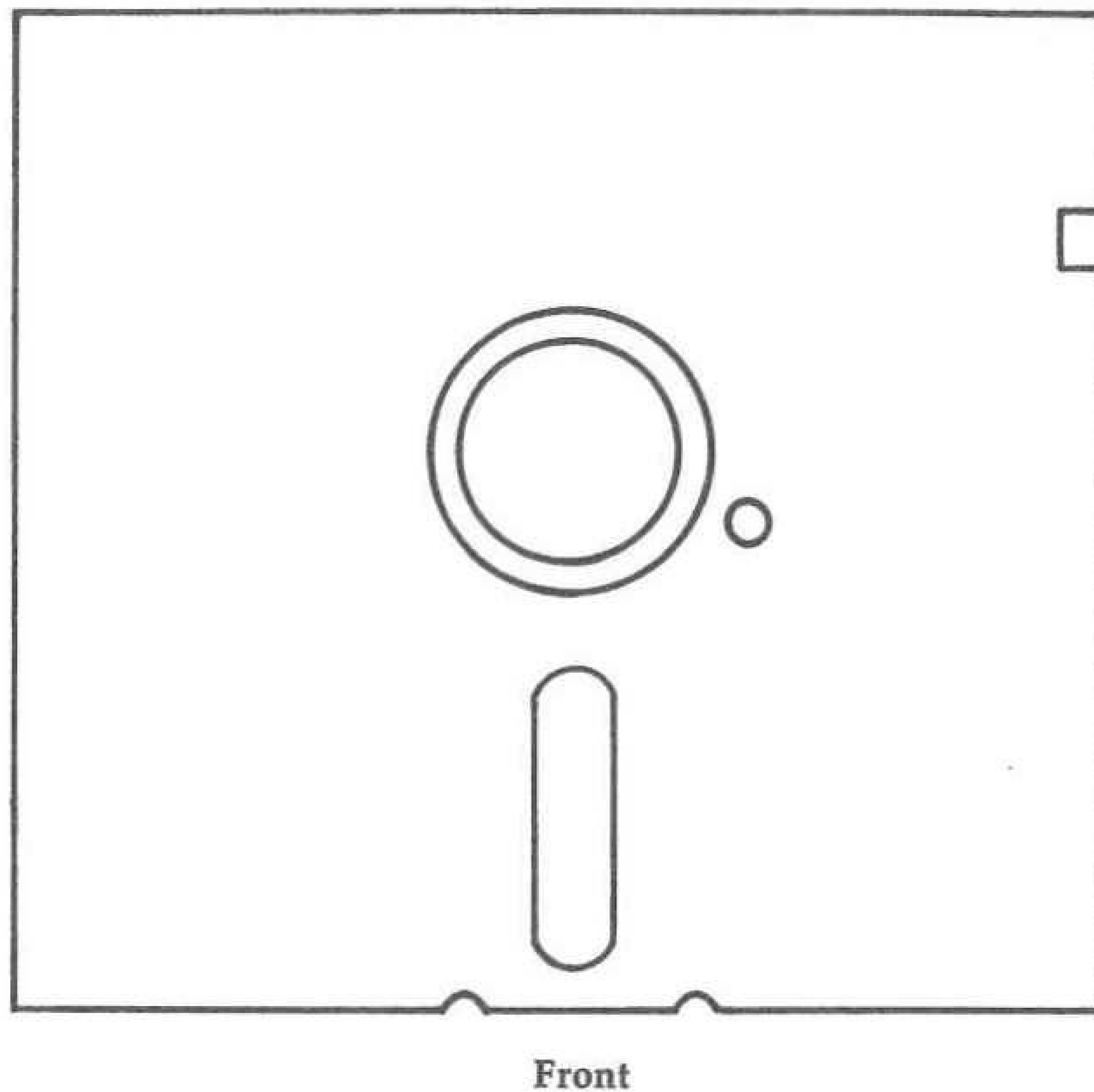
Figure 1-4. 3½-Inch Disk



The front, or top, of the disk is the side that faces upward when you insert it into your drive. This is the side on which you should place the label. In the upper right corner of the disk is the write-protect window. When the window is closed, you can change what's on the disk; when it's open, all of the files are protected from being changed. For example, if you have some files that you do not want to be accidentally deleted, you may wish to physically write-protect them by opening the write-protect window. If you attempt to format the disk, you will be informed that it is write-protected.

On the back side, or bottom, of the disk, you can see the slot with the write-protect window. Slide it up and down to see how to open and close the write-protect window. You'll want a blank disk (one with no programs or files saved on it) with a closed write-protect window (*not* write-protected) for practice. If you're using a 3½-inch-disk system, skip the following section on 5¼-inch disks (see Figure 1-5) and move ahead to see how to format a disk.

Figure 1-5. 5¼-Inch Disk



The item to notice on your blank 5½-inch disk is the square notch in the upper right corner. This is the write-protect notch. If there is no notch on a disk, or if the notch is covered by a little tab (called the write-protect tab), you cannot change files on that disk—even if all the files are unlocked. This is an important feature since it protects you against accidental erasure or overwriting. Be careful not to touch any of the exposed disk surface in the window of the disk. Touching can damage the disk, making it impossible to read the files on the disk.

Now follow these steps to format a disk without using the DeskTop:

1. Boot your ProDOS disk *(System Utilities)*.
2. When the menu appears, choose the option *Format a Disk.*
3. REMOVE the ProDOS disk.
4. Place your blank disk in the drive and close the door. Be sure that the write-protect window is closed on 3½-inch disks (or the write-protect notch is uncovered on 5¼-inch disks.)
5. Follow the prompts on the menu.

Once your disk has been formatted, you'll never need to format it again unless you want to destroy all the files on that disk.

To format a disk from BASIC in DOS 3.3, do the following:

1. Boot a DOS 3.3 disk.
2. Place an unformatted disk in the drive.
3. Key in the following, pressing the return key after each line:

```
NEW
10 TEXT : HOME
INIT HELLO
```

Every DOS 3.3 disk must have a HELLO program. Actually, the HELLO program doesn't have to be named HELLO. You could just as easily use INIT HOWDY or INIT STARTING-PROGRAM. However, HELLO is the name almost all DOS 3.3 programmers use (it's a *convention*, a rule that most people follow voluntarily). The one-line program

```
10 TEXT : HOME
```

can be any program you want to be the startup program. Don't ever delete the HELLO program from a DOS 3.3 disk.

Similar to a Typewriter

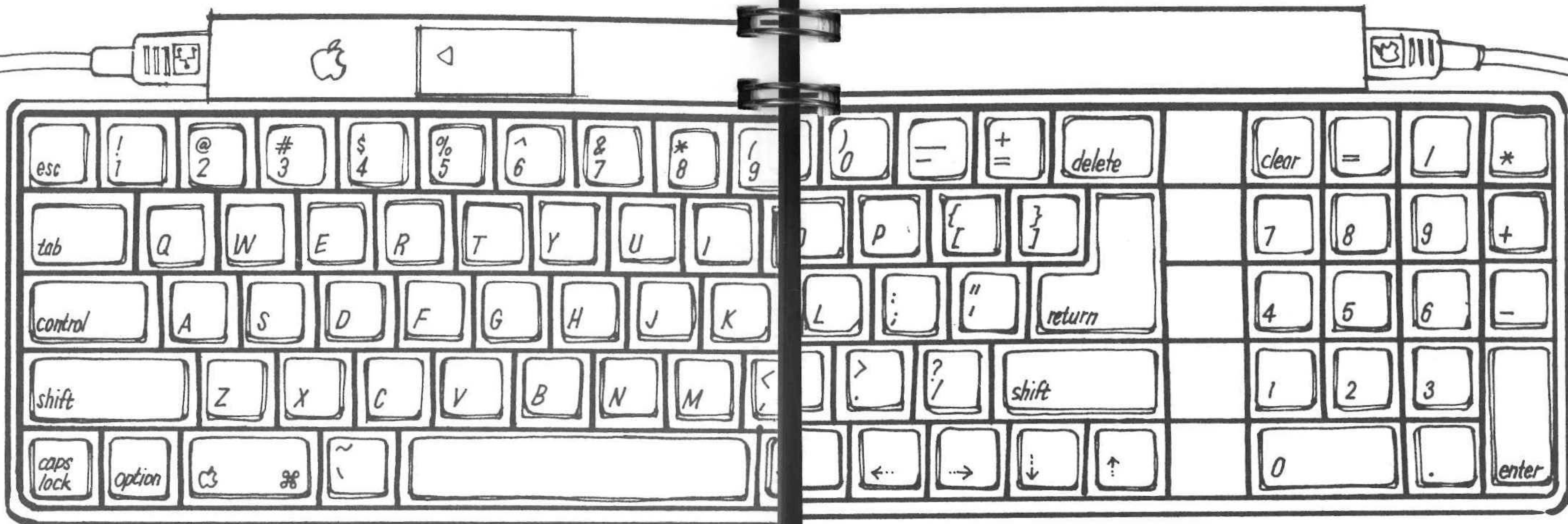
If you're familiar with a typewriter keyboard, you will see most of the same keys on your Apple IIGS. For the most part, these keys do almost the same things as typewriter keys. Of course, you cannot type just anything on the screen. If you start typing away, you'll get a ?SYNTAX ERROR unless you put in the proper statements. Otherwise, though, think of your keyboard as you would a typewriter keyboard.

Keys You Won't See on a Typewriter

While most of the keys on your IIGS look like those on a typewriter, many do not, and they are important. The following keys are peculiar to computers; you will soon get used to them even though they will be a bit mysterious at first. Refer to Figure 1-6 for the keys discussed.

Esc. The esc key in the upper left corner stands for *escape*. Depending on the program in memory or the particular task you're working on, it will do different things. You'll use it when you're editing programs (discussed in Chapter 2). For now, you should know that when you press esc, you will affect the next keyboard entry. Also, to operate the esc key, you don't have to hold it down while simultaneously pressing another key.

Figure 1-6. The IIGS Keyboard



Control. To use this key, you must hold it down while pressing another key. Try holding it down and pressing the G key. Does that ring a bell? Later, you will learn the use of *control characters* that are initiated by pressing the control key and another key simultaneously.

Reset. The reset key is often a panic button. When things freeze up on your computer, sometimes the only way to unjam them is to hit control–reset. Depending on the program in memory, hitting the reset key will have different effects. With certain programs in memory, strange things can happen if you hit reset. Don't be afraid of it, but for now, use it only when instructed or when a program freezes up on you. As a general rule, *never* press reset when your disk drive light is on and the door is closed.

Arrow keys and space bar. The left, right, up, and down arrows move the cursor without affecting the characters on the screen. If you advance the cursor with the *space bar* (the long

key at the bottom of your keyboard), you will wipe out any characters you see on the screen. This is because a space is entered as the character. The arrows are used extensively in editing. If you retrace a line with the right arrow, you will verify the line. Later, in the discussion of editing programs, you will see that there are many uses for the arrow keys.

Return. The return 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 a manner analogous to a typewriter's carriage return: After you've pressed it, the cursor bounces back to the left-hand side of the display screen. However, there are other uses for the return key.

Apple key. The open-Apple key on your IIGS operates something like the control key in that, when pressed in conjunction with other keys, it affects the way other keys work.

New Meanings for Old Keys

Some of the familiar keys have different meanings for computers from those we usually associate with the key symbols. Many are math symbols that you may or may not recognize. The next chapter will illustrate how these keys can be operated, and they will be discussed in detail. For now, here's a quick look at the math symbols:

Symbol	Meaning
+	Add
−	Subtract
*	Multiply (different from conventional)
/	Divide (different from conventional)
^	Exponentiation (the symbol is called a caret)

In addition to some of the new representations for math symbols, other keys will be used in ways that you are not used to. As we continue, we will explain the meanings of these keys. Just to accustom you to the idea that the IIGS has special meanings for certain keys, here are some others that have special meanings:

Symbol	Meaning
$	String variable or hexadecimal value
:	End of a BASIC program statement
%	Integer variable
?	Can be used as a BASIC PRINT statement

Don't try to understand what these symbols do. Just be prepared to think about 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. The first step is being aware that the different meanings exist.

On the Screen

We've mentioned the cursor, but haven't told you much about it. If you've not already noticed, the cursor is a blinking checkerboard square. At other times it can be a solid square or a blinking underline. Perhaps the easiest way to understand the cursor is to think of it as a marker that tells you where you are and what your computer is doing. When the cursor disappears, your computer is either running a program or is hung up and not operating. If you think it's hung up, hit the control-reset key combination or reboot your system by turning it off and then on again. In addition, there are different prompts representing different states.

Different Prompts

As you have seen, the Applesoft prompt is the bracket-shaped marker—]—meaning that the language Applesoft is ready to go. If you have Integer BASIC in residence, your prompt looks like an arrowhead: >. Whenever you see the > prompt, you know that Integer BASIC, and not Applesoft, is in operation. Finally, if your prompt is an asterisk *, you are in the monitor. This is a new meaning for monitor; it's where machine-level programming is done. For the most part, you *do not* want to be in the monitor. If you see the *, press control-C and press the return key to get back into BASIC—either Applesoft or Integer. (Just for fun, if you find yourself in monitor, press L, and you'll get an assembly listing. This may be useful. If you're showing off your IIGS to a friend and you accidentally bomb and end up in monitor, press L and list the assembly/machine code. Your friend will think you know what you're doing.)

Controlling the Control Panels

Whenever you wish, you can open up your Desk Accessories and change certain parameters. Usually, you'll have to do this only once when you choose your basic system—for example, to change the disk drive, monitor, and printer. You've seen how to set the slots, but there are several more options that you may need to change periodically. Let's see what they are. (Remember, to get to the Desk Accessories, press control–open Apple and the esc key.

When you see the various options that can be controlled, some will have a checkmark (√) next to them. This indicates the standard, or default, condition. Use the arrow keys to move the cursor, the escape key to cancel and back out of a file, and the return key to save a set of changes you've made. Moving the cursor over a choice with the left- and right-arrow keys will toggle the various options available. For example, moving the cursor over the 40-column default in the Display file will toggle it to 80 columns. The checkmark will disappear, and your text and programs will be shown in 80 columns instead of 40. The option in braces—{ }—is the nondefault option available when a single alternative is available. Thus,

√ **Type: Color** {**Monochrome**}

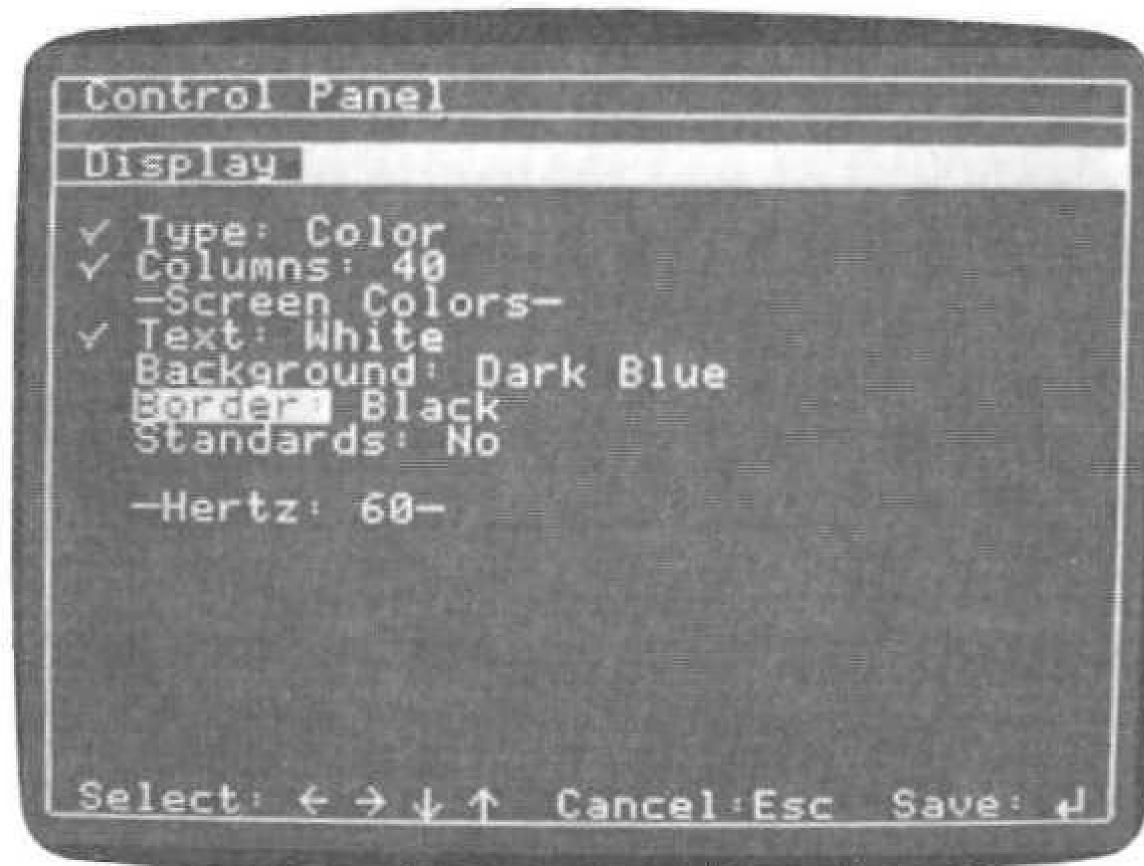
means that the default type of display screen is a color one, while the single alternative is monochrome.

We'll go over the various files in the Control Panel. Most of the options are self-explanatory; we'll try to clarify those that may cause confusion.

Display

The Display choices are for the most part self-explanatory. The Standards option will change everything back to the default parameters, and if any *single* default parameter is changed, the checkmark next to Standards will disappear, and the Yes will be changed to No. If you want to undo several changes, you can just toggle the No to Yes, and everything will go back to the default condition. Figure 1-7 gives the Display options.

Figure 1-7. The Control Panel: Display



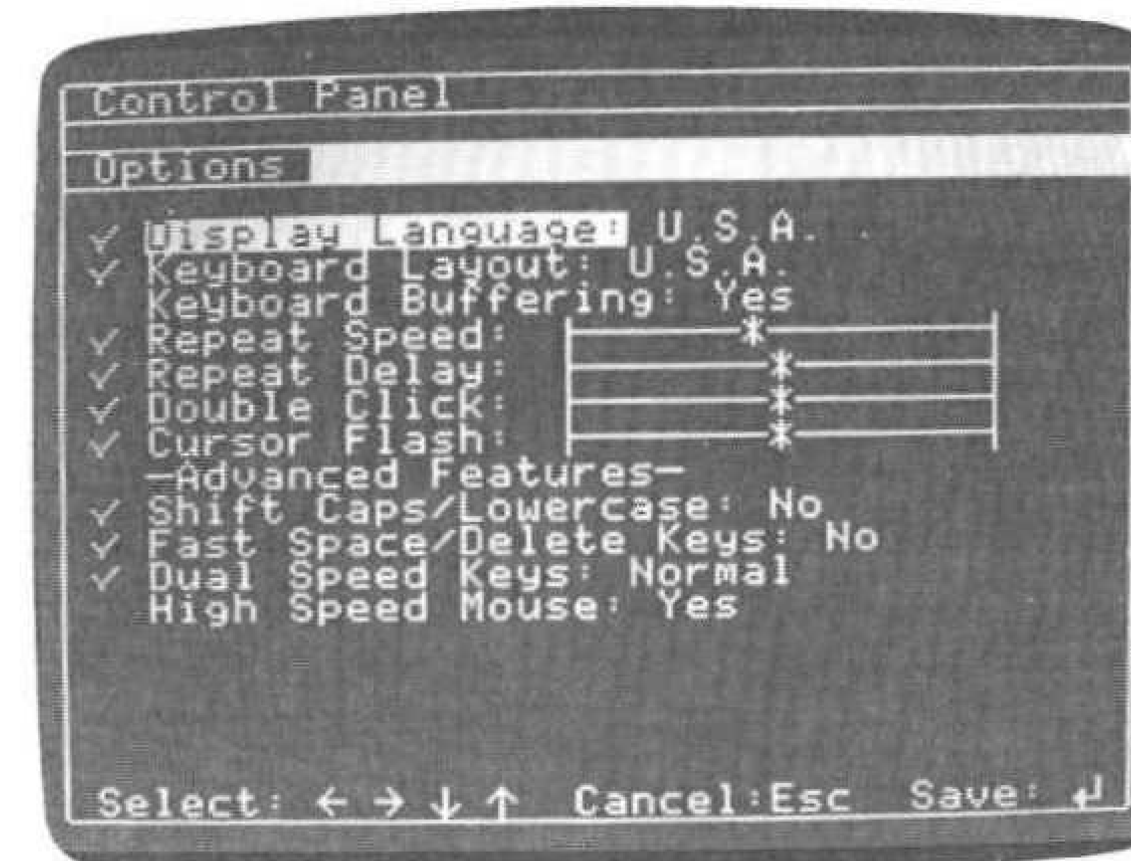
Options

The Options parameters are of the "set and forget" variety. Once you have *comfortably* set them, you don't have to worry about doing so again. Most users will incorporate the U.S.A. keyboard, but you can change it to the one desired. Keyboard buffering refers to a "type ahead" buffer for fast typists. If you really type fast, you can have the keystrokes stored in a small buffer to be sent to the screen when the computer catches up with your typing speed.

You can adjust the next four parameters to your own comfort. For example, if you want very fast repeats for keys held down, move the asterisk to the right. If you want slower repeats so that you won't get too many of the same thing racing across your screen, move the asterisk to the left. Do the same

with the other speed controls. Figure 1-8 shows the Options parameters.

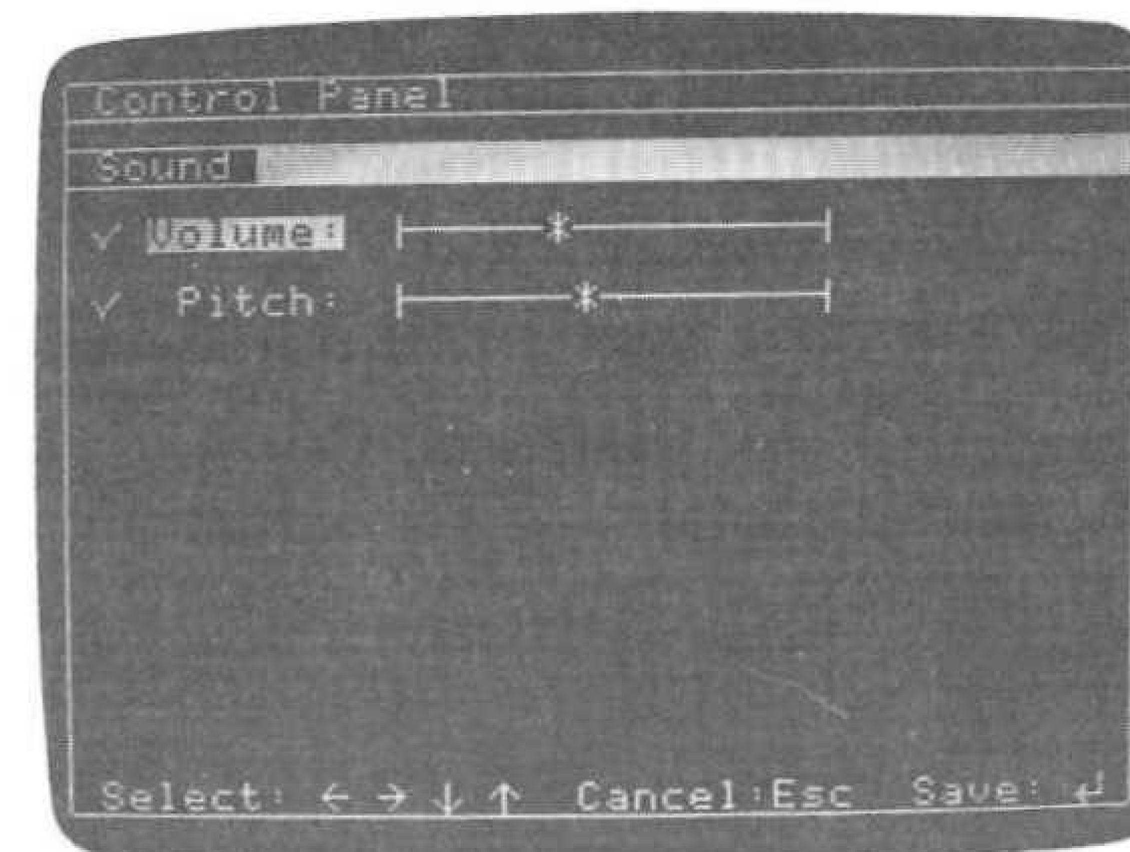
Figure 1-8. The Control Panel: Options



Sound

The sound panel (Figure 1-9) lets you change the sound of the bell. This may sound like a rather silly option, but if you can make the sound more pleasant, why not?

Figure 1-9. Sound Panel



System Speed

This is another option that you'll probably leave alone most of the time:

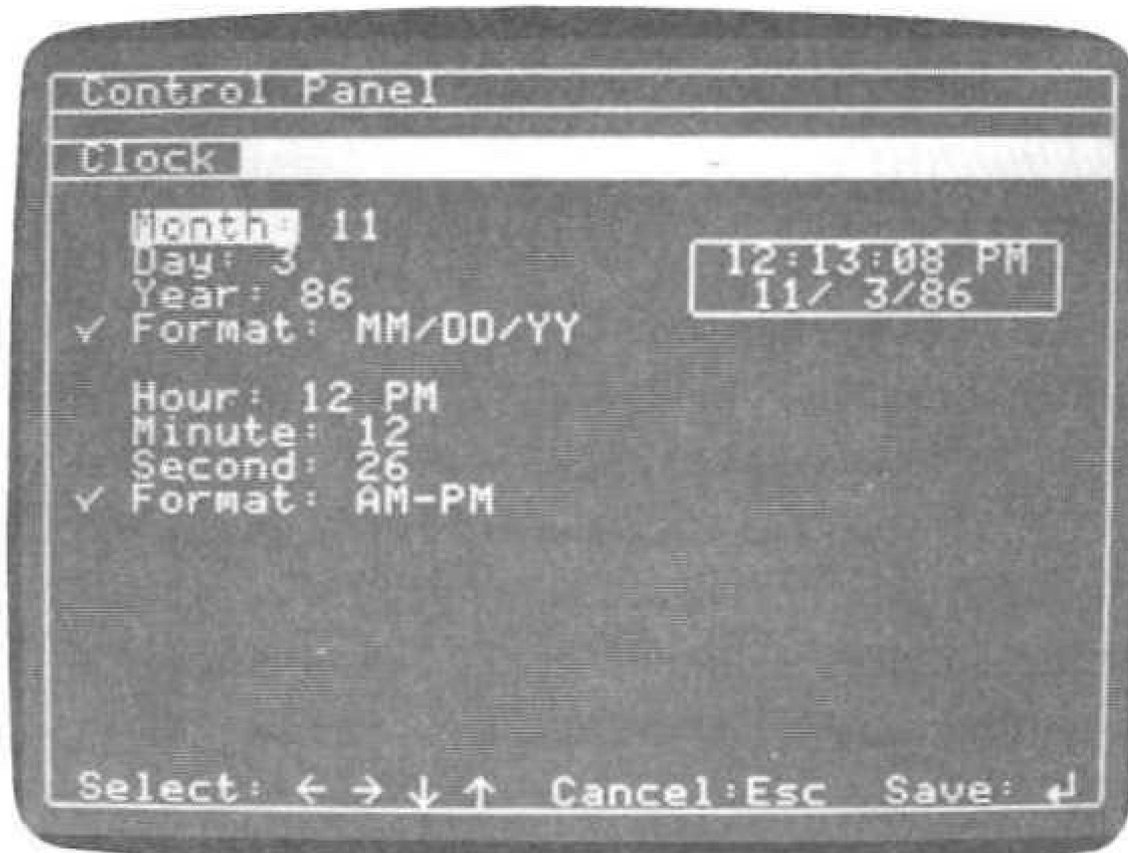
System Speed: Fast/Normal

Your IIGS can run at either 2.8 MHz (megahertz) or 1 MHz. You might as well use the faster speed unless you come across a program written for an older Apple II that has a timing routine, or unless you have an older interface card, such as a disk interface card, that has a timing-dependent routine. If you have a program or device that does not seem to be working correctly, try changing the speed to see if that has any effect.

Clock

Setting the clock (Figure 1-10) is fairly self-explanatory. About the only thing to decide is whether you want 24-hour (military) or AM/PM format.

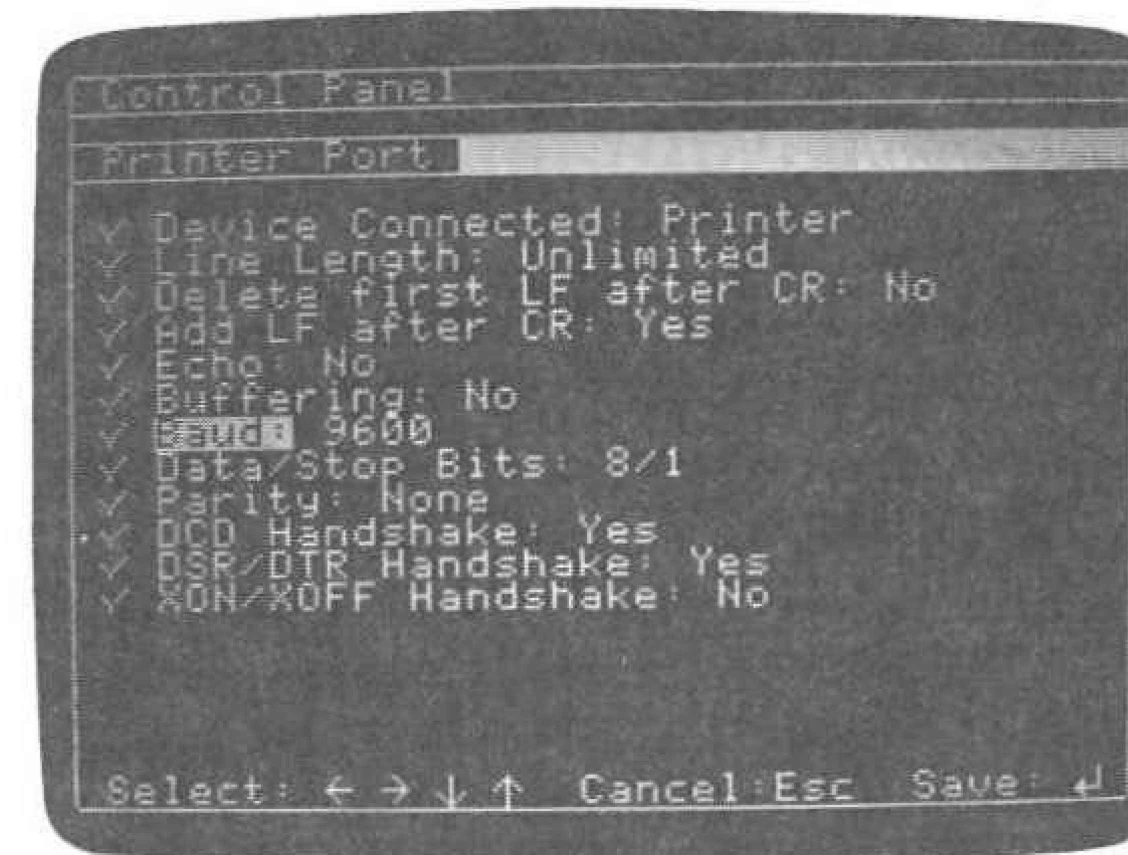
Figure 1-10. Clock Settings



Serial Printer Port

If you have an ImageWriter or ImageWriter II printer, you can leave the printer port at the default settings. If you have a parallel printer, you will need a printer interface card, and you can use this port for another device if you want. You may also have to make adjustments for software with this control panel. It all depends on your printer software. For example, if the software adds a linefeed (LF) after a carriage return (CR), you may not want another one added automatically. This problem will be especially prevalent with public domain (free) software. If your output is not what you expect, you may want to adjust some of these values. Figure 1-11 illustrates the printer port panel.

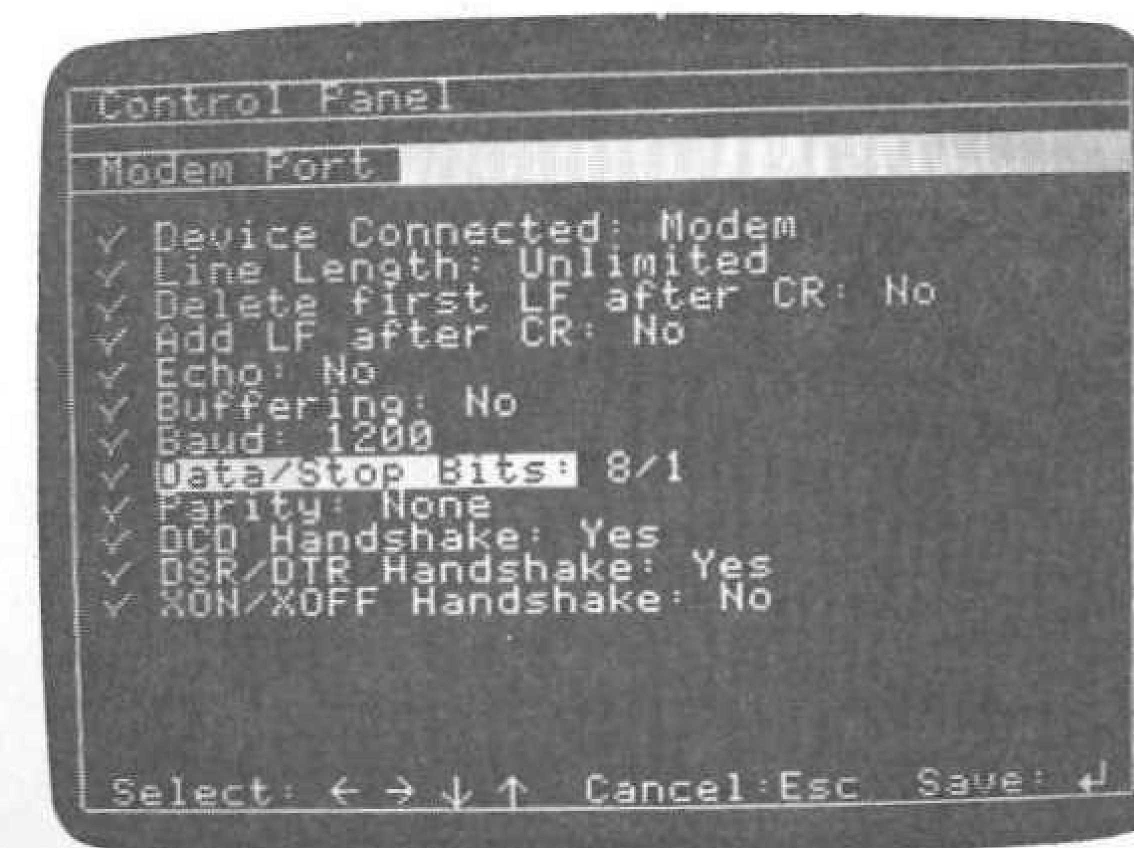
Figure 1-11. The Printer Port Panel



Modem Port

The modem port panel (Figure 1-12) is just like the printer port, except it sets the parameters for the second serial port. The default configuration is the same as for the printer port, except the default device is the modem and the baud rate is 1200 instead of 9600. Depending on the type of modem you have, the baud rate is the most likely to be set. I recommend a modem with a baud rate of 1200 since it has the best cost/speed ratio on the market, but you may have a 300- or 2400-baud modem, depending on your needs and finances. Remember, though, higher-rated modems *can* run at lower speeds, but lower-rated modems *cannot* run at higher speeds.

Figure 1-12. Modem Port Panel



Summary

This first chapter has been an overview of your new machine. You should know how to hook up the different parts of your IIGS and get it running. Also, you should be able to boot and initialize a disk, CATALOG (or CAT) the contents of a disk, and run a program from your disk drive. You should know some of the basic ProDOS commands for manipulating files on your disk. Finally, you should be familiar with the keyboard and know what the cursor and prompts mean.

At this point there is still much to learn. Don't feel bad if you don't understand everything. As we go along, you will pick up more and more, and 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 Apple. It is vitally important that you enter and run the sample programs. Also, it is recommended that you make changes in them after you have first tried them out to see whether 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.

2

Getting Started

Getting Started

In this chapter you'll be introduced to writing programs in the language known as Applesoft BASIC, also called floating-point BASIC. The best way to learn programming is to actually do it and to keep it simple. By breaking a large problem into a series of little problems, you can do just about anything you want. The trick is to solve one problem at a time. By simplifying the problem, you simplify the solution. Just ask yourself, "What do I want done? What steps are involved in getting the job done? In what sequence do the steps occur?"

The First Statement: PRINT

Probably the most often used statement in Applesoft is PRINT. Words that are enclosed inside quotation marks following a PRINT statement will be printed to your screen. Numbers and variables will be printed if they are preceded by a PRINT statement. This statement commands your computer to print output to the screen or to the printer from within a program or in the immediate mode. You may well ask what the difference is between the *immediate* and *program* modes. We'll explain, but first turn on your computer and make sure the bracket prompt is on your screen and blinking. Also, to clear memory and make sure nothing is there that will get mixed up with your statements, type in the word NEW and press the return key.

Immediate mode. The Immediate mode executes a command as soon as you press return. For example, try the following:

```
PRINT "THIS IS THE IMMEDIATE MODE"
```

After you've pressed return, if everything is working correctly, your screen should look like this:

```
PRINT "THIS IS THE IMMEDIATE MODE"
THIS IS THE IMMEDIATE MODE
```

See how easy that was? Now try PRINTing some num-

bers, but don't put in the quotation marks. Try the following (pressing return after each line):

```
PRINT 6
PRINT 54321
```

As you can see, you can enter a number without quotation marks, but the actual value of the number is placed in memory rather than a "picture" of it (we'll discuss this later).

Program mode. In this mode the execution of statements is delayed until your program is run. All statements that 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"
```

Nothing happens, right?
Type in RUN and press return. Your screen should look like this:

```
10 PRINT "THIS IS THE PROGRAM MODE"
RUN
THIS IS THE PROGRAM MODE
```

Clearing the Screen and Writing Your Name

Let's write a program and learn three new statements: TEXT, HOME, and END. The TEXT statement clears any graphics from the screen and puts you in the text screen, placing the cursor in the lower left corner. The HOME statement clears the screen and places the cursor in the upper left corner. The END statement tells the computer to stop executing statements. From the immediate mode write in the TEXT and HOME statements to see what happens. Remember to press return after each statement in the immediate mode.

Now, let's write a program using TEXT, HOME, END, and PRINT. (Be sure that you press the return key at the end of each line.)

```
10 TEXT
20 HOME
30 PRINT "<YOUR NAME>".
40 END
RUN
```

All you should see on the screen are your name and the blinking cursor. Now, here are two shortcuts that will save you time in programming and in memory. First, instead of entering new line numbers, you can put multiple statements on the same line by using a colon (:) between statements. Also,

instead of typing in PRINT, you can enter a question mark (?). Try the following program to see how this works.

```
10 TEXT : HOME
20 ? "<YOUR NAME>" : END
```

These two lines do exactly the same thing, but you don't have to put in as many lines or write out the word PRINT. As a rule of thumb, always begin your programs with TEXT : HOME. This will help you get into a habit that will pay off later when you're running all kinds of different programs with text and graphics everywhere. There will be exceptions to the rule, but for the most part, by beginning your programs with TEXT : HOME, you will start off with a nice, clear screen in the proper mode for most applications. While you're just getting started, it's probably a good idea to use the colon sparingly since it's easier to understand a program with a minimum number of statements in a single line. Some programmers like to string together as many statements in a line as possible, but that's not a good idea in the long run. It is much easier to debug (find mistakes in) a program that has more separate lines than one with a lot of lines strung together with colons.

Also, make liberal use of the REM statement. When the computer sees a REM statement in a line, it goes on to the next line number, executing nothing until it comes to an executable statement. The REM statement works as a REMark in your program lines so that other users of your program will know what you are doing and as a reminder to yourself of what you have done. Here's how REM works:

```
10 TEXT : HOME : REM This sets the full-screen text mode
and clears the screen.
20 PRINT "<YOUR NAME>" : END
30 REM This fabulous program was created by <Your name
here>
```

When you run the program, you will see that the REM statements have no effect at all. However, what your program is doing is much clearer since you can read what the statements do in the program listing.

A case of case. When you write a program, your IIGS doesn't care whether you use lowercase or uppercase letters. Keywords will automatically be changed to uppercase. Words enclosed in quotation marks or following REM statements remain in whatever case they were originally typed in.

Setting Up a Program

Now that you've written a program, let's take a look at using line numbers. In your first program, you used the line numbers 10, 20, and 30. You could have used line numbers 1, 2, and 3, or 0, 1, and 2, or even 1000, 2000, and 3000. In fact, there is no need at all to have regular intervals between numbers. Line numbers 1, 32, and 1543 would work just fine. However, it's usually a good idea to number programs by tens, starting at 10. Wouldn't it be easier to number them 1, 2, 3, 4, 5, and so forth? In some ways perhaps it would be, but overall, that's definitely not a good policy. Here's why. Type in the word LIST and press the return key. 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 that prints your home address. Rather than rewriting the entire program, you can just enter a line number with a value between 20 and 30 (such as 25) and enter the line. Try it, but first remove the END statement in line 20.

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

Your name and address are now printed on the screen. All you had to do was to write in one line instead of retyping the whole program. If you had numbered the program by ones instead of tens, you would not have been able to do this. There would have been no room between lines 2 and 3 as there was between lines 20 and 30. You would have had to rewrite the whole program. With a small program, rewriting is not much of a problem, but when you have hundred- and thousand-line programs, you'll be glad there are spaces between line numbers.

Listing Your Program

As you just saw, entering the word LIST produces a listing of your program. To make it neat, type in HOME : LIST and press return. You'll get a listing on a clear screen. However, once you start writing longer programs, you won't want to list everything—just portions. Here are the options available with the LIST statement:

Statement	Result
LIST	Lists entire program
LIST 20	Only line 20 is listed (or any line number you choose)
LIST 20,30	Lists all lines from 20 to 30 inclusive (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- (or 40,)	Lists from line 40 (or any other line number chosen) to the end of the program

Try listing different parts of your program with the options available to see what happens. These statements will give you some examples of the different options:

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

Weird programming. Generally, you will want to use the LIST statement from the immediate mode as you write your program. However, you can use it from within a program. Just for fun, add this line to your program:

```
40 LIST
```

Run it and see what happens. Believe it or not, this has some very practical applications, which you will see in some programs later in the book. For the time being, though, it's just for fun.

Saving Programs

Suppose you write a program, get it working perfectly, and then turn off your computer. Since the program is stored in 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's a simple matter to save a program to your disk. We'll use your program as an example. Make sure your program is still in memory by LISTing it. If it's not, you'll need to write it in again. Now make sure a formatted disk is in the drive and enter the following:

```
SAVE MY.PROGRAM
```

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, enter the word CAT and press the return key. You should see

```
MY.PROGRAM  BAS  1
```

which means that your program has been successfully saved to disk.

Sometimes you will want to save your program to a certain subdirectory. For example, you may want to organize your disk into different types of programs. One program might be a graphics program and another a practice program for using variables. If you want to save a program called COLORS in a subdirectory called GRAPHICS, you will enter

```
SAVE /GRAPHICS/COLORS
```

Using subdirectories will make it easier for you to find things in a given category. If you want to save another graphics program—say, one that draws lines—you can save it under the GRAPHICS subdirectory. For example, you can save it as LINES, using the format

```
SAVE /GRAPHICS/LINES
```

Now both the program COLORS and the program LINES have been saved in the same directory.

Recalling Programs

The best way to be sure that you have saved a program to disk is to turn off your Apple completely, and then turn it on again. Go ahead and do it; then CAT your disk. You should be able to see your program (MY.PROGRAM) in the catalog. Now, enter the word RUN and the name of your program. The dash (–) works just like RUN.

`RUN MY.PROGRAM` *or* `- MY.PROGRAM`

The disk drive will whirl for a while, and then your program will be executed. In this case the screen should go blank. Your name and address should appear at the top of the screen along with a listing of your program. This means that you have successfully saved a program to disk.

If you use a subdirectory, you must name the path to your program. For example, if you have saved a program called COLORS under the subdirectory GRAPHICS, you must

```
RUN /GRAPHICS/COLORS
```

While using ProDOS, you can save time by establishing a PREFIX using the PREFIX command. For instance, to make GRAPHICS the default directory, just enter

```
PREFIX /GRAPHICS/
```


Until you name another PREFIX, all RUN commands will run the file you have named that is in the subdirectory called GRAPHICS. Thus, if you have established GRAPHICS as your prefix directory, you will be able to enter

```
RUN COLORS
```

and you will get the program without having to specify the subdirectory GRAPHICS. Also, any program you save will be placed in the prefixed directory.

Now, try something else. Enter the following:

```
NEW                     Clears memory
LIST                    Shows that nothing is in memory
LOAD MY.PROGRAM
LIST
RUN
```

You can just run the program directly from disk, but you can also load a program and then run it. There will be many occasions when you will want only to load and list a program, but not run it. When you begin working on larger programs, you may spend several days working on the same one. Since you do not want to leave your computer turned on the entire time, you will save parts of the program as you go along.

A Safety Net

As you begin writing longer programs, you should save a copy of your program to disk occasionally—every 25 lines or so, for example. If you do this, when your dog accidentally trips over your cord and turns off your computer, you won't lose your entire program.

Now that you have saved and loaded programs, here's another neat trick. Remembering that you have saved your file under the name MY.PROGRAM, you can change the contents of that file. First, add the following line and then LIST your program:

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

Since line 27 has been added, your program is now different from the program you saved in the file MY.PROGRAM. Now, using lowercase letters, enter this line, followed by pressing the return key:

```
save my.program
```

Clear memory with NEW, LOAD the file MY.PROGRAM, and LIST it. Line 27 is now part of MY.PROGRAM. All you

have to do to update a program is to LOAD it, make any changes you want, and then save it under the same filename. However, be very careful. No matter what program is in memory, this program will be saved when you enter the SAVE command. If your disk has PROGRAM.A and you write a different program, and then save it under the title PROGRAM.A, PROGRAM.A will be destroyed and the saved program will actually be the new program. The best way to avoid overwriting a cherished program is to LOCK it. Then if you try to save a program to that filename, your computer will tell you FILE LOCKED. It will not overwrite that file with any other program until you UNLOCK it. Also, if you have a really important program, it's a good idea to make a backup file. For example, if you had saved your current program under the filenames, MY.PROGRAM and MY.PROGRAM.B, you would have two files with exactly the same program. To play it really safe, save the program on two different disks.

I told you so department. Sooner or later this will happen to you: You'll have several disks, one of which you want to initialize. You'll pick up the wrong disk, one with valuable programs on it—possibly one you bought for $157.23. There will be no write-protect tab on the disk, and after you initialize it and blow away everything on it, you will realize your mistake. You cannot avoid that happening at least once, believe me.

Therefore, to insure that such a mistake is not irreversible, here's what to do: *make backups*. The copy program on system master disk that came with your computer is very handy. Simply run it and copy valuable programs from one disk to another. Take your original and put it somewhere out of reach. Then, when you accidentally erase a disk, you can make a copy.

Fixing Mistakes: Error Messages

By now you have probably entered something and received a message saying ?SYNTAX ERROR IN 30 or some such number (referring to line 30 or wherever an error was detected). This message will occur in the immediate mode as soon as you hit return and in the program mode as soon as you run your program. Depending on the error, you will get different types of messages. As you continue, you will see various messages, depending on the operation. For now, we'll concentrate on fixing errors in program lines rather than on the nature of the errors themselves. This process is referred to as *editing* programs.

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

```
NEW
10 TEXT : HOME
20 PRINT " AS LONG AS SOMETHING CAN"
30 PRINT : "GO WRONG" : REM LINE WITH ERROR
40 PRINT "IT WILL"
50 END
RUN
```

If you write the program exactly as it appears above, you will get a ?SYNTAX ERROR IN 30 message. Now, enter these lines:

```
30
LIST
```

What happened to line 30? You've just learned how to delete a line. Whenever you enter a line number and nothing else, you delete the line. You already know how to insert a line; so, to fix the program, just enter the following line:

```
30 PRINT "GO WRONG"
```

Now when you run the program, it should work fine. The error was inserting the colon between the PRINT statement and the words to be printed. Another way to fix the program is simply to reenter line 30 correctly without first deleting it, but you can see how simple it is to delete a line by entering the line number.

Now suppose there are several lines that you want to delete. Instead of having to write all of the line numbers, you can use the DEL statement. Try the following:

```
DEL 20,50
LIST
```

Only line 10 is left. To DELete a range of lines, enter the first line you want deleted, a comma, and the last line you want deleted.

The Apple Editor

Within your IIGS is a trusty little editor. Before reading a description of it, however, look through the discussion of utilities in Chapter 11. The editor in your Apple works just fine, but there are some really powerful ones available. The *Program Line Editor (PLE)* from Beagle Bros. is an excellent editor and is available commercially. Using *PLE* instead of the editor in

your IIGS will save you a considerable amount of time and speed in your progress toward learning to use programming statements. For now, though, we'll concentrate on the editor in your computer. It has one huge advantage—it's right in front of you.

To begin, we'll write another bad program and fix it. To keep things consistent, use the 40-column mode (if you're in 80 columns, just press the esc key and 4 to switch to 40 columns). Also, just for fun, here is something new. In line 40, after you've put in the first quotation marks, press the control key, and while holding it down, press the G key. You will hear a bell, but you will not see the G. Now, type in the following program and run it:

```
NEW
10 TEXT : HOME
20 PRINT "IF I CAN GOOF UP A PROGRAM ";
30 PRINT "I CAN" : FIX IT : REM BAD LINE
40 PRINT " ": REM control-G BETWEEN QUOTATION MARKS
50 END
RUN
```

You should see ?SYNTAX ERROR IN 30. To repair it, instead of rewriting line 30, follow these six steps:

1. LIST your program.
2. Press the esc key.
3. Press the up-arrow key repeatedly until the cursor is on line 30; then use the left- or right-arrow key to move the cursor to the start of the line.
4. Press the esc key again.
5. Press the right-arrow key until you get to the second quotation mark in line 30. Then press the space bar twice. This will erase the quotation mark and the colon.
6. Continue pressing the right-arrow key until the cursor is to the right of the word *IT*, type a quotation mark ("), and then press return.

LIST the program again. Line 30 should be correct and the REM statement should be gone. Now run the program. You should see the statement IF I CAN GOOF UP A PROGRAM I CAN FIX IT and hear a bell ring. When you LISTed your program, you should also have heard a bell ringing. The bell rings whenever you run or LIST a program with the hidden control-G. In longer programs, it is sometimes helpful to use control-G's in REM statements to flag various parts of the program. This makes locating parts of long programs easier.

However, there's a mystery. What happened to the REM statement in line 30? When you use your Apple's editor, you must retrace the entire line with the cursor if it is to be maintained in the line. Essentially, this is like rewriting the line; as soon as you hit the return key, the line is terminated. Since you did not retrace the REM statement, it was erased.

Let's learn more about the editor. Enter the following program. (Remember, in Applesoft, you can use a question mark to replace PRINT statements. If you LIST the program before you run it, you will see that all of the question marks have magically been transformed to PRINT statements.)

```
10 TEXT : HOME
20 ? "SOMETIMES I LIKE TO WRITE LONG, LONG LINES
   AND REALLY EXPRESS MYSELF AND TELL THE
   COMPUTER THE WAY THE WORLD WAS CREATED "
   : WHEW!
30 ? "AND SOMETIMES I LIKE SHORT LINES"
40 ? "DING, DING" : PRINT "      ": REM ENTER control-G
   TWICE
   INSIDE THE QUOTATION MARKS
50 END
LIST          (See what happened to the question marks?)
RUN
```

If you ran the program, it bombed. The problem is that the WHEW! is thrown in without a PRINT statement or quotation marks after the colon has terminated the line. To repair it, LIST the program, hit esc, walk the cursor up to line 20 (using the up-arrow key), then hit esc again. Starting at line 20, retrace the line to the point where the mistake has been made. To make it simple, remove the second quotation mark and, leaving the colon in place, add a quotation mark after the word WHEW!. Since the colon is now inside the quotation marks, it will be printed as part of the PRINT statement and will be ignored as a line-termination statement. After you've run the cursor over the rest of line 20, press return. Now run the program.

Something strange happens to the statement in line 20. It has a big space in it, but you did not put it there. Here's what has happened. When you retraced the line, the cursor went out into "space" as you traced past the end of the first line, and the computer thought you were entering spaces. To avoid this problem in the future, before entering the editor, enter

```
POKE 33,33
```

This statement resets the text window so that, as soon as the cursor hits the end of a line, instead of going into space, it immediately wraps around to the start of the next line. To test it, do the following:

1. Reenter line 20 in the original wrong way.
2. Type HOME and press return; then type POKE 33,33 and press return.
3. LIST the program, hit esc, walk the cursor to line 20, and press esc again.
4. Now retrace the line, and make the same repairs you did the first time. Run the program and notice how the line is now correctly formatted. The TEXT statement in line 10 resets the text window to its full size. If there were no TEXT statement in line 10, the lines would extend only part of the way. Generally speaking, whenever you are editing lines—especially with longer lines to repair—use POKE 33,33 first.

More Editing

As long as you are in the edit mode (after you have hit esc and before you press any key other than the arrow keys), the cursor, which is solid and has a + on it, can be moved without affecting anything on the screen. To get used to it, LIST a program, press esc, and walk around the screen using the arrow keys. None of the characters on the screen will be affected. Now, try out the editor on something a little trickier, but very useful. Suppose you have already typed in a program and run it when you realize that you left out a word in the middle of a line. Even with the tricks you've learned so far, you would need a long time to insert a single word since you'd have to retype the entire line. Here's how you can use the editor to do insertions. Try the following little program:

```
10 TEXT : HOME
20 PRINT "NOW IS THE TIME FOR ALL GOOD MEN TO COME
   TO THE AID OF THEIR COUNTRY"
30 END
```

So far, so good. But you meant to include *women* as well as *men* in line 20. You could retype the entire line, but all you really need to add is AND WOMEN after MEN. Do the following to make the change with the editor:

1. Enter

```
HOME : POKE 33,33 : LIST 20
```


2. Press esc, walk the cursor up to the beginning of line 20; then press esc again.
3. Trace over the line to the point where you want to insert AND WOMEN (right after the word MEN).
4. At the point of insertion, press esc and then the up-arrow key. This will take the cursor right above the line you're fixing.
5. Once you are above the line, press the space bar to release you from the editor, type in a space, and then enter AND WOMEN.
6. Now, press esc again to get back into the editor. Press the down-arrow to take you down into the line and the left-arrow key to take you back to the point where you want the insertion. Once at the insertion point, press the space bar to get out of the editor. Then retrace the remainder of line 20 with the right-arrow key.

The diagram illustrates the above steps:

...GOOD MEN ^ TO COME ...	(^ indicates point of insertion—when cursor is here, press esc and up-arrow)
AND WOMEN ...GOOD MEN TO COME...	
AND WOMEN ...GOOD MEN ^ TO COME ...	(^ points to exit the editor and begin retracing line with right-arrow key)

Now enter

```
TEXT : HOME : LIST 20
```

When you press return, you will find that line 20 now reads

```
20 PRINT "NOW IS THE TIME FOR ALL GOOD MEN AND
   WOMEN TO COME TO THE AID OF THEIR COUNTRY"
```

You will save yourself a great deal of time if you use the editor instead of retyping every mistake you make. (You'll save even more time if you use a commercial editor). For practice, there are a several pairs of lines below that need repair. The first line in each pair shows the wrong way and the second line shows the correct way. Since seemingly little things can make a big difference, there are a number of small changes to

be made. However, as you will soon see, those little glitches are the ones that are most likely to cause snags. Practice on these examples until you feel comfortable with the editor—time spent now will save you a great deal later.

Note: Once you have entered POKE 33,33, strange things will sometimes happen on your screen when you LIST a program. To set everything back to normal, enter the statement TEXT.

Editor Practice

```
50 PRINT NOBODY EVER WENT BROKE UNDERESTIMATING
   THE TASTE OF THE AMERICAN PUBLIC."
50 PRINT "NOBODY EVER WENT BROKE UNDERESTIMATING
   THE TASTE OF THE AMERICAN PUBLIC."

10 TEXT HOME
10 TEXT : HOME

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

HOME : TEXT PRINT "We're off
40 TEXT : HOME : PRINT "We're off!"
```

If you can fix 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 have done is to print out text, but that's not very different from having a fancy typewriter. Now, let's do some simple math operations to show that your computer can compute. Enter the following:

```
HOME
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 five-dollar calculator and my 12-year-old kid). Who said computers are smart? The programmer (you) is the one who's smart. Okay, let's give it a little tougher problem.

```
HOME
PRINT 7.87 * 123.65
```


Still nothing your calculator can't do, but it'd be a little rough on the 12-year-old.

As you progress, you can work with more aspects of mathematical problems. In the next chapter, you will learn how to store values in variables as well as several other things that would choke your calculator. For now, though, we'll just introduce the format of mathematical manipulations. The plus and minus signs work just as they do in regular math. For multiplication, the × is replaced by the asterisk (*), and for division, ÷ is replaced by the slash (/).

When you deal with more complex math, you will need to observe a certain order, called *precedence*, in which problems are executed. Depending on the operations used and the results you are attempting to obtain, you will use one order or another. For example, let's suppose you want to multiply the sum of two numbers by a third number—say, the sum of 15 and 20, multiplied by 3. If you enter

3 * 15 + 20

you will get 3 multiplied by 15 with 20 added on. That's not what you want. The reason for that is precedence: Multiplication precedes addition. To help you remember the precedence, here's a little program that you can run. With it you can refer to a sort of precedence chart on the screen and practice some math problems in the immediate mode to see the results. (This program is quite handy; you might want save it to disk to use later.)

```
10 TEXT : HOME
20 PRINT "1. — (MINUS SIGNS FOR NEGATIVE NUMBERS —
   NOT SUBTRACTION)"
30 PRINT "2. ^ (EXPONENTIATIONS)"
40 PRINT "3. * / (MULTIPLICATION AND DIVISION)"
50 PRINT "4. + — (ADDITIONS AND SUBTRACTIONS)"
60 PRINT "NOTE: ALL PRECEDENCE IS FROM LEFT TO
   RIGHT"
70 PRINT "YOUR COMPUTER FIRST EXECUTES THE
   NUMBERS IN PARENTHESES, WORKING ITS WAY FROM
   THE INSIDE OUT IN MULTIPLE PARENTHESES."
80 POKE 34,10 : END : REM THE POKE STATEMENT RESETS
   YOUR TEXT WINDOW
90 REM SO THAT YOU CAN DO MATH OPERATIONS
95 REM AND KEEP YOUR PRECEDENCE CHART ON THE
   SCREEN
```

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

Reordering Precedence

Once you get the knack of the order in which math operations work, there is a way to simplify organizing math problems. By placing two or more numbers in parentheses, you can move them up in priority. Go back to the example of adding 15 and 20 and then multiplying by 3, but this time use parentheses:

```
PRINT 3 * (15 + 20)
```

Since the multiplication sign has precedence over the addition sign, without the parentheses, you would get 3 times 15 plus 20. However, since all operations inside parentheses are executed first, your computer adds 15 and 20 and then multiplies the sum by 3. If more than a single set of parentheses is used in an equation, then the innermost is executed first, with the computer 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 operation is a prisoner enclosed by walls of parentheses. To escape the dungeon, the prisoner must first get out of the innermost cell. Then the prisoner goes to the right and releases 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 analogy is lethal for our poor escaping prisoners. Try 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)
```

Three Exercises

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

1. Multiply the sum of 4, 9, and 20 by 15.
2. Multiply 3.14159265 by 35; then multiply the result by itself. (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.)
3. 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 checks.

Summary

This chapter has covered the most basic aspects of programming. At this point you should be able to use the editor in your IIGS, and write statements in the immediate and program (deferred) modes. Also, you should be able to manipulate basic math operations. However, we have just begun to uncover the power of the IIGS, and at this stage it is being treated more as a glorified calculator than as a computer. Nevertheless, this chapter is extremely important, for it is the foundation upon which your understanding of programming will be built.

If you do not understand something, review it 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 first try to get everything to do what it is supposed to do and what you want it to do. Chapter 3 will take you into the realm of computer programming and increase your understanding of the IIGS considerably. If you proceed one step at a time, you will be amazed at the power you have at your fingertips, and will discover how easy it is to program. You'll be leaving the realm of calculator-like statements and getting down to some honest-to-goodness computer work—where the fun really begins.

3

Moving Along

Moving Along

In the last chapter you learned how to execute statements in both the immediate and program modes. From now on, we will concentrate on the program mode, tying various statements together within a program. We'll still use the immediate mode to provide simple examples of how certain statements work.

As you learn more about statements, it would be a good idea to start saving the example programs on your disk. You can use them for review or for a quick lookup of examples. Use filenames that you can recognize, such as VARIABLE EXAMPLE or HOW-TO SUBROUTINES. Remember that each file must have a different name; be sure to number example filenames (for example, ARRAYS 1, ARRAYS 2).

Variables

Perhaps the single most important function of the computer is its use of variables. Basically, a variable is a symbol that can have more than a single value. If you say, for example, X = 10, you assign the value of 10 to the variable called X. Try this:

```
X = 10
PRINT X
```

Your computer's response will be 10. Now type in

```
X=55.7
PRINT X
```

This time you'll get 55.7.

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

```
X = 10
Y = 15
PRINT X + Y
```

The IIGS will respond with 25.

As you see, variables can be treated in the same way as math problems using numbers. However, instead of using the

numbers, you use the variables. Now try this short program that uses variables to calculate the area of a circle.

Area of a Circle

```
10 TEXT : HOME
20 PI = 3.14159265
25 REM YOU REMEMBER THE VALUE OF PI FROM YOUR
   GEOMETRY CLASS
30 R = 15 : REM R IS THE RADIUS OF CIRCLE
40 PRINT (PI * R) * (PI * R)
45 REM THIS GIVES THE SQUARE OF PI TIMES THE RADIUS
50 END
```

When you run the program, you will get the area of a circle with a radius of 15. By changing the value of R in line 30, you'll find that it's a simple matter to quickly calculate the area of any circle you want. Since the example squares a result, you can use the exponential sign—^. Change line 40 to

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

That saves typing, doesn't it? Run the program again and see if you get the same results. You should. Also, try changing the value of R to see the different areas of circles.

Variable Names

When you name a variable, the computer looks at only the first two characters. For example, if you name a variable NUMBER, all your computer is interested in is NU. Try this:

```
NUMBER = 63
PRINT NU
```

You get 63 even though you entered only the first two characters of the variable called NUMBER. Now try this one:

```
NUMBER = 123
PRINT NUTTY
```

The value 123 is printed because the only characters of interest to the computer are still the first two.

You may think that the best thing to do is to use variable names of only two characters. While you're getting used to variables, that's probably not a bad idea. However, as you advance into more sophisticated programs, you'll find it helpful to use descriptive variable names. For example, the following program uses MEAN as a descriptive variable name:


```
10 TEXT : HOME
20 A = 15 : B = 23 : C = 38
30 MEAN = (A + B + C ) / 3
40 PRINT MEAN
50 END
```

Even if the program were a hundred lines or more, you would know what the variable MEAN does—it calculates a mean. Of course, you'd have to be careful not to have another variable named MEATBALL, or some other name beginning with *ME*. But assigning a meaningful name certainly makes it easier to understand what a variable does. There are two other considerations in naming variables: Don't use reserved words (commands, statements, and other keywords), and begin variable names with an uppercase letter. Table 3-1 gives some examples of valid and invalid variable names:

Table 3-1. Naming Variables

TEXTURE = 987	Invalid	TEXT, a reserved word, is part of the variable name
TE = 99	Valid	Even though reserved word TEXT begins with TE, only part of the reserved word is used in variable name
R1 = 321	Valid	First character is an uppercase letter
1R = 55	Invalid	First character is not an uppercase letter
HOMEWORK = 222	Invalid	Variable name contains reserved word HOME
TO = 983	Invalid	TO is a reserved two-character word
ADFTDCVRRWDAF = 10	Valid	But really dumb

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

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

Types of Variables

There are three types of variables: real, integer, and string. Let's look at how they are used.

Real variables. So far, we have used only real or floating-point variables in our examples. Any variable that begins with an uppercase letter and does not end with a dollar sign or percent sign is a real variable. The value for a real variable can range from 0 to ±9.99999999E+37. The *E* is scientific notation for very large 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.) Think of real variables as being able to hold just about any number you would need, including the decimal fractions.

Integer variables. Integer variables contain only integers, or whole numbers (without fractions). Here are some examples:

```
AB% = 345
K% = R% + N%
ADD% = ADD% + NUM%
WXY% = 88 + LR%
```

The values of integer variables can range from −32767 to + 32767, and, as with real variables, only the first two characters are read. However, the % is always read, no matter how many characters are used. So, a variable named WA% is the same as WAX%. Yet a variable named ABC is different from one named ABC%. Therefore, you could use both variables in the same program, and each would be considered unique.

Since their range is smaller than that of real variables, integer variables have limited applications. However, because integer variables take up less memory and execute faster than real variables, they are often useful. You can use them in mathematical operations in the same way you use real variables, but integers don't store fractions—a factor you must take into account when using operations that involve division and similar fraction operations. Try some of the following operations from the immediate mode to see how they work:

```
A% = 15 : B% = 21 : C% = B% + A% : PRINT C%
        36
LL% = 17 :JJ% = LL% / 5 : PRINT JJ%
        3
Z% = −11 :XY% = Z% + 51 : PRINT XY%
        40
```


String variables. String variables are extremely useful in formatting what you will see on the screen. Like real and integer variables, they are sent to the screen by the PRINT statement. However, rather than printing numbers only, string variables send all kinds of characters, called strings, to the screen. String variables are indicated by a dollar sign at the end of a variable. For example, A$, BAD$, and PULL$ are all legitimate string variables. (In computer parlance, we say *string* instead of dollar sign. Thus, our examples are called A string, BAD string, and PULL string.)

String variables are defined by placing the string inside quotation marks, just as we did with the other messages that were printed out. Let's try a few examples from the immediate mode:

```
ABC$ = "ABC" : PRINT ABC$
KAT$ = "CAT" : PRINT KAT$
NUMBER$ = "123456789" : PRINT NUMBER$
B1$ = "5 + 10 + 20" : PRINT B1$
```

Like real and integer variables, string variables use only the first two characters, and must begin with an uppercase letter and use nonreserved words. More important, you may have noticed in the examples that numbers in string variables are not treated as numbers, but as words or messages. For example, you probably noticed that when you printed B1$, instead of printing 35 (the sum of 5, 10, and 20), B1$ printed out exactly what you put inside the quotation marks: 5 + 10 + 20. Don't attempt to do math with string variables. (In later chapters, you'll see some tricks for converting string variables to numeric variables, but for now just treat them as messages.)

Starting a Checkbook Program

Let's put your accumulated knowledge together and write a program that uses variables. We will begin writing a program that subtracts the amount of a check from your checkbook balance and prints the new balance. This program will be the basis for a check-balancer to be developed later.

```
10  TEXT : HOME
20  BALANCE = 571.88
25  REM ANY FIGURE WILL DO
27  REM BALANCE (BA) IS A REAL VARIABLE
30  CHECK = 29.95
35  REM WHAT YOU LAST SPENT IN THE COMPUTER STORE
37  REM CHECK (CH) IS A REAL VARIABLE
```

```
40  B$ = "YOUR BEGINNING BALANCE IS $"
50  C$ = "YOUR CHECK IS FOR $"
60  NB$ = "YOUR NEW BALANCE IS $"
65  REM B$, C$, AND NB$ ARE STRING VARIABLES
70  PRINT B$;BALANCE
80  PRINT C$;CHECK
90  N = BALANCE - CHECK
100 PRINT NB$; N
110 END
```

Since this is a fairly long program, make sure you type everything in correctly. It is critical that you distinguish between commas, semicolons, and periods. Save the program to disk. To play with it, you can change the values in lines 20 and 30. Here's a quick review of what the program includes:

Step 1. First, we define the real variables BALANCE and CHECK (which your IIGS reads as BA and CH since it cares about only the first two characters).

Step 2. Next, we define string variables B$, C$, and NB$ to use as labels in screen formatting (lines 40–60).

Step 3. Finally, we print out all the information using our variables, with one new variable, N, defined as the difference between BALANCE and CHECK.

Note how we format the output (what you see on your screen) of the PRINT statements. The semicolon between the variables accomplishes two things: It tells the computer where one variable ends and the next begins, and it tells the computer to print the second variable right after the first one. Thus, it takes the string variable NB$ (YOUR NEW BALANCE IS $#) and places the value of the real variable N right after the dollar sign (exactly where we placed the pound sign, #).

Formatting Output with Punctuation

Later, there will be more detail about formatting output, but for now here's a brief look at using punctuation to format text. We'll use the comma and semicolon and "new lines" to illustrate basic formatting. Enter the following program:

```
NEW
10 TEXT : HOME
20 A$ = "HERE" : B$ = "THERE" : C$ = "WHERE"
30 PRINT A$; : PRINT B$; : PRINT C$; : REM SEMICOLONS
35 PRINT
40 PRINT A$, : PRINT B$, : PRINT C$,: REM COMMAS
45 PRINT : REM A PRINT BY ITSELF
```



```
47 REM GIVES A VERTICAL SPACE IN FORMATTING
50 PRINT A$ : PRINT B$ : PRINT C$ : REM 'NEW LINES'
60 END
```

When you run the program, you can see that the little differences in lines 30, 40, and 50 make big differences on the screen. The first set is all crammed together, the second set is spaced evenly across the screen, and the third set is stacked one on top of the other. As you saw in the previous program, semicolons put numbers and strings right next to one another. However, using commas after a PRINTed variable will space output in groups of three across the screen; using colons or new line numbers will make the output start on a new line. A PRINT statement used alone will put a vertical *line feed* between statements. Try the following program to see how PRINT statements can be used by themselves.

```
NEW
10 TEXT : HOME
20 PRINT "WHENEVER YOU PUT IN A PRINT STATEMENT";
25 REM NOTE PLACEMENT OF SEMICOLON
30 PRINT "ALL BY ITSELF, IT GIVES A LINE FEED."
40 PRINT
50 PRINT "SEE WHAT I MEAN?"
60 END
```

Play with commas, semicolons, and colons with variables and string variables until you get the hang of using them. They are very important and are often the source of program bugs.

Bugs and bombs. We've mentioned bugs and bombs in programs, but have never explained what they are. These terms are computer lingo. *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.

Input/Output

Input and output, often referred to as I/O, are ways of putting something into your computer and getting it out again. Generally, you put *in* information from the keyboard, save it to disk or tape, and then later put it in again from the disk drive or cassette recorder. When you want information *out* of the computer, you want it to go to the screen or printer.

Until now you have entered information into the computer from the keyboard, either in the program or immediate

mode. Using the PRINT statement, you have sent information out to the screen. There are other ways in which you can INPUT information with a combination of programming and keyboard statements. Let's look at some of them and make the checkbook program a lot simpler to use.

INPUT Statements

When the INPUT statement is placed in a program, it expects a response from the keyboard and then a RETURN. It must be part of a program and cannot be used from the immediate mode. Here's a simple example:

```
NEW
10 TEXT : HOME
20 INPUT X : REM X IS A NUMERIC VARIABLE SO ENTER A
   NUMBER
30 PRINT X
40 END
```

Run the program. Your screen will go blank, and a question mark and blinking cursor will sit there until you enter a number. Then the computer will print the number you just entered. Now let's try INPUTting the same information, but this time 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 this program:

```
NEW
10 TEXT : HOME
20 INPUT "ENTER YOUR AGE "; X
30 HOME : PRINT : PRINT : PRINT
40 PRINT "YOUR AGE IS "; X
```

When you run the program, you will see that the presentation is a little more interesting. Also notice that there is no END statement at the end of the program. Applesoft BASIC does not require an END statement, though usually it's a good idea to add one. As you get into more advanced topics, you'll see that a program can jump around. You may want it to end in the middle, and an END statement will be necessary to prevent your program from crashing into an area where it shouldn't go. Using an END statement has not really been necessary at this point, but it's a good habit to develop.

Let's use the INPUT statement to soup up the program a little more.


```
NEW
10 TEXT : HOME
20 INPUT "ENTER YOUR NAME -> "; NA$
30 PRINT
40 INPUT "ENTER YOUR AGE -> "; AG%
50 PRINT
60 INPUT "PRESS <RETURN> TO CONTINUE "; RT$
70 HOME : ? : ? : ? : ? : ? : REM USING "?" AS SUBSTITUTES
   FOR PRINT
80 PRINT NA$; " IS "; AG% ; " YEARS OLD."
90 REM BE CAREFUL WHERE YOU PUT QUOTATION MARKS
95 REM AND SEMICOLONS IN LINE 80
100 END
```

Now you're making progress. You can enter information as numeric or string variables, and the output will be formatted so that you know what's going on. As your programs become larger and more complicated, it's very important to connect your string variables and numeric variables in such a way that you can easily see what the numbers on the screen mean. A computer wouldn't be very helpful if it filled the screen with numbers, but you didn't know what they meant.

Line 60 in the program is the format for a pause in the program. RT$ doesn't hold any information, but INPUT statements expect something from the keyboard. A variable, RT$ (for RETURN), is as good as any.

GETting Information

The GET statement is something like the INPUT statement, except it accepts only a single key and doesn't require that you press return. To see how it works, try this program:

```
NEW
10 TEXT : HOME
20 ? : ? : ? : ?
30 PRINT " ENTER A NUMBER FROM 0-9 "; : GET N
40 ? : ?
50 PRINT " HIT ANY KEY TO CONTINUE "; : GET K$
60 HOME : ? : ? : ? : ?
70 PRINT "YOUR NUMBER IS ->" ; N
80 END
```

As soon as you hit a key, the GET statement records the key value and the program proceeds. With an INPUT statement, you first enter information and then press the return key before the program executes. The good thing about the GET statement is that it provides a faster way to enter and

execute from the keyboard. The problem is that you can enter only a single character before the program takes off again. If you press the wrong key, you'll have no chance to correct your error before pressing the return key as you do with the INPUT statement.

READing In DATA

A third way to enter data into a program is with READ and DATA statements. However, instead of being entered 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 statement is associated with a variable that looks at the next DATA statement and places the numeric value or string in the variable. Look at the following example:

```
NEW
10 TEXT : HOME
20 READ NA$ : REM READS NAME
30 READ OC$ : REM READS OCCUPATION
40 READ SN : REM READS STREET NUMBER
50 READ ST$ : REM READS STREET NAME
60 READ CT$ : REM READS CITY
70 READ SA$ : REM READS STATE
80 READ ZIP : REM READS ZIP CODE
90 PRINT : PRINT : PRINT
100 REM BEGIN PRINTING OUT WHAT 'READ' READ IN
105 REM BE CAREFUL TO PUT IN EVERYTHING EXACTLY AS
    IT IS LISTED
110 PRINT NA$
120 PRINT OC$
130 PRINT SN; " " ; ST$
140 PRINT CT$ ; ", " ; SA$ ;" "; ZIP
150 END
1000 DATA Sam Spade, Detective, 112, Post Street
1010 DATA San Francisco, California, 92929
```

In the DATA statements, a comma separates the various elements, unless the DATA statement is at the end of a line. If one of the elements is out of place or if a comma is omitted, strange things can happen. For example, if a 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 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 the 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 it's in memory. Then enter the following line:

85 READ EX$

Now run the program. You should get an OUT OF DATA ERROR. This means that you have a READ statement without enough DATA statements (or elements). Be sure that your DATA statements contain enough elements to take care of the READ statements, and that the variables in your READ statements are compatible with the elements of the DATA statements. (In other words, numeric variables must read numbers and string variables must read strings.) To repair the program, simply enter

1020 DATA WORD

This will give it something to READ. (Of course, you could delete line 85.)

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

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

Both numbers and commas will happily be accepted by a READ statement with a string variable since everything is enclosed within quotation marks.

Now remove the quotation marks and run it again. This time it will print only up to the first comma, 10 DOWNING ST, but the string variable EX$ has no problem accepting a numeric character. (However, since 10 is read as a string, you cannot use it 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'll quickly discover that they can go anywhere and are READ in their order of placement within the program.

Looping with FOR-NEXT

The FOR-NEXT loop is one of the most useful operations in BASIC programming. With it, you can 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. Here's a simple example to get started:

```
NEW
10 TEXT : HOME
20 NA$ = "<YOUR NAME>"
30 FOR X = 1 TO 10 : REM BEGINNING OF LOOP
40 PRINT NA$
50 NEXT X : REM LOOP TERMINAL
60 END
```

When you run the program, you will see your name printed ten times along the left side of the screen. Okay, not very impressive, but you'll see how useful this can be. But, first, here's another simple illustration that shows what's happening to X as the loop is executed:

```
NEW
10 TEXT : HOME
20 FOR X = 1 TO 10
30 PRINT X
40 NEXT X
```

When you run the program, you'll see that the value of X 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 1 and, in our example, ending with 10.

A Practical Use for Loops

You can do something practical with a loop: You can fix up the checkbook program. But, first, you should become acquainted with some fancy output statements:

INVERSE
FLASH
NORMAL

Until now, your output has been in NORMAL mode, the default mode. However, you can also make text INVERSE or FLASH if you want, and then reset it with a NORMAL statement. Here's a quick look at how these output statements work:


```
NEW
10 TEXT : HOME
20 INVERSE : PRINT " INVERSE "
30 FLASH : PRINT : PRINT " FLASH "
40 NORMAL : PRINT : PRINT "NORMAL"
```

These statements help highlight output to make it easier to see what you're supposed to do. Now back to work.

In the souped-up checkbook program, we'll use variables in several ways. First, the FOR-NEXT loop will use a variable; let's stick with tradition and use *I*. Second, a variable will indicate the number of loops to be executed. We'll use N%, an integer variable. Finally, variables will be used for the balance, the amount of the check, and the new balance.

```
NEW
10 TEXT : HOME
20 CB$ = "CHECKBOOK"
30 PRINT : PRINT : FLASH : PRINT CB$ : NORMAL
40 INPUT "HOW MANY CHECKS? ->" ; N%
50 INPUT "WHAT IS YOUR CURRENT BALANCE? ->" ;BA
60 REM BEGIN LOOP
70 FOR X = 1 TO N%
80    PRINT "YOUR BALANCE IS NOW $";BA
90    INVERSE : PRINT " AMOUNT OF CHECK #";X; "-> "; :
      NORMAL
100 INPUT CK : REM VARIABLE FOR CHECK
110 BA = BA - CK : REM KEEPS A RUNNING BALANCE
120 NEXT X : REM LOOP TERMINAL
130 HOME : REM CLEAR SCREEN WHEN ALL CHECKS ARE
    ENTERED
140 PRINT : PRINT : PRINT
150 PRINT "YOU NOW HAVE $"; BA ; " IN YOUR ACCOUNT"
160 PRINT : FLASH : PRINT " THANK YOU AND COME AGAIN
    " : NORMAL
170 END
```

The checkbook program is becoming easier to use; and that, after all, is the purpose of computers. Now, let's look at some more loops.

Nested Loops

With certain applications, you must have one or more FOR-NEXT loops working inside each other. Here's a simple application: Suppose you have two teams with ten members on each team. You want to make a team roster indicating the team number (1 or 2) and member number (1–10). Using a nested loop, you can do this in the following program:

```
NEW
10 TEXT : HOME
20 FOR T = 1 TO 2 : REM T FOR TEAM #
30   FOR M = 1 TO 10 : REM M FOR MEMBER #
40     PRINT "TEAM #" ; T ; "PLAYER #"; M
50   NEXT M
60 NEXT T
70 END
```

It's important to keep the loops straight when you use nested loops. The innermost loop (the M loop in the example) must not have any other FOR or NEXT statement inside 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, down to the smallest fish.

This is the 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
```

Each loop begins (a FOR statement is executed) and is terminated (a NEXT statement is encountered) in a nested sequence. If you have ever stacked a set of different-sized cooking bowls, you've seen how each one fits inside the other; this 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 Backward

Loops can go one step at a time, as we have seen, or they can step at different increments. For example, this program steps by ten:

```
NEW
10 TEXT : HOME
20 FOR X = 10 TO 100 STEP 10
30   PRINT X
40 NEXT X
```

You can increment the count by whatever amount you want. You can use variables or anything else that has a numeric value. For example,


```
NEW
10 TEXT : HOME
20 K = 5 : N = 25
30 FOR X = K TO N STEP K
40    PRINT X
50 NEXT
```

Go ahead and run the program. But, wait. In line 50, you detect a bug—a typo and a big mistake. After the word NEXT, there should be an X, but there is none, right? Actually, in Applesoft BASIC you really don't need it, and you can save a little memory if you use NEXT statements without the variable name. Even in nested loops, as long as you put in enough NEXT statements, you can run your program without variable names after NEXT statements. However, it's good programming practice to use variable names after NEXT statements, especially in nested loops, so that you can keep everything straight. It's also possible to go backward. Try this program:

```
NEW
10 FOR X = 4 TO 1 STEP -1
20     PRINT "FINISHING POSITION IN RACE =";X
30 NEXT X
```

As you get into more sophisticated (and useful) programs, you'll begin to see how these different features of Applesoft BASIC can be very convenient. Often, at first, you may not see the practicality of a statement, but when you need it later, you'll wonder how you could program without it.

What happened to the indentions? You may have noticed that the program lines inside the loops are indented. If you tried indenting on your IIGS, you probably found that as soon as you LISTed your program, all the indentions were gone. Unfortunately, that will happen, and without special utilities, there's nothing you can do about it. However, don't worry. Indenting loops is a programming convention to make clearer what the program is doing, but indentions have no effect at all on your programs.

Counters

Sometimes, in your programs, you'll need to count the number of times a loop is executed and keep a record of it 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 that are incremented, usually by one, each time a loop is executed.

This next program illustrates the use of a counter:

```
NEW
10 TEXT : HOME
20 FOR X = 3 TO 99 STEP 3
30   PRINT X
40   N = N + 1 : REM THIS IS THE COUNTER
50 NEXT X
60 PRINT : PRINT "YOUR LOOP EXECUTED "; N ; " TIMES."
```

The first time the loop is entered, the value of N is 0. When the program gets to line 40, the value of 1 is added to N to make it 1 (0 + 1 = 1). The second time through the loop, the value of N begins at 1, then 1 is added, and at the top of the loop (line 50), the value of N is 2. This continues until the program exits the loop. After all the looping is finished, presto—your N tells you how many times the loop has been 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 + (X * 2)
```

Then run the 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 you have started to learn real programming. One of the most important concepts is that of the variable. The significant feature of variables is that they vary (they change depending on what your program does). This is true not only of numeric variables, but also of string variables. The various input statements show how you enter values or strings into variables depending on what you want the computer to compute.

Finally, you have learned how to use loops. You can, with minimal effort, tell the computer to go through a process several times with a single set of instructions. With loops, you can set the parameters of an operation at any increment you want, and then sit back and let the IIGS go to work.

However, you have only just begun programming. In the next chapter you'll be introduced to more statements and operations that allow you to delve deeper into the capabilities of the Apple IIGS and that make programming jobs easier. The more statements you know, the less work it takes to write a program.

4

Branching Out

Branching Out

In this chapter we'll begin exploring new programming techniques that will geometrically increase your programming ability. You'll be learning more sophisticated techniques, but by taking each a step at a time, you will be able to use them with ease. Later, when you're 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 already mastered to get a job done. After all, why use "complicated" commands to do what simpler ones can do?

The answer to that is simplicity. If one complicated command can do the work of ten simple commands, which one is simpler? As you advance into more sophisticated applications, your programs will become longer and subject to more bugs. The more commands you have to sift through, the more difficult it will be to find the bugs. While you're learning it is perfectly okay to write a long program that uses many simple commands, but you should begin thinking about shortcuts that come through the use of more advanced commands.

As well as maximizing your knowledge of various commands, let your IIGS perform the computing. This may sound strange, but novices often will figure everything out for the computer and use it as a glorified calculator. In Chapter 3, we set up a counter that counted the number of times a loop was executed when a STEP 3 loop was used. 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. As you learn new commands, think about how you can use them to perform the calculations you have had to work out yourself.

Branching

So far, with the exception of loops, our programs have gone straight from the top to the bottom. However, if the IIGS is to do any real decision making, you must be able to give it options. When a program leaves a straight path, it is either *looping* or *branching*. You already know the purpose of a loop, so

let's turn our attention to branching, using IF-THEN and GOTO commands.

Consider the following program (by now you know to clear memory with NEW, so they'll no longer appear at the beginning of each program):

```
10 TEXT : HOME
20 PRINT "CHOOSE ONE OF THE FOLLOWING BY NUMBER: "
30 PRINT
40 PRINT "1. APPLES"
50 PRINT "2. ORANGES"
60 PRINT "3. PEACHES"
70 PRINT "4. WATERMELONS"
80 PRINT
90 INPUT "WHICH? "; X
100 HOME
110 IF X = 1 THEN GOTO 200
120 IF X = 2 THEN GOTO 300
130 IF X = 3 THEN GOTO 400
140 IF X = 4 THEN GOTO 500
150 GOTO 10
160 REM THIS IS A TRAP
165 REM TO MAKE SURE THE USER CHOOSES 1, 2, 3, OR 4
200 PRINT "APPLES" : END
300 PRINT "ORANGES" : END
400 PRINT "PEACHES" : END
500 PRINT "WATERMELONS" : END
```

As you can see, the IIGS branches to the appropriate place, does what it has been told, and ends. Not very inspiring, admittedly, but it is a clear example. Now, try something a little more practical for your kids to play with in their math homework:

Addition Game

```
10 TEXT : HOME
20 TI$=" ADDITION GAME ": INVERSE : PRINT TI$ :
   NORMAL
30 PRINT : PRINT
40 INPUT "ENTER FIRST NUMBER ->" ; A
50 PRINT
60 INPUT "ENTER SECOND NUMBER ->" ; B
70 PRINT
80 PRINT "WHAT IS "; A ; "+" ; B ; : INPUT C
90 IF C = A + B THEN GOTO 200
100 PRINT : INVERSE : PRINT "THAT'S NOT QUITE IT. TRY
    AGAIN."
110 NORMAL : PRINT
```



```
120 GOTO 80
200 FLASH : PRINT " THAT'S RIGHT! VERY GOOD " :
    NORMAL
210 PRINT
220 PRINT "WOULD YOU LIKE TO DO MORE? (Y/N): "; :
    GET
    AN$
230 IF AN$ = "Y" THEN HOME : GOTO 30
240 HOME : PRINT : PRINT : PRINT
250 PRINT "HOPE TO SEE YOU AGAIN SOON" : END
```

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

What's your name? Children of all ages like to have their names displayed. See if you can change the "Addition Game" so that it asks the child's name. When the program responds to an input with either a correction or affirmation, it will mention the child's name (for example, THAT'S RIGHT! VERY GOOD, SAM). Use NA$ as the name variable.

Look carefully at the program to learn something about IF-THEN statements. First, note in line 230 that the branch is to clear the screen (HOME) if AN$ = "Y". If any other response is encountered, the program ends. You may wonder why the program does not branch to line 30, regardless of the response, since GOTO 30 comes after a colon, making it a new line. The reason is that after an IF statement for which the specified condition is not met, the program immediately drops to the next line number. That is, any statements after a colon in a line that begins with an IF statement will be executed only if the condition of the IF statement is met.

Second, AN$ is tested against "Y", and not simply a Y without quotation marks. Remember that AN$ is a string and not a numeric variable. In setting the conditional, you must remember what kind of variable is used. On the other hand, if you use a numeric variable, such as AN or AN%, you could use a line such as

IF AN = 1 THEN....

Relationals

So far, we have used only the equal condition (=) to determine whether or not a program should branch. There are other states, referred to as *relationals*, that can also be used. This is a complete list of the relationals:

Symbol	Meaning
=	Equal to
<	Less than
>	Greater than
<>	Not equal to
>=	Greater than or equal to
<=	Less than or equal to

Let's try some of these out and then examine them for their full power. Here are three quick programs:

```
10 TEXT : HOME
20 INPUT "NUMBER 1->";A
30 INPUT "NUMBER 2->";B
40 IF A > B THEN GOTO 100
50 IF A < B THEN GOTO 200
60 IF A = B THEN GOTO 300
100 PRINT "NUMBER 1 IS GREATER THAN NUMBER 2" : END
200 PRINT "NUMBER 1 IS LESS THAN NUMBER 2" : END
300 PRINT "NUMBER 1 IS EQUAL TO NUMBER 2"
```

```
10 TEXT : HOME
20 INPUT "DO YOU WANT TO CONTINUE? (Y/N)"; AN$
30 IF AN$ <> "Y" THEN END
40 GOTO 10
```

```
10 TEXT : HOME
20 INPUT "HOW OLD ARE YOU? "; AG%
30 IF AG% >= 21 THEN GOTO 100
40 HOME : PRINT : PRINT "Sorry, you must be 21 or older to
   come in here."
50 END
100 HOME : PRINT
110 PRINT "Welcome to the adult programming center!"
```

Now you have an idea of how relationals can be used with IF-THEN statements. Note that they work with strings as well as with numeric variables. However, there is still another way to use relationals. Try the following from the immediate mode:

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

Your computer should respond with 0. This is a logical operation: If a condition is false, your IIGS responds with 0; if it is true, it responds with 1. Now try the following:

```
10 TEXT : HOME
20 A = 10
30 B = 20
40 C = A > B
50 PRINT C
```


When you run the program, you again get 0. This is because the variable C is defined as the result of the test of A being greater than B. Since A is less than B, the variable C is 0, or false. Now, take it a step further:

```
10 TEXT : HOME
20 A = 10
30 B = 20
40 C = A > B
50 IF C = 0 THEN PRINT "A IS LESS THAN B" : END
60 IF C = 1 THEN PRINT "A IS GREATER THAN B"
```

Later, you will see further applications of these logical operations. For now, though, it's important to understand that a true condition is represented by 1 and a false condition by 0.

AND, OR, and NOT

Sometimes, you'll need to set up more than a single relational. Suppose, for example, that you are organizing your finances into three categories of expenses: those under $10, those between $10 and $100, and those over $100.

With relationals, it is simple to compare input under $10 and over $100. But what if you want to do something in between? In this case you might have some difficulty without additional commands. The AND, OR, and NOT statements permit you to set ranges with relationals.

AND	**If all conditions are met, then true**
OR	**If one condition is met, then true**
NOT	**If condition is not met, then true**

Here's an example:

```
10 TEXT : HOME
20 INPUT "ENTER AMOUNT ->$"; A
30 IF A < 10 THEN 100
40 IF A >= 10 AND A <= 100 THEN 200
50 IF A > 100 THEN 300
100 PRINT " PETTY CASH " : GOTO 400
200 INVERSE : PRINT " GENERAL EXPENSES " : NORMAL :
GOTO 400
300 FLASH : PRINT " BIG BUCKS " : NORMAL
400 PRINT " DO YOU WISH TO CONTINUE? ";
410 GET AN$
420 IF AN$ < > "Y" AND AN$ < > "N" THEN PRINT
    "ANSWER 'Y' OR 'N' PLEASE " : GOTO 400
430 IF AN$ = "Y" THEN 10
440 HOME : PRINT "GOODBYE"
```

In line 40, the conditional branch is set to be both greater than 10 *and* equal to or less than 100. The variable A has to meet both conditions to branch. Similarly, in line 420, again using the AND statement, the response must be either Y or N.

If you look carefully, you may wonder about some dubious formatting in the program. There are several conditional IF-THEN lines that simply say THEN 100, THEN 200, and so forth. Shouldn't there be GOTO statements as well? This brings up another feature of Applesoft BASIC. When you use IF-THEN statements and want to branch to another line if the comparison is true, you can drop the GOTO and simply put in the line number. Until you become more familiar with programming, you might want to continue using GOTO statements after IF-THEN statements, but they are not required.

You may also have a question about the AND statement in line 420. When we say, in English, if something is Y or N, we mean that it must be one or the other. However, in programming, if we use OR, we are telling the program to branch if *either* condition is met. Thus, if line 420 is written as

```
IF AN$ < > "Y" OR AN$ < > "N" THEN PRINT "ANSWER 'Y'
   OR 'N' PLEASE "
GOTO 400
```

the program will branch if AN$ is not equal to *either* Y or N. For example, if you respond with Y, that Y will not be equal to N; and so the program will branch to "ANSWER 'Y' OR 'N' PLEASE"—not what was intended. To check this, change the AND to OR in line 420, and run the program again.

Now, let's use OR and NOT statements in a program:

```
10 TEXT : HOME
20 READ A
30 READ B
40 READ C
50 DATA 10,20,30
60 IF A + B = C OR A < B OR A - B = C THEN 100
70 END
100 HOME : PRINT "ONE OF 'EM MUST BE TRUE"
```

Looking at line 60, you can see that A − B does not equal C. However, A + B does equal C, and A is less than B. When the OR statement is used, only one statement must be true to cause a branch. Now, try the following program:

```
10 TEXT : HOME
20 READ A : READ B : READ C
30 DATA 10,20,30
```



```
40 Z = A - B = C
50 IF NOT Z THEN 100
60 END
100 PRINT " THAT'S RIGHT! A - B = C IS NOT RIGHT! DID I
   SAY THAT RIGHT?"
```

As you can see from the example, it's possible to use the negation of a formula to calculate a branch condition. In most cases, you will use < > (not equal) or the positive case, but at other times, it will be simpler to employ NOT.

Subroutines

Frequently, you'll want your computer to perform an operation at several different places in a program. You can repeat the instructions again and again, or you can sprinkle GOTOs to return to the original spot after a branch to the operation.

On the other hand, you can set up *subroutines* and jump to them by using GOSUB, and then get back to the starting point by using RETURN. Up to a point, GOSUB works much like GOTO since it sends a program bouncing off to a line out of sequence. The RETURN command also is something like GOTO since it sends your program to an out-of-sequence line. However, the GOSUB–RETURN pair is unique in what it does. This simple example illustrates how it works:

```
10 TEXT : HOME
20 A$ = "HELLO" : GOSUB 100
30 A$ = "HOW ARE YOU TODAY?" : GOSUB 100
40 A$ = "I'M FINE" : GOSUB 100
50 END
100 PRINT A$
110 RETURN
```

The example shows that a GOSUB statement works exactly like a command 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 makes it 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, change line 100 to something more elaborate. (This is getting a bit ahead, but the example illustrates something very useful.) Try the following:

```
100 HTAB 20 - LEN (A$)/2 : PRINT A$
```

Now when you run the program, all of the strings will be centered. A single routine handles the centering. Instead of your having to rewrite the routine every time you want a string centered, just use a GOSUB to line 100.

Block it. We've not discussed program structure very much—in part, because it's not been necessary. However, as the instruction set grows, so too does the possibility for errors. By now if you haven't made an error, you haven't been entering the programs. One way to minimize errors, especially involving GOSUBs, is to organize them into coherent blocks. Basically, a block is a subroutine within a range of lines. For example, you might block subroutines by hundreds or thousands, depending on how long the subroutines are. You might have subroutines beginning at lines 500, 600, and 700. It doesn't matter whether a subroutine is one line or ten lines. As long as it's confined to a particular block, it will be easier to debug, easier for you and others to understand what is happening in the program, and generally, good programming practice.

Computed GOTO and GOSUB

Now we're going to get a little fancier, with some additions that will result in clearer and simpler programming. As you have seen, you can GOTO or GOSUB on a conditional (for example, IF A = 1 THEN GOTO 200). The easier way to make a conditional jump is to use *computed* branches using the ON statement. For example,

```
10 TEXT : HOME
20 INPUT "ENTER A NUMBER FROM 1 TO 5 " ; A
30 IF A < 1 OR A > 5 THEN 20 : REM TRAP
40 ON A GOSUB 100,200,300,400,500 : REM COMPUTED GOSUB
50 PRINT "DO YOU WISH TO CONTINUE? (Y/N)" ; : GET AN$
   : IF AN$ < > "Y" THEN END
60 GOTO 10
100 PRINT "ONE" : PRINT : RETURN
200 PRINT "TWO" : PRINT : RETURN
300 PRINT "THREE" : PRINT : RETURN
400 PRINT "FOUR" : PRINT : RETURN
500 PRINT "FIVE" : PRINT : RETURN
```

The format for a computed GOSUB or GOTO is for a variable to be entered after 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 forth. It's much easier than using a series of IF statements:


```
IF A = 1 THEN GOSUB 100
IF A = 2 THEN GOSUB 200 : etc. "
```

However, computed GOTOs and GOSUBs become difficult to use if you have many options. If your program is computing large numbers, all you have to do is to convert the larger numbers into smaller ones by changing the variables. For example,

```
10 TEXT : HOME
20 INPUT "ENTER ANY NUMBER -> "; A
30 IF A < 100 THEN B = 1
40 IF A >= 100 AND A < 200 THEN B = 2
50 IF A >= 200 THEN B = 3
60 ON B GOSUB 100, 200, 300
65 REM COMPUTED GOSUB ON 'B' VARIABLE
70 PRINT "DO YOU WISH TO CONTINUE? (Y/N)"; : GET AN$ :
   IF AN$ < > "Y" THEN END
80 GOTO 10
100 PRINT "LESS THAN 100" : RETURN
200 PRINT "MORE THAN 100 BUT LESS THAN 200" : RETURN
300 PRINT "MORE THAN 200" : RETURN
```

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.

Relationals and Computed GOSUBs

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

```
10 TEXT : HOME
20 X = 1 : Y = 2 : Z = 3
30 A = X < Z
40 B = Y > Z
50 C = Z > X
60 PRINT "A + A =" ; A + A
70 PRINT : PRINT "A + B =" ; A + B
80 PRINT : PRINT "A + B + C =" ; A + B + C
90 END
```

Before you run the program, see if you can determine what will be printed by lines 60, 70, and 80. Then run the program and see what happens. Let's go over it step by step:

1. Since X is less than Z, A will be true with a value of 1. Therefore A + A (or 1 + 1) will equal 2.
2. Since Y is not less than Z (remember, Y = 2 and Z = 3), B will be false with a value of 0. Therefore, A + B (or 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 (or 1 + 0 + 1) will equal 2.

Congratulations if you got it right. If not, go over it again. Remember, very simple things are happening, so don't look for a complicated explanation.

Now that you see how you can get numbers by manipulating relationals, try using them in computed GOSUBs. This program shows how:

```
10 TEXT : HOME
20 INPUT "HOW BIG WAS THE HOME CROWD?"; HC
30 R = 1 + (HC => 500) + (HC >= 1000)
40 ON R GOSUB 100,200,300
50 PRINT : PRINT "DO YOU WISH TO CONTINUE? (Y/N) "; :
   GET AN$
60 IF AN$ < > "Y" THEN END
70 GOTO 10
100 HOME : PRINT "THE HOME CROWD WAS NOT VERY BIG
   — LESS THAN 500" : RETURN
200 HOME : PRINT "THE HOME CROWD WAS A PRETTY GOOD
   SIZE — BETWEEN 500 AND 1000." : RETURN
300 HOME : PRINT "THE HOME CROWD WAS VERY BIG —
1000 OR OVER " : RETURN
```

This program hinges on the formula, or algorithm, in line 30. Here's how it works:

1. There are three conditions:

   HC (home crowd) is less than 500
   HC is between 500 and 1000
   HC is greater than 1000
2. If the first condition exists, both HC >= 500 and HC >= 1000 are false. Thus, 1 + 0 + 0 = 1. Therefore, R = 1.
3. If HC is >= 500, but less than 1000, then HC >= 500 is true, but HC >= 1000 is false. Thus, 1 + 1 + 0 = 2 . Therefore, R = 2.
4. Finally, if HC is both >= 500 and >=1000, the formula results in 1 + 1 + 1 = 3. Therefore, R = 3.

How to program and not worry. 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 judgments. If a program does what you want, no matter how slowly or how long it takes you to write it, it is right.

In the example above an algorithm was used with relationals to do something that could have been 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'll learn all kinds of shortcuts and formulas, but if you get stuck along the way, just keep going. 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 leaving computed GOTOs and GOSUBs with relationals, let's see how relationals handle strings. Try the following:

```
A$ = "A" : B$ = "B" : PRINT B$ > A$
```

Surprised? As well as comparing numeric variables, relationals can compare alphabetic string variables with *A* being the lowest and Z the highest. So if you ask whether B$ is greater than A$, you get a 1 (true) since B$ is a *B* and A$ is an *A*. In sorting strings (as in alphabetizing names) such an operation is crucial. Later, you'll find a routine for sorting strings, but for now, let's make a simple string sorter for sorting two strings.

```
10 TEXT : HOME
20 INPUT "WORD #1 -> " ; A$
30 INPUT "WORD #2 -> " ; B$
40 PRINT : PRINT : PRINT
50 IF A$ < B$ THEN PRINT A$ : PRINT B$
60 IF A$ > B$ THEN PRINT B$ : PRINT A$
```

(Just what you need—a program that will put two words in alphabetical order.)

Arrays

The best way to think about arrays is to regard them as a kind of variable. As you've seen, you can name variables A, D$,

KK%, X1, and so forth. An array uses a single name with a number to differentiate variables. Consider the following two lists, one using regular string variables and the other using a string array:

String Variable	String Array
P$ = "PIG"	AM$(1) = "PIG"
C$ = "CHICKEN"	AM$(2) = "CHICKEN"
D$ = "DOG"	AM$(3) = "DOG"
H$ = "HORSE"	AM$(4) = "HORSE"

If you PRINT H$, you'll get HORSE; if you PRINT AM$(4), you'll also get HORSE. Likewise, you can use arrays for numeric variables:

A(1) = 1
A(2) = 2
A(3) = 3
A(4) = 4
and so on

You may wonder why not just use regular numeric or string variables instead of arrays. Arrays help you organize your information, they make the computer do the work of generating unique variable names, and they are more efficient in certain types of applications. To illustrate a useful application, the following program INPUTs a list of ten names using a string array.

```
10 TEXT : HOME
20 FOR X = 1 TO 10
30 PRINT "NAME #"; X ;
40 INPUT NA$(X)
50 NEXT X
100 REM ******
110 REM OUTPUT
120 REM ******
130 FOR X = 1 TO 10 : PRINT NA$(X)
140 NEXT X
```

Now, try writing a program that does the same thing using nonarray variables. The program will be much longer. Use the variables N0$–N9$ for the names just to see what it will take.

If you rewrite the program, you see how much time arrays can save. But before continuing, let's look more closely at how the program works with the FOR-NEXT loop and array variable:

1. The FOR-NEXT loop generates the numbers sequentially, so the array will look like this:

```
FOR X = 1 TO 10
NA$(1)  First time through loop
NA$(2)  Second time through loop
NA$(3)  Third time through loop
NA$(4)  .
NA$(5)  .
NA$(6)  .
NA$(7)  .
NA$(8)  .
NA$(9)  .
NA$(10) Tenth time through loop
NEXT X
```

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

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

```
A(10) = 432 : PRINT A(10)
XYZ(9) = 2.432 : PRINT XYZ(9)
R2D2$(1) = "BEEP!" + CHR$(7) : PRINT R2D2$(1)
J%(5) = 321 : PRINT J%(5)
```

The DIMension of an Array

We've not gone over the number 10 in any of the array examples. The reason is that, once an array is larger than 10, it's necessary to use the DIM (DIMension) statement to reserve space for the array. (You actually get 11 "free" array slots—0 through 10.) This is an example of the format for DIMensioning an array:

```
10 TEXT : HOME
20 DIM AB(150) : REM DIMENSION OF ARRAY VARIABLE 'AB'
30 FOR I = 1 TO 150
40   AB(I) = I
50 NEXT I
60 FOR I = 1 TO 150
70   PRINT AB(I),
80 NEXT I
```

Run the program as it is written. It should work fine. Now delete line 20 by simply entering 20. (Remember how you

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. Whenever your arrays will have more than 11 values from 0 through 10, be sure to DIM them.

Better safe than sorry. 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, not because they are necessary, but because the programmer thinks it's a good idea to DIM all arrays as part of programming style and clarity. This practice will save some memory space, and if you program in other versions of BASIC, they may require it.

Multidimensional Arrays

It is possible to have arrays with two or more dimensions. Let's begin with two-dimensional arrays and learn 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 an array ranges from 1 to 3 on two dimensions, the entire set will 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, you can think of the first number as a row and the second as a column. This diagram will make it clearer:

	Col 1	**Col 2**	**Col 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's important to remember that each element in the array is simply a type of variable. To make this easier to keep in mind, do the following:

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

Remember, arrays must be envisioned as an orderly set of variables. Now, let's use a two-dimensional array in a program. The purpose of the program is to line up people in a nine-member marching band:

```
10 TEXT : HOME
20 DIM BA$(3,3) : REM MAKE 3 'ROWS' AND 3 'COLUMNS'
30 FOR I = 1 TO 3 : REM ROWS
```



```
40   FOR J = 1 TO 3 : REM COLUMNS
50     READ BA$(I,J)
60   NEXT J
70 NEXT I
80 DATA MARY, TOM, SUE, PETE, JACK, NANCY, BETTY,
BILL, RALPH
100 REM OUTPUT BLOCK
110 FOR I = 1 TO 3 : REM ROWS
120   FOR J = 1 TO 3 : REM COLUMNS
130     PRINT BA$(I,J) , : REM COMMA WILL FORMAT
        OUTPUT 3 ACROSS
140   NEXT J
150 NEXT I
```

When you run this program, all of the band members will be lined up. However, you could do the same thing with a single-dimensional array since all that lines them up is the use of the comma to format the PRINT statement in line 130. So, what's the big deal about a two-dimensional array? To see, add some lines to the program:

```
160 PRINT : PRINT "HIT ANY KEY TO CONTINUE "; : GET
    AN$
170 HOME : PRINT "WHAT ROW & COLUMN WOULD YOU
    LIKE TO SEE? " : INPUT "ROW #-> ";R :INPUT "COL #->
    ";C
180 PRINT : PRINT BA$(R,C); " IS IN ROW "; R; " COLUMN ";
    C
190 PRINT : INVERSE : PRINT "MORE? (Y/N) "; : NORMAL :
    GET M$
200 IF M$ = "Y" THEN 170
```

Now you can locate the value, or contents, of a specific array on two dimensions. In the 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 matrices is an important addition to your programming commands.

It's also possible to have several more dimensions in an array variable. As you add more and more dimensions, you must be careful not to confuse the different aspects of a single array. Sometimes, when a multidimensional array becomes difficult to manage (or use), you'll do better to break it down into several one- or two-dimensional arrays. But just for fun, see what you might want to do with a three-dimensional array with the following program.

```
10 TEXT : HOME
20 INVERSE : PRINT "FILE CABINET LOCATOR" : NORMAL
30 PRINT : PRINT "HOW MANY CABINETS, DRAWERS,
   FOLDERS?"
35 INPUT "(ENTER EACH SEPARATED BY A COMMA)";X,Y,Z
40 DIM FC$(X,Y,Z)
42 FOR I = 1 TO X: FOR J = 1 TO Y: FOR K = 1 to 2
44 FC$(I,J,K) = "EMPTY"
46 NEXT K,J,I
50 INPUT "HOW MAY ITEMS WOULD YOU LIKE TO FILE?
   ";N%
60 PRINT : FOR I = 1 TO N%
70 INPUT "CABINET #-> ";C
80 INPUT "DRAWER #-> ";D
90 INPUT "FOLDER #-> ";F
100 INVERSE : INPUT "NAME OF ITEM TO FILE : ";NI$:
    NORMAL
110 FC$(C,D,F) = NI$
120 NEXT I
200 REM ROUTINE FOR CHECKING CONTENTS OF CABINET
210 HOME : INPUT "WHICH CABINET # WOULD YOU LIKE TO
    CHECK? ";W
220 FOR I = 1 TO X
230 FOR J = 1 TO Y
250   PRINT "CABINET #";W;" DRAWER #";I;" FOLDER #";J;"
      CONTAINS ";FC$(W,I,J)
260   NEXT J
270 NEXT I
280 PRINT : PRINT : PRINT "CHECK MORE CABINETS?
    (Y/N)": GET AN$: PRINT AN$
290 IF AN$ = "Y" OR AN$ = "y" THEN GOTO 210
300 END
```

That's a fairly long program; go over it carefully to make sure you understand what it is doing. Again, let me remind you that the three-dimensional array is a variable with a lot of numbers in parentheses. Also, note on line 35 how several values are INPUT with a single INPUT statement. The format is

INPUT *A, B, C*

As long as the program user is told to enter the appropriate number of responses and separate each with a comma, everything will work fine. You might want to save this program on a disk as an example of a multidimensional array.

Summary

This chapter has covered a lot of ground, and if you understand everything, excellent! If not, don't worry; it will all become clear with practice. 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 stored your programs, the worst that can happen is that you will have to reboot.

You've learned that your IIGS can indeed compute. By using IF-THEN and relationals, you can give the computer the power of decision making. The use of subroutines makes it possible to branch at decision points to anywhere you want within a program. Computed GOTOs and GOSUBs allow the execution to move appropriately with a minimal amount of programming.

Finally, you've been introduced to array variables. Arrays allow you to enter values into sequentially arranged variables (or elements). Using FOR-NEXT loops makes it possible to quickly program multiple variables up to the limits of the DIMensions. Arrays not only assist you in keeping variables orderly, but they save a good deal of work as well. In Chapter 5, you'll begin to work with commands that help arrange everything. As your programs become more sophisticated, you will need to keep better track of what you're doing. By organizing programs into small, manageable chunks, you can create clear, useful programs.

5

Organizing the Parts

Organizing the Parts

In programming, as with most other tasks, good organization allows you to handle larger and more complex problems. As you learn more commands, you can do more things, but the more you do, the more likely you are to get tangled up and lost.

An area that is likely to be among 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, this book will deal extensively with text and string formatting. Not only will you be able to put things where you want them, but you'll learn to do it with style.

The second major area prone to disorganization is input/output (I/O). Some of the problem comes from formatting, but even more elementary is the problem of organizing the input and output so that data is properly analyzed. Data must be connected to the proper variables and must be subject to the correct computations. Thus, in addition to examining string formatting, you will also get a careful look at organizing data manipulation.

Formatting Text

Chapter 1 pointed out that in many ways the IIGS keyboard works like a typewriter. One feature of a typewriter is its ability to set tabs so that text can automatically be placed a given number of spaces from the left margin. With your IIGS, not only can you TAB horizontally, but you can also HTAB, VTAB, and SPC. Let's look at what each of these means:

Command	Meaning
TAB(*N*)	Used within PRINT statement to place next character *N* spaces from left margin
HTAB	Sets horizontal placement of next output character
VTAB	Sets vertical placement of next output character
SPC(*N*)	Used within PRINT statement to create a specified number of spaces (SPC starts printing nonspace characters one space after *N*)

To see how these commands format text output, try them out:

```
10 TEXT : HOME
20 PRINT TAB (20);"TAB TO HERE"
30 HTAB 20 : PRINT "HTAB TO HERE"
35 REM NOTE DIFFERENT FORMAT BETWEEN TAB & HTAB
40 VTAB 18 : PRINT "VTAB DOWN HERE"
50 PRINT SPC(20);"SPC TO HERE"
```

Now, have some fun with the commands. This little program will have a strange effect on your cursor.

```
10 TEXT : HOME : VTAB 20
20 FOR I = 38 TO 1 STEP -1
30 FOR J = 1 TO 100 : NEXT : REM DELAY LOOP
40 HTAB I : PRINT " ]"; SPC(1);
50 NEXT I
60 HTAB 2 : PRINT "HIT ANY KEY TO CONTINUE OR 'Q' TO
   QUIT-> ";: GET A$
70 IF A$ < > "Q" THEN 10
```

Here's another one to play with:

```
10 TEXT : HOME : VTAB 4
20 INPUT "ENTER MESSAGE-> "; MS$
30 PRINT : INPUT "HORIZONTAL PLACEMENT (1-40) -> "; H
40 PRINT : INPUT "VERTICAL PLACEMENT (1-24) -> "; V
50 HOME
60 HTAB H : VTAB V : PRINT MS$
70 PRINT : PRINT "HIT ANY KEY TO CONTINUE OR 'Q' TO
   QUIT "; : GET A$
80 IF A$ < > "Q" THEN 10
90 END
```

As you can see, variables can be used with formatting statements. Thus, HTAB H is read in the same way as HTAB 10 or HTAB 15 or any other number between 1 and 40. In the above program, what do you think will happen if you enter "THIS IS A LONG STRING", and specify a horizontal placement of 39, and a vertical placement of 24? Since the maximum HTAB is 40 and the maximum VTAB 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 HTAB and VTAB with this program to get a clearer understanding of their parameters.

Paddle Formatting

If you do not have game paddles, skip this section.

In some applications, you may want to use your game paddles to output numbers or text. By turning the paddle wheel, you can change the value of variables. Later, in the discussion of graphics, you will see how to use the paddles to develop interesting graphics effects and even games. But for now, we will simply look at some commands that will show you how your paddles can be used with a program. Try the following program:

```
10 TEXT : HOME
20 X = PDL(0)
30 HTAB 10 : VTAB 10
40 PRINT X; SPC(2)
50 FOR I = 1 TO 15 : NEXT I : REM PADDLE 'REFRESH' LOOP
60 GOTO 20 : REM ENDLESS LOOP
```

Note the FOR-NEXT loop in line 50. For some reason, the IIGS paddle readings need to be "refreshed" with a little loop. Anyway, run the program and turn the dial on your paddle to see the effect.

Unraveling Strings

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

When I first learned about all the 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 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 string formatting. Now, however, I find it difficult even to conceive of programming with-

out these powerful commands. So, trust me. String formatting commands are terrific little devices to have, and if you don't see their applicability right away, you will as you begin writing more programs.

String Formatting

We'll divide our discussion of string formatting into four parts:

- Calculating the length of a string
- Locating parts of strings
- Changing strings to numeric variables and back again
- Tying strings together (concatenation)

Calculating the LENgth of Strings

Sometimes it's necessary to calculate the length of a string for formatting output. Happily, your IIGS is very good at telling you the length of a particular string. By issuing the command PRINT LEN (A$), you will be given the number of characters (including spaces) that your string contains. Try the following program to see how this works:

```
10 TEXT : HOME
20 INPUT "NAME OF STRING -> "; A$
30 PRINT A$; " HAS "; LEN(A$); " CHARACTERS"
40 PRINT : INVERSE : PRINT " MORE? (Y/N) ";: NORMAL :GET
   AN$
50 IF AN$ = "Y" THEN 20
```

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

```
10 TEXT : HOME
20 INPUT "40- OR 80-COLUMN SCREEN? "; CS
30 PRINT "ENTER A STRING OF FEWER THAN "; CS ;"
   CHARACTERS" : INPUT"-> "; S$
40 HTAB (CS/2) - LEN(S$)/2 : PRINT S$
50 VTAB 22 : PRINT "HIT ANY KEY TO CONTINUE OR 'Q' TO
   QUIT ";: GET AN$
60 IF AN$ < > "Q" THEN HOME : GOTO 30
70 END
```

Now that you can see how to compute the LENgth of a string and then use that LENgth to compute the tabbing, let's look at how you 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 30 characters. You want to make sure that none of your entries is over 30 characters in length, including spaces. To do this, we will write a program that checks the LENgth of a string before it is accepted:

```
10 TEXT : HOME
20 PRINT "ENTER A NAME FEWER THAN 30 CHARACTERS
   INCLUDING
   SPACES" : INPUT "DO NOT USE COMMAS -> ";
   NA$
30 IF LEN (NA$) > 30 THEN GOTO 100 : REM TRAP
40 PRINT : INVERSE : PRINT NA$ : NORMAL
50 PRINT : PRINT "ANOTHER NAME? (Y/N) "; : GET AN$
60 IF AN$ < > "Y" THEN END
70 GOTO 10
100 HOME : FLASH
110 PRINT "PLEASE USE 30 CHARACTERS OR FEWER " :
    NORMAL
120 PRINT : GOTO 20
```

The first thing you should do is break the rule. Enter a string of more than 30 characters to see what happens. If the program has been entered properly, you'll find that it's impossible to enter a string with more than 30 characters.

From these examples, you can see that LEN can be useful in several ways. The key to understanding its usefulness is to experiment with it and see how other programmers use the same command. You'll find that there are many other ways in which you can employ such commands to reduce programming time, clarify output, and compute information.

Finding the MID$, LEFT$, and RIGHT$ of a String

Suppose you want to use a single string variable to describe three different conditions, such as POOR, FAIR, or GOOD, but you want to use only part of that string to describe an outcome. By using MID$, LEFT$, and RIGHT$, you can print only the part of the string that you want. For example, the next program lets you use a single string to describe three different conditions:

```
10 TEXT : HOME
20 X$ ="POOR FAIR GOOD"
30 PRINT "HOW DO YOU FEEL TODAY? (<P>OOR, <F>AIR, OR
   <G>OOD) "; : GET F$
40 IF F$ = "P" THEN PRINT LEFT$(X$,4)
50 IF F$ = "F" THEN PRINT MID$(X$,6,4)
60 IF F$ = "G" THEN PRINT RIGHT$(X$,4)
70 VTAB 22 : PRINT "ANOTHER GO? (Y/N) "; : GET AN$
80 IF AN$ = "Y" THEN 10
```

Let's face it; it would be easier simply to branch to a PRINT GOOD, FAIR, or POOR, and no less efficient. But, no matter; let's see what the new commands do.

Statement	Meaning
MID$(A$,N,L)	Finds the portion of A$, beginning at *Nth* character, *L* characters long
LEFT$(A$,L)	Finds the portion of A$, *L* characters long, starting at the left side of the string
RIGHT$(A$,L)	Finds the portion of A$, *L* characters long, starting at the right side of the string

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

```
W$="WHAT A MESS" : PRINT LEFT$(W$,4)
G$="BURLESQUE" : PRINT MID$(G$,4,3)
X$="A PLACE IN SPACE" : PRINT RIGHT$(X$,5) : PRINT
   RIGHT$(X$,3)
```

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

```
10 TEXT : HOME
20 BIG$ = "LONG LONG AGO AND FAR FAR AWAY"
30 LITTLE$ = MID$(BIG$,11,3)
40 AWY$ = RIGHT$(BIG$,4)
50 LG$ = LEFT$(BIG$,4)
60 VTAB 10 : PRINT AWY$;" ";LG$;" ";LITTLE$
70 REM BEFORE YOU RUN IT, SEE IF YOU CAN GUESS THE
   MESSAGE.
```

For an interesting effect, try this little program:

```
10 TEXT : HOME : VTAB 10
20 INPUT "YOUR NAME-> "; NA$
30 INVERSE : FOR I = LEN(NA$) TO 1 STEP -1 : PRINT
   MID$(NA$,I,1); : NEXT I
40 NORMAL : FOR I = 1 TO 1000 : NEXT I : REM DELAY LOOP
```



```
50 HTAB 1 : VTAB 11 : FOR I = 1 TO LEN(NA$) : PRINT
   MID$(NA$, I,1); : FOR K = 1 TO 50 : NEXT K : NEXT I
55 REM 'K LOOP' SLOWS IT DOWN FOR SLOW-MOTION EFFECT
60 HTAB 7 : VTAB 20 : PRINT "WANNA DO IT AGAIN? (Y/N)
   "; : GET AN$:
IF AN$ = "Y" THEN 10
```

You've probably been wondering ever since you got your IIGS how to make it print your name backward. Now you know. (If your name is Bob, you probably didn't notice it was printed backward—try Robert.) Actually, the above exercise does a couple of things besides goofing off. First, it demonstrates how loops and partial strings (or substrings) can be used together for formatting output. Second, it shows how output can be slowed down for either an interesting effect or simply to give the user time to see what's happening.

Since we're on the topic of speed, this would be a good time to learn the SPEED command. SPEED can be from 0 to 255; the default is 255. We have been operating at a speed of 255 from the outset. Let's try some SPEED tests:

```
10 TEXT : HOME : SPEED = 255 : REM RESETS SPEED
20 K$ ="ONCE UPON A TIME IN THE KINGDOM OF SNEW..."
30 INPUT "WHAT SPEED WOULD YOU LIKE? (0-255) "; S
40 SPEED = S
50 FOR X = 1 TO LEN(K$) : VTAB 10 : HTAB X : PRINT
   MID$(K$,X,1) : NEXT X
60 VTAB 20 : FLASH : PRINT "AGAIN? (Y/N) "; : NORMAL :
   GET AN$ : IF AN$ ="Y" THEN 10
70 SPEED = 255 : END
80 REM WHEN YOU CHANGE THE SPEED, IT IS ALWAYS A
   GOOD IDEA
90 REM TO RESET IT TO 255 AT THE END OF THE PROGRAM
```

Here's a program that will do all kinds of things with strings:

```
10 TEXT : HOME : GOTO 20
20 PRINT CHR$ (7): VTAB 10: INPUT "STRING > ";S$: IF S$ =
   " " THEN GOTO 90
30 L = LEN (S$): IF L > 10 THEN PRINT : PRINT "TEN OR
   FEWER CHARACTERS PLEASE!": GOTO 20
40 HOME : PRINT S$;: INVERSE : HTAB 20 - LEN (S$) / 2:
   PRINT S$;: FLASH : HTAB 41 - LEN (S$): PRINT S$:
   NORMAL : PRINT
50 FOR I = 1 TO LEN (S$): VTAB I + 2: PRINT MID$ (S$,I,1):
   HTAB 19: VTAB I + 2: INVERSE : PRINT MID$ (S$,I,1):
   HTAB 40: VTAB I + 2: FLASH : PRINT MID$ (S$,I,1):
   NORMAL : NEXT I
```

```
60 FOR I = 1 TO LEN (S$): HTAB I: PRINT MID$ (S$,I,1):
   NEXT I
70 FLASH : FOR I = 1 TO LEN (S$): VTAB LEN (S$) + 3 + I:
   HTAB 41 - I: PRINT MID$ (S$,I,1): NEXT : INVERSE
80 FOR I = 1 TO LEN (S$): HTAB 19: VTAB LEN (S$) + 3 + I:
   PRINT MID$ (S$,1 + LEN (S$) - I,1): NEXT : NORMAL :
   GET A$: PRINT A$: GOTO 10
90 HOME : END
```

Strings to Numbers and Back Again

Now we're going to learn about changing strings to numbers and numbers to strings. When I first found out about these commands, I thought they were pretty useless. After all, I thought, 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 the value of these commands, you'll wonder how you did without them. To get started, run this program:

```
10 TEXT : HOME
20 FOR I = 1 TO 5 : READ NA$(I) : NEXT I
30 FOR I = 1 TO 5
40 X(I) = VAL(RIGHT$(NA$(I),1))
50 NEXT I
60 FOR I = 1 TO 5 : PRINT "OVERTIME PAY= $"; X(I) * (1.5 *
   7) : NEXT I
70 DATA SMITH 7, JONES 8, MCKNAP 6, JOHNSON 2, KELLY
   3
```

By using DATA elements that were originally in a string format, you can change a portion of that string array to a numeric array. By making such a conversion, you are able to use mathematical operations in line 60 to figure out the overtime pay for someone receiving time and a half at seven dollars an hour. Well, that's pretty interesting, but there's no list of who got what and the total overtime paid. Why don't you try adding those features yourself? Change the program so that everyone's name appears with the amount of overtime received and a total for overtime paid is given. *Hint:* You are looking for the substring LEFT$ (NA$(I), LEN (NA$(I)−2)) since you want to drop the number and space after each name.

With a new statement it's always helpful to do a few exercises immediately to get the right feel:

```
A$ = "123" : PRINT VAL(A$) + 11
Q$ = "99.5" : PRINT VAL(Q$) * 7
SALE$ = "44.95" : PRINT "ON SALE AT HALF PRICE ->$ ";
```



```
   VAL(SALE$) / 2
DO$ = "$103.88" : DN$ = "$18.34" : PRINT VAL
   (RIGHT$(DO$,6)) + VAL (RIGHT$(DN$,5))
```

If you want to save the examples on disk, just add line numbers and save them as programs.

From Numbers to Strings

Now let's go the other way. You saw why you might want to change strings to numbers, but you might also want to change numbers to strings. To make the conversion, use the STR$ function. For example, look at the following program:

```
10 TEXT : HOME
20 INPUT "ENTER A NUMBER WITH 5 NUMBERS AFTER THE
   DECIMAL POINT " ; A
30 A$ = STR$(A)
40 PRINT : PRINT LEFT$ (A$,4)
```

The number is truncated to four characters. Now, try some examples in the immediate mode to set the idea firmly into your mind. A little later you'll see how to do something very practical with these statements.

```
A = 5.00 : A$ = STR$(A) : PRINT A$
V = 2345 : V$ = STR$(V) : PRINT V$
BUCKS = 22.36 : BUCKS$ = STR$(BUCKS) : PRINT
   LEFT$(BUCKS$,2)
```

Remember these statements. When you're dealing with decimal points, you will often find them convenient.

Concatenation: Tying Strings Together

You have seen how to 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 plus sign with strings. For example,

```
10 TEXT : HOME
20 INPUT "YOUR FIRST NAME -> "; NF$
30 INPUT "YOUR LAST NAME -> "; NL$
40 NA$ = NF$ + NL$
50 PRINT NA$
```

A little messy, perhaps? However, you can see how NF$ and NL$ are 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 you concatenate string variables, but you also concatenated strings themselves. For example, it is perfectly all right to do this:

```
PRINT "ONE" + "ONE"
```

There's not much you can do with ONEONE, but you see the principle behind concatenating strings.

One of the problems with the way the Apple IIGS formats numbers is that it drops zeros from the end. For example, try the following:

```
PRINT 19.80
PRINT 5.00
```

When you're dealing with dollars and cents, this can be a real problem, and it doesn't look very good. So, by using concatenation and the VAL and STR$ functions, let's see if we can find a solution:

```
10 TEXT : HOME
20 INVERSE : PRINT "BE SURE TO INCLUDE ALL CENTS!":
   NORMAL : PRINT : PRINT
30 INPUT "HOW MUCH SPENT?-> $"; S
40 T = T + S
50 T$ = STR$(T)
60 T$ = "000" + T$
65 REM LINE 60 IS TO INSURE THAT LEN(T$) IS LONG
   NOUGH
70 IF MID$ (T$, (LEN (T$) - 1),1) = "." THEN T$ = T$ + "0"
   : GOTO 100
80 IF MID$ (T$, (LEN (T$) -2),1) < > "." THEN T$ = T$ +
   ".00"
90 PRINT : PRINT
100 PRINT "YOU NOW HAVE SPENT $"; RIGHT$(T$, LEN(T$)
    -3)
```

This program looks complicated until it's broken down.

This program may seem somewhat complicated just to get the zeros back, but the entire process is done in five lines (50–90). Save the program, and when you need those zeros in your output, just include those lines. (Be careful, though; this procedure will not work with subtraction when you get below $1.00).

Lines	Result
Lines 30–40	Numeric variables are entered in line 30, and their sum is computed in line 40.
Line 50	The sum represented by T is then converted into a string variable T$.
Line 60	T$ is padded with three zeros to give it a minimum length needed in lines 70 and 80.
Line 70	The second from the last character in T$ is examined. If that character is a decimal point, the value in the string must be a figure that dropped off the last cent digit (for example, 5.4, 19.5, and so forth). In this case an additional zero is tacked on, and we jump to line 100.
Line 80	The third from the last character is computed. If it is not a decimal point, the value in the string has no cents digits—thus, it is an even dollar number. In this case, we tack on the decimal point and two zeros (.00).
Lines 90–100	Line 90 prints two blank lines. Finally, the results are printed out in line 100, but first we use RIGHT$ to drop the extra padding characters we added in line 60.

Setting Up Data Entry

Now that you have a firm grip on several statements, it's time to begin thinking seriously about program organization. The first step is to arrange your data entry in a manner easily understood by yourself and others. This involves blocking elements of a program and deciding what variables and arrays you will use. Also, when you enter data, make sure that you are entering the correct type of data. You'll need to set traps so that any input that is over a certain length or amount can be checked against your parameters.

One of the easiest ways to set traps is with the ONERR statement. This statement will set certain conditions when an error occurs. Try the following program (that's right—there is an error if you follow instructions):

```
10 TEXT : HOME
20 ONERR GOTO 100
30 INPUT "ENTER A LETTER FROM A-Z ->"; LE
40 PRINT : PRINT : PRINT "SEE WHAT'D I TELL YOU" : END
100 HOME : PRINT : PRINT : FLASH :
110 PRINT " THERE IS AN ERROR! " : NORMAL
120 PRINT : PRINT "TRY AGAIN USING A NUMBER SINCE
    'LE' IS A "
```

```
130 PRINT "NUMERIC VARIABLE"
140 VTAB 20 : INVERSE : PRINT "PRESS ANY KEY TO
    CONTINUE"
150 CLEAR : GET A$ : GOTO 10
```

Trapping yourself. The error messages built into your Apple IIGS are very useful for debugging programs. The messages tell you what kind of error occurred and which line they occurred on. ONERR statements used within programs can cause difficulties since they will jump to your error routine no matter what error they encounter—even if the error is one your program isn't expecting. To avoid this and still use the handy ONERR statement, put a temporary GOTO statement around the ONERR statement while you're developing a program and remove it when your program is complete. For example,

```
25 GOTO 40
30 ONERR GOTO 100
40 INPUT "ENTER NAME "; NA$
Etc.
```

Now, all you have to do is to delete line 25 when you're finished, and your ONERR statement can work the way it's supposed to.

The most common—and useful—type of ONERR trapping occurs with ?OUT OF DATA errors. Sometimes, when you're setting up data entry, you'll have an unknown amount of data to be READ into a program. For example, say you want a menu program that reads in files on your disk from DATA statements. Since you will be adding filenames all the time, you don't want to bother with having to count them each time you add a new DATA element. To read in all the DATA with an unknown number of elements, you can use a FOR-NEXT loop that reads 100 array elements before exiting the loop. Since you are unlikely to have that many files on a single disk side, you will run out of data before exiting the loop. Since it is generally *not* a good idea to jump out of a loop, you will have to take another precaution that will satisfy the loop count as well. Finally, if the expected error is not an ?OUT OF DATA error, you will want your program to do something else. Look at this example:

```
10 REM MENU PROGRAM
20 TEXT : HOME
30 ONERR GOTO 200: REM BEGIN ERROR TRAP
40 DIM NA$(100)
50 FOR I = 1 TO 100: READ NA$(I)
```



```
60 N = N + 1: REM COUNTER
70 NEXT I
80 I = 100: REM RESET RESISTERS IN FOR-NEXT LOOP
90 FOR I = 1 TO N
100 PRINT I;".";NA$(I): NEXT
110 PRINT : PRINT "CHOOSE PROGRAM BY NUMBER": INPUT
    " AND PRESS <RETURN>";C
120 D$ = CHR$ (4): PRINT
130 PRINT D$;"RUN";NA$(C)
140 END
150 DATA FILE A, FILE B, FILE C, FILE D, FILE E : REM
    ENTER THE FILES ON YOUR DISK
200 ER = PEEK (222): REM ADDRESS OF ERROR ROUTINES
220 IF ER = 42 THEN 80: REM 42 IS THE CODE FOR ?OUT OF
    DATA
230 PRINT "THERE WAS AN ERROR IN YOUR CHOICE OR IN
    THE PROGRAM,": PRINT " TRY AGAIN-> HIT ANY KEY
    TO RETURN TO PROGRAM OR 'Q' TO QUIT ";: GET A$
240 IF A$ < > "Q" THEN GOTO 10
```

This program uses a PEEK statement that looks into the location where the error statement codes are stored. (PEEK and POKE statements will be discussed in Chapter 6.) If the correct code is located, the program branches to where you want it. Otherwise, it stops and gives you another chance to INPUT a choice or to quit.

Give yourself an out. People like to have the option of exiting a program—even at the beginning. You can usually hit the reset button or control-C to exit a program, but those are inelegant solutions—like exiting through a wall instead of a door. Therefore, in setting up your program, always leave an option for exiting gracefully.

Now back to setting up data entry. Let's look at a way to make strings a certain length (no shorter or longer than the length you want). We've already discussed how to keep strings to a maximum length; now let's see how to keep them to a minimum. This new process is called *padding*.

```
10 TEXT : HOME
20 VTAB 8 : INPUT "YOUR COMPANY->"; CO$
30 IF LEN(CO$) > 10 THEN PRINT "10 OR FEWER
    CHARACTERS PLEASE" : PRINT "HIT ANY KEY TO
    CONTINUE-> ";:GET A$ : HOME : GOTO 20
40 IF LEN(CO$) < 10 THEN CO$ = CO$ + "X" : GOTO 40 : REM
    PADDING
50 VTAB 10 : PRINT "THE COMPUTER HAS DECIDED THAT "
60 PRINT CO$; " SHOULD GIVE YOU A RAISE"
```

If YOUR COMPANY (CO$) contains fewer than ten characters, you will see some X's stuck on the end. They were put there to show you how padding works. Now in line 40 change the X to " " (a space) and see what happens. The second time you run the program, if your company's name contains fewer than ten characters, there will be several blank spaces after the company name. To remove the spaces, enter this line:

```
60 IF MID$(CO$,LEN(CO$),1) = " " THEN CO$ =
   LEFT$(CO$,(LEN(CO$)-1): GOTO 60
```

Setting Up Data Manipulation

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

Numeric	Manipulating numeric data with mathematical operations
String	Manipulating strings with concatenation and substring statements

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: additions, subtractions, and running balance. This is the checkbook program we started earlier:

```
10 TEXT : HOME
20 REM ### BEGIN INPUT & HEADER BLOCK ###
30 CB$ = " =COMPUTER CHECKBOOK= ": HTAB 20 - LEN
   (CB$) / 2: INVERSE : PRINT CB$: NORMAL : REM
   =HEADER=
40 VTAB 4: INPUT "ENTER YOUR CURRENT BALANCE->
   $";BA
50 VTAB 6: PRINT "1. ENTER DEPOSITS": PRINT : PRINT "2.
   DEDUCT CHECKS": PRINT : PRINT "3. EXIT"
60 VTAB 20: INVERSE : PRINT " CHOOSE BY NUMBER ";:
   NORMAL : GET A
70 ON A * (A<4) GOTO 100,200,400
80 GOTO 60: REM TRAP
90 REM END OF INPUT BLOCK
100 REM ### DATA MANIPULATION ROUTINE NO. 1
    ###
110 HOME : VTAB 6: INPUT "ENTER AMOUNT OF DEPOSIT
    $";DP
120 BA = BA + DP: REM RUNNING BALANCE
130 VTAB 8: PRINT "YOU NOW HAVE $ ";BA;" IN YOUR
    ACCOUNT"
140 PRINT : VTAB 10: PRINT "MORE DEPOSITS? (Y/N) ";: GET
    AN$
150 IF AN$ = "Y" THEN 110
160 PRINT : VTAB 10: PRINT "WOULD YOU LIKE TO DEDUCT
    CHECKS? (Y/N) ";: GET AN$
170 IF AN$ = "N" THEN GOTO 400
180 IF AN$ = "Y" THEN GOTO 200
190 HOME : GOTO 160: REM TRAP & END OF DATA
    MANIPULATION ROUTINE NO. 1
200 REM ### DATA MANIPULATION ROUTINE NO. 2 ###
210 HOME : VTAB 6: INPUT "ENTER AMOUNT OF CHECK
    $";CK
220 BA = BA - CK: REM RUNNING BALANCE
230 PRINT : PRINT "YOU NOW HAVE $";BA;" IN YOUR
    ACCOUNT"
240 PRINT : VTAB 10: PRINT "MORE CHECKS? (Y/N) — 'Q' TO
    QUIT ";: GET AN$
250 IF AN$ = "Y" THEN 210
260 IF AN$ = "Q" THEN 400
270 PRINT : PRINT "ANY DEPOSITS? (Y/N) ";: GET AD$
280 IF AD$ = "Y" THEN 100
290 GOTO 240: REM TRAP & END OF DATA MANIPULATION
    BLOCK NO. 2
400 REM ### TERMINATION BLOCK ###
410 HOME : FOR I = 1 TO 400: PRINT "$";: NEXT
420 PRINT "YOU NOW HAVE A BALANCE OF $";BA
```


This program provides a simple illustration of how to block data manipulation. However, there are some problems with it in the output. You will not get the zeros on the end of the balance. This is an output problem that will be discussed in the next section, but before continuing, be sure that you understand how the data manipulation was blocked. Only three variables were used:

BA = BALANCE
CK = CHECK
DP = DEPOSIT

When you subtract a check, you simply subtract CK from BA, and when you enter a deposit, you add DP to BA. In this way you can keep a running balance, and at the very end BA will be the total of all deposits and checks. By keeping it simple and in blocks, you will be able to jump around and still keep everything straight.

Organizing Output

Let's return to the checkbook program and repair it so that the balance will show the zeros where they belong. This is es-

sentially an output problem: All of the computations have been done, and they correctly tell the balance, but it just doesn't look right with the missing zeros. However, you don't want to have to enter the lines for converting the balance into a string variable every time the running balance is printed. Therefore, put the subroutine for the conversion into a block. Looking at the checkbook program, you see that a block is available in the 300s. Luck is with you. You can use that block to format the output.

```
300 REM ### FORMAT OUTPUT ###
310 BA = BA + .001:PLACE = 1:BA$ = STR$ (BA): IF BA <
    .01 THEN BA$ = "0.00": GOTO 340
320 IF MID$ (BA$,PLACE,1) < > "." THEN PLACE = PLACE +
    1: GOTO 320
330 BA$ = LEFT$ (BA$,PLACE + 2)
340 RETURN
350 REM END OF OUTPUT BLOCK
```

Now, just change a few lines in the program so that when there is an output of the balance, it will jump to the subroutine between lines 300 and 350, and then RETURN to output BA$. The following lines in the program should be changed or added:

```
125 GOSUB 300
130 VTAB 8 : PRINT "YOU NOW HAVE $"; BA$; "IN YOUR
    ACCOUNT"
225 GOSUB 300
230 PRINT : PRINT "YOU NOW HAVE $"; BA$; " IN YOUR
    ACCOUNT"
415 GOSUB 300
420 PRINT "YOU NOW HAVE A BALANCE OF $"; BA$
```

If you put everything together properly, you should have a handy program for working with your checkbook. Just to make sure you have everything, here's the complete program with all the subroutines and changes incorporated:

Checkbook Program

```
10 TEXT : HOME : REM COMPLETE CHECKBOOK PROGRAM
20 REM ### BEGIN INPUT & HEADER BLOCK ###
30 CB$ = " =COMPUTER CHECKBOOK= ": HTAB 20 - LEN
   (CB$) / 2: INVERSE : PRINT CB$: NORMAL : REM
   =HEADER=
40 VTAB 4: INPUT "ENTER YOUR CURRENT BALANCE->
   $";BA
50 VTAB 6: PRINT "1. ENTER DEPOSITS": PRINT : PRINT "2.
   DEDUCT CHECKS": PRINT : PRINT "3. EXIT"
```



```
60 VTAB 20: INVERSE : PRINT " CHOOSE BY NUMBER ";:
   NORMAL : GET A
70 ON A * (A< 4) GOTO 100,200,400
80 GOTO 60: REM TRAP
90 REM END OF INPUT BLOCK
100 REM ### DATA MANIPULATION ROUTINE NO. 1 ###
110 HOME : VTAB 6: INPUT "ENTER AMOUNT OF DEPOSIT
    $";DP
120 BA = BA + DP: REM RUNNING BALANCE
125 GOSUB 300
130 VTAB 8: PRINT "YOU NOW HAVE $";BA$;" IN YOUR
    ACCOUNT"
140 PRINT : VTAB 10: PRINT "MORE DEPOSITS? (Y/N) ";: GET
    AN$
150 IF AN$ = "Y" THEN 110
160 PRINT : VTAB 10: PRINT "WOULD YOU LIKE TO DEDUCT
    CHECKS? (Y/N) ";: GET AN$
170 IF AN$ = "N" THEN GOTO 400
180 IF AN$ = "Y" THEN GOTO 200
190 HOME : GOTO 160: REM TRAP & END OF DATA
    MANIPULATION ROUTINE NO. 1
200 REM ### DATA MANIPULATION ROUTINE NO. 2 ###
210 HOME : VTAB 6: INPUT "ENTER AMOUNT OF CHECK
    $";CK
220 BA = BA - CK: REM RUNNING BALANCE
225 GOSUB 300
230 PRINT : PRINT "YOU NOW HAVE $";BA$;" IN YOUR
    ACCOUNT"
240 PRINT : VTAB 10: PRINT "MORE CHECKS? (Y/N) — 'Q' TO
    QUIT ";: GET AN$
250 IF AN$ = "Y" THEN 210
260 IF AN$ = "Q" THEN 400
270 PRINT : PRINT "ANY DEPOSITS? (Y/N) ";: GET AD$
280 IF AD$ = "Y" THEN 100
290 GOTO 240: REM TRAP & END OF DATA MANIPULATION
    BLOCK NO. 2
300 REM ### FORMAT OUTPUT ###
310 BA = BA + .001:PLACE = 1:BA$ = STR$ (BA): IF BA <
    .01 THEN BA$ = "0.00": GOTO 340
320 IF MID$ (BA$,PLACE,1) < > "." THEN PLACE = PLACE +
    1: GOTO 320
330 BA$ = LEFT$ (BA$,PLACE + 2)
340 RETURN
350 REM END OF OUTPUT BLOCK
400 REM ### TERMINATION BLOCK ###
410 HOME : FOR I = 1 TO 400: PRINT "$";: NEXT
415 GOSUB 300
420 PRINT "YOU NOW HAVE A BALANCE OF $";BA$
```

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 off the top of the screen:

```
10 TEXT : HOME
20 FOR X = 1 TO 100 : PRINT X : NEXT
```

Instead of numbers, suppose you have a list of names you have sorted or some other type of output that you want to see before it zips off the top of the screen. There are several ways to control the scroll, depending on the desired output, screen format, and so forth. Consider the following:

```
10 TEXT : HOME
20 FOR I = 1 TO 100
30 IF I = 20 THEN GOSUB 100
40 IF I = 40 THEN GOSUB 100
50 IF I = 60 THEN GOSUB 100
60 IF I = 80 THEN GOSUB 100
70 PRINT I : NEXT I
80 END
100 VTAB 23 : INVERSE : PRINT " HIT ANY KEY TO
    CONTINUE " ; : GET A$ : PRINT A$
110 HOME : RETURN
```

Remember that you, and 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 tabbed to another column after the vertical screen is filled. For example,

```
10 TEXT : HOME
20 FOR X = 1 TO 40
30 IF X > 20 THEN GOSUB 100
40 PRINT X : NEXT X
50 END
100 HTAB 10 : VTAB X - 20
110 RETURN
```

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

Summary

The way a program is formatted makes the difference between a useful and not-so-useful computer application. The extent to which your program is well organized and clear determines your chances 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 Apple IIGS and yourself. After all, if you can't make heads or tails of what your computer has computed, the best calculations in the world are of absolutely no use.

It is equally important to write your programs so that you and others can understand what is happening. By using blocks, you'll find it easier to organize and later understand exactly what each part of your program does. Obviously, you can write programs sequentially so that each statement and subroutine is in ascending order of line numbers, but this means that you must repeat simple and/or complex operations that would better be handled as subroutines. Also, locating bugs and making appropriate corrections will be considerably more difficult. 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 statements for substrings, and why all the fuss is made about tabs. These are handy tools for organizing the various parts in a manner that gives you complete control over your computer's output. What may at first seem like a trivial, even silly statement in Applesoft BASIC, given a useful application, can be appreciated as an excellent tool. As you delve deeper into the Apple IIGS, look at the variety of statements as mechanisms of more efficient and ultimately simpler control, not as a gobbledegook of computerese meant for geniuses. If you've come this far, you should realize that what you know now looked like the work of computer whizzes when you began.

6

Some Advanced Topics

Some Advanced Topics

The topics in this chapter are more codelike than the ones you've learned before and contain statements that can look scary. Many of the functions can be done with statements you already know. Others can better be accomplished with some of the new statements. Like so much else, what may at first appear to be impossible is really quite simple once you get the idea. More important, by playing with the statements, you can quickly learn their use.

The first thing to learn about is the ASCII code. ASCII (pronounced *AS-KEY)* stands for the American Standard Code for Information Interchange. Essentially, ASCII is a set of numbers that have been standardized to represent certain characters. In Applesoft, the CHR$ (character string) function ties into ASCII and can be used to directly output ASCII. The CHR$ function is very useful not only in disk statements, but also for outputting special characters.

The next statements concern direct access of locations in your computer's memory. The first, POKE, puts values into memory, and the second, PEEK, looks into memory addresses and returns the values there. We will examine several different uses of these two statements.

The third statement discussed here, CALL, executes machine-level routines in memory. The CALL statement can be used to execute such routines from within a BASIC program. In addition, you will learn how to use the BSAVE statement to save binary files to the disk, and to BLOAD and BRUN them. Finally, you will see the uses of the WAIT statement for conditional pauses within Applesoft programs.

The ASCII Code and CHR$

In some programs, you might see something that looks like this:

```
PRINT " ": REM CONTROL-D
```

This means that you should enter the control-D between the quotation marks. Unfortunately, you cannot see the control-D when you list the program to printer or screen, so you must use a REM statement to let you know what's there.

Another way to access any characters you want, including control characters, is to use CHR$ functions and the ASCII code. In Appendix B is a complete listing of ASCII, from 0 to 255, in both decimal and hexadecimal. Whenever you want to access a character, all you have to do is to enter the CHR$ and the decimal value of the character. For example, enter the following:

```
PRINT CHR$(65)
```

You'll get an *A*. That's simple enough, but not very interesting. On the other hand, try this program, which you can't do without using CHR$:

```
10 TEXT : HOME
20 QU$ = CHR$(34) : REM USES ASCII VALUE FOR QUOTE
   MARKS
30 VTAB 20 : PRINT "HIT ANY KEY TO CONTINUE OR "; QU$
   ; "Q" ; QU$ ; " TO QUIT ";
40 GET AN$
50 IF AN$ = "Q" THEN END
60 PRINT CHR$(7) : GOTO 10
```

Run the program, and look and listen carefully. Look at the quotation marks around the *Q*. If you try to print a quotation mark, the computer will think it has received a statement to begin printing a string. But by defining QU$ as CHR$(34), you'll be able to slip in the quotation marks and not confuse the output. Also, did you notice the bell? Until now, if you wanted to ring the bell, you had to include an invisible control-G. Since the ASCII value of control-G is 7, by PRINTing CHR$(7) you can ring the bell and see the function that makes it ring.

To see the different characters that are available, run the following program:

```
10 TEXT : HOME
20 FOR I = 32 TO 127
30 PRINT CHR$(I); : NEXT
```

Voilà. There are all your symbols. Now, to watch funny things happen to your screen, run the following program:


```
10 TEXT : HOME
20 FOR X = 0 TO 31
30 PRINT CHR$(X) ; : NEXT
```

You'll get a ?SYNTAX ERROR IN 30 with that routine. You smack into a lot of control characters in that range, and while some, such as CHR$(7), cause no problems, others do. There are important codes in this range that you will use, but be careful when exploring them.

Try the following programs to become accustomed to your increased power over your computer.

```
10 TEXT : HOME
20 LB$ = CHR$ (93) : RB$ = CHR$(91)
30 APPLE$ = "APPLE" + CHR$(32) + LB$ + RB$
40 HTAB 20 – LEN (APPLE$)/2 : PRINT APPLE$
45 REM FOR 80 COLUMNS USE HTAB 40
50 VTAB 23
```

```
10 TEXT : HOME
20 FOR I = 1 TO 4 : PRINT CHR$(7);:NEXT
30 VTAB 10: PRINT CHR$(34);"WAS THAT THE PHONE OR
   DOORBELL?";CHR$(34)
```

```
10 TEXT : HOME
20 FOR X = 65 TO 90
30 PRINT CHR$(X);CHR$(X+32)
40 NEXT X
```

In the last instruction, you should have discovered that by using an offset of 32, you can get the lowercase equivalent of the uppercase character.

Printing CHR$ Values

The next program is a handy device for printing out all of the CHR$ values to screen. The inverse ^ (caret) indicates that the character is a control character. Notice how "negative offsets" are used to print control-character representations. This is because, if you attempt to use the CHR$ function to print actual control characters, they will have an unusual effect on your output. Save this program so that you can quickly look up CHR$ values. In Chapter 11 you'll find a program that will print all of these characters to your printer.

```
10 TEXT : HOME
20 F = 0
30 FOR I = 1 TO 3: PRINT "CHR$/S",: NEXT
40 FOR I = 1 TO 40: PRINT "=";: NEXT : POKE 34,2
50 FOR I = 64 TO 95
```

```
60 IF I < 74 THEN PRINT " ";I - 64;". ";: INVERSE : PRINT
   "^";: NORMAL: PRINT CHR$ (I):F = F + 1: GOTO 80
70 PRINT I - 64;". ";: INVERSE : PRINT "^";: NORMAL :
   PRINT CHR$ (I):F = F + 1:
80 IF F > 19 AND F < 40 THEN GOSUB 600
90 IF F > 39 THEN GOSUB 700
100 NEXT
110 FOR I = 32 TO 127: PRINT I;". " + CHR$ (I):F = F + 1
120 IF F > 19 AND F < 40 THEN GOSUB 600
130 IF F > 39 THEN GOSUB 700
140 IF F = 60 THEN GOSUB 500
150 NEXT I
160 VTAB 21: HTAB 1: PRINT "HIT ANY KEY TO RESTART OR
   " + CHR$ (34) + "Q" + CHR$ (34)" TO QUIT ";: GET AN$
170 IF AN$ < > "Q" THEN GOTO 10
180 VTAB 22: TEXT : END
500 F = 0: INVERSE : HTAB 1: VTAB 24: PRINT "HIT ANY
   KEY TO CONTINUE OR 'Q' TO QUIT";: GET A$: NORMAL
510 IF A$ = "Q" THEN 180
520 HOME
530 RETURN
600 HTAB 17: VTAB F - 17
610 RETURN
700 HTAB 40 - 7: VTAB F - 37
710 RETURN
```

Using CHR$ with ProDOS

An important control character that you should become acquainted with is control-D accessed by CHR$(4). By printing this character from within a program, you can access your disk system. For example, try the following program:

```
10 TEXT : HOME
20 D$ = CHR$(4)
30 PRINT D$; "CAT"
40 END
```

Your program CATaloged your disk. Now let's see how you can do more with CHR$ and your disk system:

```
10 TEXT : HOME
20 D$ = CHR$(4)
30 PRINT D$; "CAT"
40 INVERSE : INPUT " WHICH PROGRAM WOULD YOU LIKE
   TO RUN? "; PG$ : NORMAL
50 PRINT D$; "RUN"; PG$
```

```
10 TEXT : HOME : HD$ = " FILE FIXER ": HTAB 20 -
   LEN(HD$) / 2 : INVERSE : PRINT HD$ : NORMAL
```



```
15 REM USE HTAB 40 FOR 80-COLUMN SCREENS
20 D$ = CHR$(4)
30 VTAB 8: PRINT "1. LOCK" : PRINT : PRINT "2. UNLOCK" :
   PRINT : PRINT "3. DELETE" : PRINT : PRINT "4. EXIT "
40 VTAB 20 : PRINT "CHOOSE BY NUMBER " ; : GET A
50 ON A GOSUB 100, 200, 300, 400
60 GOTO 10
100 HOME
110 PRINT D$ ; "CAT"
120 INVERSE : INPUT "LOCK WHICH FILE?"; L$ : NORMAL
130 PRINT D$; "LOCK"; L$
140 RETURN
200 HOME
210 PRINT D$; "CAT"
220 INVERSE : INPUT "UNLOCK WHICH FILE? ";U$ :
   NORMAL
230 PRINT D$; "UNLOCK"; U$
240 RETURN
300 HOME
310 PRINT D$ ; "CAT"
320 INVERSE : INPUT "DELETE WHICH FILE? "; DE$ :
   NORMAL
330 FLASH : PRINT " ARE YOU SURE? (Y/N) ";: GET AN$ :
   NORMAL
340 IF AN$ < > "Y" THEN RETURN
350 PRINT D$; "DELETE" ; DE$
360 RETURN
400 HOME : END
```

POKEs and PEEKs: Inside Memory

At first you won't find many uses for POKEs and PEEKs, but as you begin exploring the full range of your IIGS, you'll use them more and more. Basically, a POKE statement places a value into a given memory location, and a PEEK function returns the value stored in that location. For example, try the following:

```
POKE 768, 255 : PRINT PEEK (768)
```

You should get 255 since the POKE statement entered that value into location 768. That's relatively simple, but more is happening than number storage.

The key importance of POKE and PEEK involves what occurs in a given memory location when a given value is entered. In some locations nothing other than the storage of the number will happen, as in the example above. However, with some memory locations, very precise events occur. For example,

PRINT PEEK (222) was used to find the type of error that occurred when the ONERR statement was used. So if you POKE memory location 222, it will not be the same as POKEing locations that are "free space" available for the programs you write. In the remainder of this section, we'll examine some of the more useful locations for POKEing and PEEKing in your IIGS. We will not, however, get into the more complex elements of POKEs and PEEKs.

A Tale of Two Number Systems

When working with POKEs and PEEKs, you'll use decimal numbers for accessing memory locations. However, much of what is written about special locations in the computer's memory is written in hexadecimal, generally referred to as *hex*. Since we've used decimal notations all our lives for counting, it seems to be a natural way of doing things. However, decimal is simply a base-10 counting method, and we could use a base of anything we wanted. For reasons we won't get into here, base 16, called hexadecimal, is an easier way to think about using a computer's memory, and that's why so much of the notation is in hex.

Hex is counted in the same way as decimal, except it is done in groups of 16, and it uses alphanumeric characters instead of just numeric ones. You can usually tell whether a number is hex since it is typically preceded by a dollar sign (for example, $45 is not the same as decimal 45), and often alphabetic characters are mixed in with numbers (for example, FC58, AAB, 12C). The following is a list of decimal and hexadecimal numbers.

Dec	Hex	Dec	Hex
0	$0	9	$9
1	$1	10	$A
2	$2	11	$B
3	$3	12	$C
4	$4	13	$D
5	$5	14	$E
6	$6	15	$F
7	$7	16	$10
8	$8		

Instead of beginning with double-digit numbers at 10, hexadecimal begins using double digits at decimal 16 with $10. In the major memory locations of interest in your IIGS, both the decimal and hexadecimal numbers are given.

Now let's look at some places to POKE. We will begin with your text screen. In the program, above, that gives the

different CHR$ values, you used a POKE to set the text screen so that you could keep a portion of it above the new information appearing on the screen. Using POKEs, you can set the left margin, width, top, and bottom of your text window. The following program is an example of setting the text window and scrolling "under" the defined "window."

```
10 TEXT : HOME
20 PRINT "This is the protected part of the screen";
30 POKE 34,4 : REM SET THE TOP OF YOUR WINDOW TO
   VERTICAL POSITION 4
50 FOR X = 1 TO 100 : PRINT X : NEXT
```

The message and first three numbers from the loop in line 50 are in a window "on top" of the bottom window. The characters scroll under the top window since they think the window ends at vertical position 4. To get to the top window, add the following line:

```
60 VTAB 2 : PRINT "Back up here!"
```

The following four POKEs are for window dressing. To reset everything to normal, use the TEXT statement:

POKE 32,N Left edge of window (0–39/79)
POKE 33,N Width of text window (1–40/80)
POKE 34,N Top edge of window (0, 22)
POKE 35,N Bottom of text window (1–24)

If you are in the 40-column mode and POKE a window, your window will disappear as soon as you enter the 80-column mode. The same is true with windows POKEd in the 80-column mode when you switch to 40 columns. If you make this switch several times, you'll get some very strange results.

Try the following programs to get an idea of some different windows and their effects:

```
10 TEXT : HOME
20 POKE 32, 10 : REM SET LEFT EDGE TO 10
30 POKE 33, 12 : REM SET WIDTH TO 12
40 POKE 34, 8 : REM SET TOP EDGE TO 8
50 POKE 35, 15 : REM SET BOTTOM EDGE TO VERTICAL
   POSITION 15
60 VTAB 9 : INVERSE : FOR I = 1 TO 6 : PRINT SPC (12); :
   NEXT
65 REM FILL UP WINDOW WITH INVERSE SPACES
70 FOR I = 1 TO 2000 : NEXT : REM DELAY LOOP TO LOOK AT
   WINDOW
80 HOME : PRINT "THIS LITTLE WINDOW IS YOUR TEXT
   WINDOW — NOT MUCH ROOM, IS IT?"
```

```
90 FOR PAUSE = 1 TO 3000 : NEXT : REM DELAY
100 NORMAL : LIST
```

```
10 TEXT : HOME
20 FOR I = 1 TO 11: INVERSE : PRINT SPC(40) : NEXT I :
   NORMAL
30 POKE 35,11
40 T$="TOP" : B$="BOTTOM"
50 VTAB 5 : HTAB 20 - LEN(T$)/2 : PRINT T$
60 INVERSE : VTAB 17 : HTAB 20 - LEN (B$)/2 : PRINT B$ :
   NORMAL
70 VTAB 23 : END
```

When you first started this book, it was suggested that beginning programs with TEXT : HOME was a good idea. You've seen the advantage of having HOME at the beginning of a program since it clears the screen. Now, you can see the advantage of having TEXT since it is used to reset the text window to its full size. So, after you have changed the text window, as you did in this section, you'll find it convenient to run other programs with different-sized windows and know that TEXT is waiting at the beginning to put everything right.

Another Tip

When you're developing programs that use POKEs to change the text window, put a TEXT at the very end of the program to reset it to full size before LISTing and editing. In fact, if you want to be all set for editing, add the following line:

```
62000 GET A$ : PRINT A$ : TEXT : HOME : POKE 33,33 : LIST
```

This is a way to automatically reset your text window, clear the screen, set the screen width to the width of program lines, and LIST your program. The GET statement simply holds whatever you have on the screen until you are ready to edit it. When the program is completed, remember to delete line 62000. The TEXT statement at the beginning of your program will reset the POKE 33,33.

POKEing the Text Screen

Another use of POKEs is to enter a character to a location on your text screen. Each character has a different value between 0 and 255. Unlike the CHR$ values, values POKEd into memory and displayed on the screen are inverse, normal, uppercase, and lowercase, plus some symbols you have not yet seen. What is an ASCII *A* with the CHR$ function 65 is a hollow-apple icon, while a normal *A* is 193. You can envision

your screen as a set of addresses on a 40 (or 80) × 24 grid, beginning with decimal location 1024. To get an idea of what you'll see, try this program:

```
10 TEXT : HOME
20 POKE 49167,0 : REM Char Set 1
30 FOR X= 1024 TO (1024 + 254)
40 POKE X,V
50 V=V+1
60 NEXT X
70 GET A$ : REM Hold for pause
80 POKE 49166,0 : REM Char Set 2
```

When you run the program, you will see several different characters printed on your screen. When the program stops, hit a key, and you will see some of the characters start flashing. What you're seeing is two different character sets in your IIGS. Just POKE 49166 or 49167 to get the set you want.

For another interesting phenomenon, run this next program from the 40-column mode:

```
10 HOME : POKE 49167,0
20 FOR X=1 TO 40
30 POKE 1023 + X, 65
40 NEXT X
```

That will put a line of apples across the top of your screen. Press esc-8 to get into the 80-column mode. You'll see the same apples lined up, but in the smaller 80-column size, taking up only half the screen. Run the same program again in the 80-column mode. This time, the 40 apples are spaced across the screen so that only every other space is used, but the entire horizontal line is taken up. Go back to the 40-column mode, and you'll get another surprise. Instead of lining up all together, the apples are still spaced in every other horizontal position, and half of them are missing. Keep the different columns in mind when you POKE in characters on your screen.

Now, since you know how to use the HTAB and VTAB statements, as well as FLASH, INVERSE, and NORMAL, this may seem a difficult way to display text to the screen. You're absolutely right; it is much simpler and just as efficient to use the statements you alreay know for displaying text. Rarely will you need to use POKEs to display information to the screen—and you can really lock things up with POKEs to the screen. However, it is interesting to know, and if you like to encode

secret messages within your programs and amaze your friends, POKEing the text screen can be amusing.

The following program will give you a handy little utility for determining the values for all the different characters:

```
10 TEXT : HOME
20 X = PDL(0)
25 REM PADDLE 0. IF YOU DO NOT HAVE PADDLES, CHANGE
   LINE 20
26 REM TO -> VTAB 2 : HTAB 1 : CALL -868 : INPUT
   "ENTER A
27 REM NUMBER FROM 0 TO 255-> "; X : IF X > 255 THEN
   20
30 HTAB 5 : VTAB 10
40 IF X < 10 THEN PRINT "POKE <1024 TO 2047>,"; X;"=  " :
   GOTO 70
45 REM 2 SPACES AFTER "=
50 IF X < 100 THEN PRINT "POKE <1024 TO 2047>,"; X ; "=
   " : GOTO 70
55 REM 1 SPACE AFTER "=
60 PRINT "POKE <1024 TO 2047>," ; X ; "="
70 POKE 1220, X
80 GOTO 20
```

Use Your Labeler

If you do have paddles, it's a good idea to label them paddle 0 and paddle 1 (and paddle 2 and 3 if you have four paddles). If you don't know which is which, simply run the above program. If the numbers change, you have paddle 0.

While you're at it, use labels as reminders for all kinds of POKEs and PEEKs as well. Use one of those label makers that produce little plastic strips with adhesive on them to punch in handy POKEs and PEEKs and stick them on your computer. Don't overdo it, but if you find yourself constantly looking up POKE and PEEK numbers, save yourself some time by making a label.

Tweaking the Sound Chip

Another point of interest involving PEEKs and POKEs is your Apple's speaker. By using a series of PEEKs and POKEs, you can produce entire tunes. Here, however, we will simply look at some fundamental ways to make your speaker produce sounds.

The most elementary way of kicking on your speaker is with PEEK (−16336). Try this:


```
FOR X = 1 TO 10 : SS = PEEK (-16336) : NEXT
```

You should have heard a little buzz. Now enter the following program and listen carefully as the sounds change.

```
10 TEXT : HOME
20 N =-16336
30 FOR I = 1 TO 100 : S = PEEK(N) : NEXT
40 FOR I = 1 TO 100 : S = PEEK(N) + PEEK(N)
50 FOR I = 1 TO 100 : S = PEEK(N) - PEEK(N) + PEEK(N)
```

By experimenting with adding and subtracting combinations of PEEK(*N*), you can arrive at a limited set of sound subroutines. The following program, however, will provide an even fuller range of sounds:

```
10 TEXT : HOME
20 FOR I = 768 TO 795: READ J: POKE I,J: NEXT
30 POKE 1013,76: POKE 1014,0: POKE 1015,3
40 VTAB 5: HTAB 1: PRINT "ENTER TWO NUMBERS LESS
   THAN 256": CALL - 868: INPUT "SEPARATED BY A
   COMMA -> ";I,K
50 VTAB 10: HTAB 1: CALL - 868: PRINT "TONE VALUE: =
   ";K
60 VTAB 12: HTAB 1: CALL - 868: PRINT "DURATION = ";I
70 & I,K: GOTO 40
80 DATA 32,70,231,134,81,164,80,152,170,202,208,253,44,48,
   192,69,81,170,202,208,253,44,48,192,136,208,236,96
```

This program contains material that you don't know about yet. It was included to show you the tones that can be produced on the IIGS. The ampersand (&) in line 70 calls a machine-level subroutine that was POKEd in with the loop on line 20 and data on line 80.

Calling a Monitor Subroutine

You've probably noticed that CALL statements have been used in various programs. For example, in the last program there was a CALL −868. A CALL "runs" a machine-level subroutine in your monitor (not your television, but the system monitor within your computer) or another machine program in memory.

When you examine the contents of memory using the built-in machine language monitor in your Apple, along the far left side, you will see the starting addresses of the various subroutines that can be CALLed from your Applesoft program. The problem is that the listing is in hexadecimal and you have to make the CALLs with decimal values. There are many pro-

grams and charts available for converting hexadecimal to decimal. However, for the beginner, it is probably more confusing than enlightening to go into either the conversion process or to explain the monitor listing. Instead, Table 6-1 is a list of some handy CALLs. When you are more advanced, you can see how these CALLs jump to a machine-level subroutine.

Table 6-1. Mini Call Chart

CALL	Effect
−151	Enters the monitor
−936	Same as HOME
−868	Clears from cursor to end of line
−958	Clears from cursor to end of screen
−1184	Puts IIGS message at top of screen
−1401	Boots system and resets pointers
−912	Scrolls up a line
−756	Waits for keypress
−678	Waits for return key to be pressed

Try some of the above CALLs in your programs to see their effect. If you CALL −151, use control-C or type in 3D0G to get back to Applesoft BASIC. Here are some programs to practice with CALL:

```
10 TEXT : HOME : FOR I = 1 TO 800 : PRINT "X"; : NEXT
15 REM SOMETHING TO FILL THE SCREEN
20 FOR I = 1 TO 24 : HTAB 20 : VTAB I : CALL −868 : FOR J
   = 1 TO 50 : NEXT J : NEXT I
30 FOR I = 1 TO 24 : HTAB 1 : VTAB 25 − I : CALL −868 :
   FOR J = 1 TO 50 : NEXT J : NEXT I
```

```
10 TEXT : CALL −936
20 FOR I = 1024 TO 2039 : POKE I, 102 : NEXT
30 FOR PAUSE = 1 TO 2000 : NEXT PAUSE : REM PAUSE
   LOOP
40 FOR I = 1 TO 24 STEP 2 : VTAB I : CALL −868 : NEXT :
   PRINT : CALL −868
50 FOR PAUSE = 1 TO 2000 : NEXT PAUSE
60 FOR J = 2 TO 23 STEP 2 : HTAB 1 : VTAB J : INVERSE :
   PRINT SPC(40): NORMAL : NEXT J
70 FOR PAUSE = 1 TO 2000 : NEXT PAUSE
80 FOR K = 1 TO 24 : CALL −912 : NEXT
```

BSAVEing a Binary File

We won't be writing any machine- or assembly-level programs, but frequently you will want to load and save a binary

file. Unlike Applesoft files, binary files must be BSAVEd with the starting address and length of the program. For example, you might have a program named SUPER.JET that begins at hex $300 and is hex $1A5 long. (Remember, the dollar sign indicates that the number is in hexadecimal.) To save the program, you would write in

```
BSAVE SUPER JET, A$300, L$1A5
```

The problem is finding the beginning address and length of the file. Fortunately, the process is relatively simple. Here's how:

1. BLOAD a binary file.
2. Get into the monitor with CALL −151. When you arrive, you will get the asterisk (*) prompt.
3. Enter: `AA72.AA73 AA60.AA61`
4. You will get two sets of numbers looking something like this:
   ```
   AA72- 03 08
   AA60- 4E 12
   ```
5. Reverse the pairs of bits:
   ```
   03 08 to -> 08 03
   4E 12 to -> 12 4E
   ```
6. Now you have the starting address, $0803 (or simply $803), and the length, $124E.
7. On another disk:
   ```
   BSAVE BFILENAME, A$803, L$124E
   ```
8. Use control-C or 3D0G (that's 3-D-zero-G) to return to Applesoft.

You might want to make a label with AA72.AA73 and AA60.AA61 on it as a reminder. There will be many times when you'll want to transfer a binary file without having to BRUN FID or some other file copy program, and this will be very handy information.

WAITing Without a Cursor

The final statement we'll examine is the WAIT statement. Essentially, WAIT halts execution of your program until a given key or keys are hit. Unlike the GET statement, it will not produce a cursor on your screen, and it makes your program that much cleverer. For example, try the following two programs to compare the differences in using GET and WAIT:

```
10 TEXT : HOME
20 FOR I = 1 TO 800 : PRINT "#"; : NEXT
30 PP$= "HIT ANY KEY TO CONTINUE" : HTAB 20 -
   LEN(PP$) / 2 : PRINT PP$
40 GET A$ : HOME
```

Now change line 40 to

```
40 WAIT -16384,128 : HOME
```

That's relatively simple, but you can see the difference between using GET and WAIT. Sometimes you'll want to use GET in order to "get" a value for a variable, or the program will look better with a flashing cursor to remind the user that input is required. It's also a good idea to clear the keyboard buffer when you use WAIT. Here's how:

```
10 TEXT : HOME
20 VTAB 10 : PRINT "HIT ANY KEY TO CONTINUE: " : WAIT
   -16384, 128 :
30 POKE -16368,0 : REM POKE TO CLEAR KEYBOARD.
40 HOME : HTAB 1 : VTAB 10 : PRINT "THANKS, I NEEDED
   THAT."
```

If you want to use the WAIT statement with a selected key input, think of WAIT as a special kind of POKE. That is, when you press a key, the ASCII screen value of that key is put into the address of −16384, almost in the same way as a POKE. Now, since we've been using WAIT −16384,128, it would seem that the value (128) is stored in location −16384. However, the 128 refers to an ASCII value greater than 128 when used with WAIT. If you look at the ASCII screen values, you will find that 128 is the beginning value of characters that can be entered from the keyboard—the control and normal characters. (To test this, try entering a control character on the above program using WAIT −16384,128.) Now to see that WAIT can be used with a selected character, try the following program:

```
10 TEXT : HOME
20 W = -16384 : P = -16368
25 REM USE VARIABLES INSTEAD OF WRITING IN VALUES
   EACH TIME
30 VTAB 10 : PRINT "HIT ANY KEY TO CONTINUE OR 'Q' TO
   QUIT->"
40 WAIT W,128 : IF PEEK (W) = 209 THEN POKE P,0 : END
45 REM 209 IS THE ASCII SCREEN VALUE OF 'Q'
50 POKE P, 0 : HOME : FOR I = 1 TO 800 : PRINT "*"; : NEXT
60 GOTO 10
```


Remember, PEEK simply looks into an address and sees what value is stored there. The address −16384 is the keyboard data location, and −16368 is the clear-keyboard strobe. By PEEKing into the keyboard data address, you can find what key was pressed last. By POKEing −16368,0, you clear the keyboard so that nothing is on the screen or in memory once you have finished using it. Also, be sure that if you enter Q, it is an uppercase Q, or the trap will not recognize that the exit letter has been hit. Alternatively, if you can trap the value for *q* which is 241. (In Appendix B is an ASCII screen chart with the values to be POKEd in for the different ASCII characters.)

Summary

This chapter has ventured into the IIGS's memory. Don't expect to understand all the nuances, but you should have a general idea of how ASCII values work, and you should know a little about addresses and locations. Most important is that you experiment with the statements introduced here and attempt to use them in your programs. The more you use different statements, the more you'll begin to understand.

The CHR$ function introduced ASCII values. CHR$ allows you to access characters not available directly from the keyboard. You can also use CHR$ to visually write control characters within a program. This is especially useful in writing disk statements from within a program.

The POKE statement enters a value to a decimal address, and the PEEK function retrieves a value from an address. Special locations in your IIGS's memory have special functions, such as the screen window setting and ASCII screen values. More advanced uses of POKE and PEEK can provide a way of virtually writing machine-level subroutines from Applesoft BASIC.

By using the CALL statement, you can run a machine program or subroutine from within an Applesoft program. Many of the CALL statements can be written with Applesoft statements, but others are accessible only with the CALL function. You have also learned how to locate the beginning address and length of binary files and to use them with BSAVE and BLOAD statements.

Finally, you saw how to stop a program with WAIT, branch on a keyboard character with PEEK, and clear the keyboard strobe with POKE. These instructions perform functions similar to earlier statements, but these new statements increase your understanding and control over the computer.

7

Using Graphics

Using Graphics

One of the nicest features of the Apple IIGS is its graphics capability. With Applesoft BASIC, you can access two kinds of graphics: low resolution and high resolution (referred to as lo-res and hi-res). Double- and quadruple-resolution graphics are also accessible, but they require a few more tricks.

Parts of the IIGS's memory are set up to provide graphics in several different combinations of lo-res, hi-res, and text. The statement that accesses lo-res graphics is GR; it gives you the lo-res graphics screen with four lines of text at the bottom. The HGR statement does the same for hi-res graphics, and the HGR2 statement accesses page 2 of hi-res. In order to get to the other screens of graphics and graphics/text, you must use POKEs. Later, we'll discuss the POKEs required to access other screens, but to get started, we will concentrate on using GR and HGR and the statements for using graphics.

Low Resolution

Low-resolution graphics are produced by little blocks on the screen. The term *low resolution* comes from the fact that the blocks can produce only rough figures due to the size of the blocks in relation to the screen size. Think of lo-res graphics as a mosaic picture. You can represent a wide variety of two-dimensional shapes, but they'll have a mosaic texture to them—hence, a low resolution.

The main advantage of lo-res graphics is the variety of colors available. There are 16 colors, numbered 0–15. To get the color you want, use the statement

COLOR = *N*

with *N* representing the color number you want. Here are the color values:

0 Black	8 Brown
1 Magenta	9 Orange
2 Dark blue	10 Gray
3 Purple	11 Pink
4 Dark green	12 Green
5 Gray	13 Yellow
6 Blue	14 Aqua
7 Light blue	15 White

If you don't have a color TV or monitor, the colors will appear as different shades of black and white (or green or amber if you have a green or amber 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 (COLOR = 15) until you get used to the statements. Later, when you get used to the line patterns created on a noncolor screen, you can mix them for different effects.

Good graphics and bad text. One of the problems that you'll often run into when using graphics involves the intensity of the color and the quality of the text. When programming with color graphics, users tend to intensify the color to get the best results. This is as it should be. But notice, however, that when you go from graphics back to text the intensified color results in blurred text. To solve this problem, once you have set your colors properly, change only the "color" dial, not the "tint" dial. By turning the color dial *up* when using graphics and turning it *down* when using text (for example, when writing programs), you can keep your color settings correct and have clear text as well.

When you've got the color set, let's look at some statements and write a program that will produce some color bars. First, we will set the computer for lo-res graphics with a GR statement. Then, using a FOR-NEXT loop, we will generate the different colors and positions on the screen for the color bars. The statement VLIN will provide a vertical line that begins at a given starting point and terminates at an ending point at a given horizonal position.

```
10 TEXT : HOME
20 GR
30 FOR X = 0 TO 15
40   COLOR = X
50   VLIN 5, 35 AT X + 3
60 NEXT X
```


Look more closely at what's been done. The FOR-NEXT loop produces the different colors by changing the value of COLOR from 0 to 15. That's simple enough. The VLIN statement draws a line from vertical position 5 to vertical position 35 at horizontal position X + 3. This will be horizontal positions 3–18 on your screen. That's what causes the bars to line up neatly. (We use an *offset* of 3 so that the bars will not be jammed up against the left side of the screen.)

You can do the same thing with horizontal lines by using the statement HLIN. Let's stack the bars on top of one another:

```
10 TEXT : HOME
20 GR
30 FOR X = 0 TO 15
40   COLOR = I
50   HLIN 5, 35 AT X + 10
60 NEXT X
```

You've probably figured out that HLIN works exactly like VLIN, except the lines are horizontal rather than vertical. Now, try drawing a frame around the screen in white using both HLIN and VLIN:

```
10 TEXT : HOME
20 GR : COLOR = 15
30 HLIN 0,39 AT 0 : REM HORIZONTAL LINE AT TOP OF
   SCREEN
40 HLIN 0,39 AT 39 : REM HORIZONTAL LINE JUST ABOVE
   TEXT PORTION
50 VLIN 0,39 AT 0 : REM VERTICAL LINE DOWN LEFT SIDE
60 VLIN 0,39 AT 39 : REM VERTICAL LINE DOWN RIGHT SIDE
```

Before continuing, let's look at something that you may not have expected. Get your screen into the 80-column mode by pressing the esc key and 8; then run this program:

```
10 TEXT : HOME
20 GR
30 COLOR = 15
40 HLIN 0,39 AT 10
50 VTAB 23
60 FOR X= 0 TO 79
70   PRINT "X";
80 NEXT X
90 GET A$ : TEXT : LIST
```

While the Apple IIGS is in the 80-column mode, the line of 80 X's extends across the screen in a single line that fills a

horizontal block. However, it takes only 40 low-resolution blocks to do the same thing. Now, press the esc key and 4 to go into the 40-column mode, and run the same program. The low-resolution line is the same, but this time there are two lines of X's across the bottom of the screen. That's to be expected, since in the 40-column mode, after 40 characters have been printed, your IIGS starts over down one row. What you've learned is that low-resolution graphics are not affected by the 40/80-column modes. However, you can have 40 or 80 columns in the four rows of text at the bottom on the screen. Now, back to some more drawing.

It's easy to make good-looking, clear horizontal and vertical lines, but what about diagonal lines? Since there is no statement to draw diagonal lines, you'll need to use another statement, PLOT, to put a little box, or plot, at a series of points. Add the following statements to the last program:

```
70 FOR X = 0 TO 39 : PLOT X, X : NEXT
75 REM DRAWS DIAGONAL LINE FROM UPPER LEFT TO
   LOWER LEFT
80 FOR X = 39 TO 0 STEP -1 : PLOT X, 39 - X : NEXT
85 REM DRAWS DIAGONAL LINE FROM UPPER RIGHT TO
   LOWER LEFT
```

Making a Bar Graph

So far, so good. You have made straight and diagonal lines, but other than providing an exercise, what practical applications do lo-res graphics have? Besides games, which we'll get to in a bit, you can easily make graphs and charts. We'll make a simple bar graph by using a combination of lo-res graphics and the text we have at the bottom of the screen:

```
10 TEXT : HOME
20 VTAB 10: INPUT "NAME OF PLOT-> ";NP$: HOME
30 VTAB 10: PRINT "HOW MANY PLOTS?(1-7) ";: GET P%:
   PRINT P%: PRINT
40 PRINT : FOR I = 1 TO P%
50 PRINT "PLOT VALUE (0-39) FOR PLOT ";I;" ->";: INPUT
   " ";V(I): IF V(I) > 39 THEN PRINT CHR$ (7): GOTO 50
60 NEXT
70 GR : COLOR= 10
80 VLIN 0,39 AT 0: HLIN 0,39 AT 39
90 FOR I = 0 TO 39 STEP 4: PLOT 1,I: NEXT
100 COLOR= 15: FOR I = 1 TO P%: VLIN 39,(39 - V(I)) AT (I *
    5): NEXT
```



```
110 HOME : FOR I = 1 TO P%: VTAB 21: HTAB (I * 5) + 1:
    PRINT I: NEXT I
120 PRINT : HTAB 20 - LEN (NP$) / 2: PRINT NP$
130 WAIT - 16384,128: TEXT : HOME : END
140 REM HOLDS GRAPH ONSCREEN UNTIL KEY IS HIT THEN
150 REM CLEARS SCREEN AND GOES BACK TO TEXT
```

Run the program and see how nicely you can present data graphically. The program is severely limited in that it does only a maximum of seven plots using values from 0 through 39. It's simple to change the number of plots above seven, however. All you have to do is to change the trap value to a higher number and the offset in line 80 to less than I * 5. You can go as low as the values of I, but then all the bars will be adjacent. Changing the values to above 39 requires more sophisticated manipulations because 39 represents the maximum length of a vertical plot. Using a two-bar plot, we will examine how to enter any range of numbers we want.

```
10 TEXT : HOME
20 INPUT "MAX VALUE->";MV
30 R = 39.9/MV : REM R=RATIO
40 FOR X = 1 TO 2
50  INPUT "PLOT VALUE-> ";PV(X)
60  PV(X) = INT (PV(X) * R)
70 NEXT X
80 GR
90 COLOR= 15
100 VLIN 0,39 AT 0: HLIN 0,39 AT 39
110 FOR X = 0 TO 39 STEP 4: PLOT 1,X: NEXT
120 FOR X = 1 TO 3: VLIN 39,(39 - PV(1)) AT 10 + X: NEXT
130 FOR X = 1 TO 3: VLIN 39,(39 - PV(2)) AT 20 + X: NEXT
140 WAIT -16384,128 : TEXT : HOME
```

We'll go over the significant lines and explain what has happened. In line 30 the program establishes a ratio using the maximum value (MV) entered in line 20. The value 39.9 represents the upper limits on the vertical screen to be used.

Two values for PV(X) are entered in lines 40–70, and in line 60, PV(X) is multiplied by R, the ratio established in line 30. The INT statement is introduced to provide an integer (whole) number for charting.

In lines 100–120 the chart outline is set up. Finally, in line 130, the charts are plotted, using three vertical lines to construct each bar. This is done by using a FOR-NEXT loop of 3, incrementing the horizontal placement by 1 each time through the loop.

Animation

We've spent a good deal of time working on charts in lo-res graphics, but it is important to see the practical applications. Often, users see lo-res graphics simply as a way to draw mosaic pictures and nothing else, but it is possible to make very good practical use of them as well. Now, let's have a little fun before going on to hi-res graphics.

You can use animation in lo-res in games and for special effects. However, we will touch upon only some elementary examples to provide you with the concepts of how animation works. Basically, you plot a position in any color except black, and then plot a new position and cover up the old position in black. This gives the appearance of a block moving, since it is plotted from one adjacent position to another, its previous position being erased with a black plot. For example, from the immediate mode, enter the following line:

```
TEXT : GR : COLOR = 15 : PLOT 20, 20
```

Next, type in

```
COLOR = 0 : PLOT 20, 20 : COLOR = 15 : PLOT 21, 20
```

If you watch carefully, you will see what appears to be a moving block. However, you can see from what you entered that it is really a matter of plotting a block in one position, erasing it, and drawing it in an adjacent position. For a more dramatic example, the following program will start in the upper right-hand corner of your screen and bounce a white block around the screen:

```
10 TEXT : HOME : GR
20 FOR I = 1 TO 38: PLOT I,I : COLOR = 15 : PLOT I + 1, I +
   1 : COLOR = 0 : NEXT
30 COLOR = 0 : PLOT 39,39 : REM CLEAR THE BLOCK IN THE
   CORNER
40 FOR I = 39 TO 1 STEP -1 : PLOT I, 38 : COLOR = 15: PLOT
   I - 1,38 : COLOR = 0 : NEXT
50 COLOR = 0 : PLOT 0,38
60 FOR I = 1 TO 39 : PLOT I, 39 - I : COLOR = 0 : PLOT I,39
   - I : COLOR = 15 : NEXT
70 COLOR = 0 : PLOT 39, 0
80 FOR I = 39 TO 1 STEP -1 : PLOT I,0 : COLOR = 15 : PLOT I
   - 1,0 : COLOR = 0: NEXT
```

By experimenting with different algorithms, you can do anything from making letters and numbers to creating animated games.

High-Resolution Graphics

Once you understand lo-res graphics, you will understand most of the concepts of hi-res graphics, though there are some important differences:

Lo-Res	Hi-Res
16 colors	8 colors
40 × 48 matrix	80 × 192 matrix
Line statements	HPLOT statements only

Those three are the main differences, but there are others that are not so easily summarized. One of the differences concerns the preservation of a graphics screen. If you exit a lo-res drawing and enter TEXT and HOME, you will destroy the graphics you have drawn. You can test this by drawing a simple lo-res graphic and then returning to the graphics "page" with POKEs. For example, the following program will draw a simple lo-res graphic, clear back to the text page, and then return to the graphic with POKEs:

```
10 TEXT : HOME
20 GR : COLOR = 15
30 FOR I = 0 TO 39 : PLOT I, I : NEXT
35 REM LINE 30 DRAWS A DIAGONAL LINE
40 PRINT "HIT ANY KEY TO CONTINUE";: WAIT -16384,128
   : POKE -16368,0
50 TEXT : HOME : LIST : REM SHOW YOU ARE BACK IN THE
   TEXT SCREEN
60 POKE -16304,0 : POKE -16300,0 : POKE -16298,0
65 REM POKES TO RETURN TO LO-RES GRAPHICS WITHOUT
   DESTROYING
67 REM WHAT IS CURRENTLY THERE
70 PRINT "HIT ANY KEY TO CONTINUE";: WAIT -16384,128
   : POKE -16368,0 : TEXT : END
```

As you can see when you run the program, the diagonal line is no longer there after you switch back to the text screen. However, if you do the same thing with hi-res graphics, you will find the graphics there waiting for you. Run the following program:

```
10 TEXT : HOME
20 HGR : HCOLOR = 3 : REM HI-RES COLOR FOR WHITE
30 HPLOT 0,0 TO 279,155 : REM DRAWS DIAGONAL LINE
40 VTAB 22 : PRINT "HIT ANY KEY TO CONTINUE";: WAIT
   -16384,128 : POKE -16368,0
50 TEXT : HOME : LIST
60 HTAB 1 : VTAB 22 : PRINT "HIT ANY KEY TO
   CONTINUE";: WAIT -16384,128 : POKE -16368,0
```

```
70 POKE −16304,0 : POKE −16301,0 : POKE −16300,0 : POKE
   −16297,0
75 REM LINE 70 POKES FOR HI-RES PAGE 1 WITHOUT
77 REM DESTROYING CURRENT GRAPHICS
80 HTAB 1 : VTAB 22 : PRINT "HIT ANY KEY TO
   CONTINUE";: WAIT −16384,128 : POKE −16368,0
90 TEXT : END
```

This time you see the diagonal line. For most applications, as long as you do not enter HGR, your graphics will remain on the graphics page. In larger programs, it is possible for Applesoft to crash into the hi-res memory area and destroy or distort your graphics. Normally, however, you can put up a hi-res graphics picture and it will stay there, even if you type in NEW or FP. To see for yourself, type the POKEs in line 70. Your hi-res graphics should still be there. We'll return to the POKEs in more detail later.

Drawing in Hi-Res

Now that you've seen an important difference between hi-res and lo-res in terms of keeping graphics in memory, let's try some drawing. The HPLOT statement will put a dot (called a *pixel)* in the coordinates you specify in a range of 0 to 279 horizontally and 0 to 191 vertically. The syntax is

HPLOT *X,Y*

with *X* the horizonal position and *Y* the vertical. We'll start with a dot in the upper left corner. From the immediate mode, enter

```
HGR : HCOLOR = 3 : HPLOT 0,0 : VTAB 22
```

You should see a little white dot. To draw lines, you HPLOT from X1,Y1 to X2,Y2. For example,

```
HGR : HCOLOR = 3 : HPLOT 0,0 TO 279,0 : VTAB 22
```

This draws a line across the top of your screen. You may be wondering why there is a VTAB 22 at the end of the line. That's because only the bottom four lines are in text; if you don't include VTAB 22, you will not be able to see your cursor. It will be "behind" the graphics screen.

With HPLOT you can begin at a point and draw all over the screen with a single HPLOT statement and several TO statements. For example, try the following program:


```
10 TEXT : HOME
20 HGR : HCOLOR = 3
30 HPLOT 0,0 TO 279,0 TO 0,150 TO 279, 150
40 Z$ ="THE MARK OF ZORRO!": VTAB 22: HTAB 20 −
   LEN(Z$)/2 : PRINT Z$
50 WAIT −16384,128 : TEXT : HOME
```

Experiment with different plots and colors until you're comfortable with them.

Random Numbers

We have not yet explored the RND function, which generates a range of random numbers. This is a good place to introduce it since it's fun to use with graphics. The basic syntax is

RND *(N)*

with *N* a number less than 1 but greater than 0. Usually, you'll be looking for a range of numbers greater than 1 and for numbers that have no fractional parts. The following is a handy little formula that generates a range of numbers from 1 to *N*, with *N* being the upper limit of random numbers to be generated:

INT (RND (1) * (N) + 1)

To see how it works, enter the following program:

```
10 TEXT : HOME
20 FOR I = 1 TO 66
30 PRINT INT ( RND (1) * (10) + 1),
35 REM LINE 30 GENERATES RANDOM NUMBERS FROM 1 TO
   10
40 NEXT
```

Now, let's use the random number generator in a program with hi-res graphics. The random number generator will be used to make the computer create a picture, using different colors and lines. In other words, we will have a program that does the graphics drawing, with each one different.

```
10 TEXT : HOME
20 REM HI-RES RANDOM GRAPHIC GENERATOR **
   STARBURST **
30 N = 0
40 X = 100 : Y = 100 : REM X,Y AXIS FOR CENTER OF
   'STARBURST'
50 HGR
60 RC = INT (7 * RND (1)) + 1 : REM GENERATES RANDOM
   COLOR VALUES
```

```
70 REM GENERATE RANDOM MAXIMUM X AND Y AXIS
   POINTS
80 X2 = INT (279 * RND (1))
90 Y2 = INT (158 * RND (1))
100 HCOLOR = RC
110 HPLOT X,Y TO X2,Y2
120 N = N + 1 : IF N = 100 THEN 140
130 GOTO 60
140 A$ = "STAR BURST" : VTAB 22 : FLASH : HTAB 20 -
    LEN (A$) / 2: PRINT A$: NORMAL : WAIT -16384,128 :
    TEXT : HOME
```

Colored Backgrounds

By now you should be able to see the different ways in which lines can be drawn on the hi-res screen. Sometimes, though, you may want to draw on a background other than black. Wouldn't it be nice to draw on a blue or green background? That's what we'll do now.

To clear the screen to the most recent HCOLOR that's been used to plot lines, use the CALL 62454 command, which CALLs a monitor subroutine that will flood the screen with the most recent HCOLOR. The next program draws a rough hourglass (actually an X with horizontal lines on the top and bottom) in the color of your choice and then clears the screen to that color. Notice that instead of CALL 62454 being used, the variable C is defined to 62454 and then CALL C is used. This is done to show you that in programs where you will be using CALL 62454 in several different places, it's a lot easier simply to write CALL C. (It's used only once in this program, but you get the idea.)

```
10 TEXT : HOME : REM **CLEAR TO COLOR**
20 HGR
30 VTAB 21 : PRINT "ENTER COLOR (0 TO 7)" : INPUT
   "(COLORS 0 AND 4 ARE BLACK) " ; HC
40 HCOLOR = HC
50 HPLOT 0,0 TO 279, 155 TO 0,155 TO 279,0 TO 0,0
60 VTAB 22 : CALL -875 : PRINT "HIT ANY KEY TO
   CONTINUE"
65 REM CALL -875 CLEARS THE LINE OF PREVIOUS TEXT
70 WAIT -16384,128 : POKE -16368,0
80 C = 62454
85 REM ADDRESS FOR CLEARING SCREEN TO MOST RECENT
   HCOLOR
90 CALL C
100 WAIT -16384,128 : TEXT : HOME : LIST
```


Drawing Board Program

The graphics power of the Apple IIGS is considerable. Several commercial programs are available that allow you to write in hi-res fonts, draw several different figures, and generally use your computer as an artist would use a canvas or a drafter a drafting board. Of course, you can write such a program yourself. To give you an idea of how graphics programs work, here is the "Very, Very Poor Person's Drawing Board." Enter it in and play with it.

```
10 TEXT : HOME
20 ONERR GOTO 10
30 HOME : HGR : HCOLOR= 3: VTAB 23
40 PRINT "LINES OR FIGURES? (L/F)": INVERSE : PRINT
   ""C"";: NORMAL : PRINT " TO CLEAR";: VTAB 23: HTAB 38:
   GET AZ$: IF AZ$ = "F" THEN 130
50 IF AZ$ = "C" THEN 30
60 HOME : VTAB 23: INPUT "HOR PLOT, VERT PLOT BEGIN->
   ";H,V
70 INPUT "HOR PLOT, VERT PLOT END -> ";HE,VE
80 HPLOT H,V TO HE,VE
90 HOME : VTAB 23: PRINT "ANOTHER? (Y/N) ";: GET AN$
100 IF AN$ = "N" THEN INVERSE : PRINT " CLEAR SCREEN?
    (Y/N)? "; : NORMAL : GET CL$: IF CL$ = "Y" THEN
    TEXT :
    HOME : END
110 IF CL$ = "N" THEN END
120 GOTO 40
130 HOME : VTAB 21: PRINT "(S)QUARE,(T)RIANGLE": PRINT
    "(R)ECTANGLE,(P)ARALLELOGRAM": PRINT "CHOOSE BY
    LETTER-> ";: GET L$:
140 IF L$ = "S" THEN HPLOT 0,0 TO 255,0 TO 255,150 TO
    0,150 TO 0,0
150 IF L$ = "T" THEN HPLOT 127,0 TO 255,150 TO 0,150 TO
    127,0
160 IF L$ = "R" THEN HPLOT 0,20 TO 255,20 TO 255,90 TO
    0,90 TO 0,20
170 IF L$ = "P" THEN HPLOT 20,20 TO 200,20 TO 180,80 TO
    0,80 TO 20,20
180 GOTO 90
```

As you can see, the Very, Very Poor Person's Drawing Board allows you to enter different figures simply by going to subroutines. This saves a lot of HPLOTting every time you want a simple figure. However, wouldn't it be nice if you could INPUT the size of the figure you wanted and the computer would do everything else? As you saw in the above program,

you can INPUT different line sizes and angles, but you're stuck with a single figure size.

The next program shows you how to INPUT a figure of different sizes. Again, it is relatively simple and always begins the figure in the upper left corner and draws only a square, but it will give you an idea of how such graphics can be done.

```
10 TEXT : HOME : REM **INPUT SQUARE**
20 INPUT "SIZE OF SQUARE (1 TO 155)-> ";SS
30 HGR : HCOLOR = 3
40 HPLOT 0,0 TO SS,0 TO SS,SS TO 0,SS TO 0,0
45 REM HPLOTS A SQUARE TO YOUR SPECIFICATIONS
50 HTAB 1: VTAB 22: CALL-875 : PRINT "ANOTHER
   SQUARE? (Y/N) ";: GET AN$: IF AN$="N" THEN TEXT :
   HOME :END
60 IF AN$ = "Y" THEN 10
70 GOTO 50 : REM A TRAP FOR NOT GETTING A 'Y' OR 'N'
```

Adding Circles

So far you've worked only with straight-lined figures, HPLOTting the sides of the figures, which is limiting. But when you start making curves and circles, you make a quantum leap in programming. Using the SIN function, which gives a sine value, and the COS (cosine) function, you can make curves and even circles. If you have a good math background, you probably understand how these calculations can be used. However, if you do not, simply make a note of the algorithm and use it whenever you need a circle. Since there are several calculations occurring, it will take a while for the circle to be completely drawn. Be patient, though; your IIGS will draw it before your eyes.

```
10 TEXT : HOME : HGR : HCOLOR = 3 : REM **HGR CIRCLE**
20 FOR I = 0 TO 6.3 STEP .01
30 R = 40 : XPLACE = 100 : YPLACE = 75 : REM R =
   RADIUS - XPLACE = HOR START - YPLACE = VERT
   START
40 X = R * COS (I) + XPLACE
50 Y = (R / 4) * SIN (I) / .3 + YPLACE
60 HPLOT X,Y
70 HPLOT TO X,Y
80 NEXT
90 VTAB 22 : HTAB 1 : PRINT "HIT ANY KEY TO CLEAR"; :
   WAIT -16384,128 : TEXT : HOME
100 REM TRY CHANGING THE VALUE OF R, XPLACE AND
   YPLACE.
```


It's important that you experiment—even with functions such as SIN and COS that you may not understand. Also, you might want to want to skim an old geometry or trigonometry book to brush up on what sine and cosine do. Just for fun, of course.

Keeping Graphics Together

Suppose you spend a lot of time with a graphics drawing, made with a program you've struggled over. As you develop your graphics, you may not want to wait while it slowly draws circles and puts your masterpiece on the screen every time you edit your program. However, with so little room at the bottom of the hi-res screen for you to examine your listing, you need to TEXT the screen to see your listing. To get back to the hi-res screen, you have to HGR, and, as you know by now, that will blast anything on the screen.

To keep your drawing intact after you have left hi-res, you will need to POKE in certain values that put you on the hi-res screen(s). As you saw earlier in the comparison of hi-res and lo-res, pictures can be kept in memory. The following POKEs are handy for doing this:

POKE	Result
−16304,0	Goes to graphics
−16302,0	Full graphics
−16301,0	Text and graphics
−16300,0	Page 1
−16299,0	Page 2
−16298,0	Lo-res
−16297,0	Hi-res

To use these POKEs, you can either enter them from the immediate mode or from within a program. It's better to have them somewhere in a program since it is necessary to enter at least four POKEs to access a hi-res screen. The following program shows how they can be used to draw on the hi-res screen, enter the TEXT mode, and then reenter the hi-res screen without destroying the graphics there.

```
10 TEXT : HOME : HGR : HCOLOR = 3
20 HPLOT 0, 70 TO 279, 70
30 HPLOT 140, 0 TO 140, 155 : REM DRAWS A CROSS
40 HTAB 1 : VTAB 22 : PRINT "HIT ANY KEY FOR TEXT-> ";
   : WAIT -16384,128 : POKE -16368,0
50 TEXT : HOME : VTAB 10 : PRINT "THIS IS THE TEXT
   SCREEN!"
```

```
60 VTAB 22 : INVERSE : PRINT "HIT ANY KEY TO RETURN
   TO YOUR PICTURE-> ";: NORMAL : WAIT -16384,128 :
   POKE -16368,0
70 POKE -16304,0 : POKE -16301,0 : POKE -16300, 0 :
   POKE -16297,0
75 REM POKES TO RETURN TO PAGE 1 HI-RES GRAPHICS
77 REM WITH TEXT AT BOTTOM
80 HOME : VTAB 22 : HTAB 1 : PRINT "WE'RE BACK TO
   GRAPHICS!!"
```

When you run this program, you will first see the cross, then the text in the middle of the screen, and finally the cross again.

In the discussion of lo-res graphics, you saw how to create animation by first PLOTting a block, drawing over it with black, and then PLOTting another point. You can do the same in hi-res to create animation. However, there is something else that you can do with hi-res and animation that we didn't do in lo-res. You can *toggle* the two hi-res screens. So far, we've used only hi-res screen 1 (or page 1). By using both screens 1 and 2 of hi-res, you can draw images on both. Then by using POKE −16300,0 (page 1) and POKE −16299,0 (page 2), you can alternate the graphics. The following program illustrates how this is done:

```
10 TEXT : HOME : HGR2 : HGR : HCOLOR= 3: REM CLEARS
   EVERYTHING
20 HPLOT 50,70 TO 100,70
30 HPLOT 55,70 TO 95,50 TO 95,70
40 HGR2
50 HPLOT 50,70 TO 100,70
60 HPLOT 55,70 TO 95,60 TO 95,70
70 FOR I = 1 TO 30: FOR J = 1 TO 100: NEXT J: POKE -
   16300,0: FOR K = 1 TO 100: NEXT K: POKE - 16299,0:
   NEXT
80 REM TOGGLES HI-RES PAGES 1 AND 2 1000 WAIT -
   16384,128: TEXT : LIST
```

Line 70 contains three loops. The I loop sets the number of times the screens will be toggled at 30. The J and K loops simply delay the screen toggles long enough so that it does not appear that a single combined picture of HGR and HGR2 images is on the screen by itself. By changing the HPLOTs in lines 20 and 30, and in 50 and 60, you can create your own animations.

BSAVEing Your Graphics

In Chapter 6, we discussed BSAVEing binary files. Graphics screen images can also be saved to the disk as 34-sector binary files. However, rather than having to determine the starting address and length of your binary graphics file, you can just remember two simple addresses and lengths:

Hi-res page 1　A$2000,L$2000
Hi-res page 2　A$4000,L$2000

Graphics on page 1 start at hex address $2000 and have a length of $2000, while those on page 2 start at $4000 and have a length of $2000. Therefore, all you have to remember are $2000 and $4000. If you don't want to do that, this program will do it for you:

```
10 TEXT : HOME : REM **PICTURE CAPTURE**
20 D$ = CHR$(13) + CHR$(4)
30 HOME : VTAB 4 : INPUT "WHAT NAME DO YOU WANT FOR
   THIS PICTURE? "; NA$
40 HOME : VTAB 4 : INPUT "WHAT DRIVE? "; R%
50 HOME : VTAB 4: INPUT "PAGE 1 OR PAGE 2? "; PG%
60 IF PG% = 2 THEN PRINT D$; "BSAVE " NA$ ",A$4000,
   L$2000,D" R%
70 IF PG% < > 1 THEN GOTO 50
80 PRINT D$ "BSAVE"; NA$; ",A$2000, L$2000,D" R%
90 INPUT " DO YOU WANT TO CONTINUE? "; AN$: IF AN$ =
   "Y" THEN GOTO 10
100 HOME : VTAB 10: PRINT "DONE FOR NOW" : END
```

To use the above program, draw your graphics on the screen, TEXT back to the text screen, and run the program. It will save your graphics on disk.

Now, to get your graphics back. You can simply BLOAD your picture (for example, BLOAD *FILENAME),* or you can BLOAD your graphics to page 1 or page 2. Since a picture BSAVEd from page 1 or 2 will automatically BLOAD to the appropriate screen, you do not have to include an address (for example, A$2000 or A$4000), but if you want to load a picture to a screen other than the one to which it was BSAVEd, you must include the loading address. For example, if you BSAVEd a file named PICTURE from hi-res screen 1 and wanted to bring it up on HGR2, you would have to

BLOAD PICTURE, A$4000

To view your pictures, you can either POKE in the appropriate addresses after the picture is loaded, or use HGR or HGR2 and BLOAD them "in the dark."

That's simple enough, but the following program will do it all for you. You will be able to see your picture as it is loaded to either hi-res screen.

```
10 TEXT : HOME : D$ = CHR$ (13) + CHR$(4)
15 REM **PICTURE LOADER**
20 L$ =
   "================================":
   VTAB 20 : PRINT L$: VTAB 22 : PRINT L$ : VTAB 21 :
   INPUT "NAME OF PICTURE-> "; PIC$
30 VTAB 23 : PRINT "PAGE 1 OR 2 " ; : GET P%
40 IF P% = 2 THEN 110
100 POKE -16304,0 : POKE -16302,0: POKE -16300,0: POKE
   -16297,0
102 REM PAGE 1 POKES
105 GOTO 200
110 POKE -16304,0 : POKE -16302,0: POKE -16299,0: POKE
   -16297,0
115 REM PAGE 2 POKES
120 GOTO 210
200 PRINT D$;"BLOAD" PIC$ ",A$2000"
205 GOTO 300
210 PRINT D$;"BLOAD" PIC$ ",A$4000"
300 END
```

Shapes: Bitmapped Graphics

There are two ways to produce shapes—an easy way and a hard way. The easy way is to get a shape editor program, such as *Apple Mechanic* (Beagle Bros.) and spend your time creating interesting shapes for animation. The hard way is what we're going to do now: map every single bit in memory. If you don't want to take on this project, that's fine; it's a large task. If you prefer, go on to the summary or the next chapter.

For those of you who enjoy delving into the abyss of your computer's memory, welcome aboard. This will be an interesting project and worth the effort. To get started, we'll be using eight bits, but it is useful to think of the eight bits as sets of 2, 3, and 3. The first two bits are generally unused, and so for the most part we're actually dealing with two 3-bit sets. The shape table we will build is based on entering codes into a series of bytes using binary and hexadecimal codes. All of these, of course, must be translated further into decimal codes that can be used from a BASIC program. You should be able at least to begin understanding shapes and see what they can do.

First of all, think of a shape as something you draw in memory in three-bit increments, with an occasional two-bit

move thrown in if you happen to be in the right place. You can move up, down, left, or right. And you can choose to move, or move and draw. If you'll think about that for a second, that gives eight choices, and, as you will see, there are eight values that represent those moves.

The procedure is very much like what you do when you draw with paper and pencil, but there you don't have to think about it. With shapes, you can first draw what you want, generally on graph paper, and then plot it in your IIGS's memory. We will first use a combination of binary and hexadecimal numbers. Once we have our hex values, we'll translate them into decimal numbers to be used in a BASIC program. To kick things off, here are the moves you have available and their values:

Move	**Plot**	**Binary**	**Hex**
Up	No	000	0
Right*	No	001 *or* 01	1
Down*	No	010 *or* 10	2
Left*	No	011 *or* 11	3
Up	Yes	100	4
Right	Yes	101	5
Down	Yes	110	6
Left	Yes	111	7

Each move or move/plot is recorded in a three-bit segment of a byte, or a two-bit segment if appropriate. Notice the moves marked with an asterisk and their binary values. If, in the sequence of plotting and moving, one of those moves is to be recorded and there is a two-bit segment as the next available segment, it can be recorded as a two-bit value. Let's look at an example.

Shape A = a move/plot to the left, move/plot up, and move to the left:

Move/plot left = 111
Move/plot up = 100
Move left = 011 or 11

The three segments of the byte will be numbered from 1 to 3 so that we can keep the sequence in order:

Segment 1 = 111
Segment 2 = 100
Segment 3 = 11

	Segment 3		**Segment 2**			**Segment 1**		
Bit	7	6	5	4	3	2	1	0
	1	1	1	0	0	1	1	1

Now, that was pretty simple. If you get organized, it is simple. However, consider what would be required if we had the following, just slightly different sequence of moves and plots.

Shape B = a move/plot to the left, move/plot up, move/plot up, and move to the left:

Move/plot left = 111
Move/plot up = 100
Move/plot up = 100
Move left = 011 or 11

Since the third action involves a three-bit operation, you cannot use segment 3. So, all you do is go to the next byte. The diagram shows what shape B will look like when mapped:

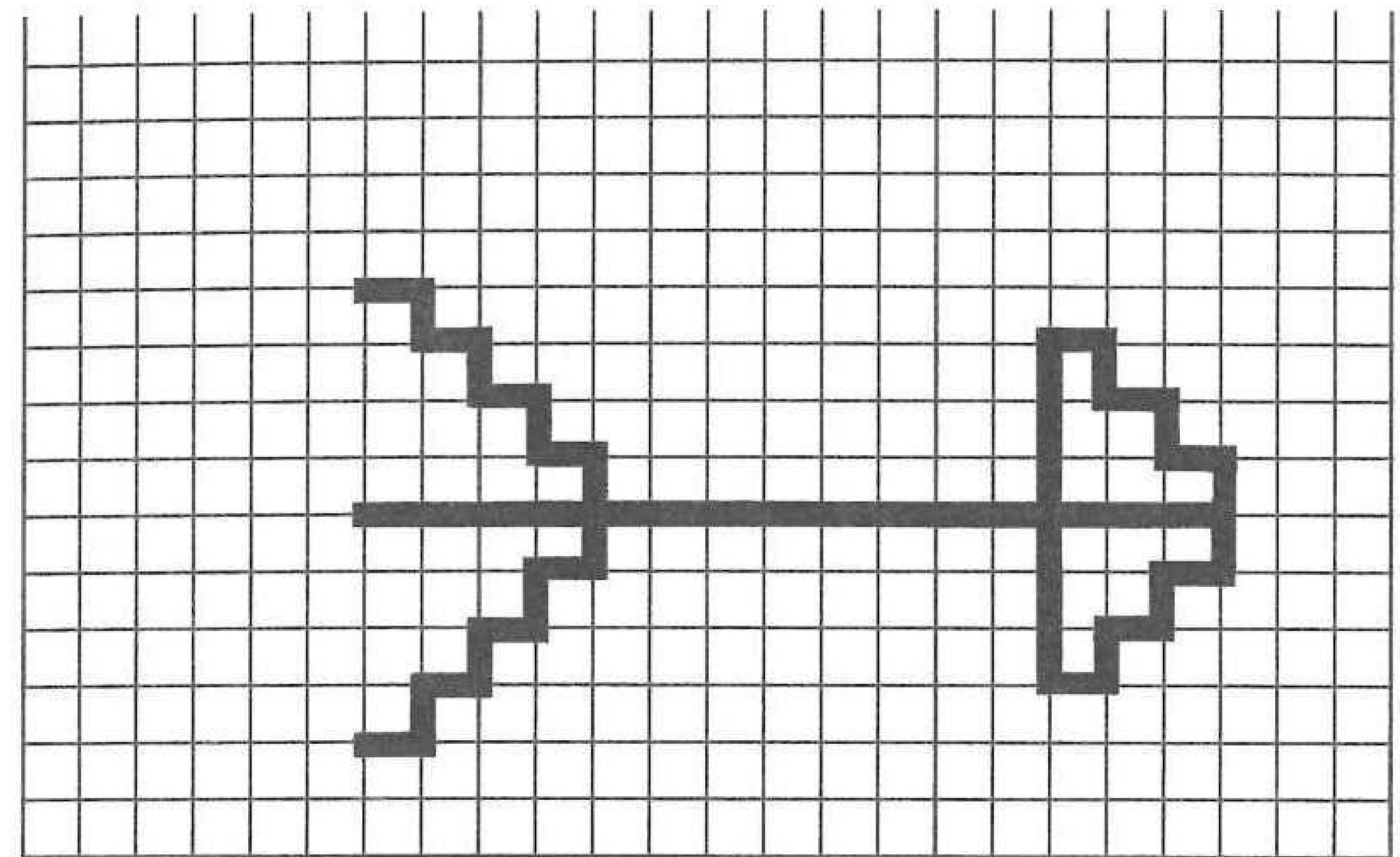
	Segment 3		Segment 2			Segment 1			
Bit	7	6	5	4	3	2	1	0	Byte #1
	0	0	1	0	0	1	1	1	
	(*)			(2)			(1)		
Bit	7	6	5	4	3	2	1	0	Byte #2
			0	1	1	1	0	0	
				(4)			(3)		

The numbers in parentheses show the sequence from one byte to the next where the values are placed. To draw any shape you can place in memory, you do the same thing, except shapes will take more steps than the little examples shown so far.

Now, you're ready to draw something you can see on your IIGS. Figure 7-1 is an arrow drawn on graph paper. Beginning with the tail of the arrow, you move/plot 16 to the right, move/plot to make the head, and end up at the tip of the arrow. Then you move 11 spaces without plotting to the left to start on the tail of the arrow, doing first the top and then the bottom.

The final step is translating the binary into hexadecimal and the hexadecimal into decimal. To do that you will have to rearrange the byte breakdown. Instead of treating it as sets of 2, 3, and 3 bits, you will now treat it as 4 and 4 bits. Let's look at the example of byte 1 in the diagram above.

Figure 7-1. Arrow



Instead of segments 1, 2, and 3, you now have a high and low nybble. (*Nybble* is the jargon term for a four-bit group—half a byte.) The binary values in the bits are the same, but to translate the byte into a hexadecimal value, you'll find that it's easier to break it into nybbles.

High Nybble				Low Nybble				
7	6	5	4	3	2	1	0	**Byte 1**
0	0	1	0	0	1	1	1	

Once you've broken them into nybbles, translating the byte into hex is very easy since you just substitute the four-bit nybble for a single-digit hex value. In the following chart notice that the full range of four binary digits (000–1111) exhausts the single-digit range of hexadecimal numbers (0–A). That is also a clue as to why hexadecimal values are used with computers.

Binary	Hex	Binary	Hex
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

To translate byte 1 into a hex value, you see that

High nybble: 0010 = $2
Low nybble: 0111 = $7
Byte 1 = $27

Now, turning to a hexadecimal-to-decimal translation table, you find the decimal value of $27, which is 39 (see Appendix C). That was a lot of effort to change a single byte, but once everything is organized, it is relatively simple. Table 7-1 is a complete shape table depicting the arrow.

Now that the shape has been translated into decimal values, let's create a program that will use those decimal numbers. This involves two things:

- Placing the shape somewhere in memory where it won't clash with other information
- Telling your IIGS where to find the shape information

First of all, there's a special register at locations $E8 and $E9 (232 and 233) where you store the starting address of your shape. We will store the shape information beginning at address $300 (decimal 768) since that is a location with some free memory. In locations $E8 and $E9, the starting address will be stored as

$E8: 00
$E9: 03

Thus, it's necessary to break up $300 into two parts and POKE the decimal equivalent into those locations. Fortunately, you can make a direct translation from hex to decimal:

Hex	Decimal
00	00
03	03

Since the low byte is stored in the first address and the high byte in the second, we will POKE 232,0 and 233,3.

Finally, you must provide a little more information at the beginning and end of the shape data:

First byte	Total number of shape definitions
Second byte	Unused
Third and fourth bytes	Relative offset for beginning of shape

In the example, we're using only one shape, and we will begin the shape data right after the relative offset information. That will be in the fifth byte, but the value will be 4 since the first byte is considered 0.

Table 7-1. Arrow Shape Table

Byte	Segment 3	Segment 2	Segment 1	Hex	Decimal
0	00	101	101	2D	45
1	00	101	101	2D	45
2	00	101	101	2D	45
3	00	101	101	2D	45
4	00	101	101	2D	45
5	00	101	101	2D	45
6	00	101	101	2D	45
7	00	101	101	2D	45
8	00	111	100	3C	60
9	00	111	100	3C	60
10	00	111	100	3C	60
11	00	110	110	36	54
12	00	110	110	36	54
13	00	110	110	36	54
14	00	100	101	25	37
15	00	100	101	25	37
16	00	100	101	25	37
17	11	011	011	DB	219
18	11	011	011	DB	219
19	11	011	011	DB	219
20	11	011	011	DB	219
21	00	111	100	3C	60
22	00	111	100	3C	60
23	00	111	100	3C	60
24	00	111	100	3C	60
25	10	010	010	92	146
26	10	010	010	92	146
27	00	010	010	12	18
28	00	100	101	25	37
29	00	100	101	25	37
30	00	100	101	25	37
31	00	100	101	25	37

Byte 0 = 01	Number of shapes
Byte 1 = 00	Unused
Byte 2 = 04	Low byte of offset
Byte 3 = 00	High byte of offset
Byte 4 = 45	First value of shape
Bytes 5–*N*	*N* = Last value of shape table
Byte *N*+1 = 00	Indicates end of shape table

After you've made these determinations, you are all set to write a program to stuff the shape information into memory and use the shapes. There are a total of 32 shape table values. Four values go at the beginning of the shape table, and one value goes at the end, giving 37 shape values to POKE in beginning at $300 (768). So you will use the following loop:

```
FOR X = 768 TO (768 + 36)
```

To make it easy, just put all the information into DATA statements; then have the loop read the data and sequentially stick it into the assigned addresses. Before that, though, you'll have to remember to indicate where the shape information is stored. Thus, $E8 and $E9 (232 and 233) will be POKEd:

```
POKE 232,0 : POKE 233,3
```

The following program does that for you:

```
10 REM $E8=232
20 POKE 232,0: POKE 233,3
30 REM $300=768
40 TEXT : HOME
50 FOR X = 768 TO (768 + 36)
60 READ S
70 POKE X,S
80 NEXT
90 DATA 1,0,4,0
100 REM 32 shape values
110 DATA 45,45,45,45,45,45,45,45
120 DATA 60,60,60,54,54,54
130 DATA 37,37,37,219,219,219,219
140 DATA 60,60,60,60,146,146
150 DATA 18,37,37,37,37
160 DATA 0 : REM end of shape
```

When you run the program, nothing that you can see on your screen will occur. You need more statements and commands to see the shape. The rest of the program is given below.

Shape Manipulation

After all of that work to create a shape, it had better be worth it, right? If you like animating the figures you create, it should be. Let's look now at the special shape statements:

SCALE. The SCALE statement sets the size of your shape. A SCALE value 1 uses the single-pixel plot resolution that you used to create your shape. Higher-value scales create larger shapes with lower resolution.

ROT. The ROT value will rotate your shape in one of eight angles. ROT recognizes values 0, 8, 16, 24, 32, 40, 48, and 56. Any other ROT values will be dropped to the next lower value (for example, 12 will be treated as 8). At angles other than 0, 90, 180, or 360, the shapes are distorted.

DRAW. Using the high-resolution pixel matrix, you plot the *X* and *Y* coordinates using the following format:

DRAW N% AT *X,Y*

where N% is the shape number and *X* and *Y* are the horizontal and vertical coordinates on your high-resolution screen.

XDRAW. XDRAW has the same format as DRAW, except it draws the complement of the color existing on the screen. In animation, XDRAW is preferable to DRAW.

By adding the following lines to the program, you can see the shape perform for you.

```
170 REM **********
180 REM DRAW SHAPE
190 REM **********
200 HGR
210 HCOLOR= 3
220 SCALE= 1
230 ROT= 1
240 DRAW 1 AT 100,50
250 GOSUB 600
260 REM ****
270 REM MOVE
280 REM ****
290 FOR X = 1 TO 279
300 XDRAW 1 AT X,80
310 XDRAW 1 AT X,80
320 NEXT
330 GOSUB 600
340 REM ******
350 REM FASTER
360 REM ******
370 FOR X = 1 TO 279 STEP 3
```

```
380 XDRAW 1 AT X,80
390 XDRAW 1 AT X,80
400 NEXT
410 GOSUB 600
420 REM ********
430 REM ROTATION
440 REM ********
450 FOR X = 1 TO 64 STEP 8
460 ROT= X
470 XDRAW 1 AT 100,80
480 NEXT
490 GOSUB 600
500 REM *****
510 REM SCALE
520 REM *****
530 ROT= 1
540 FOR X = 1 TO 5
550 SCALE= X
560 DRAW 1 AT 140,80
570 GOSUB 600
580 NEXT
590 TEXT : LIST : END
600 VTAB 22: PRINT "Hit any key"
610 GET A$
620 HGR : RETURN
```

The more you experiment, the more you can do with shapes. However, if you are lost or confused at this point, don't worry. This section has dealt with some fairly advanced aspects of programming with shapes.

Summary

In this chapter you have explored the many different graphics capabilities of the IIGS. Beginning with low-resolution graphics, you saw that it is possible to program mosaic-like images on the screen in 16 colors. Using the four lines of text at the bottom of the screen, you learned to create crisp bar charts with labels. Also, you saw how animation is possible by using black to erase images.

High-resolution graphics, while decreasing your choice of colors, increases the choice of shapes you can make. The HPLOT statement was used to create lines, figures, and even circles. The RND function was introduced to add random drawings (or anything else) to your storehouse of statements. You also saw how to use special POKEs to switch from HGR to HGR2 without erasing your graphics, and then create animation by switching the two screens. Finally, you learned how to BLOAD and BSAVE graphics from, and to, the disk. All in all, graphics can be fun, and there are numerous practical applications as well.

8

Text Files and the Disk System

Text Files and the Disk System

In this chapter you're going to learn more about working with the disk system and creating data files. Three types of files will be covered: EXEC files, sequential text files, and random access files. All of the files to be discussed are one type of text file or another, but while they are similar, they do different things in somewhat different ways. What we are calling EXEC files are actually text files that we want to access or execute. Sequential files and random access files are types of data storage systems.

You will find the functions in this chapter to be extremely practical. First, you will see how an EXEC file can be used to automatically "type" commands for you, and save programs and subroutines in text files. It's like having a robot at the keyboard. Second, you are going to make simple sequential text and random access files. These files are very useful for storing information you have entered. Rather than having to enter the data all over again, you simply OPEN the data file and READ it. Finally, to show how these files work, you'll write a simple program for keeping and updating names, cities, and states.

Create EXEC Files (and Leave the Driving to Them)

EXEC files are text files that you can EXECute with the DOS command EXEC. Depending on what's in the files, you can either put a program into memory or have your computer "type" several commands as though the computer has taken over the keyboard. First, let's examine how to put a program into a text file and then how to have the program enter commands.

Converting BAS Files to TXT Files

The purpose of this section is to show how to change a BASIC program or subroutine into a text (TXT) file. Now, you may well ask yourself, Why you would want to do that? After all, it's a lot simpler to save a program to the disk. That's a good point, except that whenever you load or run a program whatever program is in memory will be wiped out. However, by using text files and the EXEC command, you can load one program and then merge another program with it by EXECing its text file.

As an example, enter the following two partial programs and save them both to disk. Use the filename ENTER.STRINGS for this one:

```
10 TEXT : HOME : REM ENTER.STRINGS
20 VTAB 10 : INPUT "HOW MANY STRINGS TO ENTER (1-20)
   "; N%
30 DIM S$(N%)
40 HOME: FOR I = 1 TO N%
50 PRINT "STRING #"; I; " ==>"; :INPUT " "; S$(I)
60 NEXT I : HOME
70 FOR J = 1 TO N% : GOSUB 1000
80 NEXT J
90 END
```

After you've saved the above lines as a program called ENTER.STRINGS, enter NEW to clear it from memory. Next, enter the following two lines:

```
1000 HTAB 20 - LEN(S$(J)) / 2 : PRINT S$(J)
1010 RETURN
```

Now save lines 1000 and 1010 to the disk as CENTER.STRINGS. If you want to put these two pieces together, you will find it impossible without a special program or without reentering the lines by keying them in. However, if one part or the other is written as a text file, it's a simple matter to load one part and EXEC the other. So let's do that.

To create a text file, use the commands OPEN, WRITE, and CLOSE. First, issue OPEN and WRITE commands, which must be preceded by CHR$(4). Then, enter the contents of text files, and finally, close the text file using CLOSE, again preceded by CHR$(4). D$ has been defined to equal CHR$(4).

Now LOAD CENTER.STRINGS. Next DELETE CENTER.STRINGS. This will delete CENTER.STRINGS as a BAS file, but you will see that CENTER.STRINGS is still in memory. Write in the following, *but do not type in NEW:*


```
1 TEXT : HOME
2 D$ = CHR$(4)
3 PRINT D$ "OPEN CENTER.STRINGS"
4 PRINT D$ "WRITE CENTER.STRINGS"
5 POKE 33,30
6 LIST 1000,1010
7 PRINT D$ "CLOSE CENTER.STRINGS"
8 END
```

When you LIST the program now, it should look like this:

```
1 TEXT : HOME
2 D$ = CHR$(4)
3 PRINT D$ "OPEN CENTER.STRINGS"
4 PRINT D$ "WRITE CENTER.STRINGS"
5 POKE 33,30
6 LIST 1000,1010
7 PRINT D$ "CLOSE CENTER.STRINGS"
8 END
1000 HTAB 20 - LEN(S$(J)) / 2 : PRINT S$(J)
1010 RETURN
```

Run the program. You will hear your disk whirling. It is writing a TXT file called CENTER.STRINGS. When the disk stops, CATALOG your disk and you will now see the file CENTER.STRINGS.

Now, write in NEW to make sure everything is cleared from memory. First, LOAD the program ENTER.STRINGS. LIST the program to make sure it does *not* have lines 1000 and 1010 (if it does, DEL 1000,1010). Type in

```
EXEC CENTER.STRINGS
```

Your disk should whirl and soon stop. LIST your program, and *voilà*—there is your complete program. Lines 1000 and 1010 have been placed into a text file and are then "released" with an EXEC statement. This is a very handy way to store subroutines that are frequently used in your programs. Rather than having to rewrite common or lengthy subroutines every time you want them, all you have to do is EXEC them into the program portion you have keyed in. However, you must be careful as to what line numbers you use. If you have two subroutines in text files with the same range of line numbers, you'll get strange results. By blocking your important subroutines within thousand-range blocks (1000–1999, 2000–2999, for example) and keeping track of what subroutines are in what blocks, you will not have conflicts when you EXEC them.

Chaining Programs

Another way to get two (or more) programs working together is with the CHAIN command. This command does not merge two programs together as EXEC does, but it allows one to be run from another. As soon as the second program has been chained, the first one is removed.

To see how CHAIN works, we'll use another string-centering example. The first program will place strings in an array, and the second one will print them centrally on a 40-column screen. Save the first program under the name STRING1 and the second as STRING2. Be sure to put them on the same disk. When you run STRING1, you will be prompted to enter from 1 to 20 strings. After you have done that, STRING2 will automatically be chained and proceed to print the strings on the screen. After it has finished, LIST the program, and you will see that only STRING2 is still in memory.

STRING1

```
10 TEXT : HOME
20 VTAB 10
30 INPUT "HOW MANY STRINGS TO ENTER (1-20) ";N%
40 DIM S$(N%)
50 HOME
60 FOR X = 1 TO N%
70 PRINT "STRING #";X
80 INPUT " " ;S$(X)
90 NEXT X
100 D$ = CHR$ (4)
110 PRINT D$"CHAIN STRING2"
```

STRING2

```
10 FOR J = 1 TO N%
20 GOSUB 100
30 NEXT J
40 END
100 HTAB 20 - LEN (S$(J)) / 2
110 PRINT S$(J)
120 RETURN
```

The important feature of using the CHAIN command is that, when the second program is loaded into memory, all the values in the variables and array are preserved from the first program. Try deleting line 110 in STRING1, and then running STRING2 after having typed in some words in the STRING1 program. You won't get anything printed on the screen since the variables and arrays have been reset to zero.

You can organize all kinds of useful subroutines in this way. Instead of having to rewrite the routines every time they're needed in a program, you can chain them. Of course, you will have to be careful about keeping variable and array names the same, but if you spend a little time arranging the routines, you can save a lot of time later. What's more, as you can see with the above examples, when you use CHAIN, you don't have to worry about line numbers from one program chained to another.

Automatic EXECs

Another important use of EXEC files comes from their ability to issue commands as though they were coming from the keyboard. You've seen how to issue DOS commands by using control-D or CHR$(4) from within a program. You can also use EXEC files to do the same thing and much more. To make an EXEC file that will take over the keyboard, you must set up special formats, using PRINT statements to issue the correct commands. For example, this little EXEC file will catalog your disk:

```
10 TEXT : HOME
20 D$ = CHR$ (4)
30 PRINT D$"OPEN CAT"
40 PRINT D$"WRITE CAT"
50 PRINT "CAT"
60 PRINT D$"CLOSE CAT"
```

Run the program to create a text file called CAT. Now type in

```
EXEC CAT
```

Your disk drive should spin and CATalog your disk. When it does, notice that the file CAT is there.

In some programs it's necessary to load different "supporting" files to get the program to work properly. Using an EXEC file is useful since, rather than having to type in all the LOADs, BLOADs, and EXECs yourself, you can have an EXEC file do it for you. For example, let's create an EXEC file to put ENTER.STRINGS and CENTER.STRINGS together for us, and LIST and RUN the combination as well.

```
10 TEXT : HOME
20 D$ =CHR$(4)
30 PRINT D$ "OPEN COMBINE"
35 PRINT D$ "WRITE COMBINE"
50 PRINT "LOAD ENTER.STRINGS"
60 PRINT "EXEC CENTER.STRINGS"
70 PRINT "LIST"
90 PRINT "RUN"
100 PRINT D$ "CLOSE COMBINE"
```

First, run the above program to create the file COMBINE. Make sure your disk has the Applesoft file ENTER.STRINGS and the text file CENTER.STRINGS on it. Then EXEC COMBINE. Watch what happens as the EXEC file takes control of your computer.

As a final demonstration of using EXEC files, we will reach back to Chapter 6 where you learned to locate the starting address and length of the most recently BLOADed or BRUN binary file. You had to have the starting address in order to BSAVE a binary file. Now, instead of having to remember all the steps in doing that, we will create an EXEC file that will do it for us. Here's how:

```
10 TEXT : HOME
20 D$ =CHR$(4)
30 PRINT D$ "OPEN BINARY.FINDER"
40 PRINT D$ "WRITE BINARY.FINDER"
50 PRINT "HOME"
60 PRINT "CALL -151"
70 PRINT "AA72.AA73 AA60.AA61"
80 PRINT "3D0G"
90 PRINT D$ "CLOSE BINARY.FINDER"
```

First, run the program to set up the text file BINARY.FINDER. Then BLOAD a binary program and EXEC BINARY.FINDER to see the beginning address and length of the program. (Remember to reverse the two-digit sets to get the correct addresses. See Chapter 6 if you've forgotten.) As you become more experienced, you will find BINARY.FINDER to be extremely useful.

Sequential Text Files

Of the two kinds of text files that we will discuss, sequential text files are simpler to work with and use less memory. Random access files, discussed in the next section, are a little trickier, but they can be accessed faster than sequential files. Using sequential and random access text files, you can enter data

from a program and store it as a text file. You can add to the file, change it, and retrieve data from it.

Creating Sequential Text Files

The first step is to write a formatting program, a program that will create a sequential text file. Essentially, the program will look almost identical to the programs that were written to create EXEC files. There's a good reason for this: EXEC files are a type of sequential text file. To create a file, use

OPEN
WRITE
PRINT ***(PORTION THAT GOES IN TEXT FILE)******
CLOSE

These commands take care of everything that's needed in a sequential text file.

Now, to begin using sequential text files, we will write a formatting program for our text file. Let's begin with a simple example that will store the names of a known number of people who sent us Christmas cards:

```
10 TEXT : HOME
20 REM **********************
30 REM SEQUENTIAL FILE WRITER
40 REM **********************
50 D$ = CHR$ (4)
60 VTAB 10: HTAB 1: INPUT "How many entries? ";N%
70 DIM NA$(N%)
80 VTAB 10: HTAB 1: CALL - 868
90 FOR X = 1 TO N%
100 VTAB 10: HTAB 1: CALL - 868
110 PRINT "Name #";X;"=>";
120 INPUT NA$(X)
130 NEXT X
200 REM ***************
210 REM CREATE SEQ FILE
220 REM ***************
230 PRINT D$"OPEN XMAS.CARDS"
240 PRINT D$"WRITE XMAS.CARDS"
250 FOR X = 1 TO N%
260 PRINT NA$(X)
270 NEXT X
280 PRINT D$"CLOSE XMAS.CARDS"
290 END
```

After you've entered the program, run it. Remember the number of names that you have entered. Save the program

under the name CC.WRITER, for you will return to it later. CATALOG your disk and make sure there is a file called XMAS.CARDS that was created by the CC.WRITER program.

The next step is to READ the files using

OPEN
READ
INPUT ** TAKES DATA OUT OF FILE **
CLOSE

The following program will OPEN XMAS.CARDS, READ the file, CLOSE the file, and then PRINT out the contents to the screen. Notice the similarities and differences between it and the program for writing files.

```
10 TEXT : HOME
20 REM **********************
30 REM SEQUENTIAL FILE READER
40 REM **********************
50 D$ = CHR$ (4)
60 VTAB 10: HTAB 1: INPUT "How many entries? ";N%
70 DIM NA$(N%)
80 PRINT D$"OPEN XMAS.CARDS"
90 PRINT D$"READ XMAS.CARDS"
100 FOR X = 1 TO N%
110 INPUT NA$(X)
120 NEXT X
130 PRINT D$"CLOSE XMAS.CARDS"
140 FOR X = 1 TO N%
150 PRINT NA$(X)
160 NEXT X
```

After you've run the program, save it under the filename CC.READER. If you forgot the number of names you put in the XMAS.CARDS file, you might get an OUT OF DATA ERROR. By entering too many names to be read, you went beyond the limits of the file. Later, you will learn a way to make sure this doesn't happen.

So far, so good. You have a program that will WRITE a list of names in a data file and one that will READ those names back. What happens, though, if you want to add some names to your file? You could make a new file under another name, but a better way is to APPEND your current XMAS.CARDS file. Using the APPEND command, write your additional files to the bottom of the data list you have in your existing file. It is important to remember, when using APPEND, that there is an existing file to which you can APPEND your data. To do that, use the following format:

APPEND
WRITE
PRINT **ENTER APPENDED DATA **
CLOSE

Since you will need a program only slightly different from CC.WRITER, simply LOAD CC.WRITER, make some changes, and save the program under the name CC.APPENDER.

Look at the following program carefully, and note the changes made from CC.WRITER.

```
10 TEXT : HOME
20 REM **********************
30 REM SEQUENTIAL FILE APPEND
40 REM **********************
50 D$ = CHR$ (4)
60 VTAB 10: HTAB 1: INPUT "How many entries? ";N%
70 DIM NA$(N%)
80 VTAB 10: HTAB 1: CALL - 868
90 FOR X = 1 TO N%
100 VTAB 10: HTAB 1: CALL - 868
110 PRINT "Name #";X;"=>";
120 INPUT NA$(X)
130 NEXT X
200 REM ***************
210 REM APPEND SEQ FILE
220 REM ***************
230 PRINT D$"APPEND XMAS.CARDS"
240 PRINT D$"WRITE XMAS.CARDS"
250 FOR X = 1 TO N%
260 PRINT NA$(X)
270 NEXT X
280 PRINT D$"CLOSE XMAS.CARDS"
290 END
```

That doesn't take much. All you have to do is change WRITE to APPEND. Once you've saved the program as CC.APPENDER, you'll have programs that will WRITE, READ, and APPEND sequential text files.

Finally, you may want to examine only one entry or record in your sequential text file. To do this, use the POSITION command, which will allow you to go to a certain record and examine it. Since POSITION recognizes only numbers, you will have to inspect any entry by its number. The general format for POSITION is

POSITION FILE.NAME,F#

Note in line 60, below, the format for POSITION in a program that uses a variable for the position number. The ,F must follow the POSITION command, which is then followed by the position number (here, the variable N) of the record to be examined. This program will let you examine any of the entries you have in your XMAS.CARDS text file:

```
10 TEXT : HOME
20 D$ = CHR$ (4)
30 VTAB 10: HTAB 1
40 INPUT "WHICH ENTRY #=> ";N
50 PRINT D$"OPEN XMAS.CARDS"
60 PRINT D$"POSITION XMAS.CARDS,F";N
70 PRINT D$"READ XMAS.CARDS"
80 INPUT NA$
90 PRINT D$"CLOSE XMAS.CARDS"
100 PRINT : PRINT NA$
```

The first position is position 0, not 1. Thus, if there are five records, they will occupy positions 0–4.

File Manager

Now you've seen how to WRITE, APPEND, READ, and POSITION elements of a single text file. However, since text filenames are essentially nothing but strings, you could use variables to do much of the work automatically. Remember, if you can write a program that will do most of the work for you, you can save a lot of time that would have been spent writing several little programs. The following program, FILE.MANAGER, will create, append, read, and position any file you want. It handles only a single string element, but you can change that if you want. Its main purpose is to provide an example of a program that deals with all the basic aspects of sequential text file handling in one place.

```
10 TEXT : HOME
20 REM ************
30 REM FILE MANAGER
40 REM ************
50 D$ = CHR$ (4)
60 L = 200: DIM NA$(L)
80 GOTO 1000
100 REM ******************
110 REM CREATE/APPEND FILE
120 REM ******************
130 HOME : VTAB 10: INPUT "HOW MANY NAMES TO ENTER
    =>";N%
```



```
140 HOME : PRINT : PRINT
150 FOR X = 1 TO N%
160 INPUT "ENTER NAME=> ";NA$(X)
170 NEXT X
180 HOME : VTAB 10: INPUT "NAME OF FILE ";CF$
190 PRINT D$TASK$CF$
200 PRINT D$"WRITE"CF$
210 FOR X = 1 TO N%
220 PRINT NA$(X)
230 NEXT X
240 PRINT D$"CLOSE"CF$
250 GOTO 1000
300 REM *********
310 REM READ FILE
320 REM *********
330 ONERR GOTO 400
340 HOME : VTAB 10: INPUT "FILE TO READ => ";RF$
350 PRINT D$"OPEN"RF$
360 PRINT D$"READ"RF$
370 INPUT NA$: PRINT NA$
380 GOTO 370
390 PRINT D$"CLOSE"RF$
400 IF ER = 0 THEN ER = 1: GOTO 390
410 GOTO 610
500 REM *************
510 REM POSITION FILE
520 REM *************
530 HOME : VTAB 10: INPUT "NAME OF FILE=> ";PF$
540 VTAB 12: INPUT "POSITION # ";X%
550 PRINT D$"OPEN"PF$
560 PRINT D$"POSITION"PF$",F"X%
570 PRINT D$"READ"PF$
580 INPUT NA$
590 PRINT D$"CLOSE"PF$
600 PRINT NA$
610 VTAB 24: INVERSE
620 PRINT "HIT ANY KEY TO CONTINUE";: NORMAL
630 GET A$: PRINT A$
1000 REM ****
1010 REM MENU
1020 REM ****
1030 HOME :FM$ = " FILE MANAGER "
1040 INVERSE : HTAB 20 - LEN (FM$) / 2
1050 PRINT FM$: NORMAL : RESTORE
1060 FOR X = 1 TO 5: READ M$(X): NEXT
1070 VTAB 6
1080 FOR X = 1 TO 5: PRINT X;".";M$(X)
1090 PRINT : NEXT X
```

```
1100 PRINT : PRINT : INVERSE : PRINT " CHOOSE BY
     NUMBER ";
1110 NORMAL : GET A%: IF A% < 1 OR A% > 5 THEN PRINT
     CHR$ (7): GOTO 1000
1120 IF A% = 1 OR A% = 2 THEN GOSUB 1500
1130 ON A% GOTO 100,100,300,500,1600
1500 REM ****************
1510 REM CREATE OR APPEND
1520 REM ****************
1525 HOME : VTAB 10
1530 PRINT "(C)reate or (A)ppend ";: GET AN$: PRINT AN$
1540 IF AN$ < > "C" AND AN$ < > "A" THEN 1525
1550 IF AN$ = "C" THEN TASK$ = "OPEN"
1560 IF AN$ = "A" THEN TASK$ = "APPEND"
1570 RETURN
1600 TEXT : HOME : END
2000 DATA CREATE FILE,APPEND FILE,READ FILE,POSITION
     FILE,EXIT
```

Before moving on to random access files, you should be reminded that there are several more aspects to sequential data files, and you can do much more with them than the simple applications covered in this section. There are several good books that go into detail on text file applications. If you want to go beyond the elementary stage in this very important aspect of the Apple IIGS computer, it is strongly recommended that you look at one of these other sources.

Random Access Files

Random access files are like containers of equal size into which you store data. If you're familiar with the standardized boxes that are used as shipping containers, you have some idea of how random access files work. Basically, you must first decide how big a container you will need, based on the maximum size of the material you'll be entering. Since all you can put into a random access file is either numbers or strings, the problem is greatly simplified. Each character in a number or string takes one byte. Therefore, if your maximum for a given string is 10, it will be necessary to allocate a total of ten bytes: one for each of the ten characters. (As you know, a byte is a unit of measurement in the IIGS's memory.) With numbers, everything—including the decimal point—counts. For example, 99.95 takes five bytes: one for each of the four numbers and one for the decimal point. Even carriage returns count. Remember CHR$(13) in our discussion of ASCII?

For the most part, the activities involved in creating and reading random access files look very similar to those for sequential text files, but there are important differences as well. First, when you OPEN a random access file, you must include the (L)ength of the file. Then, as was done with sequential text files, PRINT D$—composed of CHR$(4)—and then place the OPEN command and the name of the file inside quotation marks. However, then it is necessary to put a comma and an *L* along with the maximum length of the file. The following example shows the format for OPENing a random access file:

```
PRINT D$ "OPEN NAME.FILE, L40"
```

The next step in creating a random access file is to enter the WRITE command, but you must include the (R)ecord number with it, such as

```
PRINT D$ "WRITE NAME.FILE, R1"
```

Using Random Access Files

To illustrate the use of random access files, we will create a series of programs that store the names, cities, and states of people. We will call the file we create HOMETOWN, using three strings:

NA$ **Person's name**
CT$ **City**
SC$ **State's mailing code**

Since people and cities have names of different lengths, you must decide on a maximum size. Any name longer than the maximum will simply be truncated. This process is extremely important in working with random acess files since you are limited to the number of bytes specfied when you OPEN a file. If your entries go over the length, they will spill over into the next record. Therefore, limit the length of a person's name to 20, a city to 10, and a state's mailing code to the 2-character abbreviation employed by the post office. If a string is longer than the specified length, convert the string by using LEFT$. If that is not done, the number of characters may be too large, causing the string to flow over into the next record. Now calculate the (L)ength to OPEN the file:

NA$ **= 20**
CT$ **= 10**
SC$ **= 2**
TOTAL **= 32**

Using these values, we will write a program that will enter a single record into a random access file:

```
10 TEXT : HOME
20 REM ******************
30 REM RANDOM ACCESS FILE
40 REM ******************
50 D$ = CHR$ (4)
60 VTAB 4: HTAB 1
70 INPUT "NAME => ";NA$
80 IF LEN (NA$) > 20 THEN NA$ = LEFT$ (NA$,20)
90 REM NAME USES 20 BYTES
100 REM IF LONGER THAN 20
110 REM IT IS TRUNCATED
120 PRINT : INPUT "CITY => ";CT$
130 IF LEN (CT$) > 10 THEN CT$ = LEFT$ (CT$,10)
140 REM CITY USES 10 BYTES
150 PRINT : INPUT "STATE CODE => ";SC$
160 IF LEN (SC$) < > 2 THEN VTAB 7: GOTO 150
170 FOR X = 1 TO 27: PRINT "-";: NEXT X
180 PRINT : PRINT "TOTAL BYTES = 32"
190 PRINT : PRINT : INVERSE : PRINT "HIT ANY KEY TO
    CONTINUE ";: NORMAL : WAIT - 16384,128: POKE -
    16368,0
200 REM ********************
210 REM CREATE SINGLE RECORD
220 REM ********************
230 PRINT D$"OPEN HOMETOWN,L32"
240 PRINT D$"WRITE HOMETOWN,R1"
250 PRINT NA$
260 PRINT CT$
270 PRINT SC$
280 PRINT D$"CLOSE HOMETOWN"
```

That was a lot of work to enter a single record, but be patient. We will do more. Now, we will READ a record from a random access file. As in WRITEing random access files, we must OPEN them with a specified (L)ength and READ them in terms of a specified (R)ecord. The following program will read the HOMETOWN file:

```
10 TEXT : HOME
20 REM ******************
30 REM READ SINGLE RECORD
40 REM ******************
50 D$ = CHR$ (4)
60 PRINT D$"OPEN HOMETOWN,L32"
70 PRINT D$"READ HOMETOWN,R1"
80 INPUT NA$
```



```
90 INPUT CT$
100 INPUT SC$
110 PRINT D$"CLOSE HOMETOWN"
120 HOME : VTAB 7
130 PRINT NA$
140 PRINT CT$;", ";SC$
```

Again, that was a lot of work just to READ a single record. However, you can see how random access files operate. Now, we will examine how to deal with multiple records. We will stick with our HOMETOWN example, though, since dealing with multiple random access records is a bit more involved than dealing with sequential text files. We will even begin the program with a reminder of the number of bytes we are using and the strings.

Multiple Records

In the program that follows, lines 10–150 are very much like the original program used to create a file, but notice the "counter" variable RN in line 70. The counter will be used to keep track of the number of records in the file HOMETOWN. By storing the value of the counter (renamed RZ in line 200, once you have entered all the records) in R0 (record 0), you will have a way of knowing how many records there are in the file. Then, when you READ the files, you will first read R0 and then make a FOR-NEXT loop with the value of RZ as the maximum number of records to read. Also notice that D$ has been PRINTed in line 170. This is to leave the WRITE mode to ask whether the user wants to enter another record. An empty PRINT CHR$(4) will allow you to leave a WRITE or READ in ProDOS.

```
10 TEXT : HOME
20 REM ***********************
30 REM MULTIPLE RANDOM RECORDS
40 REM ***********************
50 D$ = CHR$ (4)
60 PRINT D$"OPEN HOMETOWN,L32"
70 RN = 0
80 RN = RN + 1: VTAB 4: HTAB 1
90 INPUT "NAME => ";NA$
100 IF LEN (NA$) > 20 THEN NA$ = LEFT$ (NA$,20)
110 PRINT : INPUT "CITY => ";CT$
120 IF LEN (CT$) > 10 THEN CT$ = LEFT$ (CT$,10)
130 PRINT : INPUT "STATE CODE ";SC$
140 IF LEN (SC$) < > 2 THEN VTAB 7: GOTO 130
150 PRINT D$"WRITE HOMETOWN,R";RN
```

```
160 PRINT NA$: PRINT CT$: PRINT SC$
170 PRINT D$
180 INVERSE : PRINT " ANOTHER RECORD (Y/N) ";: NORMAL
190 GET AN$: PRINT AN$: IF AN$ = "Y" THEN 80
200 RZ = RN
210 PRINT D$"WRITE HOMETOWN,R0"
220 PRINT RZ
230 PRINT D$"CLOSE HOMETOWN"
```

Now that you have several records in your file, you will need to get them out again. Here's where the counter variable RZ, stored in R0, comes in handy. First, read R0 to see how many records there are, and then loop through the records to READ them all. Notice that in line 70 we first read HOMETOWN,R0. After it's INPUT into memory, it is used in the FOR-NEXT loop in line 100 to pull all the records out.

```
10 TEXT : HOME
20 REM ************************
30 REM MULTIPLE RECORD READER
40 REM **********************
50 D$ = CHR$ (4)
60 PRINT D$"OPEN HOMETOWN,L32"
70 PRINT D$"READ HOMETOWN,R0"
80 INPUT RZ
90 PRINT D$
100 FOR G = 1 TO RZ
110 PRINT D$"READ HOMETOWN,R"G
120 INPUT NA$
130 INPUT CT$
140 INPUT SC$
150 PRINT D$
160 PRINT NA$
170 PRINT CT$;",";SC$
180 PRINT : NEXT
190 PRINT D$"CLOSE HOMETOWN"
```

Now that you have seen how to build a random access file, here's a little problem to work out. By adding a few lines and calculating a few more bytes, you can create a very useful address list program. You already have a program that enters names, cities, and states. All you have to add are address and zip code, and there you have it. By attaching a subroutine to kick it out to your printer (which will be discussed in Chapter 9), you can generate your own mailing list program.

Summary

This chapter began by examining ways in which you can capture a program in a text file, and it ended by creating individual records in random access files. Most of the material covered here has been rudimentary. Several tricks and techniques are needed before you can fully use text files. Nevertheless, you've made a start. By refining and extending what you have learned, you can vastly extend your knowledge.

First, you saw how to "catch" a program or subroutine in a text file that can be brought into memory with EXEC. This allows you to load several programs simultaneously or to attach useful subroutines onto programs. Then, using CHAIN, you learned how to have one program use another program without losing variable values that were entered in the first program. Also, you found that you can create ROBOT EXEC files that take over the keyboard. These are handy when a series of operations are needed to execute a program.

Second, you learned how to OPEN, WRITE, APPEND, POSITION, and CLOSE sequential text files to store data. These files are very useful for storing data in the form of numbers or strings to be accessed for later use by programs that READ data from text files.

Finally, you began using random access files. These files, while taking up more memory and being trickier to deal with than sequential text files, have advantages in their speed of operation and uses for multiple record handling that make them practical for certain applications. With more advanced understanding of these files, you can create just about any database program.

9

You and Your Printer

You and Your Printer

Until recently, when an Apple owner bought a printer, he or she was lucky to find a manual that would give the slightest hint on how to use it. Some printer manufacturers would decide upon a certain brand of computer other than Apple, and explain the use of their printer with that brand. Other manuals were written for people with high-level understanding of printer operations, and simply provided the hardware specifications to be applied to any computer. Needless to say, there were many frustrated Apple owners, and printers were used only to their minimum capacity.

Things have improved somewhat, and Apple's ImageWriter manual is a clear exception to the general rule of fuzzy explanations. Furthermore, good printer manuals are available now, either supplied free with the printer or from independent sources. However, just in case your manual does not tell you how to work it specifically with the Apple IIGS, there are some things you should know.

We will discuss printers both generally and in terms of specific popular printers. The general discussion will cover how your IIGS communicates with a printer. This will give you a chance to decipher some of the manuals that came with your printer, allowing you to enter the correct code in BASIC that will get your printer to do your bidding. Once you have the idea of how printers work, you'll find them really quite simple to operate. But until you learn a few secrets, you'll generally find them difficult to use to their full capacity.

Why bad printer manuals? So many beginners are frustrated with printer manuals that provide only vague hints as to their operation. Here's an explanation of why the manuals are, in general, so bad. Only ten years ago, computers in the home were a rarity. Printers developed for businesses came with technical manuals designed for the company technicians who set up the printers. Likewise, many printers were simply hooked up to a dedicated word processor that had all the code for printer operation already installed in the program. Furthermore, the small, inexpensive printers that came on the market

at the beginning of the 1980s, and were intended for home or small-business use, were designed to be used with any number of different computers. After all, if you were selling a printer, you would not want to market it for a single brand of computer. Rather than producing separate manuals for different computers, the companies wrote manuals for the technicians who could get them to work on any computer. Currently, some printer companies are providing manuals that explain their use on several different computers, including the Apple, but others do not.

Printing Text

In Chapter 1 is a simple example of how to print something out to your printer. As you remember, by entering PR#1 and a PRINT statement, you are able to print text to your printer. To return to your screen, you simply enter PR#0. This will "disconnect" your printer. However, your printer will not always be in slot 1, and when you're writing programs for the IIGS, there should be a choice of which slot the printer will be connected to. (PR#0 will always disconnect your printer, no matter what slot it is in—except slot 0, where it should *never* be.) The following little program allows you to enter text and print it to the printer in any slot used by the printer or to the screen (slot 0):

```
10 TEXT : HOME : D$ = CHR$(4)
20 VTAB 10 : INPUT "ENTER STRING -> "; MS$
30 GOSUB 1000
40 PRINT MS$
50 PRINT D$; "PR#0"
60 VTAB 22 : INVERSE : PRINT " ANOTHER ENTRY? (Y/N) "; :
   NORMAL : GET AN$ : PRINT AN$
70 IF AN$ = "N" OR AN$ = "n" THEN END
80 GOTO 10
1000 REM ******************
1010 REM PRINTER SLOT ROUTINE
1020 REM ******************
1030 HOME : VTAB 10 : PRINT "PRINTER SLOT # (0 TO PRINT
     TO SCREEN ): "; : GET S% : PRINT S%
1410 IF S% > 7 THEN PRINT CHR$(7) : GOTO 1000
1050 PRINT D$; "PR#";S%
1060 RETURN
```

The printer slot subroutine in lines 1000–1060 is handy to have. You might want to save it as a text file and EXEC it into programs that require the printer.

Program-Listing Utility

Listing programs to a printer is a good way to debug a program or to send it to a friend by mail. It would be convenient to have a utility program that enables you to do just that. It's a nuisance to enter the six-liner above, so let's make an EXEC file that can handle the chore automatically. To create the EXEC file, type in the program below and then enter RUN 10. The program will create a new file named PRG.LISTER on the disk. If you run the program from line 0 (the beginning), the EXEC file will not be created. (If you save the EXEC file generator program, *do not* use the name PRG.LISTER.)

```
0 HOME : INPUT "PRINTER SLOT #=>";PR%
1 HOME : VTAB 10: INPUT "PROGRAM NAME=> ";PG$
2 D$ = CHR$ (4)
3 PRINT D$"PR#"PR%: PRINT PG$: FOR X = 1 TO LEN (PG$):
  PRINT "*";: NEXT : PRINT
4 LIST 6,
5 PRINT D$"PR#0": END
10 REM **************
20 REM PROGRAM LISTER
30 REM **************
40 D$ = CHR$ (4)
50 PRINT D$"OPEN PRG.LISTER"
60 PRINT D$"WRITE PRG.LISTER"
70 LIST 0,4
80 PRINT "RUN"
90 PRINT D$"CLOSE PRG.LISTER"
```

To test the EXEC file, load one of your programs, make sure your printer is turned on and is online, and enter EXEC PRG.LISTER. Enter the slot number of your printer and then the program's name. Your printer should chug out a listing of your program with a neat-looking title.

Special Printer Characteristics

We mentioned in Chapter 1 that there are several kinds of printers. Our focus here will be on dot-matrix printers, since they are the most popular. However, much of the information also applies as well to daisywheel printers. Finally, we will briefly discuss using a language called PostScript to control most laser printers, such as Apple's LaserWriter.

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,

Table 9-1 is a partial list of codes provided with an Epson printer:

Table 9-1. Epson Printer Codes

Code	Action
8	Back space
10	Line feed
12	Form feed
13	Carriage return
14	Double width
15	Condensed
18	Turn off condensed
20	Turn off double width
27	Escape key used in conjunction with the following:
	E Emphasized printing
	F Turn off emphasized
	G Double-strike printing
	H Turn off double-strike printing
	K Normal-density printing
	L Dual-density printing
	Q Set column width

For most first-time computer owners, that list of codes could have been written by a visitor from another planet for all the good it does. However, there is important information there, and once you get to know how to use the codes, you'll find that they are relatively easy to understand.

To tell your printer you want emphasized print, for example, you use CHR$(27) + "E". To kick that into your printer, you do the following:

```
PR#1 (or whatever slot your printer is in)
PRINT CHR$(27) + "E" + "MESSAGE"
```

If you have an Epson printer, the word MESSAGE is printed in an emphasized typeface. Emphasized printing will remain in effect until you send the right sequence of code or turn off your printer. We'd better turn it off with

```
PRINT CHR$(27) +"F"
```

assuming you haven't done anything since your message was printed in the emphasized face.

For condensed printing you use CHR$(15) without the escape code tacked on the front and CHR$(18) to turn it off.

Once you get the decimal code, all you have to do is to enter that code to the printer. It will do anything from changing the typeface to performing a backspace function.

ImageWriter Codes

Let's look at another printer, the ImageWriter, and some of its control codes. However, rather than keying in all of the codes, we will write a program that will give us different typefaces, an underlining feature, and some other tricks. The first thing we will do is to define string variables as printer codes and then enter messages we want to go to the printer.

```
10 TEXT : HOME
20 ESC$ = CHR$ (27): REM ESC
30 EXT$ = CHR$ (110): REM n
40 PICA$ = CHR$ (78): REM N
50 ELITE$ = CHR$ (69): REM E
60 SQUISH$ = CHR$ (113): REM q
70 UN$ = CHR$ (88): REM X
80 UU$ = CHR$ (89): REM Y
90 BOLD$ = CHR$ (33): REM !
100 NOBOLD$ = CHR$ (34): REM "
110 RESET$ = CHR$ (99): REM c
120 D$ = CHR$ (4)
200 REM *************
210 REM TYPEFACE MENU
220 REM *************
230 FOR X = 1 TO 9
240 READ M$
250 PRINT X;".";M$
260 PRINT : NEXT X
270 PRINT : RESTORE
280 PRINT "Choose by number:";
290 GET N
300 ON N GOSUB 1100,1200,1300,1400,1500,1600,1700,
    1800,1900
310 DATA Extended,Pica,Elite,Condensed,Underline
320 DATA Stop Underline,Bold,Stop Bold,Reset All
330 HOME
340 INPUT "Printer slot number ";SN%
350 PRINT D$"PR#"SN%
360 PRINT PT$ + "This is your typeface Bucko!";
370 PRINT
380 PRINT D$"PR#0"
390 PRINT "Another go Sport? (Y/N) ";
400 GET AN$
410 IF AN$ = "Y" OR AN$ = "y" THEN 10
```

The Zero Character

There is an extremely important switch on many models of dot-matrix printers. It is the switch that prints out the zero character with a slash through it. When you use your printer to make program listings, you'll often find it essential to be able to easily differentiate between zeros and letter *O*'s. For example, if you have a variable O, and you want to have a FOR-NEXT LOOP from the variable O to 255, you would enter

FOR X = O TO 255

Most people reading that line would assume that the *O* was a zero and would enter the wrong character. By flipping the appropriate switch on your printer, you can be sure that all zeros are differentiated from the letter *O*. On Epson MX-100 printers, for example, when switch SW1-7 is in the on position, the zeros have slashes through them, and when it is off, the zeros have no slashes.

Don't use *O* as a variable name. Since there are enough problems in getting programs correctly copied from printed listings in the first place, there is no reason to make it more difficult by using the *O* character as a variable anyway. Even if your zeros have slashes through them, if you have an *O* for a variable, when it comes time to PRINT O, chances are, you will enter "PRINT 0" (slash and all). Use other characters—you have plenty available.

Multiple Address Labels

Before going on to examine printer graphics, look at the following program, designed to give you single or multiple address labels. Use it to experiment with your printer's special characteristics. See if you can rewrite portions of it to get interesting effects on your printout.

```
10 TEXT : HOME
20 REM ********************
30 REM MULTIPLE LABEL MAKER
40 REM ********************
50 D$ = CHR$ (4)
60 RESTORE
70 FOR X = 1 TO 5
80 READ LB$
90 VTAB X + 7: PRINT LB$
100 NEXT X
110 DATA Name,Address,City,State,Zip
120 FOR X = 1 TO 5
```



```
130 HTAB 11: VTAB X + 7
140 INVERSE : PRINT SPC(25)
150 NEXT : NORMAL
160 HTAB 11: VTAB 8: INPUT " ";NA$
170 HTAB 11: INPUT " ";AD$
180 HTAB 11: INPUT " ";CT$
190 HTAB 11: INPUT " ";SA$
200 HTAB 11: INPUT " ";ZIP$
300 REM ***********
310 REM Check Label
320 REM ***********
330 VTAB 22: PRINT "Is this correct (Y/N)? ";
340 GET A$
350 IF A$ = "N" OR A$ = "n" THEN 10
360 PRINT
370 INPUT "How many labels would you like ";N%
400 REM ****************
410 REM Print the Labels
420 REM ****************
430 PRINT D$"PR#1"
440 FOR P = 1 TO N%
450 PRINT : REM Adjust to label size
460 PRINT NA$
470 PRINT AD$
480 PRINT CT$;", ";SA$; SPC(1);ZIP$
490 PRINT : REM Adjust to label size
500 NEXT P
510 PRINT D$"PR#0"
600 HOME
610 INPUT "Would you like more labels printed? ";AN$
620 IF AN$ = "Y" OR AN$ = "y" THEN 10
630 VTAB 10: PRINT "See you when you need more labels."
640 END
```

Look at this next program for sending information to your printer. Use it to make disk labels, and note the format used.

```
10 POKE 49167,0
20 TEXT : HOME
30 L$ = " Label Maker ":S = LEN (L$)
35 S = 20 - S / 2
40 FOR X = 1 TO LEN (L$)
50 K$ = MID$ (L$,X,1)
60 K = ASC (K$)
65 IF K = 32 THEN POKE 1024 + S + X,K: NEXT
67 IF X = > S THEN 100
70 IF K < 97 THEN POKE 1024 + S + X,K - 64
80 IF K = > 97 THEN POKE 1024 + S + X,K
90 NEXT
```

```
100 VTAB 5: INPUT "Message: ";M$
110 INPUT "How many labels";N%
120 D$ = CHR$ (4)
130 INPUT "Printer slot #";PN%
140 PRINT D$"PR#"PN%
150 FOR X = 1 TO N%
160 PRINT M$
170 NEXT
180 PRINT D$"PR#0"
```

Printer Graphics

It's important to understand that not all printers do graphics, and while daisywheel printers can print graphics, they are very slow. Furthermore, printing graphics on most printers requires special programs (or hardware) beyond the programming skill of novices. Therefore, we will discuss programs and interfaces that you can purchase to get your printer to do all kinds of tricks with graphics. Some printers come with the software/hardware so that it is unnecessary to buy additional or special devices. Likewise, some "graphics dump" programs operate only with certain printers and not with others. We'll try to use illustrations of graphics programs that work on most popular printers and provide some insight into what you can do with them.

Dumping Your High-Resolution Screen

When you print out graphics, your computer sends a series of bytes to your printer reflecting the pixels on your hi-res screen. Since most printers print graphics only in black and white (black on white paper), the colors may seem not to matter. Different colors, however, give different textures on the printer.

By using different colors, you can get interesting effects and shading on your printer, even though all your printer is doing is printing dots in black ink. Unfortunately, you cannot do this with the same ease with which you can print out text.

One graphics printing program that works well on every printer tested, including daisywheel printers, is *The Printographer* (Roger Wagner Publishing). For ease of use, cropping, different magnification, and horizontal and vertical printing, it is excellent. *Triple-Dump* (Beagle Bros.) is another outstanding program for sending graphics to your printer. There are others as well, and I suggest that you examine any one you intend to purchase first and consider the following points:

1. Will it work on my printer?
2. Will it crop pictures easily?
3. Is it easy to use?
4. Will it magnify graphics?
5. Will it place graphics in different horizontal locations on the printout?
6. Will it condense graphic files on the disk?
7. Will it access either drive?
8. Will it print both inverse and normal?
9. Can it be used within a program?

You might also consider whether it will work with color printers since some really good ones are now available at reasonable prices.

Laser Printing: Using PostScript

The Apple LaserWriter printer has a computer inside it that understands a language called PostScript. PostScript is a special "page description" language that reads instruction sets just as BASIC reads the instructions you give it when you write a program. The best thing about the LaserWriter, and other laser printers with PostScript interpreters, is that you can write the programs on your Apple IIGS and send them to the laser printer, using all of the laser printer's high-quality capabilities.

To accomplish this, all you do is to write the programs and save them as sequential text files. Then, you send the text files to the LaserWriter, and it prints them out. Most people probably cannot afford a laser printer, but for a small charge many copy shops make laser printers available.

The following quick tutorial will show how to print laser text and provide a program for writing PostScript programs, automatically setting up the desired laser font and font size. For a full description of how to program in PostScript, see *PostScript: Language Tutorial and Cookbook* (Adobe Systems, Addison-Wesley, 1985).

Like BASIC, PostScript has strings and the equivalent of a PRINT statement (the PRINT statement is *show* instead of PRINT). To create a string, just put the message you want printed inside parentheses. Finally, you have to tell the printer where on the page you want the message to go. There are 72 printing dots per inch on the LaserWriter, and the first printing position is in the lower left corner of the page. The horizontal axis (x-axis) increases to the right and decreases to the left, and the vertical axis (y-axis) increases as you go up and decreases

as you go down—exactly the way Cartesian coordinates work. Since the starting point is 0,0, the top right corner of an 8½-by-11-inch page then would be X = 612 (72 * 8.5), Y=792 (72 * 11), or coordinates 612,792.

To get to a given spot, use the PostScript statement *moveto*. For example, suppose you want to print your name and address in the upper left corner of a page for a letterhead, with one-inch margins from the top and left side. This program will do that:

```
%==Letterheader==
/Times-Roman findfont 12 scalefont setfont
72 720 moveto
(Your Name) show
72 707 moveto
(Your Address) show
72 694 moveto
(Your City, State and Zip Code) show
72 681 moveto
(Your phone number) show
showpage
```

The first line is like a REM statement in BASIC. Any line with the percent sign (%) as the first character is ignored by the PostScript interpreter, just as any lines in BASIC ignore everything after REM. The second line chooses the font and font size—in this case, Times Roman in 12-point font. Also notice that all of the statements are in lowercase letters. This is not an option. The moveto statement takes the format

x y **moveto**

expecting to be preceded by two numbers—separated by a space—that specify the horizontal and vertical positions for the next printing position on the page. Finally, the statement *showpage* issues a command to the printer to actually print the page.

There's a great deal more in the PostScript language, including graphic statements. You might want to look at *PostScript: Language Tutorial and Cookbook,* and use the following program to try your hand at creating laser pages and graphics:

```
10 TEXT : HOME : RESTORE
20 D$ = CHR$ (4)
30 REM ************************
40 REM SIMPLE POSTSCRIPT WRITER
50 REM ************************
60 DIM PS$(254)
```



```
70 FOR X = 1 TO 4
80 READ F$(X)
90 PRINT X;". ";F$(X): PRINT
100 NEXT
110 INVERSE : PRINT " Choose font by number ";
120 GET A: NORMAL
130 DATA Times,Helvetica,Courier,Symbol
140 FT$ = F$(A)
150 PRINT : PRINT : INPUT "Font point size ";FZ$
200 REM ********************
210 REM Enter PostScript Program
220 REM ************************
230 HOME
240 PRINT "When you are ready, begin typing in your
    PostScript"
250 PRINT "program. Keep the lines short, and press return
   at the
260 PRINT "end of each line. Press 'Q-return' to quit."
270 INVERSE : PRINT : PRINT SPC( 79);
280 NORMAL
290 PRINT : PRINT
300 INPUT "=> ";PS$(V)
310 IF PS$(V) = "Q" OR PS$(V) = "q" THEN 500
320 V = V + 1: IF V > 250 THEN GOSUB 400
330 GOTO 300
400 REM *******
410 REM Warning
420 REM *******
430 PRINT CHR$ (7)
440 INVERSE
450 PRINT "You are about to run out of array space: finish
    up now."
460 NORMAL : RETURN
500 IF FT$ = "Times" THEN FT$ = "Times-Roman"
510 FT$ = "/" + FT$
520 FS$ = FT$ + " findfont " + FZ$ + " scalefont setfont"
530 HOME
540 INPUT "Program name=> ";P$
550 PRINT D$"OPEN"P$
560 PRINT D$"WRITE"P$
570 PRINT FS$
580 FOR X = 0 TO V - 1
590 PRINT PS$(X)
600 NEXT
610 PRINT D$"CLOSE"P$
```

By the way, if you have a 1200-baud modem, and there's one attached to a friend's laser printer, you can send your files over the modem and have them printed at a remote location.

Printing Fonts

One of the more useful (and just plain fun) aspects of printing graphics on dot-matrix printers is the special graphics fonts. They are not in the same league as laser fonts, but they give you a big variety of typefaces to use for everything from posters to Christmas cards.

To use these fonts, first load a font, and then with PRINT statements from within a program, print out various messages. *Apple Mechanic* (Beagle Bros.) has an "Xtyper" program that works something like a little word processor for hi-res fonts. Also, hi-res fonts are useful for labeling hi-res graphics.

Printing Photographs

A final use of hi-res printing is in computer photographs. However, instead of using photographic paper and chemicals to reproduce your pictures, you use your printer dump program and regular paper. Using a video camera and digitizer, you can put high-resolution graphic photographs on your computer screen and save them to disk. They can be printed on regular printer paper just like any other high-resolution graphic.

Summary

In this chapter, we have covered many aspects of working with your printer. We've touched upon only a few of the printers that work with the Apple IIGS; and some printers are far more flexible than others. However, you should now have a good idea of how your computer sends messages to your printer to get it to print with different typefaces, as well as to access other special features available on your particular machine and interface card.

Since printer manuals are often too technical for beginning users, it is important to know how to translate them so that you can use all their special features. Basically, all that's required is a chart in the manual that provides the decimal codes necessary for turning on the various printer devices—from typefaces to linefeed. If you can find these codes in your manual, it will be a relatively simple matter to encase them in CHR$ functions to communicate your desires to the printer. Sometimes this will involve using hexadecimal code, but that

too is simply a matter of knowing what the code is for a particular feature.

For printing graphics, you will need to purchase a special program, hardware interface, or learn a higher level of programming than is covered in this book. However, with many software and hardware packages available for the Apple IIGS and printer graphics, it is easiest to begin with a commercial graphics dump routine. With a printer capable of doing graphics, you can print out anything on your hi-res screen. You can even print near-typeset-quality text and graphics with PostScript programs on a LaserWriter. The versatility of your Apple's printing ability is outstanding, and it will serve you for years to come.

10

Super High-Res Graphics and Sound

Super High-Res Graphics and Sound

Your IIGS has made a quantum leap over the other Apple IIs when it comes to graphics and sound. The reason is that it can place pixels on a 640 × 200–dot matrix for four-color programs to gain the highest resolution on an Apple II yet. And it has an Ensoniq sound chip for creating unsurpassed digitized sound. There are some excellent programs available for accessing these new features, but you cannot easily work with them directly from Applesoft BASIC. Applesoft BASIC remains the same as it has always been, so the thousands of programs written in it can also run on the IIGS.

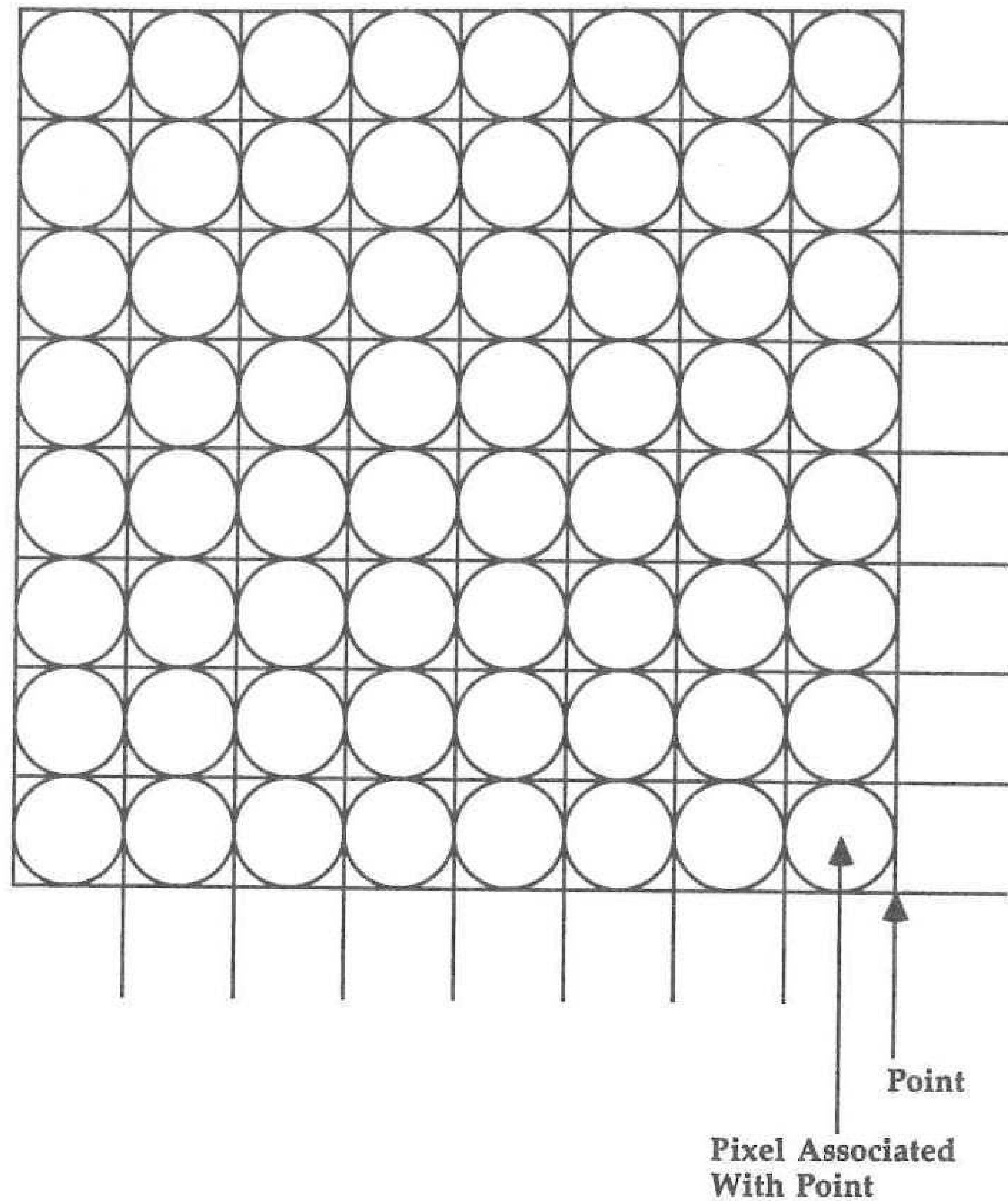
Apple has made it simple for advanced programmers to use the super high-resolution graphics and sound. However, because of the advanced skills required for accessing the "toolboxes," we'll introduce just the concepts here so that you can get an idea of whether you're interested in pursuing the advanced programming levels required to access the IIGS's super high resolution and sound. Remember, there are application programs available for you to do all of this without any programming skills at all. However, you might find doing the programming yourself to be a real adventure.

The QuickDraw II Toolbox: Super Hi-Res Graphics

The collective set of the subroutines built into the IIGS ROM have been named QuickDraw II. By making *calls* to these routines, you can create lines, rectangles, polygons, arcs, and other figures. In Chapter 7 you saw how involved it is to create shapes. However, with the toolbox routines in QuickDraw II, much of the work has been done for you.

The basic building block of the routines is the pixel matrix made up of points and associated pixels. Each point referenced through the toolbox routines has an associated pixel above and to the left of the point. A pattern is composed of an 8 × 8 matrix of points and pixels (Figure 10-1).

Figure 10-1. Pixel Matrix



Depending on the values associated with a given pixel matrix, different patterns, shapes, and figures can be created. The boundaries of the drawings created in QuickDraw II are (−16384,−16384) and (16383,16383) on a Cartesian matrix with memory space of 32768 × 32768 pixels. Various calls made in assembly language, C, or another language that can easily access the QuickDraw II tools are used. The calls are given various names associated with addresses in memory. Let's look at some of these calls to get an idea of what is available in the graphics toolbox inside your IIGS.

QuickDraw II Calls

The first set of functions in the QuickDraw II toolbox is intended for housekeeping purposes. These functions set up the

various registers and pointers to allow access to the graphics tools. They include QDBootInit, which initializes the QuickDraw II tools when the system is booted; QDStartup, which initializes QuickDraw II and sets the standard port and clears the screen; and QDShutDown, which turns QuickDraw II off and frees the buffers. QDVersion and QDStatus, respectively, provide information on the version of QuickDraw II and specify whether or not it is active.

A second set of Quickdraw routines allows you to control the characteristics of the screen display. Each of the 200 horizontal *scan lines* that make make up the display has its own scan-line control byte (SCB), so the characteristics of each line can be specified independently. The GetStandardSCB routine returns information about the SCB. The first four bits (0–3) are used for color table 0; bit 4 is reserved; bit 5 controls the fill option; bit 6 specifies whether an interrupt can occur; and bit 7 controls the color mode (320 pixels or 640 pixels). The call SetMasterSCB sets the low byte of the master SCB, and GetMasterSCB returns the same information. SetSCB, GetSCB, and SetAllSCBs are further scan-line control-byte calls. For setting the color table, either in the 320 or 640 mode, the InitColorTable call is used. Table 10-1 lists the values (in hexadecimal) for the two modes:

Table 10-1. Hexadecimal Values for Setting Color Table

320 Mode

Pixel	Color	Code
$0	Black	0 0 0
$1	Dark gray	7 7 7
$2	Brown	8 4 1
$3	Purple	7 2 C
$4	Blue	0 0 F
$5	Dark green	0 8 0
$6	Orange	F 7 0
$7	Red	D 0 0
$8	Flesh	F A 9
$9	Yellow	F F 0
$A	Green	0 E 0
$B	Light blue	4 D F
$C	Lilac	D A F
$D	Periwinkle blue	7 8 F
$E	Light gray	C C C

640 Mode

Pixel	Color	Code
$0	Black	0 0 0
$1	Red	F 0 0
$2	Green	0 F 0
$3	White	F F F
$4	Black	0 0 0
$5	Blue	0 0 F
$6	Yellow	F F 0
$7	White	F F F
$8	Black	0 0 0
$9	Red	F 0 0
$A	Green	0 F 0
$B	White	F F F
$C	Black	0 0 0
$D	Blue	0 0 F
$E	Yellow	F F 0
$F	White	F F F

Calls to SetColorTable, GetColorTable, SetColorEntry, and GetColorEntry all access the routines to set and get information about the colors.

The other global calls deal with the fonts, clearing the screen, and turning the super hi-res graphics mode on and off. The calls include SetSysFont, GetSysFont, ClearScreen, GrafOn, and GrafOff. The functions of the calls are fairly self-explanatory, making it easier to use the graphics than to use more obtuse codes.

Port of Calls

Now you have some idea of a few of the functions and how the name of the call is connected to the functions. For the rest of the QuickDraw II functions, refer to Table 10-2, just to see the extent of the toolbox. Further description here would do little good, since you do not yet have the programming skills necessary to use the calls.

Table 10-2. GrafPort Calls

OpenPort	InitPort	ClosePort	SetPort	GetPort
SetPortLoc	GetPortLoc	SetPortRect	GetPortRect	SetPortSize
MovePortTo	SetOrigin	SetClip	GetClip	ClipRect
HidePen	ShowPen	GetPen	SetPenState	GetPenState
SetPenSize	GetPensize	SetPenMode	GetPenMode	SetPenPat
GetPenPat	SetSolidPenPat	SetPenMask	GetPenMask	SetBackPat
GetBackPat	SetSolidBackPat	SolidPattern	PenNormal	MoveTo
Move	SetFont	GetFont	SetFontID	GetFontID
GetFontInfo	GetFGSize	GetFontGlobals	SetFontFlags	GetFontFlags
SetTextFace	GetTextFace	SetTextMode	GetTextMode	SetSpaceExtra
GetSpaceExtra	SetCharExtra	GetSpaceExtra	SetForeColor	GetForeColor
SetBackColor	GetBackColor	SetBufDims	ForceBufDims	SaveBufDims
RestoreBufDims	SetClipHandle	GetClipHandle	SetVisRgn	GetVisRgn
SetVisHandle	GetVisHandle	SetPicSave	GetPicSave	SetRgnSave
GetRgnSave	SetPolySave	GetPolySave	SetGrafProcs	GetGrafProcs
SetUserField	GetUserField	SetSysField	GetSysField	

Drawing Calls

This section will give you a better idea of what kinds of drawing shapes are supported by the QuickDraw II toolbox. This set of calls is what most programmers will use often when they're creating graphics directly or when they're writing a graphics drawing program.

Lines
LineTo Line

Rectangles
FrameRect PaintRect EraseRect InvertRect FillRect

Regions
FrameRgn PaintRgn EraseRgn InvertRgn FillRgn

Polygons
FramePoly PaintPoly ErasePoly InvertPoly FillPoly

Ovals
FrameOval PaintOval EraseOval InvertOval FillOval

Rounded-Corner Rectangles
FrameRRect PaintRRect EraseRRect InvertRRect FillRRect

Arcs
FrameArc PaintArc EraseArc InvertArc FillArc

Pixel Transfers
ScrollRect PaintPixels PPToPort

Text Drawing and Measuring
DrawChar DrawText DrawString DrawCString CharWidth
TextWidth StringWidth CStringWidth CharBounds TextBounds
StringBounds CStringBounds

Mapping and Scaling Utilities
MapPt MapRect MapRgn MapPoly ScalePt

Miscellaneous Utilities

Rectangle Calculations
SetRect OffsetRect InsetRect SectRect UnionRect
PtInRect Pt2Rect EqualRect EmptyRect

Point Calculations
AddPt SubPt SetPt EqualPt
LocalToGlobal
GlobalToLocal

Region Calculations
NewRgn DisposeRgn CopyRgn SetEmptyRgn
SetRectRgn RectRgn OpenRgn CloseRgn
OffsetRgn
InsetRgn SectRgn UnionRgn DiffRgn
XorRgn
PtInRgn RectInRgn EqualRgn EmptyRgn

Polygon Calculations
OpenPoly ClosePoly KillPoly OffsetPoly

Other
Random SetRandSeed GetPixel

By looking over these calls from the QuickDraw II toolbox, you can become acquainted with what is available and get a clue as to the graphics power of your Apple IIGS computer.

The Sound of the IIGS

In Chapter 6 you saw an example of sound produced with your computer, but that was simply a little routine for tweaking the speaker. That technique works on all the versions of

the Apple II, but doesn't take advantage of the 5503 Ensoniq Digital Oscillator Chip (DOC). The DOC has 32 digital oscillators that give you everything from beeps and buzzes to a talking computer and symphonic orchestra. However, as with the super high-resolution graphics, you must use the DOC toolbox to take full advantage of this feature.

To get started, let's look quickly at the registers used to control the sounds in DOC.

Frequency control (low and high). Two registers control frequency; joined together they form a 16-bit value used for the 24-bit accumulator. The value of this register pair is added to the current value stored in the 24-bit accumulator.

Address: $00–$1F (low)
$20–$3F (high)

Volume. This register set controls the volume level of the sound created.

Address: $40–$5F

Waveform data sample. This reads the last value from the waveform table.

Address: $60–$7F

Address pointer. These registers are used to determine where in RAM the waveform tables are located. Each waveform table begins with the first address of a page and must continue upward through RAM and cannot wrap around over 64K. The register keeps track of where the table ends.

Address: $80–9F

Control register. Channel assignment, oscillator mode, and halt bit are all controlled by this register. Bits 4–7 make up the channel assignment. Those four bits can assign up to 16 channels for sound. Bit 3 is the interrupt enable used for ordering output when more than a single oscillator has generated output. It helps keep all the different sounds organized. Bits 1 and 2 set the oscillating mode for each oscillator, and bit 0 is the halt bit indicating when an oscillator has been stopped by the microprocessor or DOC.

Address: $A0–$BF

Bank select/resolution/waveform registers. Each register uses seven bits for controlling three major functions (bit 7 is not used). Bit 6 determines whether the DOC address range is 0–64K (0) or 65–128K (1). Bits 3–5 specify the size of the

waveform table, ranging from 256 bytes to 32K bytes. Finally, bits 0–2, called the *resolution determination bits,* actually determine the final address for the waveform table.

Address: $C0–$DF

Oscillator interrupt, oscillator enable, and A/D converter registers. These three registers *(not* bits) control the oscillators and analog-to-digital conversion.

Address: $E0–$E2.

Sound Tools

From the above brief description of the sound registers and the digital oscillator chip, you can see that it's not simple to crank up the kinds of sound heard in musical demonstrations on the IIGS. As an aid to programmers, the sound tools have been provided. There are 18 sound function calls and six low-level routines for accessing the power of DOC. The sound toolkit works through a sound tool set with a specified number. The tool locator finds this number in order to use the sound tools. Again, this requires a higher level of programming skills than you now possess, but to give you an idea of what's in the sound tool set, the calls in Table 10-3 are available:

Table 10-3. Sound Tool Calls

Function Calls	
SoundBootInit	SoundStartup
SoundShutdonw	SoundVersion
SoundReset	SoundToolStatus
WriteRamBlock	ReadRamBlock
GetTableAddress	GetSoundVolume
SetSoundVolume	FFStartSound
FFStopSound	FFSoundStatus
FFGeneratorStatus	SetSoundMIRQV
SetUserSoundIRQV	FFSoundDoneStatus
Low-Level Routines	
Read Register	Write Register
Read Ram	Write Ram
Read Next	Write Next

For some of you, it may be frustrating not to be able to program sound and super high-resolution graphics on your IIGS with what you now know about programming, but be patient. You will learn the more advanced techniques in time. Books and programming utilities will be available in the future to help you.

11
Utility Programs, Hints and Help

Utility Programs, Hints, and Help

By now most of the commands used for programming in Applesoft on the IIGS and many tricks of the trade have been covered. However, if you are seriously interested in learning more about your computer and using it to its full capacity, there's more to learn. In fact, the purpose of this chapter is to give you some direction beyond the scope of the book.

First, you will be introduced to the best thing since silicon—Apple IIGS user groups. These are groups of people that are interested in maximizing their computer's use. Second, I want to suggest some periodicals that will help you learn more about your IIGS. Third, you'll become acquainted with some languages other than BASIC that can be used on your IIGS. BASIC has many advantages, but like all computer languages it has limitations, and you should know what else is available.

Next, you'll be able to examine some more programs that you may find useful, enjoyable, or both. These program listings were chosen to show you some applications of what you've learned in the previous ten chapters, enhancing what you already know.

Then we'll focus on some of the different types of programs that you can purchase. These are programs written by professional programmers that do everything from making your own programming simpler to keeping track of your taxes. Finally, we'll examine some hardware peripherals.

Apple User Groups

Of all of the things I did when I bought my Apple, the most helpful, economical, and useful was joining an Apple user group. Not only did I meet a wonderful group of people, but I was taught how to program and generally what to do and not to do. Each month my club, the Apple Corps of San Diego, has a "disk-of-the-month," with programs from all over the world on both sides of the disk. The programs are contributed

by members in the International Apple Core. At monthly general meetings, speakers inform us about everything from new peripherals available for the Apple IIGS to programming techniques. A beginner's group meets monthly to teach novices how to program, and several other special interest groups meet to discuss hardware, various programming languages, and anything else people want to learn more about. We receive a monthly magazine with club news, tutorials, and programs. The club also makes group buys on various supplies, programs, and hardware for the Apple IIGS, saving members a good deal of money.

Usually, the easiest way to contact your nearest Apple user group is through local computer stores. Often, stores that sell Apple computers will have application forms; some even serve as the meeting site for the clubs. You can also call Apple's toll-free user group referral number—800-538-9696, ext. 500—and you will be given the name and contact information for the nearest Apple user group. To start your own user group, post a notice, giving meeting time and site, in your local computer store, and call the user group referral number and give them the information about the person to contact and the time and location of meetings.

Apple Magazines

There are several periodicals with information about the Apple. Some microcomputer magazines are general and others are for the Apple only.

COMPUTE! and *COMPUTE!'s Apple Applications Special* (800-247-5470; in Iowa, 800-532-1272) are two magazines of interest to IIGS owners. The monthly *COMPUTE!* magazine contains several articles and programs for the Apple. Some programs are written for several computers, among which you will find one for your IIGS. There is a special semiannual issue of *COMPUTE!'s Apple Applications Special* dedicated to the Apple computer. In these issues you can get an in-depth look at the features of your computer, plus reviews, programs, tips, and lots of other information to maximize your computer's utility.

CALL-A.P.P.L.E. (800-426-3667; in Washington state, 206-251-5222) contains some very good articles and programs. This monthly magazine is a professionally produced club magazine with over 10,000 members nationwide. Next to the excellent articles, programs, and tips, the best aspect of belonging to CALL-A.P.P.L.E. Co-op (Apple Puget Sound Pro-

gram Library Exchange) is the savings on software and hardware. In addition, CALL-A.P.P.L.E. Co-op provides a hotline which you can call for help with your Apple, and a free bulletin board service.

A+: The Independent Guide for Apple Computers; Nibble; and *In-Cider* are three more excellent resources for information on your Apple IIGS. Each has a mix of reviews, program listings, program-use articles, and do-it-yourself hardware projects. There's so much to do with a IIGS that the source of interesting material is almost unlimited.

Additionally, there are computer magazines for specialized applications such as education, publishing, robotics, and business that may have general articles that you can apply to your IIGS. Again, just go to your local book or computer store and browse through the table of contents of the various publications to see whether there is something of interest to you.

Apple Speaks Many Languages

Besides BASIC, your IIGS can be programmed and can run programs in several other languages. In some cases special hardware devices are required to run the languages, and special software will be required as well. Here is a brief 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 discussed here. To write in assembly language, you must have an assembler to enter code. Your Apple IIGS has a built-in miniassembler that you can use to get started in assembly language. However, you're going to need something a bit more powerful to do any serious assembly language programming. This language gives you far more control over your Apple than does BASIC, but it is more difficult to learn, and a program takes more instructions to operate than BASIC. By having a good editor/assembler package, you can do a lot more and do it better. Two popular assemblers are:

Merlin 16 (Roger Wagner Publishing). This is the IIGS version of the popular Merlin assembler used for years by Apple programmers. The new version works with the 65816 microprocessor in both the native and emulation (6502) modes. It has an outstanding editor and comes with "Sourceor," a utility that transforms binary code into source listings.

ProDOS ORCA/M (Byteworks). This assembler is another popular assembler for 65816 programming.

High, Low, and Intermediate Languages

When you hear computer people talk of high- and low-level languages, think of high-level being close to talking in normal English and low-level in terms of machine language—binary and hexadecimal. The intermediate-level languages are used extensively by developers, giving them access to both the low-level control and high-level speed of writing programs. (Another way to think of them is to remember that the lower-level the language, the harder it is to learn.)

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. Apple markets a version of U.C.S.D. Pascal, along with manuals on how to use it. However, other versions of Pascal are available. The following are good beginner's books:

Apple Pascal: A Hands-On Approach, by Arthur Luehrmann and Herbert Peckham (McGraw-Hill, 1982). This is a step-by-step introduction to Apple Pascal, showing how to program in Pascal using a single disk drive (many books assume the user has two drives).

Elementary Pascal: Learning to Program in Pascal with Sherlock Holmes, by Henry Ledgard and Andrew Singer (Vintage Books, 1982). This is an enjoyable 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 (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 intermediate-level language, developed to create programs that 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 that execute routines. New words incorporate previously defined words into Forth programs.

The best part of Forth is that several versions are in the public domain. Many Apple clubs have a Forth language system in their disk libraries based on George Lyons' fig (Forth Interest Group) Forth. No special hardware is required to run it since it is an extremely compact language. Several good books on learning Forth are available:

Mastering Forth, by Anita Anderson and Martin Tracy (Brady, 1984). This book covers the latest version of Forth, called Forth-83. It is clearly written and has lots of good examples.

Starting Forth, by Leo Bodie (Prentice-Hall, 1982). Well-written and well-illustrated work on Forth for beginners. Uses a combination of words from fig, 79-Standard, and polyForth.

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 people seriously interested in learning the language.

The C language. Many of the applications used on the Apple IIGS are written in C. This intermediate-level language is highly structured and very transportable (transportability refers to having a single language's code work on different computers and microprocessors). Like Forth, C provides speed to the finished product and the process of programming. Here are two good books on learning C:

From BASIC to C, by Harley M. Templeton (COMPUTE! Books, 1986). This book provides a simple bridge from BASIC to C. Once you understand BASIC, you can quickly apply the same logic to C.

C Made Easy, by Herbert Schildt (Osborne/McGraw-Hill, 1985). This is clear and simple with lots of good examples.

PostScript. The PostScript language is built into several laser printers, including Apple's LaserWriter. With it, you can write programs on your word processor and send them over to a laser printer with a PostScript interpreter and do all kinds of things with text and graphics. The language looks something like Forth, C, and Logo combined. For a good introduction to the language, see *PostScript Language: Tutorial and Cookbook,* by Adobe Systems Incorporated (Addison-Wesley).

Miscellaneous languages and compilers. Besides the languages discussed above, you can get disks with COBOL, FORTRAN, Logo, and other languages for specialized and general applications. Before you spend time, money, and effort on another language, however, it is highly recommended that you carefully examine your needs. If your main interest is in developing your own programs, first learn BASIC thoroughly and see what you can do with it. If BASIC fits your requirements

and its relatively slow speed is sufficient for your uses, your time will be better spent improving your BASIC programming skills. If your main interest is in using application programs, the language capability will depend on the programs you are using.

Finally, if you determine that BASIC programming is best suited to your needs, but you want to speed up your programs, a compiler provides a simple way. Essentially, a compiler is a program that transforms your code into a binary file that will run four to five times faster than Applesoft. All you do is write the program in BASIC, compile it, and then BSAVE the compiled program. From then on, you will BRUN your compiled program.

Some Useful and Enjoyable Programs

This section consists of programs that contain many of the programming practices we have discussed throughout this book. The ones that are included here do something useful either in terms of applications or in demonstrating something about your computer.

Quick Apple-Betize

This program sorts a list of strings (words) for you. It uses the quick-sort 2 algorithm for the sort. It is fairly limited in that it accepts only single strings and you cannot enter commas, but if you ever want to sort a list of names, recipes, or anything else, you will find it handy.

```
10Ø D$ = CHR$ (4)
2Ø Z = Ø:F = Ø
3Ø TEXT : HOME
4Ø DIM A$(1ØØØ)
5Ø N = N + 1
6Ø INPUT "ENTER WORD  ";A$(N)
7Ø IF A$(N) = "/" THEN 1ØØ
8Ø Z = Z + 1
9Ø GOTO 5Ø
1ØØ REM QUICKSORT 2
11Ø HOME :MS$ = "ALPHABETIZING": FLASH : VTAB 9: HT
    AB 2Ø - LEN (M$) / 2: PRINT MS$: NORMAL
12Ø S1 = 1
13Ø L(1) = 1
14Ø R(1) = N
15Ø L1 = L(S1)
```



```
16Ø R1 = R(S1)
17Ø S1 = S1 - 1
18Ø L2 = L1
19Ø R2 = R1
2ØØ X$ = A$( INT ((L1 + R1) / 2))
21Ø C = C + 1
22Ø IF A$(L2) > = X$ THEN 25Ø
23Ø L2 = L2 + 1
24Ø GOTO 21Ø
25Ø C = C1
26Ø IF X$ > = A$(R2) THEN 29Ø
27Ø R2 = R2 - 1
28Ø GOTO 25Ø
29Ø IF L2 > R2 THEN 36Ø
3ØØ S = S + 1
31Ø T$ = A$(L2)
32Ø A$(L2) = A$(R2)
33Ø A$(R2) = T$
34Ø L2 = L2 + 1
35Ø R2 = R2 - 1
36Ø IF L2 < = R2 THEN 21Ø
37Ø IF L2 > = R1 THEN 41Ø
38Ø S1 = S1 + 1
39Ø L(S1) = L2
4ØØ R(S1) = R1
41Ø R1 = R2
42Ø IF L1 < R1 THEN 18Ø
43Ø IF S1 > Ø THEN 15Ø
44Ø REM SORT COMPLETE
45Ø HOME
46Ø GOSUB 9ØØØ
47Ø FOR N = 2 TO Z + 1
48Ø F = F + 1
49Ø IF F > 22 THEN GOSUB 1ØØØ
5ØØ PRINT A$(N)
51Ø NEXT N
52Ø PRINT D$;"PR#Ø"
53Ø END
1ØØØ INVERSE : PRINT "HIT ANY KEY TO CONTINUE  ": G
     ET AN$: NORMAL
1ØØ5 F = Ø
1Ø1Ø RETURN
9ØØØ HOME : PRINT "ENTER PRINTER SLOT (Ø FOR SCREEN
     )": INPUT " ";PR
9Ø1Ø IF NOT PR THEN RETURN
9Ø2Ø IF PR > 2 THEN RETURN
9Ø3Ø PRINT D$;"PR#";PR: RETURN
```

Sort Menu

This menu program uses the Shell-sort algorithm instead of the quick sort. To use it, you must enter the files on your disk as DATA statements within the program. It will automatically run any Applesoft program, but not binary or text files. Of course, with your newly acquired programming skills, you can fix that.

```
1Ø TEXT : HOME
2Ø MU$ = " SORTING MENU ": VTAB 1Ø: HTAB 2Ø - LEN (
   MU$) / 2: FLASH : PRINT MU$
3Ø DIM A$(4Ø):L$ = CHR$ (91):R$ = CHR$ (93)
4Ø ONERR GOTO 7Ø
5Ø FOR I = 1 TO 4Ø: READ A$(I)
6Ø N = N + 1: NEXT
7Ø I = 4Ø
1ØØ REM SHELL SORT
11Ø L = (2 ^ INT ( LOG (N) / LOG (2))) - 1
12Ø L = INT (L / 2)
13Ø IF L < 1 THEN 26Ø
14Ø FOR J = 1 TO L
15Ø FOR K = J + L TO N STEP L
16Ø I = K
17Ø T$ = A$(I)
18Ø IF A$(I - L) < = T$ THEN 22Ø
19Ø A$(I) = A$(I - L)
2ØØ I = I - L
21Ø IF I > L THEN 18Ø
22Ø A$(I) = T$
23Ø NEXT K
24Ø NEXT J
25Ø GOTO 12Ø
26Ø NORMAL : REM SORT DONE
27Ø ONERR GOTO 31Ø
31Ø HOME : FOR I = 1 TO N
32Ø IF I < 1Ø THEN INVERSE : PRINT L$;" ";I;R$;: NO
   RMAL : PRINT " ";A$(I): NEXT
325 IF N < 1Ø THEN 36Ø
33Ø IF I > 2Ø THEN GOTO 35Ø
34Ø INVERSE : PRINT L$;I;R$;: NORMAL : PRINT " ";A$
   (I): NEXT
345 IF N < 2Ø THEN 36Ø
35Ø VTAB (I - 2Ø): HTAB 2Ø: PRINT " ";: INVERSE : P
   RINT L$;I;R$;: NORMAL : PRINT " ";A$(I): NEXT
36Ø GOSUB 6ØØ
37Ø IF C = 99 THEN END
38Ø PRINT CHR$ (4);"RUN";A$(C)
385 END
4ØØ REM ******************************
41Ø REM ENTER FILENAMES HERE, SEPARATED BY A COMMA.
42Ø REM ******************************
43Ø REM FOR EXAMPLE, SEE LINE 44Ø
```



```
44Ø DATA FILE PROGRAM,MUSIC GENERATOR, GRAPHIC DRAW
   , SORT PROGRAM
5ØØ REM BY ADDING FILENAMES TO YOUR DATA LIST, YOU
   CAN AUTOMATICALLY
51Ø REM UPDATE YOUR FILE MENU.
52Ø REM THIS WORKS ONLY WITH 4Ø OR FEWER BAS FILES.
6ØØ VTAB 22: INVERSE : PRINT " ENTER 99 TO EXIT ":
   NORMAL
61Ø VTAB 23: FOR W = 1 TO 5ØØ: NEXT : FOR I = 38 TO
    1 STEP - 1: FOR J = 1 TO 1Ø: NEXT : HTAB I: PR
   INT " ";: NEXT : HTAB 2: INPUT "CHOOSE FILE BY
   NUMBER-> ";C: RETURN
```

Disk Locator

This program incorporates the Shell sort again, but it does so with a sequential text file. When you begin to accumulate disks, you may find it sometimes difficult to remember where everything is, but using "Disk Locator," you can keep track of everything on both sides of your disk.

Note: You must first create a file before READing or APPENDing a file.

```
1Ø TEXT : HOME : VTAB 7
2Ø HD$ = " DISK LOCATOR "
3Ø FOR I = LEN (HD$) TO 1 STEP - 1: PRINT MID$ (HD$
   ,I,1);: NEXT
4Ø INVERSE : FOR W = 1 TO 1ØØØ: NEXT : VTAB 7: HTAB
    1: FOR I = 1 TO LEN (HD$): PRINT MID$ (HD$,I,1)
   ;: NEXT : NORMAL
5Ø D$ = CHR$ (4):S$ = "/"
6Ø DIM DI$(2ØØ),D1$(2ØØ),D2$(2ØØ),ST$(2ØØ)
7Ø PRINT : VTAB 1Ø
8Ø FOR R = 1 TO 4: READ R$(R): NEXT
9Ø FOR R = 1 TO 4: PRINT CHR$ (91);R; CHR$ (93);" "
   ;R$(R): PRINT : NEXT
1ØØ FOR I = 38 TO 1 STEP - 1: FOR J = 1 TO 1Ø: NEXT
    : HTAB I: PRINT " ";: NEXT : HTAB 2: PRINT "CH
   OOSE BY NUMBER->";: GET A%: PRINT A%
11Ø IF A% = 3 THEN PRINT "DO YOU WANT THIS ON THE P
   RINTER? (Y/N) ";: GET AA$: PRINT AA$
12Ø IF A% = 3 THEN 45Ø
13Ø IF A% = 4 THEN END
14Ø TEXT : HOME : REM DISK STORAGE DATABASE
15Ø N = Ø
16Ø FOR I = 1 TO 2ØØ
18Ø HOME : VTAB 1Ø: INVERSE : PRINT "FILE #";N + 1;
   : NORMAL : PRINT SPC( 5);: INVERSE : PRINT "ENT
   ER 'XX' TO QUIT": NORMAL : PRINT
19Ø INPUT "SIDE 1    -> ";D1$(I)
2ØØ IF D1$(I) = "XX" THEN I = 2ØØ: GOTO 27Ø
21Ø INPUT "SIDE 2    -> ";D2$(I)
```

```
220 INPUT "LOCATION -> ";ST$(I)
230 DI$(I) = D1$(I) + S$ + D2$(I) + S$ + ST$(I)
240 PRINT : PRINT DI$(I): INVERSE : PRINT : PRINT "
    CORRECT? (Y/N)";: NORMAL : GET AN$
250 IF AN$ = "N" THEN 180
260 N = N + 1
270 NEXT I
280 I = 200:I = 0
290 HOME :MS$ = " SORTING ": FLASH : VTAB 10: HTAB
    20 - LEN (MS$) / 2: PRINT MS$
300 GOSUB 610
310 IF A% = 2 THEN 380
320 PRINT D$"OPEN DISK.FILES"
325 PRINT D$"WRITE DISK.FILES"
330 FOR I = 1 TO N
340 PRINT DI$(I)
350 NEXT I
360 PRINT D$"CLOSE DISK.FILES"
370 GOTO 430
380 PRINT D$"APPEND DISK.FILES"
385 PRINT D$"WRITE DISK.FILES"
390 FOR I = 1 TO N
400 PRINT DI$(I)
410 NEXT I
420 PRINT D$"CLOSE DISK.FILES"
430 HOME : VTAB 10: PRINT "DISK FILES CLOSED": VTAB
     23: END
440 DATA "CREATE DISK FILES","APPEND DISK FILES","R
    EAD DISK FILES","EXIT"
450 PRINT D$;"OPEN DISK.FILES"
455 PRINT D$"READ DISK.FILES"
460 N = 0
470 ONERR GOTO 520
480 FOR I = 1 TO 200
490 INPUT DI$(I)
505 N = N + 1
510 NEXT I
520 ER = PEEK (222): IF ER = 5 THEN I = 200: GOTO 5
    10
530 HOME : FLASH : VTAB 10:MS$ = " SORTING ": HTAB
    20 - LEN (MS$) / 2: PRINT MS$
540 GOSUB 610
550 HOME
560 IF AA$ = "Y" THEN GOSUB 810
570 FOR I = 1 TO N: PRINT I;". ";DI$(I): NEXT
580 IF AA$ = "Y" THEN PRINT D$;"PR#0"
590 PRINT D$"CLOSE DISK.FILES"
600 END
610 REM SHELL SORT
620 L = (2 ^ INT ( LOG (N) / LOG (2))) - 1
630 L = INT (L / 2)
640 IF L < 1 THEN 790
650 FOR J = 1 TO L
```



```
660 FOR K = J + L TO N STEP L
670 I = K
680 T$ = DI$(I)
690 C = C + 1
700 IF DI$(I - L) < = T$ THEN 750
710 DI$(I) = DI$(I - L)
720 S = S + 1
730 I = I - L
740 IF I > L THEN 690
750 DI$(I) = T$
760 NEXT K
770 NEXT J
780 GOTO 630
790 NORMAL : REM  SORT DONE
800 RETURN
810 PRINT D$"CLOSE DISK.FILES"
815 PR# 0: HOME : PRINT "PRINTER SLOT(0 FOR SCREEN)
    ";: INPUT PR
820 IF NOT PR THEN RETURN
830 IF PR > 7 THEN RETURN
840 PRINT D$;"PR#";PR: RETURN
```

Program Header

This program is given simply for fun and for adding a little interest to your program (or disk) headers. After playing with it awhile, rather than INPUTing the title of your program, simply define YP$ as the title of your program and GOSUB to it as a subroutine at the beginning of your own program. It will put an end to boring program headers. It's set up for 40-column programs. If you want it for 80 columns, just change the values.

```
10000 TEXT : HOME :F$ = "=": PRINT "ENTER PROGRAM N
      AME:": PRINT "(MUST BE EVEN # OF CHARACTERS.)
      "
10005 INPUT "==> ";YP$:YP$ = "**" + YP$ + "**":P =
      LEN (YP$)
10010 HOME :LM = 20 - LEN (YP$) / 2
10020 IF LEN (F$) < > 40 THEN F$ = F$ + "=": GOTO 1
      0020
10030 PRINT F$;: FOR I = 1 TO 15: PRINT "I"; SPC( 3
      8);"I";: NEXT : PRINT F$
10040 INVERSE : FOR I = 2 TO 16: HTAB 2: VTAB I: PR
      INT SPC( 38): NEXT : NORMAL : FOR PAUSE = 1 T
      O 1000: NEXT
10050 FOR K = 2 TO 16: FOR W = 20 TO 21: VTAB K: HT
      AB W: PRINT SPC( 1): NEXT : NEXT : FOR I = 2
      TO LEN (YP$) / 2
10060 FOR J = 2 TO 16: VTAB J: HTAB I + 20: PRINT S
      PC( 1): VTAB J: HTAB 21 - I: PRINT SPC( 1): N
      EXT : NEXT
```

```
10070 VTAB 22
10080 SS$ = "*"
10090 IF LEN (SS$) < 40 THEN SS$ = SS$ + "*": GOTO
      10090
10100 FLASH : HTAB 1: VTAB 18: PRINT SS$: NORMAL
10110 SPEED= 150
10120 L$ = LEFT$ (YP$, LEN (YP$) / 2)
10130 R$ = RIGHT$ (YP$, LEN (YP$) / 2)
10140 FOR V = 1 TO ( LEN (YP$) / 2)
10150 VTAB 9: HTAB 20 + V: PRINT MID$ (R$,V,1): GOS
      UB 10230
10160 IF V = ( LEN (L$)) + 1 THEN 10180
10170 VTAB 9: HTAB 21 - V: PRINT MID$ (L$, LEN (L$)
       - (V - 1),1): GOSUB 10230
10180 NEXT : SPEED= 255
10190 INVERSE : VTAB 22:H$ = " <HIT ANY KEY TO CONT
      INUE> ": HTAB 20 - LEN (H$) / 2: PRINT H$: NO
      RMAL : WAIT - 16384,128: POKE - 16368,0
10200 VTAB 16: HTAB LM: POKE 32,LM: POKE 35,16: POK
      E 33,P: POKE 34,2
10210 FOR I = 1 TO 16: FOR J = 1 TO 50: NEXT J: CAL
      L - 912: NEXT
10220 VTAB 22: TEXT : FOR I = 1 TO 24: FOR J = 1 TO
       50: NEXT J: CALL - 912: NEXT : END
10230 BZ = 49200: FOR I = 1 TO 15: FOR J = 1 TO I *
       (J - 1): NEXT :B = PEEK (BZ): NEXT
10240 RETURN
```

Calendar Plot

This program is an extension of what you learned in Chapter 7 about using lo-res graphics for plotting. It will translate the data you enter so that the graph will fit on the 40-column screen, regardless of the range of input values. You can begin and end with any month you wish, but you are limited to a single year.

```
10 TEXT : HOME
20 VTAB 8: INPUT "MAXIMUM VALUE TO BE ENTERED -> ";
   MV
30 N = 1:NN = MV: FLASH : PRINT "CALCULATING ": NOR
   MAL
40 IF NN > 39 THEN N = N + .1:NN = MV / N: GOTO 40
50 HOME : VTAB 10: INPUT "NAME OF PLOT-> ";NP$
60 HTAB 1: VTAB 10: CALL - 868: INPUT "YEAR-> ";Y
70 VTAB 1: HTAB 20 - LEN (NP$) / 2: INVERSE : PRINT
    NP$: NORMAL : POKE 34,2
80 HTAB 1: VTAB 10: CALL - 868: INPUT "BEGIN WITH W
   HAT MONTH? (1-12) ";B%: PRINT
90 HTAB 1: VTAB 10: CALL - 868: INPUT "END WITH WHA
   T MONTH? (1-12) ";P%: HOME
100 DIM V(P%)
```



```
110 CA$ = "JFMAMJJASOND"
120 FOR I = B% TO P%: PRINT "PLOT VALUE (MAX= ";MV;
    ") FOR ";: GOSUB 230: INPUT " ";V(I): PRINT
130 V(I) = INT (V(I) / N): NEXT
140 GR : COLOR= 10
150 VLIN 0,39 AT 0: HLIN 0,39 AT 39
160 FOR I = 0 TO 39 STEP 4: PLOT 1,I: NEXT
170 COLOR= 15: FOR I = B% TO P%: VLIN 39,(39 - V(I)
    ) AT (I * 3): NEXT
180 HOME : FOR I = B% TO P%: VTAB 21: HTAB (I * 3)
    + 1: PRINT MID$ (CA$,I,1): NEXT I
190 PRINT "HATCH MARKS=";MV / 10
200 HTAB 20 - LEN (NP$) / 2: PRINT NP$;: PRINT " ";
    : INVERSE : PRINT SPC( 1);Y; SPC( 1): NORMAL
210 WAIT - 16384,128
220 TEXT : HOME : END
230 INVERSE : ON I GOSUB 250,260,270,280,290,300,31
    0,320,330,340,350,360
240 NORMAL : RETURN
250 PRINT "JANUARY";: RETURN
260 PRINT "FEBRUARY";: RETURN
270 PRINT "MARCH";: RETURN
280 PRINT "APRIL";: RETURN
290 PRINT "MAY";: RETURN
300 PRINT "JUNE";: RETURN
310 PRINT "JULY";: RETURN
320 PRINT "AUGUST";: RETURN
330 PRINT "SEPTEMBER";: RETURN
340 PRINT "OCTOBER";: RETURN
350 PRINT "NOVEMBER";: RETURN
360 PRINT "DECEMBER";: RETURN
```

ASCII Hexadecimal/Decimal

This program lists ASCII in both hexadecimal and decimal, either to your printer or to the screen. It lists only the first 128 (0–127) codes, but it is handy for looking up CHR$ codes, and it will give you an idea of one way to translate decimal into hexadecimal.

```
10 TEXT : HOME
20 GOSUB 400
30 HOME : PRINT
40 GOSUB 300
50 DIM H$(127)
60 FOR H = 0 TO 126
70 READ H$(H): NEXT H
80 FOR I = 0 TO 63
85 IF PEEK ( - 16384) > 128 THEN GOSUB 500
90 J = I:K = I + 64
100 IF I < 27 THEN J = I + 64
```

```
105 IF I < 1Ø THEN PRINT "$Ø";H$(I);: HTAB (5): PRI
     NT " ";I;: HTAB 11: PRINT CHR$ (J); CHR$ (99);:
     GOTO 15Ø
11Ø IF I < 27 THEN PRINT "$";H$(I);: HTAB (5): PRIN
     T I;: HTAB 11: PRINT CHR$ (J); CHR$ (99);: GOTO
     15Ø
12Ø PRINT "$";H$(I);: HTAB (5): PRINT I;: HTAB (11)
     : IF I > 26 AND I < 32 THEN PRINT "ESC";: GOTO
     15Ø
13Ø IF I = 32 THEN PRINT "SPACE";: GOTO 15Ø
14Ø HTAB (11): PRINT CHR$ (J);
15Ø HTAB (17): PRINT "$";H$(K);: HTAB (23): PRINT K
     ;: HTAB (29): PRINT CHR$ (K)
16Ø NEXT
17Ø PRINT CHR$ (4);"PR#Ø"
18Ø END
2ØØ DATA Ø,1,2,3,4,5,6,7,8,9,ØA,ØB,ØC,ØD,ØE,ØF,1Ø,1
     1,12,13,14,15,16,17,18,19,1A,1B,1C,1D,1E,1F,2Ø,
     21,22,23,24,25,26,27,28,29,2A,2B,2C,2D,2E,2F,3Ø
     ,31,32,33,34,35,36,37,38,39,3A,3B,3C,3D,3E,3F,4
     Ø,41,42,43,44,45,46,47,48,49
21Ø DATA 4A,4B,4C,4E,4F,5Ø
22Ø DATA 51,52,53,54,55,56,57,58,59,5A,5B,5C,5D,5E,
     5F,6Ø,61,62,63,64,65,66,67,68,69,6A,6B,6C,6D,6E
     ,6F,7Ø,71,72,73,74,75,76,77,78,79,7A,7B,7C,7D,7
     E,7F
3ØØ HEX$ = "HEX":DEC$ = "DEC":CH$ = "CHAR"
31Ø L$ = "================================"
32Ø PRINT HEX$; SPC( 1);DEC$; SPC( 2);CH$;" ";: INV
     ERSE : PRINT "/";: NORMAL : PRINT " ";HEX$; SPC
     ( 3);DEC$; SPC( 2);CH$: PRINT L$
33Ø POKE 34,3: IF PR < > Ø THEN RETURN
34Ø VTAB 1Ø: HTAB 33: PRINT "HIT": HTAB 33: PRINT "
     ANY": HTAB 33: PRINT "KEY": HTAB 33: PRINT "TO
     STOP": HTAB 33: PRINT "OR START": POKE 33,32: V
     TAB 4: RETURN
4ØØ HOME : PRINT "ENTER PRINTER SLOT (Ø DISPLAYS TO
     SCREEN)"": INPUT "";PR
41Ø IF NOT PR THEN RETURN
42Ø IF PR > 7 THEN RETURN
43Ø D$ = CHR$ (4)
44Ø PRINT D$;"PR#";PR
45Ø RETURN
5ØØ GET A$: GET B$: RETURN
```

Utility Programs

Programs that help you write other programs more efficiently are called *utility* programs. In this section we will discuss some of the more useful ones that come under this category.

One of the handiest utilities you will find is *Program Line Editor* or *Global Program Line Editor* (Beagle Bros.), better

known simply as *PLE*. The built-in editor in your IIGS is fine for simple editing, but it's not very powerful compared with *PLE*. For example, to change a mistake with *PLE*, all you have to do is to press control-E and the line number with the error. That line will then pop up on your screen, ready to be edited. There are dozens of features to recommend *PLE* to someone interested in programming in Applesoft BASIC.

Another clever utility that can get your Applesoft programs up to near-professional quality is the Toolbox series of utilities (Roger Wagner Publishing). The Toolboxes consist of machine language subroutines that you can append to your BASIC programs. Instead of having to learn machine language, you can get machine language speed, plus improved BASIC routines, just with BASIC and the Toolbox utilities. (This series is not the same as the built-in toolbox utilities in your IIGS, which were described in Chapter 10; they are a lot easier to access and use than the built-in toolbox.)

Many more utilities are expected to be available for the Apple IIGS soon. In the meantime, take a look at those available for Apple II–series machines. Remember, the best source of information is your Apple user group.

Word Processors

Your Apple computer can be turned into a first-class word processor with a program designed for that use. 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 are simple, and 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.

Before we suggest some word processing packages, here's a word of caution. If you meet other Apple users who have a certain type of word processor they have worked with for a while, they will swear it is the best available. Never argue about politics, religion, or word processors with Apple users. This is because word processors are fairly sophisticated programs, and it takes a while to become accustomed to their strengths and weaknesses. By the time a person does, he or she has invested a good deal of time and has become quite skilled in the program's use. Thus, users are convinced that their own word processor is the best there is, and it can be difficult to determine which one is best for you. To give you

some help in making up your mind, Table 11-1 lists some of the more important features you might want to look for in a word processing program.

Generally, the more a word processor can do, the more it will cost. If you want only to write letters and short documents, there is little need to buy an expensive word processor. However, if you are writing longer and more complex documents or a wider variety of documents, the investment in a more sophisticated word processor is worth the added cost. If you have specialized needs (for example, producing billing forms), you will want to look for those features in a word processor that meet those needs. 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 IIGS before laying out your hard-earned cash.

Of the word processors actually available and written specifically for the Apple IIGS, *MouseWrite* (Roger Wagner Publishing) has all of the above features and more. It makes excellent use of the RAM disk available either with the 256K RAM that comes with your IIGS or additional RAM you can add later. It is easy to learn how to use, yet very powerful.

Database Programs

When you want a program for creating and storing information, a database program is what's needed. Essentially, professionally designed database programs are either sequential or random access files. When you use one, all you have to do is to use the predefined fields provided or create your own fields. For example, a user may want to keep a database 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, date of last purchase, how much money is owed, date of last payment, and so forth.

Probably more than most other packages, database programs should be examined carefully before you make a purchase. Some of the more expensive databases can be used with virtually any kind of application, but if you're going to be using your database only to keep a list of names and addresses to print out mailing labels, for example, a database program designed to do that one thing will usually do it better and for a lot less money. Some word processors, such as *MouseWrite*, can be used as mailing-list database systems as well as word processors, so you may not even need a database

Table 11-1. Word Processing Options

Option	Description
Find/replace	Finds any string in your text and/or finds and replaces any one string with another string. Good for correcting spelling errors and locating sections of text to be repaired.
Block moves	Moves blocks of text from one place to another (for example, moves a paragraph from the middle to the end of a document). An extremely valuable editing tool.
Link files	Automatically links files on disks. Very important for longer documents and for linking standardized shorter documents.
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 you to begin editing anywhere on the screen. Screen-oriented editing is important for large documents and where extensive editing is normally required.
Automatic page numbering	Pages are automatically numbered without your having to determine page breaks in writing text.
Imbedded code	In word processors, this enables the user to send special instructions directly to the printer for changing tabs, enabling special characters on the printer, and doing other things to the printed text without having to set the parameters beforehand and/or without having the capability of overriding set parameters.
Spelling checker	More and more word processors have spelling checkers. The size of the dictionary determines the number of words they will check against.
Communications	Now that we're in the Communications Age, having a built-in communications package in your word processor makes it more convenient to send material back and forth. For example, instead of sending a letter through the post office, you can easily send it over your modem if your word processor has a communications program built into it.
Printer spooling	While your printer is printing out a file, this feature allows you to work on other things.
Mail merge	With this feature you can easily create form letters and then automatically print out different names and addresses.
Downloadable fonts	This allows you to get different fonts, including graphics fonts, for your printer. If you use your IIGS for doing newsletters, this is an important feature.
Mouse interface	When I first started using a mouse on a word processor, I was not impressed. Now, I don't know how I worked without it. It is far easier to use the intuitive "mouse arm" to grab text than it is to bip the cursor around with the keys.

program if your word processor can handle your needs. On the other hand, if your needs are varied and involve sophisticated report generation and require changing record fields, then don't expect a simple, specialized program to do the job. Several database programs, including some public domain ones, are available through your user group.

Business Programs

Business programs have such a wide variety of functions that it is best to start with a specific business need and see whether there is a program that will meet that need. On the other hand, there are general business programs that are applicable to many different businesses. Specific business programs include ones that deal only with real estate, stock transactions, or nutritional planning. More general programs include electronic spreadsheets, financial planning, and, as discussed above, database programs.

Unfortunately, business people often spend far too much for systems that do not work. There's a common belief that if you spend a lot of money on software and hardware, you'll get a better system than a less expensive, simpler setup would provide. This thinking is based upon a "you get what you pay for" mentality, and it can lead to systems that are not used at all. Here is where a good dealer or consultant is necessary. Since computers are getting more sophisticated and less expensive, often you do not "get what you pay for" when you purchase a big, expensive one. Frequently, all the business person ends up with is a dinosaur system that is outmoded, too big, and too expensive for the needs.

Some computer dealers specialize in helping business people. They will set up a system in your place of business, help train 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, someone will be available to help you. Since the Apple IIGS is comparatively inexpensive to begin with, the extra money spent buying from a supportive dealer is worth the extra cost.

Alternatively, there are consultants available for setting up your system. If you use a consultant, get one who is an independent, with no connection to a vested interest in selling computers. Contact one through your phone book. Say that you want to set up an Apple IIGS system in your office and describe exactly what your needs are. If the consultant is fa-

miliar with your system, he or she will know the available software and peripherals you need. If the consultant tries to sell you another computer, that's probably an indication of unfamiliarity with your system, and you might want to try another consultant.

However, if several consultants tell you that your needs cannot be met by an Apple IIGS, you may indeed need a larger system. I have encountered many unhappy business people who bought the wrong system for their needs. One man said he paid $14,000 for a computer system that never did work for his requirements, and he finally bought a microcomputer system for about one-tenth that price and everything worked out fine. This does not mean that a business may not require an expensive system to handle certain business functions, and the IIGS certainly has limitations.

However, before you buy any system, make sure it does what you want and have it shown to you working the way you expect it to work. You will often find that the less expensive new micros like the IIGS actually work better than do larger machines. With the addition of a relatively low-cost hard disk and increased RAM, the IIGS can handle a lot of big business files.

Many small and individual businesses can use integrated programs such as *AppleWorks* (Apple Computer) or *VIP Professional* (VIP Technologies) to keep track of everything in a single program—from their customers to spreadsheet analysis. Integrated programs gained popularity with the famous Lotus *1-2-3* (Lotus Development) on the IBM PC. Several programs were melded together; instead of existing as a number of separate programs for different tasks, such as spreadsheet and database, these were integrated so that different parts could be used together.

Graphics Programs

Chapter 7 discussed some of the Apple IIGS's capabilities with graphics. However, certain uses require either highly advanced programming skills or a good graphics package. For example, it is possible to draw on the screen in super high-resolution graphics, just as you would with a palette. The pictures produced can then be saved to disk or printed out to your printer. These programs allow you to concentrate on the graphics themselves rather than on the programming techniques necessary to produce them.

DeluxePaint (Electronic Arts) is a good example of a graphics drawing program. You can "mix" the 16 colors into a combination of 4096 colors available on the Apple IIGS. Using the mouse as a super paintbrush, you can create incredible paintings with *DeluxePaint*. This is a tremendous leap over previous versions of the Apple II.

Hardware

The Apple IIGS is expandable, which means you can add various attachments to it to make it do more than it does normally. In the back and inside of your IIGS there are a number of ports where hardware extensions can be attached. Joysticks are used for games as well as other programs. For games, they guide rockets, spaceships, and characters against the forces of evil. However, they are also used for drawing graphics and for input in other programs. Of course, your mouse will probably be used as the input device for most such input. Other hardware attachments include interfaces for various peripherals.

Another important peripheral that you may want to consider is a hard disk. Since your IIGS has a built-in hard disk controller, the price for one is far below what is normally charged on microcomputer systems. A hard disk will greatly increase your storage capacity and significantly speed up a lot of your work with large programs and files.

Music buffs may want to look at *Deluxe Music Construction Set* (Electronic Arts). The IIGS has a fabulous music chip, but it is not easily programmed from Applesoft. Thus, to concentrate on the score and composition, you will need a good music program, and *Deluxe Music Construction Set* is highly recommended.

Modems and Communications

One of the most exciting things you can do with your computer is to communicate with another computer. Not only can you communicate with another IIGS, but you can access other micros and even tie into big mainframes. The Apple Personal Modem, made by Apple, plugs easily into the wall, and a cable plugs into your computer's serial port. However, with the IIGS you are in luck, for if you get the right cable, you can hook up just about any modem to the serial port and inexpensively have modem communications. You will have no problems communicating between the IIGS and other computers.

A good deal. Currently, some modem packages include free membership in CompuServe and Dow Jones News/Retrieval, plus one free hour of use. This is a good way to check out these services to see whether they are of interest to you. If they're not, you're not out the cost of membership. This deal may not still be available when you buy your modem or communication package, but it's worth checking before you purchase a modem.

Summary

The most important thing to understand from this chapter is that you have just scratched the surface of what is available for the Apple IIGS. There is much more than a single chapter could possibly cover. As you come to know your Apple, you will find that the choice of software and peripherals is limited only by the confusion in making up your mind. The software and hardware suggested here are based on personal preferences, and I 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 has found to be of interest. Then, after your own sampling, examination, and testing, get exactly what you need.

You should now 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 recreation. 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 don't understand everything about how your car's engine works either, and that has never prevented you from driving. Think of your computer, like your car, as a vehicle that will take you where you want to go. Never again consider it a machine that you must follow.

Appendices

Appendix A

Applesoft BASIC Token Chart

The following chart shows the tokens for each Applesoft BASIC keyword, along with the address of the routine called to execute that statement. Values are given in both hexadecimal and in decimal. When you examine an Applesoft program with the monitor, a value of 136($88) will indicate a GR statement or a value of 221($DD) will indicate the EXP function. To use one of these commands from the monitor, type the address corresponding to the proper token, followed by G. For example, use F390G to execute the GR statement.

Note: Some functions and the error messages do not have the jump routines.

Token Values			Jump Addresses	
$80	128	END	$D870	55408
$81	129	FOR	$D766	55142
$82	130	NEXT	$DCF9	56569
$83	131	DATA	$D995	55701
$84	132	INPUT	$DBB2	56242
$85	133	DEL	$F331	62257
$86	134	DIM	$DFD9	57305
$87	135	READ	$DBE2	56290
$88	136	GR	$F390	62352
$89	137	TEXT	$F399	62361
$8A	138	PR#	$F1E5	61925
$8B	139	IN#	$F1DE	61918
$8C	140	CALL	$F1D5	61909
$8D	141	PLOT	$F225	61989
$8E	142	HLIN	$F232	62002
$8F	143	VLIN	$F241	62017
$90	144	HGR2	$F3D8	62424
$91	145	HGR	$F3E2	62434
$92	146	HCOLOR=	$F6E9	63209
$93	147	HPLOT	$F6FE	63230

APPENDIX A

Token Values			Jump Addresses	
$94	148	DRAW	$F769	63337
$95	149	XDRAW	$F76F	63343
$96	150	HTAB	$F7E7	63463
$97	151	HOME	$FC58	64600
$98	152	ROT=	$F721	63265
$99	153	SCALE=	$F727	63271
$9A	154	SHLOAD	$03F5	1013
$9B	155	TRACE	$F26D	62061
$9C	156	NOTRACE	$F26F	62063
$9D	157	NORMAL	$F273	62067
$9E	158	INVERSE	$F277	62071
$9F	159	FLASH	$F280	62080
$A0	160	COLOR=	$F24F	62031
$A1	161	POP	$D96B	55659
$A2	162	VTAB	$F256	62038
$A3	163	HIMEM:	$F286	62086
$A4	164	LOMEM:	$F2A6	62118
$A5	165	ONERR	$F2CB	62155
$A6	166	RESUME	$F318	62232
$A7	167	RECALL	$03F5	1013
$A8	168	STORE	$03F5	1013
$A9	169	SPEED=	$F262	62050
$AA	170	LET	$DA46	55878
$AB	171	GOTO	$D93E	55614
$AC	172	RUN	$D912	55570
$AD	173	IF	$D9C9	55753
$AE	174	RESTORE	$D849	55369
$AF	175	&	$03F5	1013
$B0	176	GOSUB	$D921	55585
$B1	177	RETURN	$D96B	55659
$B2	178	REM	$D9DC	55772
$B3	179	STOP	$D86E	55406
$B4	180	ON	$D9EC	55788
$B5	181	WAIT	$E784	59268
$B6	182	LOAD	$03F5	1013
$B7	183	SAVE	$03F5	1013
$B8	184	DEF	$E313	58131
$B9	185	POKE	$E77B	59259
$BA	186	PRINT	$DAD5	56021
$BB	187	CONT	$D896	55446
$BC	188	LIST	$D6A5	54949
$BD	189	CLEAR	$D66A	54890
$BE	190	GET	$DBA0	56224
$BF	191	NEW	$D649	54857
$C0	192	TAB(		

Token Values			Jump Addresses	
$C1	193	TO		
$C2	194	FN		
$C3	195	SPC(		
$C4	196	THEN		
$C5	197	AT		
$C6	198	NOT		
$C7	199	STEP		
$C8	200	+		
$C9	201	−		
$CA	202	*		
$CB	203	/		
$CC	204	^		
$CD	205	AND		
$CE	206	OR		
$CF	207	>		
$D0	208	=		
$D1	209	<		
$D2	210	SGN	$EB90	60304
$D3	211	INT	$EC23	60451
$D4	212	ABS	$EBAF	60335
$D5	213	USR	$000A	10
$D6	214	FRE	$E2DE	58078
$D7	215	SCRN(	$D412	54290
$D8	216	PDL	$DFCD	57293
$D9	217	POS	$E2FF	58111
$DA	218	SQR	$EE8D	61069
$DB	219	RND	$EFAE	61358
$DC	220	LOG	$E941	59713
$DD	221	EXP	$EF09	61193
$DE	222	COS	$EFEA	61418
$DF	223	SIN	$EFF1	61425
$E0	224	TAN	$F03A	61498
$E1	225	ATN	$F09E	61598
$E2	226	PEEK	$E764	59236
$E3	227	LEN	$E6D6	59094
$E4	228	STR$	$E3C5	58309
$E5	229	VAL	$E707	59143
$E6	230	ASC	$E6E5	59109
$E7	231	CHR$	$E646	58950
$E8	232	LEFT$	$E65A	58970
$E9	233	RIGHT$	$E686	59014
$EA	234	MID$	$E691	59025

Error Message Tokens

$EB	235	NEXT WITHOUT FOR
$EC	236	SYNTAX
$ED	237	RETURN WITHOUT GOSUB
$EE	238	OUT OF DATA
$EF	239	ILLEGAL QUANTITY
$F0	240	OVERFLOW
$F1	241	OUT OF MEMORY
$F2	242	UNDEF'D STATEMENT
$F3	243	BAD SUBSCRIPT
$F4	244	REDIM'D ARRAY
$F5	245	DIVISION BY ZERO
$F6	246	ILLEGAL DIRECT
$F7	247	TYPE MISMATCH
$F8	248	STRING TOO LONG
$F9	249	FORMULA TOO COMPLEX
$FA	250	CAN'T CONTINUE
$FB	251	UNDEF'D FUNCTION
$FC	252	ERROR IN

Appendix B
ASCII Characters

Characters preceded by carets (^) are control characters.

0	^@	32	(space)	64	@	96	
1	^A	33	!	65	A	97	a
2	^B	34	"	66	B	98	b
3	^C	35	#	67	C	99	c
4	^D	36	$	68	D	100	d
5	^E	37	%	69	E	101	e
6	^F	38	&	70	F	102	f
7	^G	39	'	71	G	103	g
8	^H	40	(	72	H	104	h
9	^I	41	)	73	I	105	i
10	^J	42	*	74	J	106	j
11	^K	43	+	75	K	107	k
12	^L	44	,	76	L	108	l
13	^M	45	–	77	M	109	m
14	^N	46	.	78	N	110	n
15	^O	47	/	79	O	111	o
16	^P	48	0	80	P	112	p
17	^Q	49	1	81	Q	113	q
18	^R	50	2	82	R	114	r
19	^S	51	3	83	S	115	s
20	^T	52	4	84	T	116	t
21	^U	53	5	85	U	117	u
22	^V	54	6	86	V	118	v
23	^W	55	7	87	W	119	w
24	^X	56	8	88	X	120	x
25	^Y	57	9	89	Y	121	y
26	^Z	58	:	90	Z	122	z
27	^[	59	;	91	[	123	{
28	^\	60	<	92	\	124	|
29	^]	61	=	93	]	125	}
30	^^	62	>	94	^	126	~
31	^_	63	?	95	_	127	DEL

Appendix C

Hex-to-Decimal and Decimal-to-Hex Conversion

Decimal-to-Hex Conversion

Decimal	Hex	Decimal	Hex	Decimal	Hex
0	$0	34	$22	68	$44
1	$1	35	$23	69	$45
2	$2	36	$24	70	$46
3	$3	37	$25	71	$47
4	$4	38	$26	72	$48
5	$5	39	$27	73	$49
6	$6	40	$28	74	$4A
7	$7	41	$29	75	$4B
8	$8	42	$2A	76	$4C
9	$9	43	$2B	77	$4D
10	$A	44	$2C	78	$4E
11	$B	45	$2D	79	$4F
12	$C	46	$2E	80	$50
13	$D	47	$2F	81	$51
14	$E	48	$30	82	$52
15	$F	49	$31	83	$53
16	$10	50	$32	84	$54
17	$11	51	$33	85	$55
18	$12	52	$34	86	$56
19	$13	53	$35	87	$57
20	$14	54	$36	88	$58
21	$15	55	$37	89	$59
22	$16	56	$38	90	$5A
23	$17	57	$39	91	$5B
24	$18	58	$3A	92	$5C
25	$19	59	$3B	93	$5D
26	$1A	60	$3C	94	$5E
27	$1B	61	$3D	95	$5F
28	$1C	62	$3E	96	$60
29	$1D	63	$3F	97	$61
30	$1E	64	$40	98	$62
31	$1F	65	$41	99	$63
32	$20	66	$42	100	$64
33	$21	67	$43	101	$65

102	$66	147	$93	192	$C0
103	$67	148	$94	193	$C1
104	$68	149	$95	194	$C2
105	$69	150	$96	195	$C3
106	$6A	151	$97	196	$C4
107	$6B	152	$98	197	$C5
108	$6C	153	$99	198	$C6
109	$6D	154	$9A	199	$C7
110	$6E	155	$9B	200	$C8
111	$6F	156	$9C	201	$C9
112	$70	157	$9D	202	$CA
113	$71	158	$9E	203	$CB
114	$72	159	$9F	204	$CC
115	$73	160	$A0	205	$CD
116	$74	161	$A1	206	$CE
117	$75	162	$A2	207	$CF
118	$76	163	$A3	208	$D0
119	$77	164	$A4	209	$D1
120	$78	165	$A5	210	$D2
121	$79	166	$A6	211	$D3
122	$7A	167	$A7	212	$D4
123	$7B	168	$A8	213	$D5
124	$7C	169	$A9	214	$D6
125	$7D	170	$AA	215	$D7
126	$7E	171	$AB	216	$D8
127	$7F	172	$AC	217	$D9
128	$80	173	$AD	218	$DA
129	$81	174	$AE	219	$DB
130	$82	175	$AF	220	$DC
131	$83	176	$B0	221	$DD
132	$84	177	$B1	222	$DE
133	$85	178	$B2	223	$DF
134	$86	179	$B3	224	$E0
135	$87	180	$B4	225	$E1
136	$88	181	$B5	226	$E2
137	$89	182	$B6	227	$E3
138	$8A	183	$B7	228	$E4
139	$8B	184	$B8	229	$E5
140	$8C	185	$B9	230	$E6
141	$8D	186	$BA	231	$E7
142	$8E	187	$BB	232	$E8
143	$8F	188	$BC	233	$E9
144	$90	189	$BD	234	$EA
145	$91	190	$BE	235	$EB
146	$92	191	$BF	236	$EC

237	$ED	244	$F4	251	$FB
238	$EE	245	$F5	252	$FC
239	$EF	246	$F6	253	$FD
240	$F0	247	$F7	254	$FE
241	$F1	248	$F8	255	$FF
242	$F2	249	$F9		
243	$F3	250	$FA		

Hex-to-Decimal Conversion

$00	0	$23	35	$46	70
$01	1	$24	36	$47	71
$02	2	$25	37	$48	72
$03	3	$26	38	$49	73
$04	4	$27	39	$4A	74
$05	5	$28	40	$4B	75
$06	6	$29	41	$4C	76
$07	7	$2A	42	$4D	77
$08	8	$2B	43	$4E	78
$09	9	$2C	44	$4F	79
$0A	10	$2D	45	$50	80
$0B	11	$2E	46	$51	81
$0C	12	$2F	47	$52	82
$0D	13	$30	48	$53	83
$0E	14	$31	49	$54	84
$0F	15	$32	50	$55	85
$10	16	$33	51	$56	86
$11	17	$34	52	$57	87
$12	18	$35	53	$58	88
$13	19	$36	54	$59	89
$14	20	$37	55	$5A	90
$15	21	$38	56	$5B	91
$16	22	$39	57	$5C	92
$17	23	$3A	58	$5D	93
$18	24	$3B	59	$5E	94
$19	25	$3C	60	$5F	95
$1A	26	$3D	61	$60	96
$1B	27	$3E	62	$61	97
$1C	28	$3F	63	$62	98
$1D	29	$40	64	$63	99
$1E	30	$41	65	$64	100
$1F	31	$42	66	$65	101
$20	32	$43	67	$66	102
$21	33	$44	68	$67	103
$22	34	$45	69	$68	104

$69 105
$6A 106
$6B 107
$6C 108
$6D 109
$6E 110
$6F 111
$70 112
$71 113
$72 114
$73 115
$74 116
$75 117
$76 118
$77 119
$78 120
$79 121
$7A 122
$7B 123
$7C 124
$7D 125
$7E 126
$7F 127
$80 128
$81 129
$82 130
$83 131
$84 132
$85 133
$86 134
$87 135
$88 136
$89 137
$8A 138
$8B 139
$8C 140
$8D 141
$8E 142
$8F 143
$90 144
$91 145
$92 146
$93 147
$94 148
$95 149

$96 150
$97 151
$98 152
$99 153
$9A 154
$9B 155
$9C 156
$9D 157
$9E 158
$9F 159
$A0 160
$A1 161
$A2 162
$A3 163
$A4 164
$A5 165
$A6 166
$A7 167
$A8 168
$A9 169
$AA 170
$AB 171
$AC 172
$AD 173
$AE 174
$AF 175
$B0 176
$B1 177
$B2 178
$B3 179
$B4 180
$B5 181
$B6 182
$B7 183
$B8 184
$B9 185
$BA 186
$BB 187
$BC 188
$BD 189
$BE 190
$BF 191
$C0 192
$C1 193
$C2 194

$C3 195
$C4 196
$C5 197
$C6 198
$C7 199
$C8 200
$C9 201
$CA 202
$CB 203
$CC 204
$CD 205
$CE 206
$CF 207
$D0 208
$D1 209
$D2 210
$D3 211
$D4 212
$D5 213
$D6 214
$D7 215
$D8 216
$D9 217
$DA 218
$DB 219
$DC 220
$DD 221
$DE 222
$DF 223
$E0 224
$E1 225
$E2 226
$E3 227
$E4 228
$E5 229
$E6 230
$E7 231
$E8 232
$E9 233
$EA 234
$EB 235
$EC 236
$ED 237
$EE 238
$EF 239

$F0 240
$F1 241
$F2 242
$F3 243
$F4 244
$F5 245

$F6 246
$F7 247
$F8 248
$F9 249
$FA 250
$FB 251

$FC 252
$FD 253
$FE 254
$FF 255

Appendix D
Error Messages

Here are the codes for the error messages that you'll encounter on your Apple IIGS. Whenever you have an error-handling routine using ONERR and PEEK (222), you will be given a code representing an error. For example, error 42 means that you are OUT OF DATA when a READ statement has tried to read more than the available number of DATA statements.

The error messages are divided into four sections: DOS 3.3 and Applesoft error messages, ProDOS 8 error messages, ProDOS 16 messages (though you may not encounter them with Applesoft programs), and finally, four fatal errors.

Applesoft and DOS 3.3 Error Messages

0	NEXT WITHOUT FOR
1	LANGUAGE NOT AVAILABLE
2, 3	RANGE ERROR
4	WRITE PROTECTED
5	END OF DATA
6	FILE NOT FOUND
7	VOLUME MISMATCH
8	I/O ERROR
9	DISK FULL
10	FILE LOCKED
11	DOS SYNTAX ERROR
12	NO BUFFERS AVAILABLE
13	FILE MISMATCH
14	PROGRAM TOO LARGE
15	NOT DIRECT COMMAND
16	PROGRAM SYNTAX ERROR
22	RETURN WITHOUT GOSUB
42	OUT OF DATA
53	ILLEGAL QUANTITY
69	OVERFLOW
77	OUT OF MEMORY
90	UNDEFINED STATEMENT
107	BAD SUBSCRIPT
120	REDIM'D ARRAY

133 DIVISION BY ZERO
163 TYPE MISMATCH
176 STRING TOO LONG
191 FORMULA TOO COMPLEX
224 UNDEFINED FUNCTION
254 BAD INPUT RESPONSE
255 CONTROL-C INTERRUPT

ProDOS 8 Error Messages

2	RANGE ERROR
3	NO DEVICE CONNECTED
4	WRITE PROTECTED
5	END OF DATA
6	PATH NOT FOUND
7	PATH NOT FOUND
8	I/O ERROR
9	DISK FULL
10	FILE LOCKED
11	INVALID OPTION
12	NO BUFFERS AVAILABLE
13	FILE TYPE MISMATCH
14	PROGRAM TOO LARGE
15	NOT DIRECT COMMAND
16	SYNTAX ERROR
17	DIRECTORY FULL
18	FILE NOT OPEN
19	DUPLICATE FILENAME
20	FILE BUSY
21	FILE(S) STILL OPEN

ProDOS 16 Error Messages

$1	Invalid call number
$5	Call pointer out of bounds
$6	Invalid caller identification
$10	Device not found
$11	Invalid device ref number
$20	Invalid request code
$25	Interrupt table full
$26	Invalid operation
$27	I/O Error (Note: different code number from DOS 3.3 above)
$28	No device connected
$2B	Write protected
$2E	Disk switched

$30–$3F	Device-specific errors
$40	Invalid pathname syntax
$42	FCB full
$43	Invalid file reference number
$44	Path not found
$45	Volume not found
$47	Duplicate filename
$48	Volume full
$49	Directory full
$4A	Version error
$4B	Unsupported storage type
$4C	End of file encountered (EOF)
$4D	Position out of range
$4E	Access not allowed
$50	File is open
$51	Directory structure damaged
$52	Unsupported volume type
$53	Parameter out of range
$54	Out of memory
$55	VCB full
$57	Duplicate volume
$58	Not a block device
$59	Invalid level
$5A	Block number out of range

Fatal Errors

$1	Unclaimed interrupt
$A	VCB unusable
$B	FCB unusable
$C	Block zero allocated illegally

Appendix E

Glossary

The glossary contains Applesoft as well as DOS 3.3 and ProDOS statements, functions, and commands; it is arranged alphabetically. (DOS indicates either DOS 3.3 or ProDOS.) The examples have been set up to show you how to use the keywords with their proper syntax. In some cases, when a command can be used in different contexts, more than a single example has been given. Some examples are shown in the immediate mode, some in the program mode (those with line numbers), and some in both modes. The more advanced commands listed here were not covered in the text, but they're provided here to give you a complete reference.

ABS( )

Gives the absolute value of a number or variable.

```
PRINT ABS(123.45)
```

AND

Logical operator.

```
140 IF A$ < > "Y" AND A$ < > "N" THEN GOTO 100
```

APPEND

Adds data to end of existing sequential text file (DOS).

```
200 PRINT CHR$(4); "APPEND NAMES"
```

ASC( )

Returns ASCII value of first character in string.

```
PRINT ASC ("W")
```

or

```
A$ = "APPLE" : PRINT ASC(A$)
```

ATN( )

Returns arctangent of number or variable.

```
PRINT ATN (123)
```

BLOAD
Loads binary file into memory. Does not need address to load, but can be included (DOS).

Immediate mode: BLOAD GRAPHICS FILE or BLOAD GRAPHICS FILE, A$4000
Program mode: 100 PRINT CHR$(4); "BLOAD GRAPHICS FILE"

BRUN
Runs binary file from disk (DOS).

Immediate mode: BRUN SPACE APES
Program mode: 100 PRINT CHR$(4); "BRUN SPACE APES"

BSAVE
Saves binary file to disk. Must include both starting address and length in either hexadecimal or decimal (DOS).

Hexadecimal: BSAVE ZOOM BLAST, A$300, L$3DF
Decimal: BSAVE SPACE SHIP, A801, L1234

CALL
Goes to machine language subroutine at a given decimal address.

CALL -936.

CAT
Shows the files in a given directory. The prefix directory is shown if you do not specify a directory (ProDOS).

CAT
CAT /SORTS

CATALOG
Prints the contents of disk to screen or printer (DOS).

CATALOG,D1

CHR$( )
Returns the character with a given decimal value.

PRINT CHR$(65)

CLEAR
All variables are reset to zero.

120 CLEAR

CLOSE
Closes text file.

210 PRINT CHR$(4); "CLOSE NAME LIST"

COLOR=
Sets color in lo-res graphics in values from 0 through 15.

30 COLOR = 9

CONT
Continues program after a STOP, END, or CTRL-C

CONT or CONT 300

COS( )
Returns the cosine of variable or number.

PRINT COS(123)

CREATE
Used primarily to make directory files, but it can be used to create any kind of file (ProDOS).

Directory file CREATE /SORTS
BASIC file CREATE SPOTS.BAS

DATA
Strings or numbers to be read.

1000 DATA 2, 345, HELLO, "WALK"

DEF FN( )
Defines a function for real variable.

DEF FN A(X) = X * X
PRINT FN A(4) (Result = 16)

DEL
Deletes range of line numbers.

DEL 120,200

DELETE
Deletes file from disk (DOS).

DELETE ADDING MACHINE

DIM
Allocates memory for array.

```
130 DIM A$ (100)
```

DRAW AT
Draws a hi-res shape table on hi-res screen at *(x,y)* coordinates.

```
50 DRAW 5 AT 100,40
```

END
Terminates running of program.

```
200 END
```

EXEC
Executes the contents of a text file without removing program in memory (DOS).

```
EXEC PROGRAM SETUP
```

EXP( )
Returns *e*(2.718289) to indicated power.

```
PRINT EXP (3)
```

FLASH
Turns on flashing mode.

```
30 FLASH : PRINT "HELLO"
```

FLUSH
Takes all data in buffers and writes data to file (ProDOS).

```
FLUSH NAMES/CUSTOMERS
```
(Writes all data in buffers to file CUSTOMERS)

FOR
Sets up beginning and limit of FOR-NEXT loop.

```
40 FOR I = 1 TO 100
```

FP
Sets to floating point or Applesoft language. Also clears memory (DOS 3.3).

```
FP
```


FRE( )
Returns available memory.

```
PRINT FRE(0)
```

GET
Halts execution until single entry received from keyboard.

```
30 GET A$  or  GET A  or  GET A%
```

GOSUB
Branches to subroutine at given line number.

```
100 GOSUB 200
```

GOTO
Branches to given line number.

```
100 GOTO 200
```

GR
Sets lo-res graphics mode.

```
120 GR
```

HCOLOR=
Set hi-res color to values of 0–7.

```
40 HCOLOR= 3
```

HGR
Sets page 1 hi-res graphics mode and clears hi-res screen 1.

```
120 HGR
```

HGR2
Sets page 2 hi-res graphics mode and clears hi-res screen 2.

```
130 HGR2
```

HIMEM
Sets memory to highest available location for a program within a range of −65535 to 65535, depending on RAM memory.

```
10 HIMEM: 40123
```

HLIN AT
Draws a horizontal line in lo-res graphics at a given vertical position.

```
40 HLIN 10,20 AT 11
```

HOME
Clears screen and places cursor in upper right corner of text window.

```
HOME
```

HPLOT TO
Plots single point or points on hi-res screen at given *(x,y)* coordinates to another set of *(x,y)* coordinates.

```
40 HPLOT 10,50 : REM SINGLE POINT
50 HPLOT 0,0 TO 279,151 : REM RANGE OF POINTS
```

HTAB
Horizontal tab position set.

```
50 HTAB 30 : PRINT "HERE"
```

IF-THEN
Sets up conditional logic for execution.

```
60 IF A$ = "Q" THEN END
```

INIT
Initializes disk (DOS 3.3).

```
INIT HELLO
```

IN#
Takes input from indicated slot number (DOS).

```
IN#6
```

INPUT
Halts program execution until strings or numbers are entered and the return key has been pressed. May enter message within INPUT statement.

```
90 INPUT "ENTER WORD-> "; W$(I)
100 INPUT "ENTER NUMBER -> "; A
110 INPUT "ENTER INTEGER NUMBER -> "; N%
120 PRINT "HIT 'RETURN' TO CONTINUE ";
130 INPUT R$
```


INT
Sets to Integer BASIC language and clears memory (DOS 3.3).

```
INT
```

INT( )
Returns the integer value of a real variable or number.

```
PRINT INT (123.45)
```

INVERSE
Turns on inverse mode.

```
50 INVERSE : PRINT "APPLE"
```

LEFT$()
Returns specifed number of characters from a given string beginning with character at far left.

```
10 A$ = "GOODBYE"
20 PRINT LEFT$ (A$,4)    (Results = GOOD)
```

LEN
Returns the length in terms of number of characters of a specified string.

```
PRINT LEN(A$)
```

LIST
Lists program currently in memory.

```
LIST
```

LOAD
Loads Applesoft or integer program specified (DOS).

```
LOAD CALENDAR PLOT
```

LOCK
Prevents file from being overwritten or deleted (DOS).

```
LOCK CALENDAR PLOT
```

LOG( )
Returns logarithm of specifed number or variable.

```
PRINT LOG (15)  or  PRINT LOG (G)
```

LOMEM
Sets memory to lowest location available for a program within a range from −65535 through 65535, depending on RAM memory.

```
LOMEM: 1234
```

MAXFILES
Reserves specified number of buffers for files within a range of 1 through 16 (DOS 3.3).

```
MAXFILES 10
```

MID$( )
Returns a portion of a string beginning with the *nth* number from the left to the length of the second number.

```
10 A$ = "WONDERFUL"
20 PRINT MID$(A$,4,3)   (Results = DER)
```

MON
Turns on screen display of computer-disk communication (DOS 3.3).

```
MONCIO  or  MON C,I,O
```

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
```

NOMON
Turns off screen display of computer-disk communication (DOS 3.3).

```
NOMONCIO   or  NOMON C,I,O
```

NORMAL
Returns screen to standard display from INVERSE or FLASH modes.

```
10 FLASH : PRINT "FLASHING MODE"
20 NORMAL: PRINT "NORMAL MODE"
```


NOT
Logical negation in IF-THEN statement.

```
60 IF NOT B THEN GOTO 100
```

NOTRACE
Turns off TRACE mode.

```
NOTRACE
```

ON
Sets up computed GOTO and GOSUB.

```
190 ON A GOSUB 1000,2000,3000
```

ONERR
Branches to specified line when error is encountered.

```
40 ONERR GOTO 1000
```

OPEN
Creates or starts new text file (DOS).

```
500 PRINT CHR$(4); "OPEN NAME LIST"
```

OR
Logical OR in IF-THEN statement.

```
130 IF A=10 OR B = 20 THEN GOTO 190
```

PDL( )
Returns value of specified paddle number 0–3.

```
PRINT PDL(0)
```

PEEK
Returns memory contents of given decimal location.

```
170 PRINT PEEK (768)
180 IF PEEK(768) = 5 THEN GOTO 200
```

PLOT
Plots point in lo-res graphics in *(x,y)* coordinates, visible in colors other than black.

```
PLOT 10,15
```

POKE
Inserts given value in specified memory location.

```
POKE 768,10    (Sets memory location 768 to decimal
                value 10)
```

POP
Used in GOSUB context, it removes top line number in the stack and makes the second-to-last line number the "return" point when the next RETURN will be encountered.

```
10 GOSUB 100
20 END
100 GOSUB 200
200 PRINT "HELLO"
210 POP
220 RETURN
```
(Results = HELLO; without POP, it would be HELLO HELLO.)

POS( )
Gives the current horizontal position of the cursor.

```
10 PRINT "THIS LINE";: PRINT POS(0)
```

POSITION
Used in sequential text files to begin reading files at specified position rather than at the beginning of the file (DOS).

```
10 D$=CHR$(13) + CHR$(4)
20 PRINT D$ "OPEN NAME FILE" : PRINT D$ "POSITION
   NAME FILE,R9" : PRINT D$ "READ NAME FILE"
```

PREFIX
Sets a directory as the current (prefix) one (ProDOS).

```
PREFIX /SORTS/SAMPLES/
CAT (Shows the files in SAMPLES
     now if directory is not specified)
```

PRINT
Outputs string, number, or variable to the screen or printer.

```
PRINT 1;2;3; "GO"; F$; A; N%
```

PR#
Sends output to specified slot number.

```
PR#1
```

READ
Enters DATA contents into variable.

```
10 READ A : READ B$
20 DATA 5, "BATS"
```


READ
Reads contents of text file (DOS).

```
40 PRINT CHR$(4) "READ NAME FILE"
```

RECALL
Loads array from tape that has been recorded with STORE.

```
RECALL Z
```

REM
Nonexecutable command. Allows remarks in program lines.

```
10 BELL$ = CHR$(7): REM RINGS BELL
```

RENAME
Renames files on disk (DOS).

```
RENAME FAST SORT, SHELL SORT
```

RESTORE
Resets position of READ to first DATA statement.

```
10 FOR I = 1 TO 5 : READ A$(I) : NEXT
20 RESTORE
```

RESTORE
Places the contents of a VAR file into the variable in memory in Applesoft programs (ProDOS).

```
RESTORE /FRIENDS/ADDRESSES
```

RESUME
Goes to statement where error has occurred in error-handling routine.

```
10 ONERR GOTO 50
20 INPUT V%
30 END
50 PRINT "ENTER ONLY INTEGER NUMBERS!" : RESUME
```

RETURN
Returns program to next line after GOSUB command.

```
500 RETURN
```

RIGHT$ ()
Returns the rightmost *n* characters of given string.

```
10 A$= "DATAMOST" : PRINT RIGHT$(A$,4)
```
(Results = MOST)

RND( )
Generates a random number less than 1 and greater than or equal to 0.

PRINT RND(5), INT (RND (1) * (N) + 1) (Generates whole random numbers from 1 to *N*, with *N* being the upper limit of desired numbers)

ROT=
Used to rotate shapes on hi-res screen. Can be set to angles between 0 and 255. Number of angular positions recognized depends on SCALE value of shape.

```
60 ROT= 64
```

RUN
Executes program in memory, or if filename is included, executes program from disk.

RUN FLYING MACHINE (Executes disk file)
RUN (Executes program in memory)
150 PRINT CHR$(4); "RUN FLYING MACHINE" (Program mode)

SAVE
Records program on disk.

```
SAVE GRAPH PLOT
```

SCALE=
Specifies size or scale of shape on hi-res screen.

```
40 SCALE= 5
```

SCRN()
Returns the color code (0–15) in lo-res graphics of plot at *(x,y)* coordinates.

```
30 PRINT SCRN (10,20)
```

SHLOAD
Command for loading shape table from tape (DOS 3.3).

```
SHLOAD
```


SIN( )
Returns the sine of a variable or number.

```
PRINT SIN(123)
```

SPC( )
Prints specified number of spaces.

```
PRINT SPC(29); "HERE"
```

SPEED
Sets speed of execution from 0 to 255 (default is 255).

```
50 SPEED = 100
```

SQR( )
Returns the square root of a variable or number.

```
PRINT SQR(64)
```

STOP
Halts execution and prints line number where break occurs. (The CONT command will restart program at next instruction after STOP command.)

```
100 STOP
```

STORE
Records array values on tape (DOS 3.3).

```
STORE Z
10 STORE Z
```

STORE
Writes variables currently in memory to VAR file to be recovered with RESTORE (ProDOS).

```
STORE FRIENDS/ADDRESSES
```

STR$( )
Converts number/variable into string variable.

```
20 T= 123 : T$= STR$(T) : TT$= "$" + T$ + ".00"
```

TAB( )
Sets horizontal tab from within a PRINT statement.

```
PRINT TAB(20);"HERE"
```

TAN( )
Provides the tangent of a number or variable.

```
40 T = 34 : V = 55
50 R = T + V : PRINT TAN(R)
```

TRACE
Turns on TRACE function for display of program execution (turned off with NOTRACE).

```
TRACE
```

UNLOCK
Removes LOCK status from file on disk, allowing it to be removed with DELETE statement or to be overwritten (DOS).

```
UNLOCK TAX CHART
```

VAL( )
Used to convert string to numeric value.

```
30 H$ = "123" : PRINT VAL(H$)
```

VERIFY
Examines disk file to check for errors. If there are no errors, nothing happens (DOS).

```
VERIFY TAX CHART
```

VLIN AT
Draws vertical line in lo-res graphics AT given horizontal position.

```
50 VLIN 1,30 AT 10
```

WAIT
Stops execution until memory location values meet given conditions.

```
80 WAIT -16384,128
```

WRITE
In making text files, specifies filename to which following PRINT statements will be written (DOS).

```
50 D$ = CHR$(13) + CHR$(4)
60 PRINT D$"OPEN NAMES": PRINT D$"WRITE NAMES"
70 PRINT "TOM" : PRINT "DICK" : PRINT "HARRY"
```


XDRAW
Used with shape tables to draw in complementary color a specified shape on hi-res screen. Similar to DRAW.

```
70 XDRAW 3 AT 150,55
```

Index

$. *See* variable, string
%. *See* percent sign
:. *See* colon
?. *See* PRINT
"Address labels" program 184–85
AND 75
animation 136
APPEND 166
Apple publications 206–7
Applesoft 15, 25
Applesoft BASIC 33, 131
  token chart 229–32
Apple user groups 205–6
arrays 81–86
  multidimensional 84–86
ASCII 113–14
  characters 233
  hexadecimal/decimal lists 217–18
BAS files 18
BASIC program 4, 207, 209–10
baud rate 13
BINARY.FINDER 164
BIN files 18
BLOAD 113
blocking 78, 101
blurred text 132
branching 71–87
  computed branches 78
BRUN 113
BSAVE 113
  binary file 124–25
  graphics 145
business programs 222–23
bytes 4
"Calendar plot" program 216–17
CALL 113, 123–24
CAT 17
CHAIN 162–63
CHR$ 113–15, 217
circles 142
CLOSE 160
colon 24, 58–59
color 131–32, 140, 186
  in QuickDraw II 197
comma 58–59
compiler 210
computer languages 207–10
  assembly 207
  BASIC 207, 209–10
  C language 209
  COBOL 209
  Forth 208–9
  FORTRAN 209
  Logo 209
  Pascal 208
concatenation 99–100
control characters 114–16
control key 22
controller card 6
control panel 12–14, 25
counters 67–68
DATA 62–63
database programs 220–22
data manipulation 104–5
DEL 41
Desktop 16
desktop publishing 9
Digital Oscillator Chip (DOC) 200
DIM 83
"Disk Locator" program 213–15
disks 7, 224
  booting 15
  booting with DeskTop 16
  booting without Desktop 17
  formatting 16
  formatting without Desktop 18–20
  installing drive 6
Display 26
DOS 15
DRAW 153
editing 40–46
editor/assemblers 207–8
END 34, 60
error messages 40, 102, 241–43
esc key 21
EXEC 160
EXEC files 159–63
  automatic commands 163
exiting 103
FILE.MANAGER 168–70
FLASH 64
fonts 190
FOR 64
formatting text 91–92
GET 61, 125–26
GOSUB 77
  computed 79–80
GOTO 72, 102
GR 131
graphics 131–54
  commercial programs 223–24
  drawing board program 141–42
  drawing in high resolution 138–39
  high-resolution 131, 137
  low-resolution 131, 134, 136
  preserving screen 137–38
  super high resolution 195
hardware 4
HELLO program 21
hexadecimal 118, 148–50
  conversion chart 235–39
HGR 131, 138
HGR2 131
HOME 34
HPLOT 138
HTAB 91
IF 72
ImageWriter 28, 179
  control codes 183
immediate mode 33–34
INPUT 60
Integer BASIC 25
INVERSE 64
joysticks 224
keyboard 21–24
LaserWriter 187, 209
LEFT$ 95–96
lines 132–35
LIST 36
loops 64–68
  nested 65–67
math operations, elementary 46–47
memory 113–14, 117
  accessing 114
  locations 113
MID$ 95–96
modem 12–13, 189, 224–25
modem port panel 29
monitors 7
  types 8
  versus television 7
mouse 13
moveto statement 188
music programs 224
NEW 72
NEXT 64, 67
NORMAL 64
NOT 75
numbering programs 36
nybble 149
ON 78
ONERR 101–2
OPEN 160
Options 26
OR 75
padding 103
paddles 93, 122
  formatting with 93
  labeling 122
PEEK 103, 113, 117–18, 122–23, 127
percent sign 24, 188
PLOT 134
POKE 43–44, 113, 117–23, 127, 131, 143
POSITION 167
PostScript 187, 209
precedence 47–48
PRG.LISTER 181
PRINT 24, 33, 35
printer
  control codes 181–83
  port panel 28
  test program 15
printers 8–11, 179–90
  dot-matrix 9
  graphics 186
  ink-jet 10
  laser 9–10
  letter-quality 9
  plotter 10
  thermal 10
printing photographs 190
PRINT LEN(A$) 94–95
ProDOS. *See* DOS
"Program header" program 215–16
program mode 34
"Quick Apple-Betize" program 210–11
QuickDraw II 195–99
  calls 195–99
RAM 4–5, 11
RAM cards 12
RAM drive 12
random access files 159, 170–71
random numbers 139
READ 62–63
recalling programs 38
relationals 73–81
  strings and 81
REM 35
reset key 22
RIGHT$ 95–96
RND 139
ROM 4–5
ROT 153

RUN 17, 38–39
SAVE 37–38
SCALE 153
scan lines 197
scroll control 108
semicolon 58–59
sequential text files 159, 164–70
setting traps 101–2
shapes 146–54
  manipulation of 153
showpage statement 188
slots 11
software 4
"Sort menu" program 212–13
sound 27, 122–23, 199–201
  registers 200–201
SPC(N) 91
SPEED 97
string 24, 57, 93–100, 104
  formatting 93–100
  length 94–95
  manipulation 104
  rationals and 81
?SYNTAX ERROR 40
system speed 27–28
TAB(*N*) 91
TEXT 34
text files 159–74
  converting BAS files into 160
TEXT:HOME 120
THEN 72
toggle 144
toolboxes 195
TXT files 18
UNLOCK 18
utility programs 218–19
VAR files 18
variables 53–58
  string 24, 57
  types 56–57
VLIN 132–33
VTAB 91
WAIT 113, 125–26
windows 119–20
word processors 219–20
WRITE 160
XDRAW 153
zero character 184