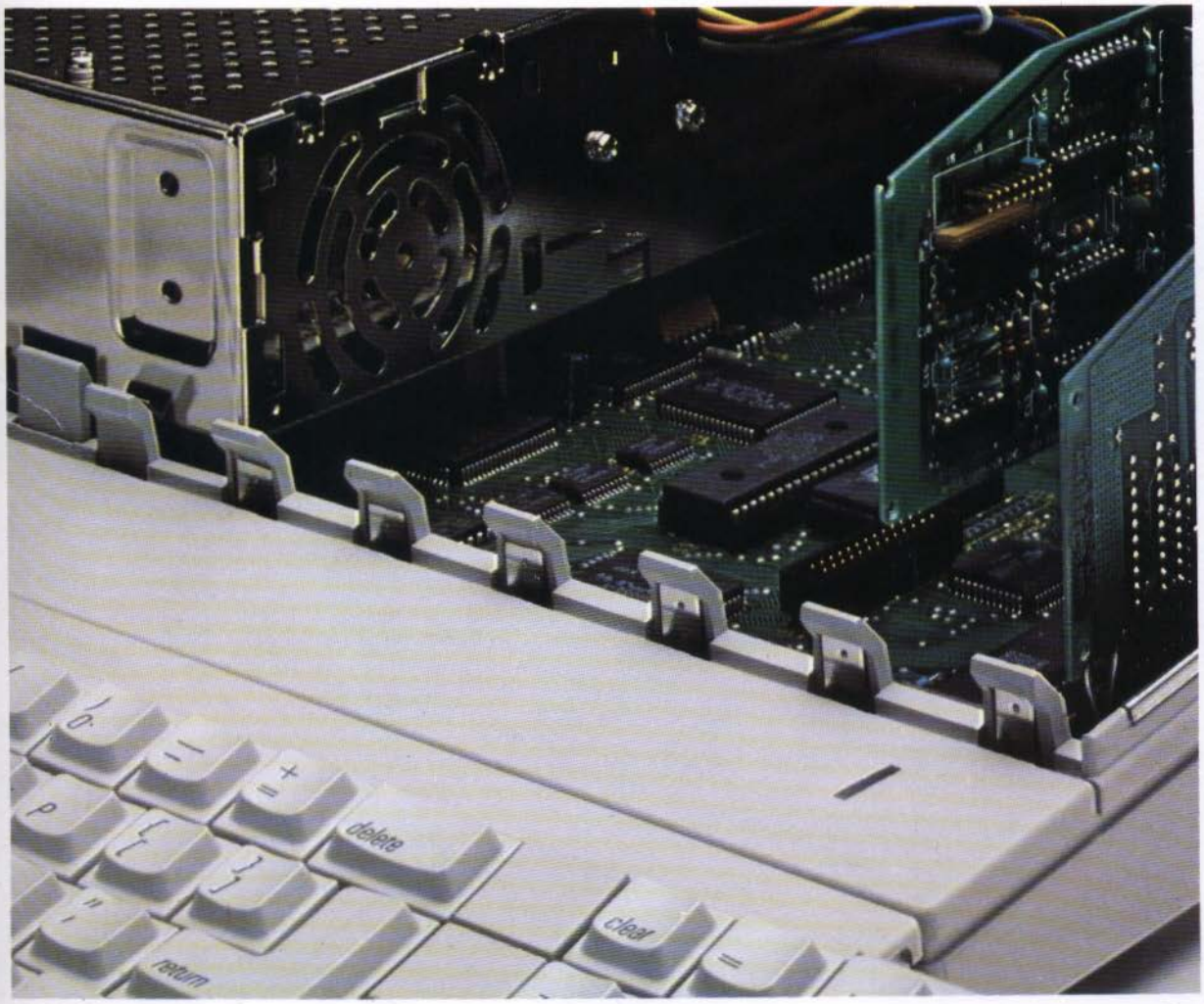
BANTAM COMPUTER BOOKS

THE Apple® IIGS Toolbox Revealed

Danny Goodman



The Apple IIGS Toolbox Revealed

Bantam Computer Books
Ask your bookseller for the books you have missed

THE AMIGADOS MANUAL
by Commodore-Amiga, Inc.
THE APPLE IIGS BOOK
by Jeanne DuPrau and Molly Tyson
THE APPLE //c BOOK
by Bill O'Brien
THE ART OF DESKTOP PUBLISHING
by Tony Bove, Cheryl Rhodes, and Wes Thomas
ARTIFICIAL INTELLIGENCE ENTERS THE MARKETPLACE
by Larry R. Harris and Dwight B. Davis
THE BIG TIP BOOK FOR THE APPLE II SERIES
by Bert Kersey and Bill Sanders
THE COMMODORE 64 SURVIVAL MANUAL
by Winn L. Rosch
COMMODORE 128 PROGRAMMER'S REFERENCE GUIDE
by Commodore Business Machines, Inc.
THE COMMODORE 128 SUBROUTINE LIBRARY
by David D. Busch
COMPUTER SENSE: *Developing Personal Computer Literacy*
by Paul Freiberger and Dan McNeill
EXPLORING ARTIFICIAL INTELLIGENCE ON YOUR APPLE II
by Tim Hartnell
FIRST STEPS IN ASSEMBLY LANGUAGE FOR THE 80286
by Robert Erskine
HOW TO GET THE MOST OUT OF COMPUSERVE, 2d edition
by Charles Bowen and David Peyton
HOW TO GET THE MOST OUT OF THE SOURCE
by Charles Bowen and David Peyton
MACINTOSH C PRIMER PLUS
by The Waite Group / Stephen W. Prata
THE IDEA BOOK FOR YOUR APPLE II: *How to Put Your Apple II to Work at Home*
by Danny Goodman
PC-DOS / MS-DOS
User's Guide to the Most Popular Operating System for Personal Computers
by Alan M. Boyd
SMARTER TELECOMMUNICATIONS
Hands-On Guide to On-Line Computer Services
by Charles Bowen and Stewart Schneider
UNDERSTANDING EXPERT SYSTEMS
by The Waite Group / Mike Van Horn

The Apple IIGS Toolbox Revealed

Danny Goodman

BANTAM BOOKS
TORONTO • NEW YORK • LONDON • SYDNEY • AUCKLAND

THE APPLE IIGS TOOLBOX REVEALED
A Bantam Book / November 1986

ISBN 0-533-34360-2

Published simultaneously in the United States and Canada

Bantam books are published by Bantam Books, Inc. Its trademark, consisting of the words "Bantam Books" and the portrayal of a rooster, is registered in U.S. Patent and Trademark Office and in other countries. Marca Registrada. Bantam Books, Inc., 666 Fifth Avenue, New York, New York 10103.

PRINTED IN THE UNITED STATES OF AMERICA

0 9 8 7 6 5 4 3 2

Acknowledgments

Many folks associated with Apple Computer helped in the writing of this book. The people I wish to thank most of all are those who devoted their own time over and above their own frenetic deadlines to answer questions, review manuscripts, and offer other guidance along the way. Tied for first place in lending helping hands were Martha Steffen and Eagle Berns. Harvey Lehtman and Dan Oliver also came through without flinching. Bill Harris provided the right ideas at the right time to solidify the book's organization. And special thanks to Jim Merritt for providing superb technical guidance in the manuscript's final stages. Off the Apple campus, Linda's undying patience and understanding made the job possible.

Contents

A Primer Road Map

The Apple IIGS is the most recent step in an evolutionary climb that dates back to the earliest days of personal computers for everyday people. The IIGS is at once compatible with its past and representative of the future. You can run thousands of programs and plug in hundreds of boards already available for earlier generation Apple IIs. At the same time, by offering casual Apple II programmers and serious developers built-in programming power matched only by the Macintosh, the IIGS paves a path to new generations of innovative and truly user-friendly software designs.

The programming power inside the Apple IIGS consists of a carefully crafted programmer's toolbox, which simplifies the design of sophisticated Apple II programs, for both newcomers and experienced programmers.

This book provides an introduction to concepts crucial to learning to program the Apple IIGS by way of its toolbox. Along the way we'll discuss principles of program design to help you start visualizing your applications right now. We won't limit our discussions to a single programming language. In fact, we'll even help you choose a language if you're still undecided. We'll compare C, Pascal, and assembly language, since these will probably be the most popular languages for toolbox programming. Later in the book, a few program examples will illustrate toolbox operations in a pseudolanguage that closely resembles both C and Pascal.

Whether you're brand new to programming on the Apple II or if you come to the toolbox as an experienced programmer, this book is where you

should begin your IIGS toolbox experience. Start your toolbox explorations in the chapter appropriate to your expertise:

1. If you are brand new to programming or if you are an experienced programmer only in BASIC on any computer, then start with Chapter 1.
2. If you have programmed earlier models of Apple II in Pascal, C, or assembly language, then you can skip Part One and head right for the toolbox discussions, beginning in Chapter 4. Of course, if you're rusty, it wouldn't hurt to start at Chapter 1, skimming over the parts you know and studying the subjects that need brushing up on.
3. Some experienced Macintosh programmers may also come by, hoping to see how the Apple IIGS toolbox is different from the Mac's. For you, Chapter 8 is the place to start.

No matter where you begin, bear in mind that this is a book about programming *concepts*. Aside from a little hands-on exploration in the early chapters, you won't be needing your IIGS nearby. It's more important that you grasp the ideas presented here. You will encounter them again, and have plenty of hands-on experience when you start real programming.

There are hardly any prerequisites for understanding this book. We recommend, however, that you spend time familiarizing yourself with the visual orientation of the Apple IIGS Finder. Notice how you work with screen menus, window, and icons. The Finder demonstrates the fundamental "feel" of the highly graphical interface that toolbox programming promotes. For you it may well be a new way of interacting with a computer. We also suggest that you acquaint yourself with the Control Panel, which you can summon from the keyboard at any time by pressing the Apple, Control, and Escape keys simultaneously.

That's all you'll need to know to begin your lessons with *The Apple IIGS Toolbox Revealed*.

Part One

Programming Fundamentals — A Crash Course In Plain English

CHAPTER 1

Under the Hood

This chapter and the others in this part of the book are intended for two types of readers: (1) those who have never programmed a computer before, and (2) those who have experience programming a personal computer in the BASIC programming language. You may think it odd that we've placed an experienced BASIC programmer in the same classroom with the complete neophyte. BASIC programmers, however, will see in the next couple chapters that the BASIC language has hidden many fundamental computing concepts. If you are a Pascal or C programmer, you will likely have worked with most of these concepts, but if it has been a while since you last typed some code, then feel free to tag along as we start by peeking under the hood of the Apple IIGS.

THE ENGINE

Whenever you open the hood of your car, you can't help seeing its main component, the engine. It is at the physical center of the engine compartment. Surrounding components attached to it perform the tasks needed to make the car move. The engine is also the functional center of the automobile, regulating such things as battery recharging, vehicle speed (in response to the press of the gas pedal), the amount of exhaust going to the tail pipe, and so on.

CPU

All computers — whether a desktop model like your Apple IIGS or a giant mainframe computer that a credit card company uses to generate monthly bills — need an engine to regulate the workings inside the machine. The computer engine, however, doesn't have any moving parts (although invisible electrons move through it) and is small enough to get lost in a desk drawer. It consists of a single *integrated circuit* chip. A common name for this chip, derived from its primary function, is the *central processing unit*, or *CPU*. This kind of chip is also frequently called a *microprocessor* because it is, in a sense, a self-contained computer on a single chip. You can locate the CPU chip on your Apple IIGS main circuit board (the *motherboard*) by using Figure 1-1 as a reference.

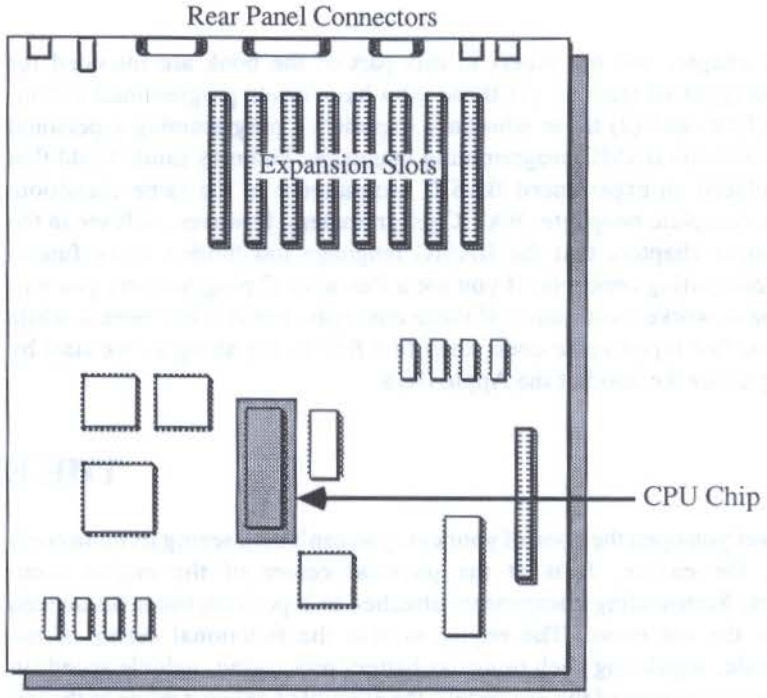


Figure 1-1. Apple IIGS motherboard and CPU chip.

The 65816 Chip

If you look closely at the actual CPU chip on the motherboard, you'll see many cryptic numbers on its top. Among them you'll find the part number, 65SC816. This part, generically known as the 65816, is an advanced and greatly enhanced version of the 6502 CPU, around which all Apple II computers have been built since the very beginning. The Apple IIe, for example, has a 6502 on its main board, while the Apple IIc has a low-power consumption version of that chip, called the 65C02, on its main board. Among the many enhancements built into the 65816 are increased processing speed and the ability to work with far more information than the 6502 was ever meant to manage. These enhancements give the Apple IIGS the power it needs for super high-resolution color graphics, responsive mouse control of a screen cursor, and many more desirable features to aid user and programmer alike.

We said that a microprocessor is essentially a "computer on a chip." That phrase grew out of a long history of gradually combining the abilities of more and more integrated circuits into fewer and fewer chips. While a microprocessor like the 65816 performs the tasks that a personal computer's central brain needs, it is hardly an entire computer that we could use directly. Just as a car engine needs a water pump, a battery, and other components to work, a microprocessor needs extra integrated circuit chips to allow us to communicate with the computer via the keyboard or mouse and to allow the computer to communicate with us via the screen or speaker. Additional circuitry manages the movement of information to and from disk drives, printers, and modems. Most of the other chips you see on the IIGS motherboard are those support chips.

MEMORIES

Among the other chips on the motherboard are two types of memory chips, called *RAM* and *ROM*. Each type performs a distinctly different function in the IIGS.

RAM

RAM stands for *Random Access Memory*. You'll find these chips arranged in two groups on your IIGS motherboard (Figure 1-2).

These chips store information while the computer is turned on. The kind of information kept there includes: the contents of the video display screen, certain numbers that the microprocessor needs for its own housekeeping, characters you type on the keyboard or load in from disk, and programs (as

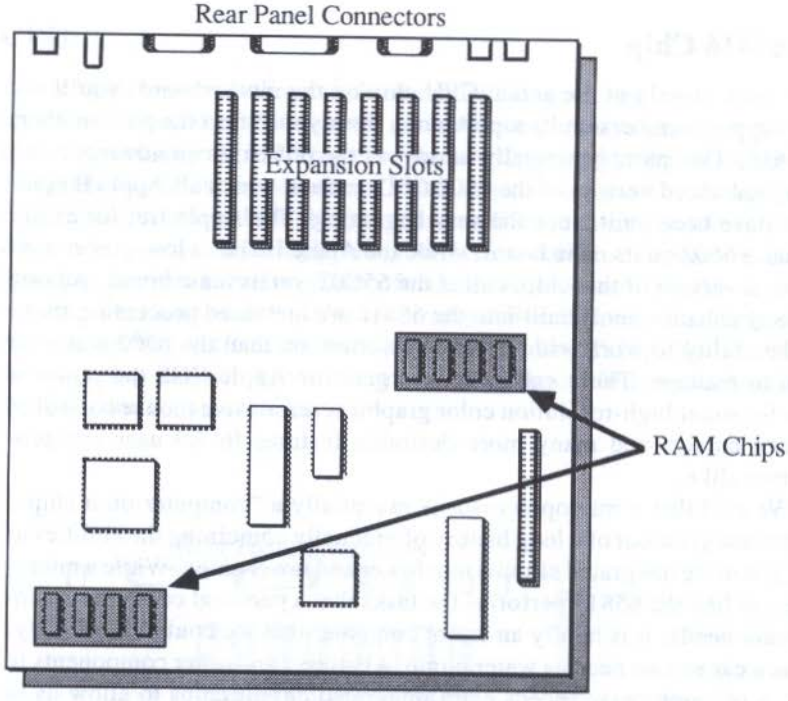


Figure 1-2. Apple IIGS motherboard and RAM chips.

you type them in or run them). We'll get into how these chips store information in the next chapter.

Reading and Writing Memory

As you run an applications program, such as a word processor or a spreadsheet, the content of the RAM chips is constantly changing. Practically any action that takes place causes information to be written to or erased from RAM. Even if you can't see explicit action on the screen, such as when a program is calculating, the CPU is frenetically writing, reading, and rewriting RAM.

(For the sake of accuracy, the term "Random Access Memory" does not accurately describe this kind of chip, because all types of memory chips can have their information accessed in a random fashion. That is, the microprocessor can fetch a character from a specific space in a memory chip without "thumbing through" the contents of other spaces. The precise name

for the kind of memory chips we're discussing here is Read/Write Memory, or RWM. That means that the microprocessor can both fetch information from a specific spot in a chip and put information there, too. Tradition, however, dictates the general acceptance of the term "RAM", so we'll use it here, too.)

RAM: The Fuel Tank

The amount of RAM inside a computer is an important measure of the machine's capabilities. The more RAM available to a programmer, the more sophisticated or complex programs can be. Lots of RAM also makes it possible to work efficiently on very large word processing documents, spreadsheets, or databases. With enough memory, all necessary data can be stored for instant, electronic recall rather than electromechanical recall of small chunks from a comparatively slow disk drive. You can say, then that RAM capacity is analogous to the size of the fuel tank on an automobile. The larger the capacity, the more you'll be able to do with the machine, and the more work you'll be able to do without refueling.

Apple IIGS users should have little trouble accommodating early IIGS commercial programs in RAM supplied with the machine. Eventually, however, commercial developers will exploit the design opportunities offered by the computer's color graphics, sound, and inexpensive RAM chips to build sophisticated programs requiring a RAM expansion card.

ROM

There's one last key component of the computer's memory circuitry that you should know about. It's called *Read-Only Memory*, or *ROM*.

If you recall what we said earlier about a RAM chip's reading and writing abilities, then the ROM's characteristics should be obvious from its name: information in ROM is for reading only. Neither you nor the microprocessor can alter the contents of ROM.

ROM's Role

While the microprocessor is a powerful chip in its own right, it doesn't really know what to do when you supply electricity to it. It's a bundle of potential energy waiting for something to do. The ROM contains a fixed list of instructions that the microprocessor follows, starting the instant you turn on the computer. In the first few seconds your IIGS comes to life, the ROM shows the microprocessor how to set up the various components on the motherboard so that the combination of chips will act as a IIGS — how information is to be displayed on the screen, how to react to presses of the

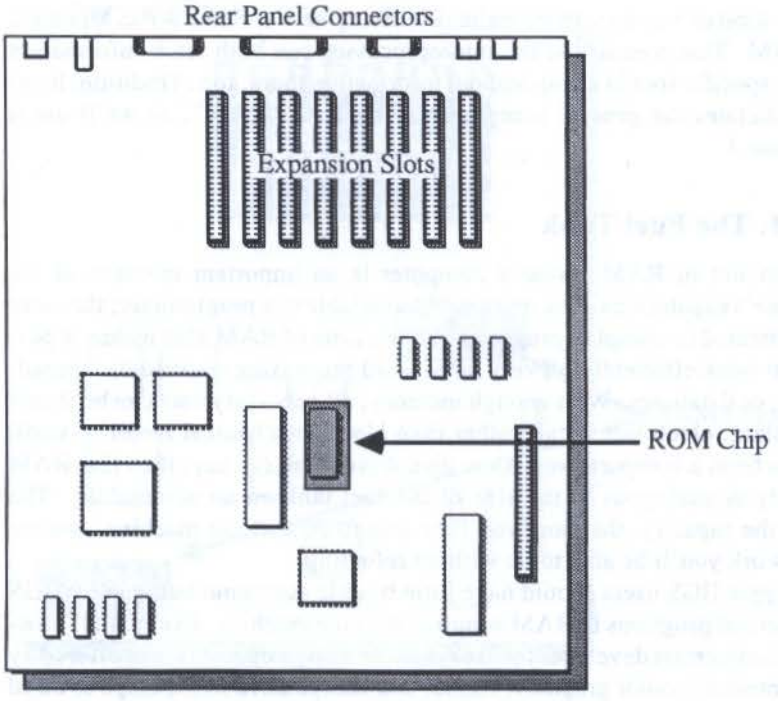


Figure 1-3. Apple IIGS motherboard and ROM chip.

keyboard keys, how to respond to a diskette that has been placed in one of the disk drives, and more.

The Apple IIGS ROM has much more in it than just the start-up instructions for the system. It's also where a large portion of the programmer's toolbox is located. That's right. Much of what programmers need for putting together fancy-looking, Macintosh-like programs is contained in that lone chip. We'll have much more to say about the tools and ROM in later chapters.

COMPATIBILITY

When you purchased your Apple IIGS, quite likely one of the selling points that clinched the deal was the machine's compatibility with most software already available for the Apple IIe and IIc. For two computers to be compatible with each other, many specific items about each machine's design must

be identical. Video display characteristics of both machines, for instance, must be the same. Since software written for one machine expects to display a well-defined number of dots on a video screen, a computer that doesn't have the requisite number of dots may not produce the program's video output at all. Disk drives, too, must be identical in the way they write information onto a disk and read information from it. While to the naked eye the 5¼-inch floppy diskette for an Apple IIc looks like a 5¼-inch diskette for an IBM Personal Computer, each machine encodes information on a disk much differently from the other. Neither machine would even recognize that a diskette recorded for the other machine had information on it.

Compatibility issues, however, reach much more deeply inside a computer. For one, ROM instructions must be sufficiently equal so that important systemwide functions, such as display characteristics and information flow in and out of the computer's connectors, are the same. Even more fundamentally, the two machine's microprocessor chips must be functionally identical. Each chip must respond to instructions the same way.

Two Modes

We noted earlier that previous Apple II computers were based on the 6502 microprocessor, while the 65816 is at the core of the IIGS. One major reason the IIGS is compatible with earlier Apple II software is that the new chip can act as if it were a 6502. When it behaves in this manner, it is said to *emulate* the 6502. When it behaves like a full-blown 65816, it operates in what is called *native mode*.

Emulation mode programming gets a boost by another chip on the IIGS motherboard. Called the Mega II, the chip was designed by Apple's engineers to place on a single chip as much of the "old" Apple II as possible. The Mega II chip provides the IIGS with Apple IIe- and IIc-compatible video display resolutions and controls communications to outside devices through expansion slots and rear panel connectors (called *ports*). As a programmer, however, you won't have to know much about the Mega II, because the 65816 (under the guidance of ROM) automatically knows when to put it in use, in both emulation and native modes.

When you program for the Apple IIGS, you have the choice of programming the machine either in its native mode, which this book emphasizes, or in emulation mode. Advanced programmers also have the choice of mixing modes by switching from one to the other and back in the middle of a program, if necessary. Toolbox programming, though, is entirely in native mode. Information on programming in emulation mode is readily available, because it is identical to programming an Apple IIe or IIc, both of which are abundantly documented in other books.

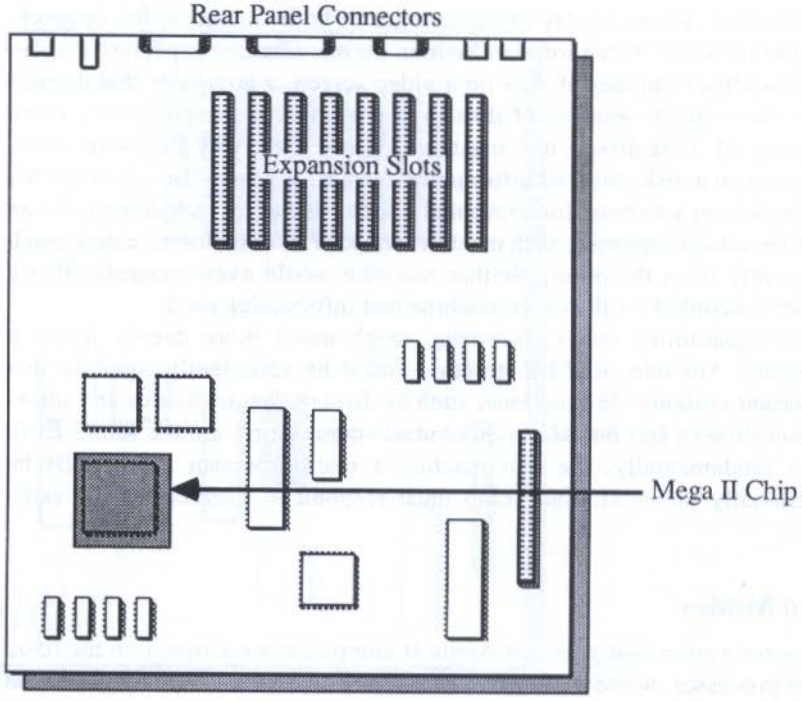


Figure 1-4. Apple IIGS motherboard and Mega II chip.

PROGRAMS: THE COMPUTER'S ROAD MAPS

So far, we've been talking mostly about the pieces that define a machine and its capabilities. Assembling a fine automobile from components, though, does not necessarily make it a productive machine. It needs someplace to go. Similarly, without a program to run, a computer, as defined by its CPU, ROM, and other chips, doesn't do a thing. You could, then, compare a program to a road map for the automobile.

Step by Step

A computer program is nothing more than a series of instructions for the microprocessor and its related chips to follow. A word processing program, for example, tells the machine to display characters you type on the keyboard so they are arranged into neat sentences and paragraphs on the screen. Other instructions tell the computer what to do when you press the mouse button

when the cursor is in a certain place on the screen. Still other instructions calculate the number of lines on a page until it is time to display a dotted line representing a page break, and begin counting lines for the next page.

The programs you'll be writing will probably be stored on either a floppy disk or a hard disk. When you wish to run a program, a command — perhaps a command you type or an action by the mouse-controlled pointer — will load a copy of the disk's program into the machine's RAM. Then the microprocessor will start obeying the instructions, one by one. Some of your instructions will cause the microprocessor to reach over into ROM for some further instructions that Apple's engineers put there for convenience, as you'll see later.

Program execution will continue until you give the stop or quit command written into your program. When the microprocessor receives that command, it instantly returns you to the screen from which you started. Even though your program may no longer be in memory, it is still safe and sound on the disk. It's ready to run again whenever you need it.

In the next chapter, we push even deeper inside the computer. You'll learn the true meaning of many buzzwords hurled at you since you first heard about personal computers. Get ready to think small.

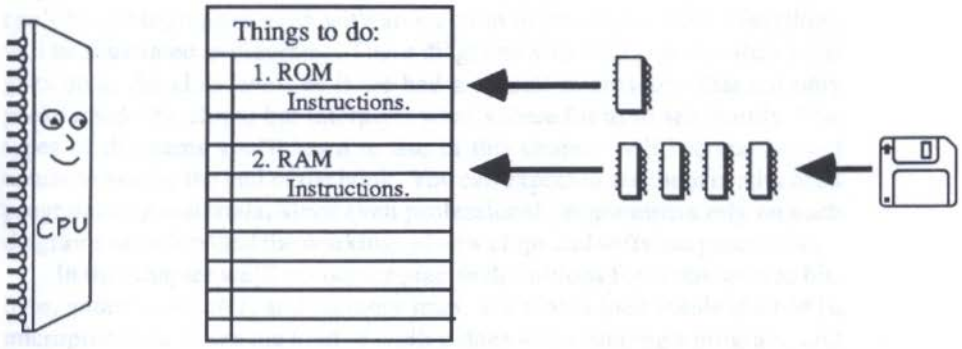


Figure 1-5. The CPU follows ROM and RAM instructions step by step.

CHAPTER 2

Under the Microscope

Until now, we've been discussing physical pieces of the IIGS — things you can see when you lift off the computer's cover. Now we'll go one giant step further by explaining key concepts about what occurs inside the chips on the motherboard. Because much of what you'll learn in this chapter can't be photographed even with an electron microscope, most everything will be illustrated in diagrams. These diagrams will help you visualize what goes on at the chip level, as if we had a special microscope that not only peeks inside the chips, but interprets what's there for us to see plainly. The types of diagrams you'll learn to use in this chapter will become second nature to you by the end of the book. You can expect to see them in advanced programming materials, since even professional programmers rely on such diagrams to understand the workings of new chips and software procedures.

In this chapter we'll encounter precise definitions for terms such as bit, byte, address, pointer, and memory map. We'll also look inside the 65816 microprocessor to see the kind of work it does when running a program, and learn the meanings of terms such as stack, pointers, and flags.

GOING WITH THE FLOW

When you look at the Apple IIGS motherboard, you probably notice that each chip has a number of connections coming from it. These connections, called *pins*, are attached to the motherboard in particular spots so that low-

voltage electricity can pass from one chip to the next. You might say that electricity is the computer's blood, the wiring its veins.

Electricity with a Message

The electricity, however, is used as a way of communicating information between components on the motherboard and to slots and ports connecting to the outside world. The language of this information is about as simple as you can get, with a vocabulary of only two entries. One entry is represented in the circuits by a high voltage level; the other is represented by a low voltage level. The terms "high" and "low" are relative to each other, and don't indicate that the high voltage is necessarily dangerous. In fact, the high-voltage signal is usually about as high as that generated by a couple of flashlight batteries, while the low voltage is usually less than a quarter of that.

CAUTION: Just because the high voltage in the chip communications lines is only a few volts, this does not mean that it is safe to stick your hands inside the computer when the power is turned on. There may be places on the circuit board that handle enough *current* to give you a shock. Also, owing to the inherent conductivity of human skin, touching the closely spaced pins on some chips may cause unexpected short circuits, which may have one or two undesirable results: (1) an abnormally high current flow into your finger, causing a shock, and (2) a blow to one or more chips on the motherboard, necessitating replacement of the entire board. Therefore, always turn off the computer before removing the cover.

The designers of the 65816 microprocessor imbedded instructions into the chip so that it performs very specific actions in response to precise sequences of these low- and high-voltage signals. You'll see examples of this later on, but for now, take it on faith that the microprocessor is "born" understanding this two-entry vocabulary.

Human Notation

The low-high notation is a bit cumbersome for us humans, so to make things easier, the low and high entries in the chip's vocabulary are represented by the digits 0 and 1, respectively. This is illustrated in Figure 2-1. From now on, we will use this 0 and 1 notation when discussing individual voltage pulses flowing through the computer.

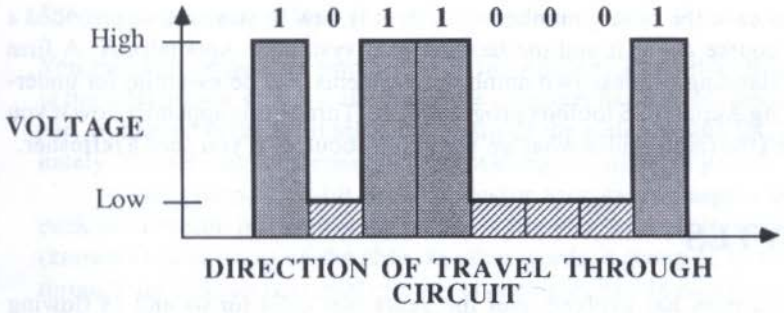


Figure 2-1. Low- and high-voltage flow with their 0 and 1 representations.

Binary Numbering

Through no accident, the 0 and 1 components of a microprocessor's vocabulary are also the two digits that make up the binary numbering system. In binary math there are no such numerals as 2, 3, or all the rest up to 9 — just 0 and 1. Instead of columns of numbers associated with ones, tens, hundreds, and so on (each increasing by a factor of 10 — hence the name of our *deci*mal math), a binary number's columns increase by a factor of 2. In Figure 2-2, compare the different ways the decimal and binary numbering systems represent the decimal number 195.

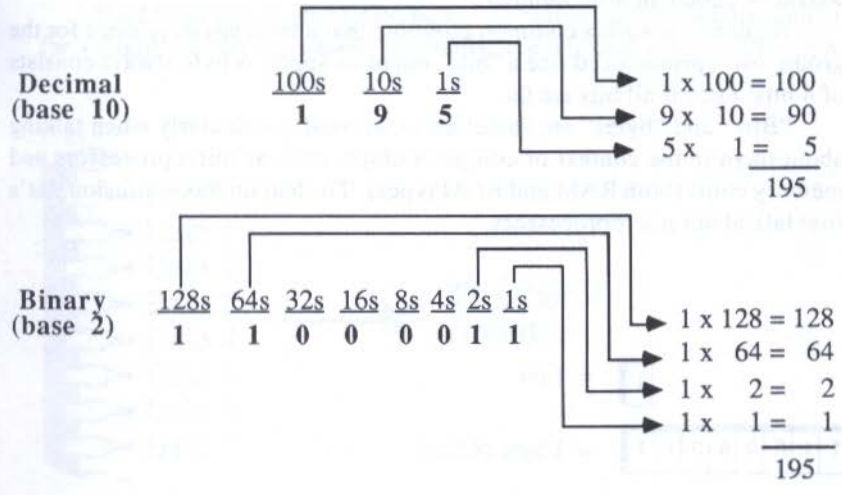


Figure 2-2. Decimal and binary representations of the decimal number 195.

In case the binary numbering system is new to you, we've provided a short course about it and the hexadecimal system in Appendix A. A firm understanding of these two numbering systems will be essential for understanding Apple IIGS toolbox programming. Turn to this appendix now if you haven't the faintest idea what we're talking about, or if you need a refresher.

BITS TO BYTES

A convention has evolved over the years that calls for 0s and 1s flowing through a computer to be grouped in batches of 8. New microprocessors, including the 65816, can deal with these pulses in groups of 16, but you will still work with them often in groups of 8. A sequence of 8 digits would look something like this:

0101 1110

The extra space between the two groups of 4 is generally provided as a means of making the long binary number more readable. Placing 8 *binary digits* together, such as 01011110, makes it harder for us to figure out exactly what number it is, although the CPU is quite content with the unbroken series.

Now, the term we just used, *binary digit*, contains a lot of syllables to describe a single 0 or 1. In common usage, these two words have been combined into one short one, *bit*. A bit is the smallest unit of information that a computer deals with. In computer jargon, the binary number shown above would be called an 8-bit number.

Eight bits is such a common grouping that a term has developed for the group: *byte*, pronounced like a "bite" out of an apple. A byte always consists of 8 bits, even if all bits are 0s.

"Bits" and "bytes" are sometimes confused, particularly when talking about them in the context of computer chips, such as microprocessors and memory chips (both RAM and ROM types). To clear up this confusion, let's first talk about microprocessors.

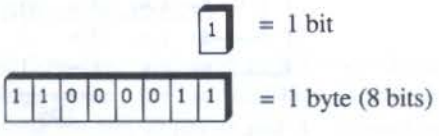


Figure 2-3. Comparative length of a bit and a byte.

Microprocessor Bits

You probably remember somewhere in the Apple IIGS product literature or in the salesperson's pitch a claim that the computer has a 16-bit microprocessor, compared to the 8-bit microprocessor of the earlier Apple IIs. Unfortunately, this terminology means different things to different people.

To one group, a 16-bit microprocessor accepts, massages, and sends back information in 16-bit-wide paths — literally through 16 separate pins (known as *data lines*) on the chip. In other words, a group of 16 bits passes through the CPU and external circuits at one time, like 16 race horses taking off from the starting gate at the bell. An 8-bit microprocessor, then, sends and receives information only 8 bits at a time. With twice as much information passing through a 16-bit computer at a given instant than through an 8-bit computer, processes generally operate much faster, when both machines operate at the same speed.

To another group of terminology makers, however, the bit rating of a microprocessor is measured by the width of information flowing only inside the microprocessor, regardless of the number of data lines connected to outside chips. For example, the model 8088 chip in the IBM PC connects to the rest of the computer via eight data lines, while inside the chip, information shuffles about in 16-bit-wide chunks. That means that certain operations, such as simple arithmetic built into the chip, will be performed in the faster 16-bit mode. Communicating the result of the arithmetic to the rest of the computer, however, will be done in only 8-bit-wide chunks. The designation for a chip operating at 8 bits externally and 16 bits internally is an *8/16-bit* microprocessor.

The 65816 inside your IIGS, however, claims its 8/16-bit denomination for a slightly different reason. Internally, the chip can run as either an 8-bit

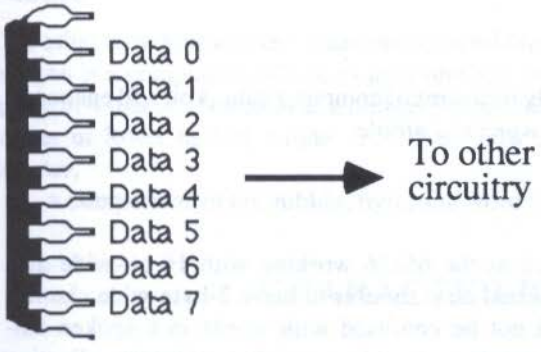


Figure 2-4. A microprocessor's eight data lines.

(emulation mode) or a 16-bit (native mode) microprocessor. Its link to the outside world is via 8 data lines. A trend seems to be developing to ignore the data line count and refer to a microprocessor only according to its best internal capabilities. Hence, the 65816 is commonly called a 16-bit chip.

RAM Chip Bits

Bit terminology is also used to describe the capacity of memory chips — not the amount of the computer's memory specified in the product literature, but of the actual chips themselves. Chips are normally rated by the number of kilobits of information they can store. Strictly speaking, a kilobit is 1024 bits. The "kilo" prefix, which should mean 1000, means 1024 in any computer measure (since computer stuff is calculated in binary, 1024 [2^{10}] is the factor of 2 nearest to 1000).

RAM Bytes

The RAM chips inside the IIGS are 256-kilobit chips, commonly labeled 256K chips. If each chip holds 256 kilobits, that means each holds 32 kilobytes (remember, 8 bits for each byte). To bring the computer's total onboard RAM to 256 kilobytes (that's the measure on the specifications sheets), Apple had to plant eight 256 kilobit chips, which it did in two separate banks on the motherboard.

Since kilobits and kilobytes are both abbreviated by the letter "K", be careful to identify what kind of K you're looking at. Chips are measured in bits; system memory as a whole is measured in bytes. Keep your bits and bytes straight.

OTHER MEASURES

Bits and bytes aren't the only measures of computer data. You will encounter two other important terms: *word* and *nibble*.

Word

With a microprocessor such as the 65816 working with 16-bit-wide data internally, there is a convenient term to refer to these 2-byte-wide chunks: *word*. This "word" should not be confused with words in a spoken language. A word inside a computer is simply any 2-byte (16-bit) collection of data.

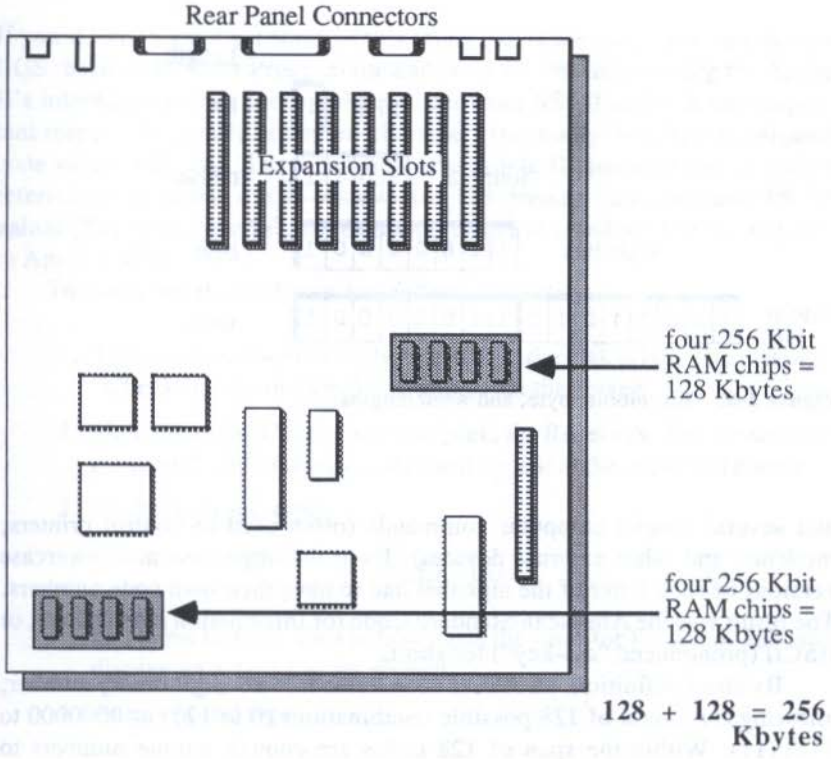


Figure 2-5. Eight 256-Kbit chips makes 256 Kbytes of RAM.

Nibble

One other term you will see occasionally is *nibble*. Just as a nibble is a small bite, so is a computer's nibble exactly one half of a byte. In other words, a group of 4 bits is referred to as a nibble. By convention, a nibble is either the higher or lower half of a byte. You'll see how a nibble is used in a later chapter.

A comparison of bit, nibble, byte, and word is illustrated in Figure 2-6.

CHARACTER BYTES — ASCII CODES

Some years ago, an industry standards committee assigned a unique code number to each letter, numeral (0 through 9), common punctuation mark,

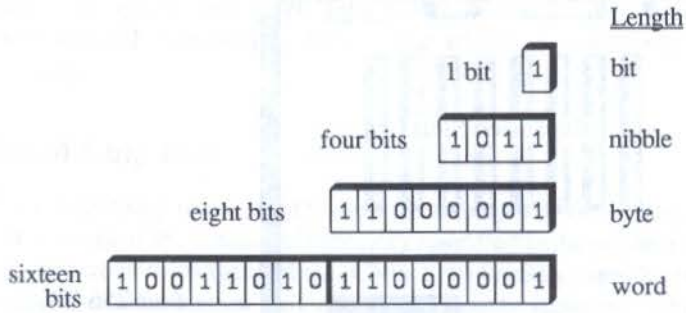


Figure 2-6. Bit, nibble, byte, and word lengths.

and several special computer commands (often used to control printers, modems, and other external devices). Even the uppercase and lowercase versions of each letter of the alphabet had to have their own code numbers. The result was the American Standard Code for Information Interchange, or ASCII (pronounced "ass-key") for short.

By strict definition, an ASCII code number is a 7-digit binary number, allowing for a total of 128 possible combinations (0 to 127, or 0000000 to 1111111). Within the span of 128 codes are enough unique numbers to accommodate all letters, numerals, punctuation marks, and control codes. Since computers generally work with bits in multiples of 8, the 7-bit code is usually embedded in an 8-bit character, with the most significant bit always being 0. In the United States, these ASCII codes are the common way of sending characters between computers, whether over the telephone line (via modem) or through direct cabling.

A number of personal computers also have extended ASCII codes, which turn a code number's most significant bit to a 1. Characters assigned to these code numbers (128–255) are not standardized among different computer models. Some machines use these codes for graphics characters, others for foreign language letters.

Together, a computer's 128 ASCII characters plus extended codes make up the machine's *character set*.

Inside the computer, each character that you type on the keyboard or that appears on your screen is really known only by its code number. Video generation circuitry has a *lookup table* that it uses to display a particular pattern of dots that to us looks like an "A" each time it receives the code number for that letter.

ASCII X-Ray

If you'd like to see what some of this looks like in memory, you can use the IIGS' built-in Monitor Program for a sneak peek. We'll be seeing the Apple II's internal character codes, which differ from ASCII codes in one important respect. Internally, standard characters (normally 0–127) are assigned code values 128–255. In other words, the Apple II character values (often referred to as *Apple ASCII* values) are 128 greater than standard ASCII values. The lowercase "e", for example, is 101 in standard ASCII, and 229 in Apple ASCII.

To check out these characters, follow these steps:

1. Turn on the computer without a disk in the disk drive. The message "Check startup device!" will appear on the screen.
2. Hold down the Control key and press the Reset key. The screen will clear, and a left-facing bracket will appear at the upper left corner.
3. At that prompt, type

   CALL -151

   and press Return. This action starts the Monitor Program, and will display an asterisk as its prompt.
4. At this point, type

   0200.Hello!

   and press Return twice. You're now looking at the contents of 16 bytes of RAM.

For the moment, ignore the group of characters along the left edge of the screen. Notice, though, that to the right of the colon is a series of 2-digit hex numbers extending most of the way across the screen. Each 2-digit number is the hexadecimal equivalent of the content of a byte of memory. Since most of these bytes represent text characters, the actual characters appear in the group of letters along the right edge of the screen. Look up each hex byte (as displayed on the screen) in the Apple II character table in Appendix B to see the way each character of the word "Hello!" is stored as its code number, instead of the character as we would recognize it.

If you want to see some more, you can use the Monitor to explore the contents of ROM. At the asterisk prompt, type

FF/D0CF

```
00/0200: 8D B2 B0 B0 AE C8 E5 EC EC EF A1 8D 00 FF FF 00 - .200.Hello!.....
```

Figure 2-7. Contents of 16 bytes of RAM.

and press Return several times. You'll see all of Applesoft BASIC's reserved words strung end to end.

HOW MEMORY WORKS

To understand how memory works, we'll start with an analogy.

You can conceive of memory as a blackboard with row after row of blank spaces. Into these spaces can go characters, like letters of the alphabet.

RAM-type memory allows you to write a character in a space, erase it, and write in another. Whenever you press a character key on the keyboard, that character is stored in a space in RAM. In the case of ROM, however, the characters are painted on the blackboard at the factory (the chip's factory) and cannot be altered. With the help of the Apple IIGS Monitor, we just saw that those character spaces in memory actually hold code numbers (when they are to represent recognizable characters) and bytes of instructions.

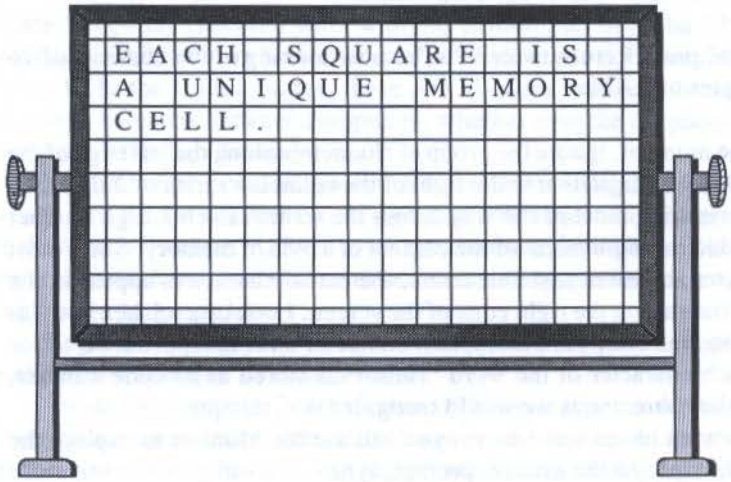


Figure 2-8. A RAM "blackboard."

The Memory Address

For a microprocessor to fetch information from memory or save information there, it must have a way of denoting a location in memory, like a box number. In the blackboard scenario, you could establish row and column numbers (in hexadecimal, of course) that would look like those in Figure 2-9.

The location for the byte $65 would be $0301, consisting of the column number, $03, and the row number, $01. This location number is called the *address* of a memory location, just like the street address on the front of your home or apartment building. The number of the address stays the same, regardless of how often the content of that cell changes or what that content is. It's exactly the same as your home address: many families may live at that address over decades, but the address of the building stays the same.

The Memory Map

We're now going move away from the blackboard analogy and orient your conception of memory to the way programmers visualize memory. Instead of thinking of memory as a grid of byte spaces, think of memory as a tall column of byte spaces, with each space having a unique address in numerical sequence.

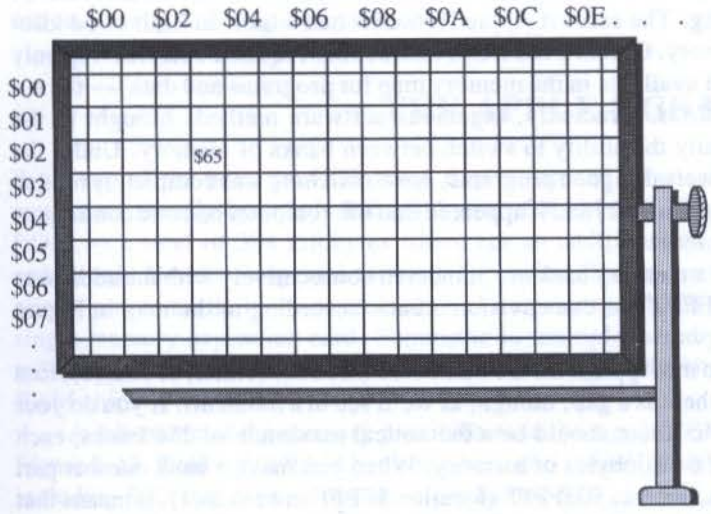


Figure 2-9. Memory "blackboard" with addresses.

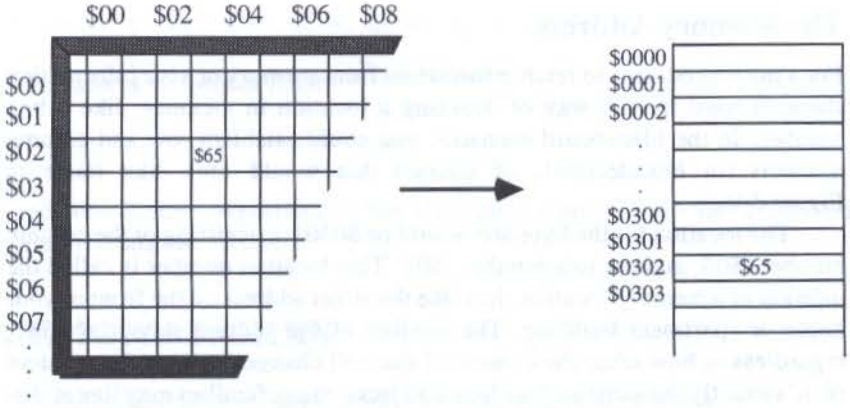


Figure 2-10. A tall column of memory spaces.

In this format, you can show the contents of individual bytes or many bytes combined into a single block. This method of visualizing memory is called a *memory map*.

Banked Memory

Partly owing to the necessity of making the IIGS compatible with earlier Apple II computers, the IIGS memory is actually divided into banks, each 64 kilobytes long. The earliest Apple II models had a built-in limit of 64 kilobytes of memory. Of that 64K, ROM instructions required 16K, leaving only 48K of RAM available in the memory map for programs and data — the so-called *user RAM*. Gradually, ingenious software methods brought to the Apple II family the ability to switch between banks of memory. Under the guidance of well-designed programs, *bank switching* was completely hidden from the program user, and it appeared that the computer had one continuous block of memory.

Cells of a memory bank are numbered consecutively with the addresses $0000 to $FFFF. You can envision a bank according to the map in Figure 2-11.

Banks in the Apple IIGS are numbered (in hexadecimal, of course) from $00 to $FF (there is a gap, though, as we'll see in a moment). If you do your hex arithmetic, there should be a theoretical maximum of 256 banks, each consisting of 64 kilobytes of memory. When you make a bank number part of an address, such as $01FFFF (location $FFFF in bank $01), it means that the Apple IIGS should address up to 256 × 64K memory cells — a total of 16 megabytes (16,777,216 addresses, to be precise). In practice, the IIGS

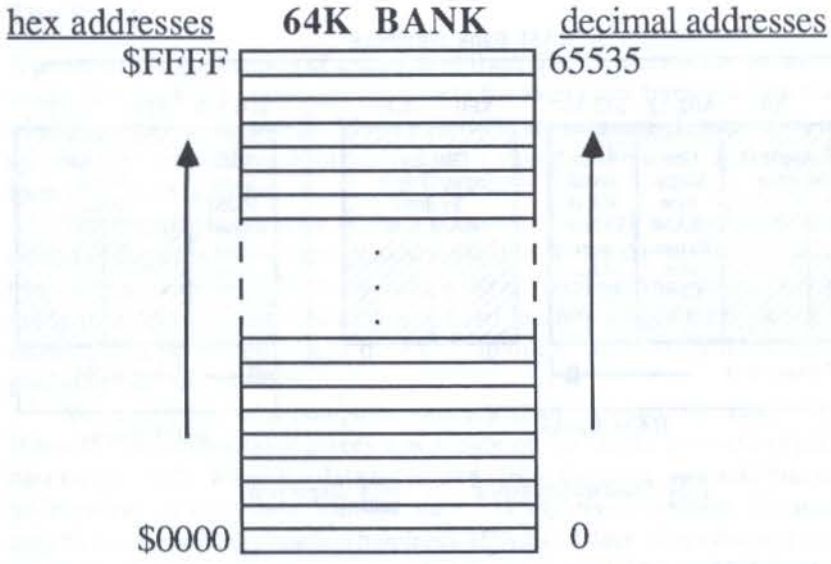


Figure 2-11. A 64K-bank memory map.

places a cap on user RAM space at 4 megabytes, plus 128 kilobytes of special RAM assigned primarily to super high-resolution video and system maintenance, plus up to 1 megabyte of ROM.

THE APPLE IIGS MEMORY MAP

Right out of the box, the IIGS has a tiny portion of its possible memory maximum already installed: 256 kilobytes of RAM and 128 kilobytes of ROM — a total of 384 kilobytes (there are an additional 64 kilobytes of sound-only RAM). A special memory expansion slot on the motherboard allows you to easily increase memory to 1 megabyte with the addition of a single memory expansion card. Expansion to multiple megabytes of RAM will be possible with memory cards built around higher-density, 1-megabit RAM chips.

A schematic of the memory possibilities in the Apple IIGS is shown in Figure 2-12. Notice that RAM bank numbers stop at $3F (a total of 64 banks, including $00) and resume for two more banks, $E0 and $E1. ROM banks $FE and $FF are the ones claimed by the IIGS ROM. ROM addresses below these banks are reserved for future use, leaving plenty of room available for

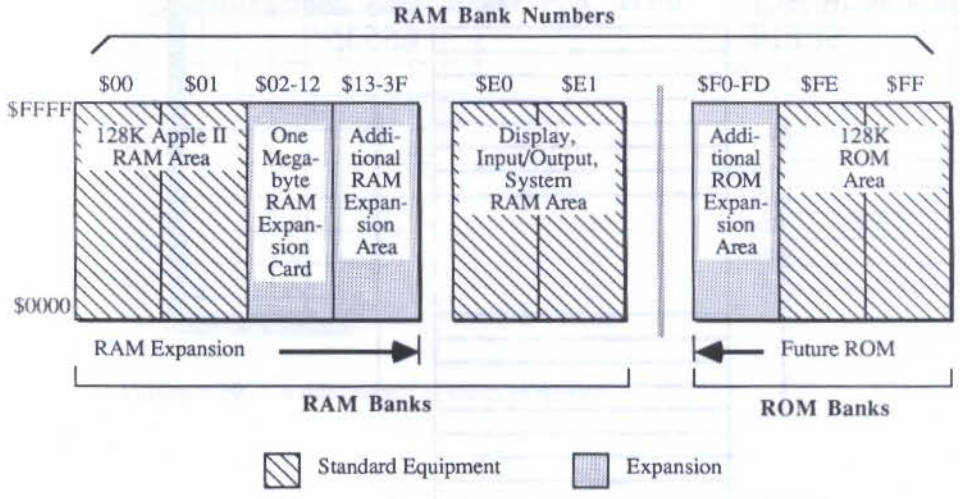


Figure 2-12. Apple IIGS memory map.

many enhancements to the IIGS for years to come. The uses of banks $00, $01, $E0, and $E1 differ depending on whether you are in native or emulation mode. Although this book deals with native mode programming, we can say that in Apple II emulation mode, banks $00 and $01 are used as if they were the 128 kilobytes of memory in an Apple IIe or IIc. In that mode they are apportioned and managed identically to the way they are in those earlier machines.

ONE SPECIAL BANK

Bank $00 has a special place in Apple IIGS programming, because it is the only bank that can hold two important chunks of memory: the *zero page* and the *stack*. We'll discuss these one at a time.

The Zero Page

Assembly language programmers in particular will need to know about the location of a small piece of bank $00 called the zero page. A holdover from Apple II days, the zero page is a kind of scratchpad that various toolbox tools will use to store temporary data while in use.

The Stack

A more visible application of a section of Bank $00 memory is for use as the stack. The stack is a temporary receptacle for even more transient data than what goes into the zero page. Moreover, there are substantial restrictions on the way you can retrieve information that is stored in the stack. An example from real life is in order.

The favorite illustration of the mechanics of a stack is a spring-loaded meal tray dispenser at the start of a cafeteria line. When the busperson places trays atop the existing pile, the weight of the new trays pushes the trays originally there out of sight. Only the tray most recently placed on the stack is showing. As people pull trays from the top of the stack, trays from below gradually become available.

You can think of a computer memory stack in much the same way. When the microprocessor places a new byte on the stack, it is said to *push* data on the stack. When the data is removed from the stack, it is said that the microprocessor *pops* data from the stack. At any given moment, the stack may be empty or it may contain hundreds of bytes of data. Since there is only one stack, it is sometimes necessary for the microprocessor to push data on the stack, then push some other data atop it temporarily. Then it pops the data in the reverse order it had been pushed. The rule with the stack is Last In, First Out. The stack is probably the most active place in memory while a program is running on your computer. Not only does its content change constantly, but it is rare for any chunk of data to sit on the stack for any length of time.

The Inverted, Solid Stack

Now that you've got a conceptual model for the workings of a stack, get ready to have that model blown away for two reasons: (1) the stack in the

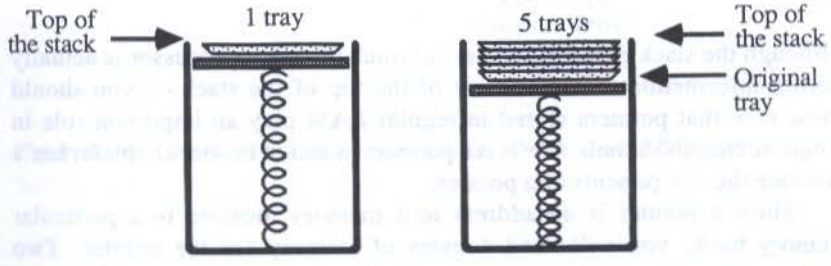


Figure 2-13. A stack of trays.

Apple IIGS grows downward; and (2) data on the stack really doesn't move in memory like the bouncing meal trays. Let's look at ways of putting these two upsetting facts into a conceptual model you can live with.

In the first instance, we can trade in the meal tray analogy for that of a paper cup dispenser — one that makes you insert fresh cups from the bottom rather than refill from the top. If you squeeze several cups in from the bottom, the one at the very bottom of the bunch you shoved into the dispenser will be the first one available for the next drink. The Last In, First Out rule still holds. That is one unbreakable rule about the stack.

The problem with using any kind of dispenser in a stack analogy, however, is that a dispenser assumes the next available tray or cup will be in the same location in space as the one before it and the one after it. That is, either a spring or gravity places the next available item on the stack at the same location every time. That's not the way it works on a computer stack.

If we start with several bytes of data on the stack in Figure 2-14A, the next available item on the stack is in memory address $007FFC. Then, if we add 2 bytes of data to the stack (Figure 2-14B), the address of the new "top" of the stack is 2 bytes less, or $007FFA. Popping 1 byte of data from the top of the stack (Figure 2-14C) makes the next available stack address $007FFB. At no time does the data on the stack shift around inside the stack.

Obviously, the microprocessor needs to keep track of where the top of the stack is at any moment. The 65816 does so with a special counter that it keeps in one of its own built-in cubbyholes. That counter is called the *stack pointer*. That CPU cubbyhole holds 2 bytes of data — just enough to specify the address of a memory location (bank $00 is assumed). At any instant during program execution, the stack pointer contains the address of the top of the stack. As an item is popped off the stack, the stack pointer increases by 1 (goes up in memory); as an item is pushed onto the stack, the stack pointer decreases by 1 (goes down in memory). This can be confusing at first, but Figure 2-15 should help bring the concept home.

POINTERS

Although the stack pointer is a case in which the microprocessor is actually storing information — the address of the top of the stack — you should know now that pointers stored in regular RAM play an important role in programming IIGS tools. We'll see pointers in action in later chapters. Let's examine the components of a pointer.

Since a pointer is an address to a memory location in a particular memory bank, you will need 4 bytes of memory for the pointer. Two bytes — the *low*, or rightmost, bytes of a four-byte number — refer to the memory location. Two *high* bytes refer to the bank number. Therefore, a

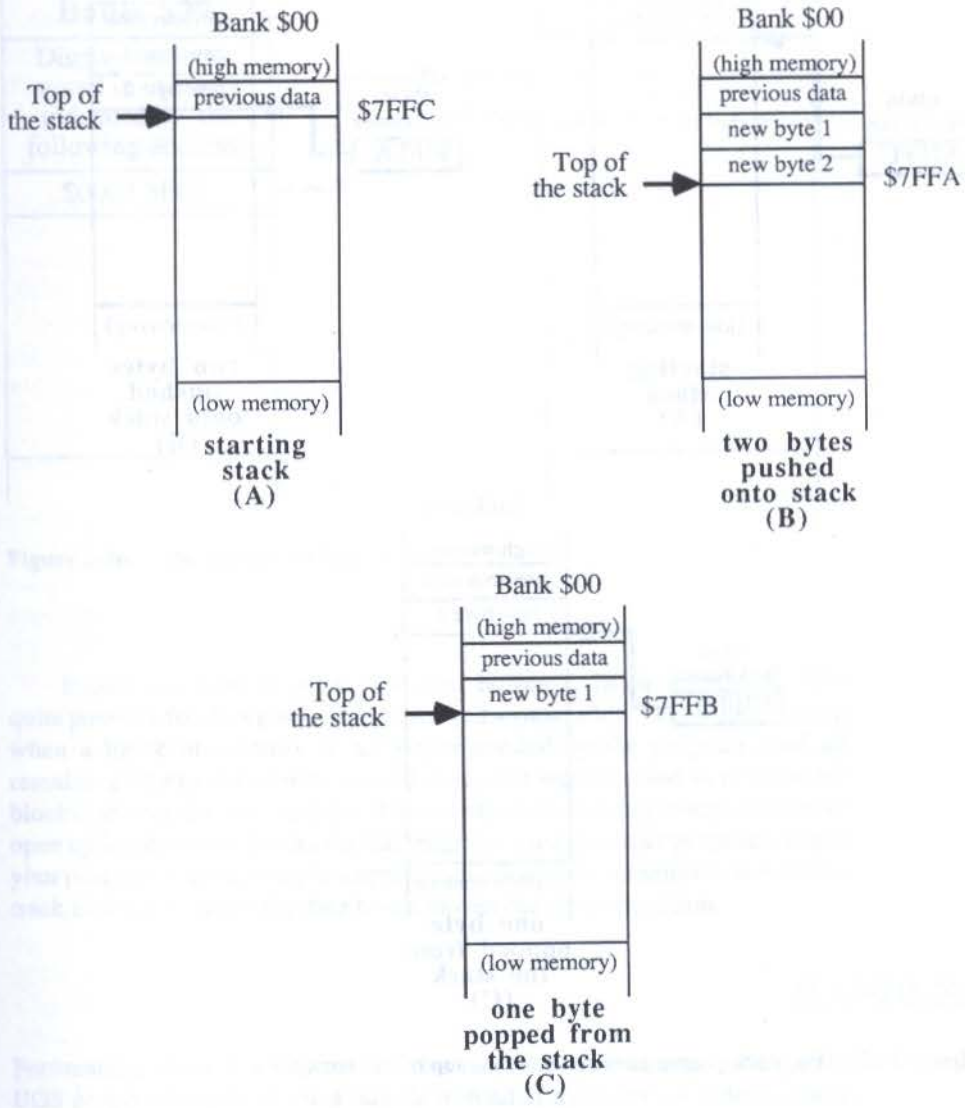


Figure 2-14. Stack manipulation before push, after push, and after pop.

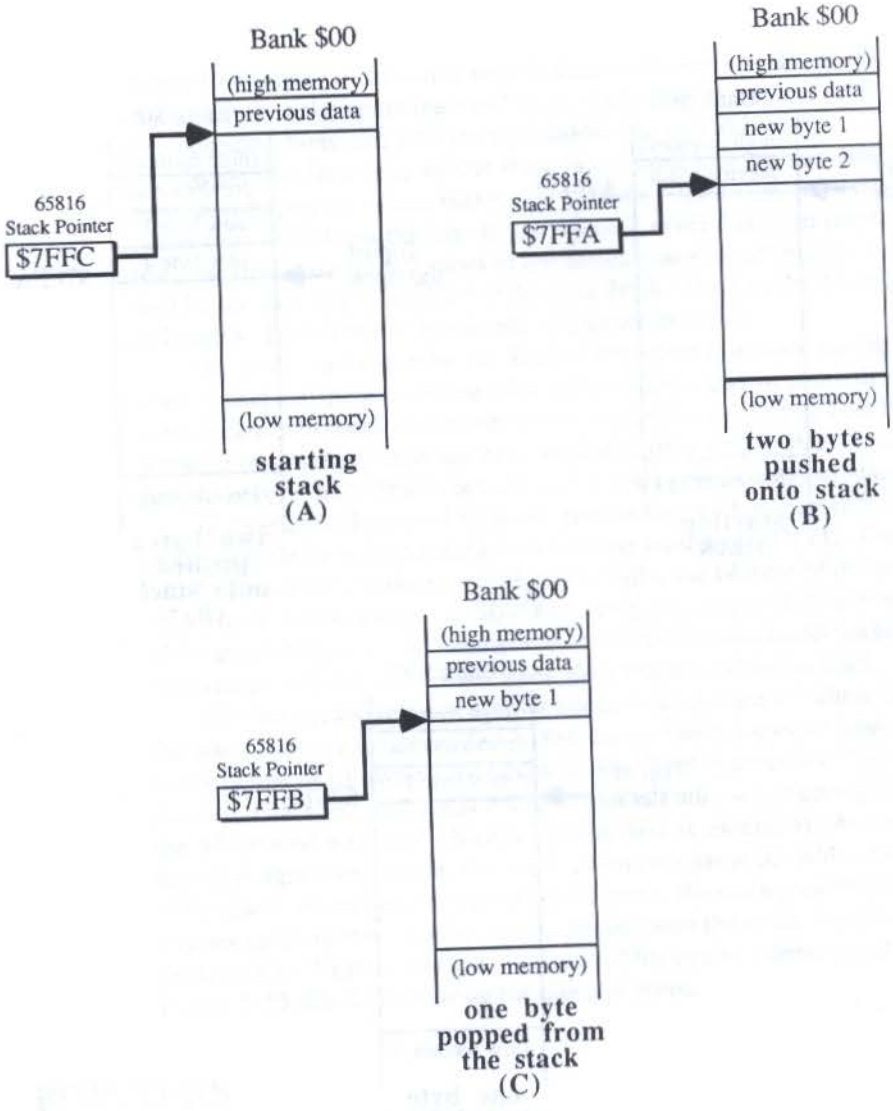


Figure 2-15. The stack pointer keeps track of the top of the stack.

pointer to address $3F02 in bank $03 will have to be placed in memory as the 4-byte number $00033F02. If hex numbers, such as this one, ever get difficult to read, simply divide them into single bytes, in this case $00 $03 $3F $02, or two-byte chunks, $0003 $3F02. This pointer mechanism is illustrated in Figure 2-16.

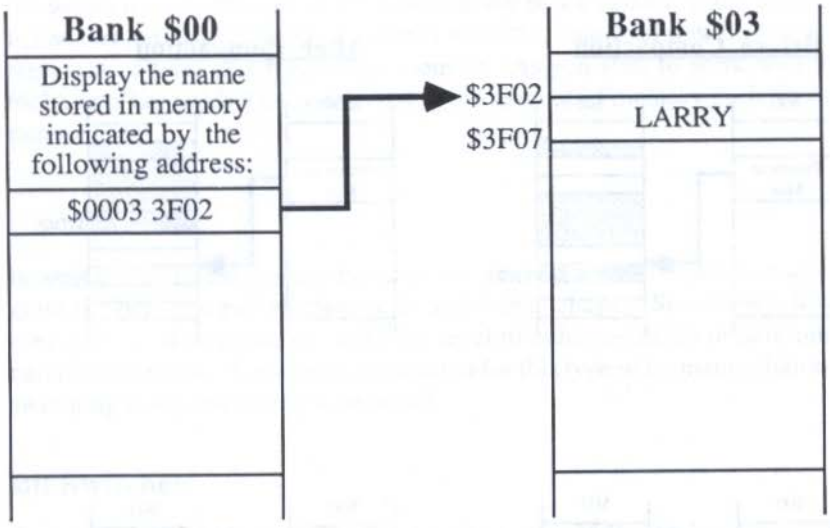


Figure 2-16. The pointer mechanism.

Pointers have to be used with care, however. In an application, it is quite possible for data portions to be moved around a bit. This would happen when a block of memory is no longer needed by the program, and all remaining blocks that can be moved (i.e., that were created as *relocatable* blocks) are compacted together. The net effect of memory compaction is to open up large, empty blocks for the program to use for other purposes. But if your program was pointing to a specific chunk of data in memory, it will lose track of the data when the data block moves during compaction.

HANDLES

Fortunately, there is a way around this dilemma. A common technique in IIGS programming is to use a *handle* instead of a pointer — indeed, many tools require the use of handles rather than pointers. Instead of using a pointer to refer to a specific data address in memory, a handle is a pointer to a *master pointer*, whose location never changes. The master pointer, in turn, keeps track of the location of the desired chunk of data as the data shifts around memory while a program runs. Figure 2-17 demonstrates the stages involved here, and compares the result of using a handle instead of a pointer.

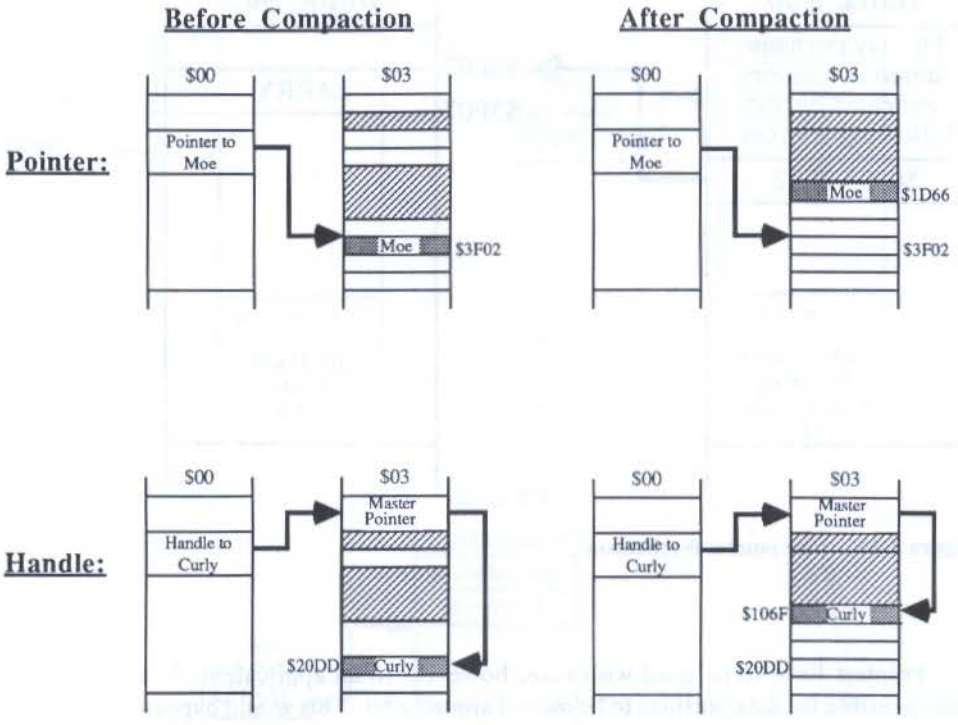


Figure 2-17. Pointer vs. handle action during memory compaction.

At first a handle might seem like a long way to go to keep track of a block of data, but because the master pointer never moves in memory, the IIGS toolbox can always find it and supply it with new information about the location of relocatable data. Since even the program that calls the handle may move during a memory compaction, the master pointer system is far more efficient than if each pointer in a program tried to track relocatable data on its own.

Notice another important matter, one that has to do with memory notation, rather than pointers and handles. In the last several figures, we've been displaying varying length items as simple blocks of memory in these vertical memory maps. The blocks are not necessarily drawn to scale, and a block can contain a chunk of information ranging from a single byte to perhaps thousands of bytes. Maps are designed to give you a bird's eye view of items stored in memory at a given instant. Therefore, it is important that you watch

the words used to describe information in any block of data. If the wording in one box indicates it holds a pointer or a handle, then you know that the box represents a total of 4 bytes from memory. As you start to work with the tools, you'll quickly become versed in the amount of memory each type of item requires.

FLAGS

In working with pointers and handles, we've seen that information in a program is often grouped together in 2- and 4-byte chunks. Sometimes, however, you need to get down to the bit level to either establish or determine certain conditions. A common application for this type of bit manipulation is switching an operating mode on or off.

Bit Switches

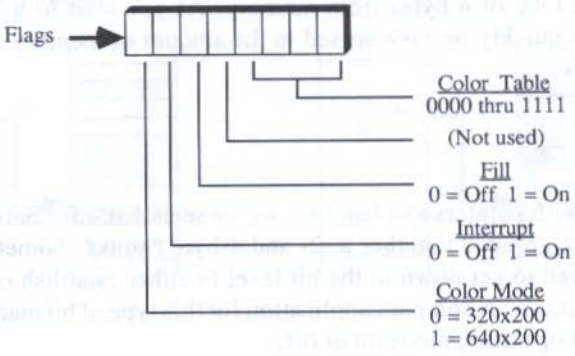
Since turning a particular mode on or off (or checking which mode is currently engaged) requires nothing more complex than an on or off signal, it can be handled by the content of a single 1 or 0 bit in memory. Usually, several of these bit switches are grouped together so that they can be "read" at once. The smallest practical grouping of bits is 1 byte. Each bit in such a byte is called a *flag*, and the byte itself is often called a *status byte*. Most data in an Apple IIGS native mode program shuffles about in word (2-byte) length. A word containing a series of flags is called a *status word*.

Each bit in a status byte or word stands for a particular condition. For example, one flag might indicate which of two video modes is engaged, while a group of four adjacent flags indicates which of sixteen color tables is in use (depending on the binary number that those four binary digits make). A status byte might look something like the one in Figure 2-18.

Bit Arithmetic

Although each bit can carry a specific message about some condition, a program has to perform some clever math on the binary number to determine the settings of the flags in each of the bits. For example, if you want to check whether the sixth bit flag is set to 1 before proceeding with a section of your program, you would perform Boolean AND arithmetic on the entire byte. The operation would look like the one in Figure 2-19.

Since the result of this operation is the binary number 0010 0000 ($20), the program detects that the flag in bit 6 is set. If the result were 0 (0000 0000), then the flag in bit 6 would be 0, and the program would know that the flag was not set.



Status Byte 10100011 ($A3) = Color mode 640x200; Interrupt off; Fill on; Color Table 3

Status Byte 00101110 ($2E) = Color mode 320x200; Interrupt off; Fill on; Color Table 14

Figure 2-18. A status byte.

Boolean arithmetic plays an important role in assembly language programming and a lesser role in Pascal and C, but you will encounter it several times in your learning about Apple IIGS tools. Be prepared for it by studying Appendix A's short course in binary and hex math.

HOW A PROGRAM WORKS

In the last chapter, we described a program as a list of instructions for the microprocessor. Now that you've seen a little more of what goes on behind the scenes, we'll get somewhat more specific about what a microprocessor does with a program.

```
      1010 0110    {Status Byte}
AND   0010 0000    {Test presence of 0010 0000}
-------------------
      0010 0000    {Yes, it's there!}
```

Figure 2-19. Boolean AND arithmetic.

Loading

The first task occurs before the program even begins. It happens at the *operating system* level. The operating system is, itself, a program that helps you manipulate disk files and load programs. It is like a master program from which you begin loading an application. Even when you start up the computer with an applications disk that appears to launch straight into the program, the operating system actually runs first, and then your program loads.

If you are using the Finder and desktop view from ProDOS 16, then you're looking at a *shell* built around the operating system, insulating you from the operating system's command language. By moving the mouse pointer onto an application's screen icon and double-clicking the mouse button, you actually give the load command to the operating system. The operating system tells the microprocessor to perform a set series of tasks that will copy some or all of the bytes stored in the selected program on the disk into RAM.

Running

The microprocessor immediately begins to follow the instructions loaded into RAM. It knows where to start because it keeps track of the address where the instructions are supposed to start and where the program is at any instant. Just as the CPU tracks the stack pointer in a special spot on the chip itself, it also keeps an address of the next program instruction it is to follow in a section of the microprocessor called the *program counter*. If an instruction in the program calls for program execution to jump to a spot far away in memory, then the program counter will adjust accordingly, pointing to the address of whatever the next instruction is to be.

The program counter works automatically. Assembly language programmers have to keep a close eye on the program counter, but C and Pascal programmers will never come in contact with it.

Quitting

At the end of the applications program, such as when you choose Quit from a menu, the final instruction of the program returns control of the computer over to the previous program — usually the operating system. When this happens, the applications program instructions and all its data are erased from memory (remnants may still be in RAM, but you'll have no access to them). If the program was one that you used to store information, such as a database or word processing program, you should, of course, save the new data in data files on disk before quitting the application.

One more mechanism is involved in running a program — the microprocessor's registers. We'll save discussion of this important concept for the next chapter, in which we will examine the languages you might use to program the IIGS.

CHAPTER 3

Talking to Your IIGS

In this chapter we'll take a slight detour from inside the Apple IIGS and investigate what you can expect to encounter when using a programming language to write instructions for your computer. We'll also examine three programming environments — assembly language, Pascal, and C — in some detail. This discussion may help you choose a language if you have not yet made this important decision.

WHY A "LANGUAGE"?

We have all learned at least one language — certainly the language in which this book is written — although we usually take that learning process for granted because it was so gradual. Language experts tell us that we learn our main language by imitating the sounds our parents make, slowly assigning meaning to those utterances. After years of constant conversational use, plus the reinforcement of reading and writing, we learn to convey meaning to virtually anyone knowing the same language — we *communicate*.

Command Languages

When you give instructions to someone to drive to the store, you are communicating the directions. The same is true when you want to give instructions to your IIGS to do something for you. You must communicate with the computer.

Even when you run a commercial applications software program, you communicate your intentions to the computer by issuing commands. Those commands may be in the form of words ("Copy"), keyboard commands (Control-Q), or mouse actions (pulling down a menu and choosing an item on that menu). Commands such as these actually constitute a small language. Often the words in the language are similar to those of your own language; other times the language is designed to be easy to remember with the help of mnemonic clues, such as Control-Q to Quit. Generally, you issue a command to produce an action; the program converts those commands into instructions for the microprocessor to follow. You, as the program's user, are completely insulated from direct contact with the microprocessor.

From User to Designer

When you're the program designer, however, the scene changes entirely. Not only do you communicate with the microprocessor, but you must be in steady touch with it. The chip feeds on your instructions and must have a constant stream of them. In other words, instead of issuing occasional indirect commands, as in an applications program, you must carry on a continual conversation. But how?

That's where a programming language comes in. It acts as an intermediary between your human language and the language the microprocessor understands.

MACHINE LANGUAGE

If you're wondering what kind of language the microprocessor has, you've already seen what it looks like. To talk directly to the chip, you'd have to use 1s and 0s. That is the total vocabulary of *machine language*, the language the microprocessor "machine" understands.

An Awkward Tongue

Programming in machine language is not impossible, but it is awfully inconvenient. In the earliest days of personal computing, most programming was done in machine language. But instead of typing 1s and 0s on a keyboard, programmers toggled switches on the computer's front panel to the desired 1 or 0 position. Once a byte of 1s and 0s was set, the press of another button on the panel stored that byte in the computer's RAM. There were no video monitors either. Programmers could review programming instructions only one byte at a time, with a panel light representing each bit's status — on or off — plus some other lights to reveal the condition of various other parts of the microprocessor (see Figure 3-1).

Aside from the tedium of writing each byte one bit at a time, errors could easily creep in. After all, it takes close scrutiny to tell the difference between 0100 1010 and 0101 1100. And in a program consisting of several thousand bytes, imagine trying to find an errant bit in a printout.

This brings up another key ingredient to using a programming language: accuracy.

LANGUAGE PRECISION

When you speak your native language to another person, the precise word selection and order are not critical, provided you follow some loose rules. For example, read these two sentences:

Yesterday I went to the grocery store.
I shopped at the supermarket yesterday.

While both sentences adhere to the rule that a subject and object agree, each sentence is constructed quite differently, yet both present the same message — at least similar enough to convey the same meaning.

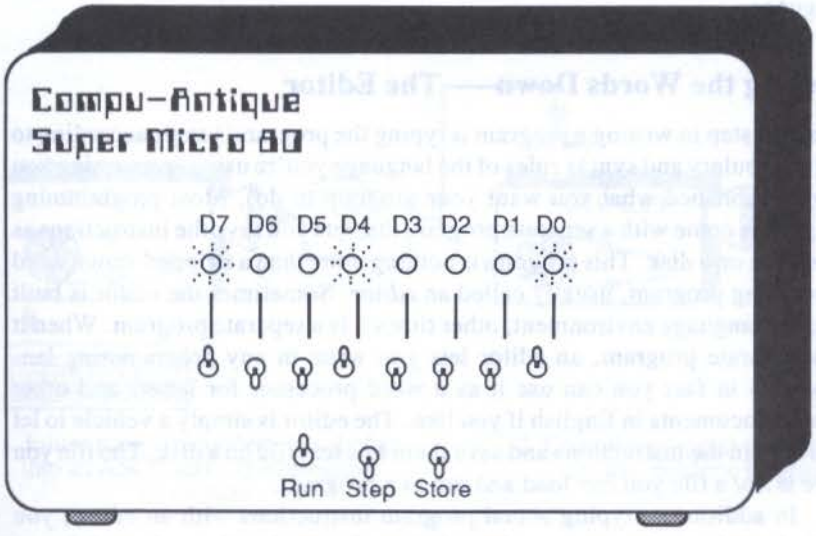


Figure 3-1. An early programming environment.

In communicating with a computer in a programming language (or in its own machine language for that matter), there is no room for ambiguities. A computer language has a vocabulary and a very specific way its words can be strung together. The structure of a message is called the *syntax*. You can make an error in syntax when talking to a fellow human, and the message will usually get across just the same. But a computer is a finicky devil, insisting that you talk to it in proper syntax. Failure to do so usually results in the computer telling you that you have made an error — often called a *syntax error*.

THE WRITING PROCESS

Up to now, we may have made it sound as if the microprocessor knew the same programming language that you learn so that the two of you could communicate. Strictly speaking, that's not true. Actually, all the microprocessor understands is machine language, and that's all it will ever know. But most language software you buy translates your words into machine language for the microprocessor. The language has a vocabulary and syntax of its own, and then translates your writings into properly constructed machine language instructions. It insulates you from the 1s and 0s of machine language. The language is usually designed to make the program writing process simpler by letting you write instructions in an English-like environment. Granted, sometimes the "English" is stilted and abbreviated, but the language still makes the programming job much easier than does machine language.

Getting the Words Down — The Editor

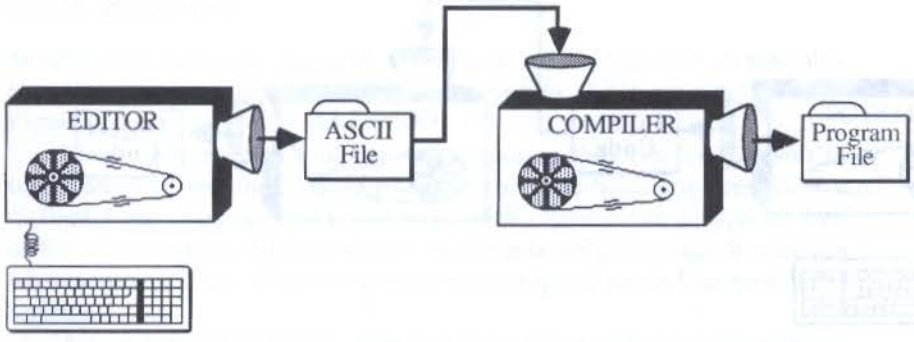
The first step in writing a program is typing the program's steps according to the vocabulary and syntax rules of the language you're using (presuming you have established what you want your program to do). Most programming languages come with a separate program that lets you save the instructions as a text file on a disk. This program is nothing more than a stripped-down word processing program, usually called an *editor*. Sometimes the editor is built into the language environment; other times it is a separate program. When it is a separate program, an editor lets you write in any programming language — in fact you can use it as a word processor for letters and other simple documents in English if you like. The editor is simply a vehicle to let you type in the instructions and save them as a text file on a disk. The file you save is *not* a file you can load and run as a program.

In addition to typing actual program instructions with an editor, you will also add explanations and reminders about what the code is doing at

various stages of the program listing. These notes are not part of the program — they do not affect the CPU's actions in any way — but they are a part of the editor text file just the same. These notes are for your own edification so you can return to the list of sometimes cryptic instructions and locate a particular section for repair or modification. The task of writing these notes (usually as you write the instructions) is called *documenting* the program. Since the notes aren't part of the program, you can use any language or syntax that makes sense to you. Depending on the language, notes are kept separate from program listings by enclosing the notes between special characters, such as curly brackets.

Translating Words into Programs

What turns an edited list of program instructions into a program is an *assembler* or *compiler* (hereafter referred to simply as compiler unless we're specifically discussing assembly language). When you start the compiling procedure, the compiler program opens a file created by the editor and converts your typed instructions into machine language. The results of this compilation are also saved to disk in a separate file. This file is in a special, compressed format that only the computer knows how to read. You won't be able to look at its contents by opening it with the editor. By the same token, since the instructions are in machine language, the computer has no way of identifying the file as coming from a particular language. Once the instructions are in machine language, they essentially lose all ties with their linguistic origins.



1. Editor program converts keystrokes into an ASCII text file of program instructions.

2. Compiler program converts your text into machine language.

Figure 3-2. Editor and compiler work flow.

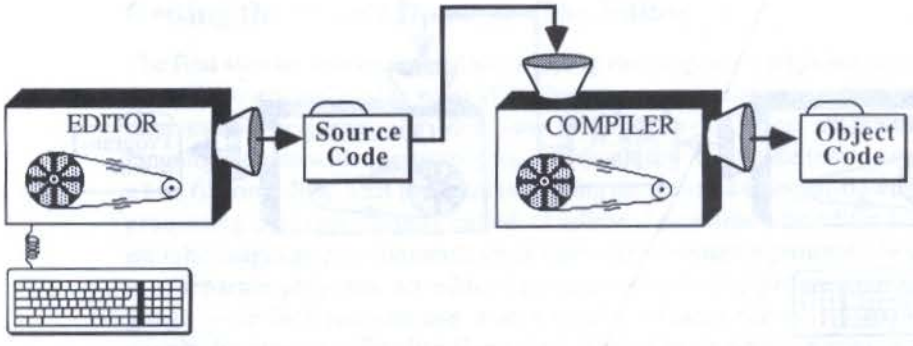
You see, then, the language, per se, is not contained in the editor, but in the compiler program. That program expects to see only its own vocabulary in the list of instructions you typed into the editor file. A "C" compiler, for instance, would not know how to handle a Pascal instruction, even though compilers from both languages could translate editor files created with the same editor program.

Two terms you will become very familiar with when you start programming are *source code* and *object code*. Source code consists of the list of instructions you type into the editor. It is the "source" from which the compilation derives its information. Source code listings are the ones you will write, print out, share with others, and use to track down errors. Object code, on the other hand, is the name of the result of compilation. You might say that it is the "objective" of doing all the programming: to create a list of instructions in machine language that the microprocessor can follow. We will be using these terms freely through the rest of this chapter, so be sure you understand them fully before proceeding.

We now take a closer look at three types of languages: assemblers, compilers, and interpreters. Each has advantages and disadvantages you should know about if you're still considering which language to settle down with.

ASSEMBLY LANGUAGE MECHANICS

You have probably heard both horror stories and praise for assembly language. Both are justified. Assembly language is often said to be difficult to



1. Editor program converts keystrokes into an ASCII text file of program instructions.

2. Compiler program converts your text into machine language.

Figure 3-3. Source code and object code in the edit and compile stages.

learn. That view is hotly debated, but it is safe to say that if you did your first programming in a language such as BASIC or Pascal, assembly language will seem much harder at first. The primary reason for this is that in assembly language you get much closer to working at the microprocessor level than you do with any other common language. For this reason, assembly language is called a *low-level* language. In contrast, a *high-level* language largely insulates you from worrying about the CPU chip.

Brick and Mortar

To understand what assembly language programs do, you need a little background in a microprocessor's internal components and their role in a program — its *architecture*. The architecture of a given chip is generally unique compared to that of other chips. A 65816, for instance, has far different architecture from that of the 68000 in the Macintosh. Among the numerous architectural features of a chip, assembly language programmers pay the closest attention to the chip's *registers*.

A register is little more than a small storage space built into the microprocessor. In the last chapter, we saw how two of the 65816's registers work: the stack pointer and the program counter. In both cases, these registers hold addresses to places in memory, one for the location of the top of the stack, the other for the location in RAM of the next instruction the CPU is to follow. These registers are largely automatic, in that certain instructions adjust their contents. Pushing a byte onto the stack, for instance, automatically reduces by 1 (*decrements*) the address held in the stack pointer.

65816 Registers

Several other registers, however, are the ones actively used by an assembly language program instruction. The registers of the 65816 are represented in Figure 3-4.

Most registers are 16-bit registers, meaning they can hold 2 bytes of information at one time. If you find that you need holding places for two distinct 8-bit characters, you can store each character in a separate half of the same register. Simply signify in the assembly language instruction whether you want to store the number in the high or in the low byte of the register.

As you learned in the last chapter, a register can be reserved for specific jobs, such as maintaining addresses of the stack and instruction pointers. Explanation of the jobs for the other registers goes beyond the scope of this book, but the documentation accompanying an assembly language programming package should describe them in detail.

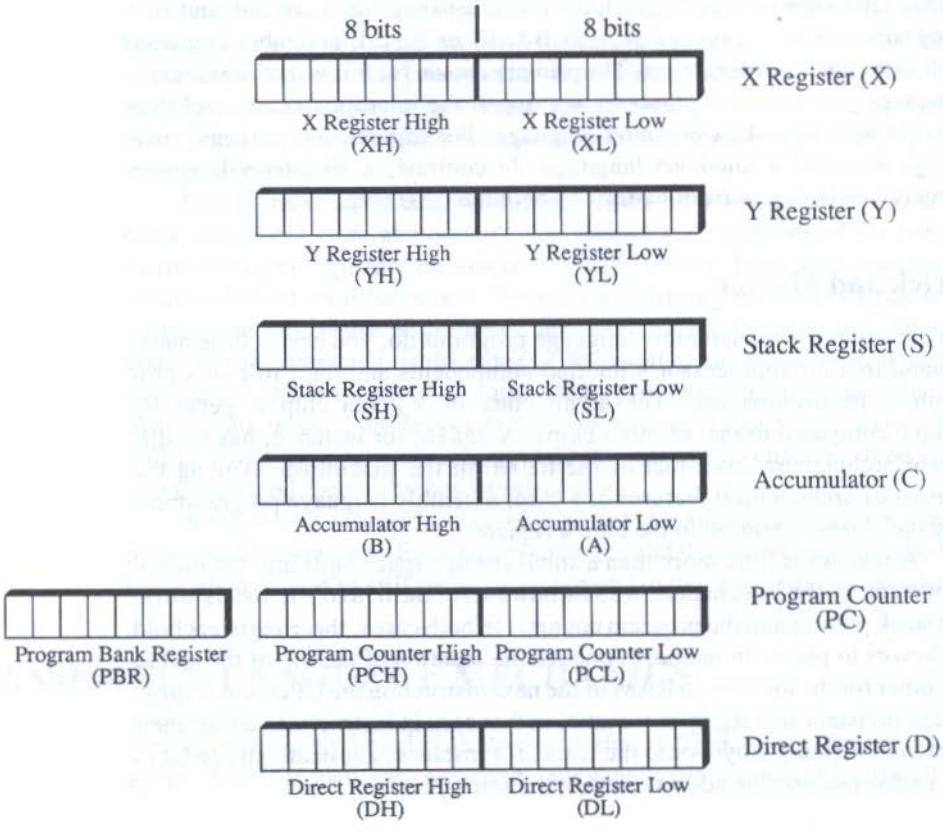


Figure 3-4. Registers of the 65816.

Shuffled Registers

It may seem too simple to be true, but the bulk of assembly language programming consists of writing instructions to move information into and out of registers on the microprocessor chip. For example, if a program is supposed to place the character "A" into a place in RAM, the first step of the program procedure will be to place the ASCII code for that character into a register. From there, another instruction will specifically store the contents of that register into a location in memory.

To the beginner, this shuffling about of data among registers and memory locations might seem like a very time-consuming endeavor when a program runs. On the contrary, because these simple instructions are actu-

ally built into the microprocessor, they are handled with blinding speed — as often as millions of times a second. Although some instructions take longer to perform, they still work at speeds we can only imagine. Tens of thousands of instructions, including the microprocessor's maintenance of the instruction and stack pointers, can be executed in less than a second.

An Assembler Package

When you buy an assembly language for the IIGS, you usually get a minimum of three separate programs, and often several more. The three basic programs are an editor, an assembler, and a linker. The editor, as we've seen, is a simple word processor that lets you write the list of instructions. Most of the instructions you write into the editor will be the same as the instructions that were built into the microprocessor by its designers. Microprocessor makers call these instructions *opcodes*, since each instruction is assigned an identifying number (the code) that signifies a specific operation (the op). Collectively, the opcodes are grouped into an *instruction set*. The instruction set of the IIGS' 65816 contains 256 different opcodes, while the older 6502 microprocessor has only 151 opcodes (the 65C02 in the IIc has 178 opcodes).

Macro Libraries

Most assembly language packages include numerous source code files containing prewritten code for common program operations. Instead of retyping a series of frequently used program instructions into your own source code listing, you can simply type the name assigned to that group of instructions wherever you want them in your program. The name you type is called a *macroinstruction*, or *macro*. When you assemble the program, the assembler reads the contents of the macro text files and inserts the detailed instructions where you typed the macro name — it essentially expands the single macro name into a full list of instructions from the macro file. A collection of macro text files is called a *macro library*, and you can insert as many as your program requires.

To incorporate a macro in your program, you'll also have to identify at the beginning of the source code file that you will be *including* macros from one or more macro files when the program is assembled. Gradually, you will also build your own library of macros added to those supplied with your assembler language. These will save you the time of retyping instructions for sections of future programs. You'll be able to include them in a new program just the way you do the macros supplied with the language software.

The Assembler

Once the program instructions are safely stored on disk as your source code file, it is time to run the *assembler* program. The assembler reads the contents of the source code file (and macro files, if any) line by line, and translates the opcodes into machine language.

The Linker

The third program in your package is called a *linker*. As its name implies, this program links together previously assembled portions of a program. It also figures out where in memory the various portions of your program will "reside" when you run it, as well as writing other instructions that the microprocessor will follow when you run the program.

Assembly and linking times vary, depending on the assembler software you're using and the length of the source file. The procedure may take only a few minutes for a modest program. Still it is practical to assemble and link only a sizable portion of new code or existing error-filled code that you've repaired to the best of your ability. Assembly and linkage of a longer program can take tens of minutes, so you're not likely to assemble each time you make only one of a series of intended changes.

The result of the assembly and linkage procedures is saved as a *load file*, which means that you can load and run it just like an application program you buy at the computer store. In the case of a native-mode program designed to behave like a Macintosh program, it means you start it by double-clicking on its icon in the Finder, do whatever the program does while you're running it, and then quit (provided you programmed such an option) to the Finder.

HIGH-LEVEL COMPILER MECHANICS

The steps involved in writing a program in a high-level compiled language, such as Pascal and C, are not far different from those of an assembly language program. The key difference is in the way you write program instructions in the editor. In both Pascal and C, the vocabularies are more English-like, and it is often easier to trace the execution of a program just by following a printout of the source code file (although documenting the source code is still highly recommended).

Portability

Another attraction of writing in a high-level, compiled language is that you can often transfer the experience of writing programs for one computer to

writing programs on another computer, even though the two machines run on completely different microprocessors. For example, if you know how to program in C on the Macintosh, you can carry that knowledge right to the Apple IIGS. Your learning time on the IIGS will be relatively short. The major differences, it turns out, are in the ways of using the built in programming tools, which we'll be getting to in the next chapter. Other than programming the tools, the language vocabulary and syntax will be almost identical on the two machines. Therefore, an experienced programmer can concentrate on the programming peculiarities of the machine at hand, rather than trying to learn an entirely new vocabulary, as would be necessary moving from the Mac's 68000 to the IIGS' 65816 assembly language instruction sets.

By staying with the same high-level language from machine to machine, the programmer will have an easier time converting programs to the IIGS. Such a conversion is often called a *port*. Of course, owing to the graphical user interface that the IIGS toolbox promotes, it will be easier to port programs from similar environments, such as the Macintosh.

High-Level Punctuation

Both Pascal and C are rich in punctuation rules, which must be followed carefully. The punctuation marks play no role in the running of the program, though. Rather, the marks send instructions to the compiler when the source file is being compiled. For instance, both Pascal and C require that a semicolon be placed at the end of each statement. When the compiler encounters a semicolon in the source file, it knows that all the text between that mark and the one previous is a single statement and it's okay to go ahead and compile it. At first, you will probably experience some frustration when the compiler encounters a missing or incorrect punctuation mark, alerting you of the mistake. Later you'll acknowledge the marks as a necessary nuisance, and learn to check punctuation prior to compiling.

Standard Languages

Designers of high-level languages often try to come close to a recognized standard vocabulary and syntax for that language — a standard generally set by the language's original developers or an industry standards group. One reason for adhering to a standard is that it might be easier for someone to use a particular compiled language if it resembles a standard version learned in school or from experience elsewhere. A developer of a new *dialect* might include a number of enhancements to the language, and discuss them in the manual as enhancements to a particular standard.

One way to enhance a language is to make it easier to use. Perhaps the dialect offers more logical menu selections, a faster compiler (compiler

design is a craft in itself), more compact object code, or more automatic performance of key operations. Any of these (and more) qualify as reasons to compare various brands of compilers in your desired language, if more than one are available.

High-Level Libraries

Another common way of enhancing a standard version of a compiled language is to add libraries of compiled routines that your program uses to perform complex functions when it runs. These routines are merged into your program at the linking stage.

For example, if your program reads and writes disk files, your source code would contain instructions such as "read" and "write" to retrieve and store data. When you run the source code through the compiler, the resulting object code will not contain the routines that do the actual disk work. Those routines are in a separate library file, which must be linked to your program.

The linker attaches the disk routines to your program, and links the "read" and "write" instructions to them. The output of the linker — the *load file* — is the final program file. When the program runs and encounters one of those disk instructions, execution branches momentarily to the disk routines. The procedure for linking library files is illustrated in Figure 3-5.

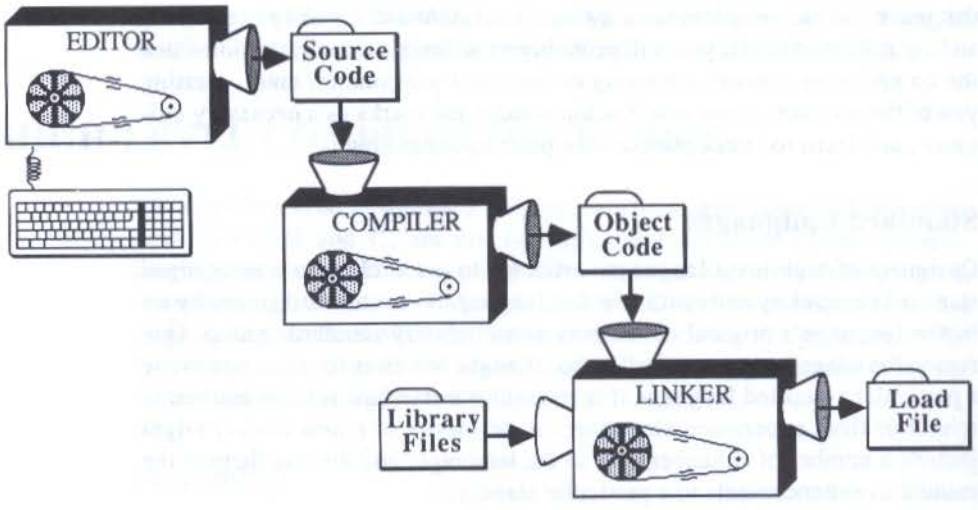


Figure 3-5. Stages of editing, compiling, and linking.

HIGH-LEVEL INTERPRETER MECHANICS

A third class of programming language (after assembler and compiler) is called an *interpreter*. An interpreter functions quite differently from either an assembler or a compiler. Assemblers and compilers, as we've seen, translate source code into machine language. The CPU then turns the machine language code into "results," which we see as a running program. An interpreter, however, does not have an explicit intermediate machine language stage. The program's source code (usually stored in a compacted, non–text file format) is turned directly into "results" while the program runs.

Interpreter Pros and Cons

Now, there are distinct advantages and disadvantages to using an interpreted language.

On the plus side, an interpreter eliminates the compiling stage. That means you can test the results of a single-character change in the program without waiting for compiling and linking. The source code is, for all practical purposes, the object code. Source code is written not in an editor, but atop the interpreter. In other words, the entire language is a single program, a single environment.

There are penalties for this convenience, however. First, the procedure to run an interpreted program is to first load the interpreter into memory; then load the program into the interpreter — like loading a spreadsheet program and then the spreadsheet into the program. That means that for you to distribute the program to others, they, too, must have the interpreter. The program file you generate is not executable, as it usually is with a compiled language. Since the copyright on the interpreter language package prevents you from distributing the interpreter (at least not without a license from the language developer), every potential user of your program must own the same language just to run the program.

Another penalty is that a program running inside an interpreted language runs at a slower pace than a compiled program doing the same operations. If you recall that when writing a compiled language program the compilation process takes a long time, then you'll understand that having an interpreter perform similar translations while the program runs is bound to make for slower execution speed.

Interpreter Experiment

If you want experience using an interpreted language, you have one built into your Apple IIGS. The language is called Applesoft BASIC. Unfortunately, you cannot program with the toolbox from Applesoft BASIC. Other inter-

preted BASIC languages designed specifically for the Apple IIGS, however, will allow you to program toolbox operations. There may eventually be BASIC compilers for the IIGS toolbox programming, which will allow you to create executable program files after developing a program in an interpreter environment for ease of testing and debugging. BASIC compilers have traditionally generated rather large executable files, because they often link a full complement of program modules, some of which your program might not need. If you are considering a BASIC compiler as your language, consider its ability to create compact executable files.

While an interpreter's relatively slow execution speed and other negative features make it a poor choice for a serious production language, many professional programmers use an interpreter, such as BASIC, as a rough draft tool. The ability to run and debug programs "on the fly" makes it easy to test many programming hypotheses quickly, without wasting compiling time.

CHOOSING A LANGUAGE

One of the most difficult decisions you will make about your computer is the programming language you wish to commit yourself to. It is a commitment, to be sure, because you'll invest time, energy, and money in the mastery of a given language. Making the choice more difficult will be advice from experienced programmers. Such advice often consists of emotionally charged statements about one language being "the only" one to use, while another is "a waste of time." Programmers tend to be passionate about the languages they use.

If you're impatient — and who of us is not? — then a high-level programming language will probably be the best place to start. Once you get experience with programming, you will be able to learn enough about assembly language to perhaps write assembly language subroutines from your high-level language programs that perform key operations faster than the compiled high-level program does.

The High Road

Between the two most popular high-level, compiled languages, C and Pascal, the choice is more a matter of support available to you. If you are surrounded by C language friends, then you will have valuable resources available to you. Of course, if you already have experience programming in Pascal, then you will have an easier time focusing on the idiosyncracies of programming with a toolbox. Apple Computer plans to support both C and Pascal for developers, so there will likely be plenty of documentation and

other support material available from both Apple and third-party publishers for both languages.

Which Compiler?

Choosing from multiple compilers in a given language is more challenging, primarily because it takes some experimentation to adequately compare the performance of one C compiler against another, for example. Poke around the electronic bulletin boards that seem to attract programmers, such as the Micronetworked Apple User Group (MAUG) on CompuServe. Find out which packages programmers there are using with success.

The most important criterion for Apple IIGS compilers is that the one you choose be able to access the entire toolbox. Additionally, some compilers come in different *levels*. One level may not offer access to the entire toolbox, but it is inexpensive. Higher levels give you more power, greater toolbox access, and perhaps the opportunity to distribute software compiled with that language. Be sure you understand a compiler manufacturer's licensing agreements for commercial software you might develop with its compiler.

The Low Road

A number of programming purists, however, will pursue assembly language, another language supported directly by Apple. Once you have the feel of controlling each movement of data around your microprocessor and RAM, you will be hard-pressed to return to a high-level language for anything other than program prototyping. Programming in assembly language gives you the supreme opportunity to fine-tune each operation of a program for maximum speed. You will likely compare two ways of doing the same operation by adding up the number of processor cycles (internal clock pulses) it takes for each, and choosing the one that runs faster.

APPLE'S PROGRAMMING WORKSHOPS

Few computer companies go out of their way to support third-party software developers as Apple is doing for the IIGS. While most companies, such as IBM, offer an assembler and hardware technical documentation for its PCs, Apple set out from the very beginning to offer an assembler, a Pascal compiler, a C compiler (each in a separate Programmers Workshop package), and volumes of technical data for professional and hobbyist programmers.

There's method to this madness, of course. By offering so many development packages, Apple will attract the widest possible audience to

the IIGS: accomplished assembler programmers, longtime Apple II Pascal adherents, and Pascal and C whizkids from the ranks of Macintosh programmers. If IIGS development parallels that of the Macintosh, then C will likely be the predominant high-level language among commercial developers. It may be easier, therefore, to find programming help at user group meetings and on electronic bulletin boards for C than for Pascal, yet you won't be alone if your choice is Pascal.

Common Ground

The codevelopment of these three programming environments has yielded a practical direction for them all. Object files emanating from the assembler or the complier of any Workshop language are in identical formats. What makes this so inviting is that you can then link object files from multiple languages into a single load file.

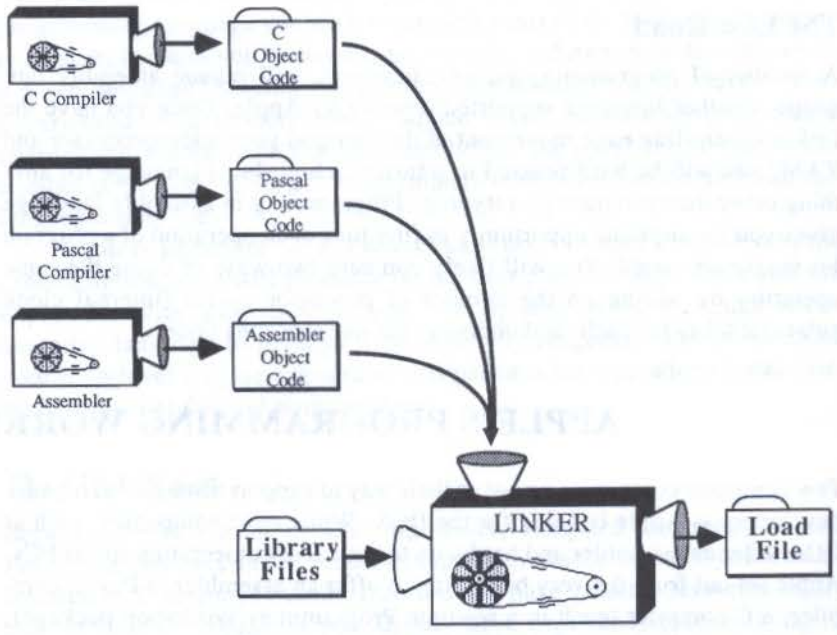


Figure 3-6. Programmer's Workshop languages produce object code that can be linked together.

This simplifies the combination of program segments written in different languages.

Workshop or Third-Party?

The availability of an Apple-produced programming environment may offer comfort to programmers, but this should not rule out using third-party assemblers and compilers. Third-party developers will surely write high-level language compilers for the IIGS as they have for the Macintosh. Some will write C and Pascal compilers, while others will focus on different languages, such as Modula 2, Lisp, Prolog, Logo, and more. For maximum flexibility, though, the language you choose should generate object code that is fully compatible with the Apple IIGS Programmer's Workshop object code. That way, you'll be able to link object code of other programmers from many different environments.

No matter which programming language you choose, you will be using it as a way to gain access to the Apple IIGS toolbox — a very powerful set of routines built into the machine. The remaining chapters will focus on the tools in preparation for your own exploration using the language of your choice.

Part Two

Key Toolbox Concepts

CHAPTER 4

What's a Toolbox?

Before we get into the specifics of the toolbox you'll eventually be using on the IIGS, we need to examine fundamental concepts about programmer's toolboxes in general. Of particular interest will be why toolbox programming is so important for IIGS program development. We'll start our toolbox discussion with an extended analogy, which should give you a solid picture of what it will be like to use a toolbox in your own programming.

THE WOODSHOP

Imagine that you want to build a wood bookcase from scratch. Perhaps you've drawn some sketches of what you want the bookcase to look like — its basic dimensions, the number of shelves, the type of base it should have for stability, and so on. Now imagine that you are provided with the resources of a fully equipped woodworking shop, decked out with racks of hand tools and several power tools for just about every step you'll go through in building the bookcase.

Thanks to the availability of those tools, you won't have to figure out how to cut the wood planks to proper length, for example. Over centuries, professional wood craftsmen have refined the design of the saw so that you can now pick it up and start using it for cutting. The same goes for essentially

every tool you will use — hammer, screwdriver, vise, even the power drill. People who knew what they were doing designed those tools to make it easier for both professional and apprentice woodcrafters to turn their ideas into finished products.

Of course, just because the tools are well designed doesn't mean that they guarantee success. It's still possible to saw a crooked line even with the most expensive and best engineered handsaw. If you mismark the spot for a drill hole, the fanciest drill press in the shop won't drill the hole in the right place by itself.

Look closely, and you may notice that some of the tools in the wood shop, as they have developed over the years, influence the final designs of items they help build. Access to a mitre (pronounced "my-ter") box, which lets you saw precise angles at the ends of two adjoining pieces of wood, has prompted many a builder to design his or her creation around mitred corners.

You could say that mitred corners are now an accepted convention or standard for joining intersecting wood sections.

FROM WOODSHOP TO COMPUTER SHOP

Now imagine that you have an idea for an Apple IIGS program. You may even have some sketches of what the screens are to look like and a diagram of the program's basic structure. Instead of going to a woodworking shop, you go into a programming shop. In the shop are numerous tools to help you create effects on the screen such as windows, pull-down menus, graphics shapes, and text in many fonts.

In this case, the tools are prewritten assembly language programs (*routines*) that your program branches to while it runs. These prewritten

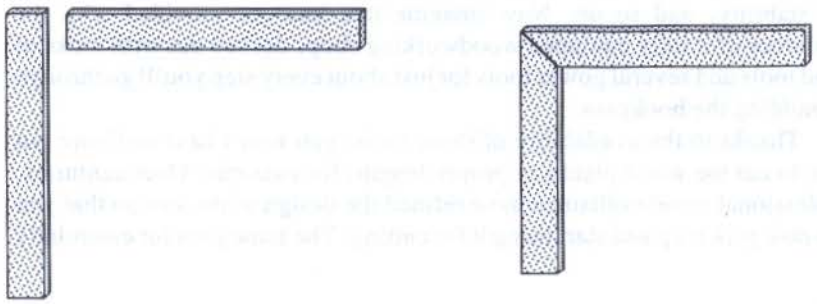


Figure 4-1. Joining two pieces of wood to form a mitred corner.

programs were crafted by experienced programmers so that they operate quickly and guide you in the direction of good program design. You don't have to write the program code for generating on-screen windows or other features you may use often. The tools that do these tasks are there for the picking. Simply branch to the ones you need.

While the tools in this Apple IIGS programmer's shop number in the hundreds, you aren't restricted to those tools only. In fact, if you have the programming experience, you can design your own tools from scratch if they will make the job of creating future programs easier. You might say that Apple provides you with a well-equipped toolbox that is big enough to hold additional tools.

TOOLS AND THE USER INTERFACE

Having a built-in set of tools at your disposal is a time-saver, especially if you are trying to design a program that approaches the standards set these days by commercial programs. The tools, however, will also influence the way your program will look and behave. Just like the woodworker's mitre box has made mitred corners a standard way of joining two pieces of wood, so too will the Apple IIGS tools point you in the direction of certain design standards.

These standards are called the *User Interface Guidelines*, a set of program design criteria established by Apple. Such guidelines were developed for the Macintosh a couple of years prior to the release of the computer, and were published for even the earliest in-house and third-party software developers to follow.

Guidelines Intentions

The purpose of formal user interface guidelines is to establish a level of commonality among programs so that a machine owner will feel at home with essential commands and tasks in virtually any application program. For example, the *User Interface Guidelines* for the Apple IIGS specify ways to use scroll bars to scroll through a document.

No Guidelines = Chaos

If you've worked with applications programs on earlier Apple II family computers or IBM PCs, then you know that scrolling can be handled many different ways. For example, to scroll a full screenful on an Apple IIe, a program may ask you to press the Down arrow in conjunction with the Shift key, Control key, or one of the Apple keys next to the space bar. It's even

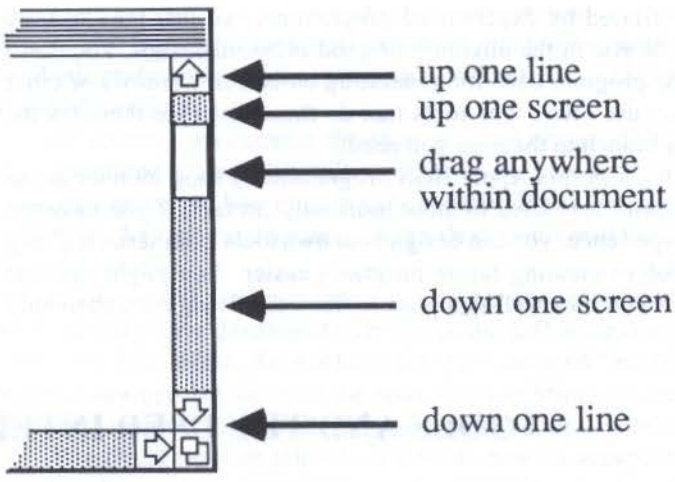


Figure 4-2. Scroll bar actions according to the *User Interface Guidelines*.

worse on the IBM PC, which has not only arrow keys, but keys labeled PgUp and PgDn (for Page Up and Page Down). Some program designers take the PgUp command to mean that a press of that key will bring into view the text above the screenful you're now looking at. Still others take the opposite frame of reference: a PgUp command means that you scroll up the page, as on an papyrus scroll, which means that you'll be looking at text below the starting screenful. How are you to remember the correct command or combination when switching among several programs that use different conventions?

Guidelines = Order

Fortunately, Apple provides a guideline for scrolling documents in a window on the IIGS, as shown in Figure 4-2. It's true that the graphical orientation helps in remembering how to scroll, but even so, we know after learning in one program that a click on the downward pointing arrow will advance the document one line; a click on the gray area underneath the white box will scroll one screenful. Without this guideline, program designers who need scrolling windows in their programs would likely come up with several different ways to bring other parts of a document into view. Nowadays, Macintosh owners expect the guidelines to be followed in software they acquire. When the guidelines are not met, reviewers and the consuming public are quick to criticize the designers for the flaws.

Guidelines and Innovation

The adherence to user interface guidelines is a controversial issue. Some programmers, particularly those who are accustomed to doing everything in a program "their way," feel that guidelines are too restrictive. Experience on the Macintosh has shown, however, that the common user interface has helped dramatically in customer perception that Mac software is easier to learn than comparably powered programs on the IBM PC. Far from telling a program designer how to design his or her program, the guidelines relieve the designer of developing schemes for basic operations: starting the program, opening existing files, starting a new file, saving a file to disk, printing, editing (cutting, copying, and pasting) text, scrolling a window, and quitting the program. The programmer can concentrate, instead, on those elements of the program that make it unique and practical. The guidelines simply provide a steady foundation atop which the program designer can build individualistic palaces and cathedrals.

Guidelines Extensions

Even though the published guidelines might seem all-inclusive with regard to basic commands and operations, several third-party developers have demonstrated on the Macintosh that guidelines can be extended logically.

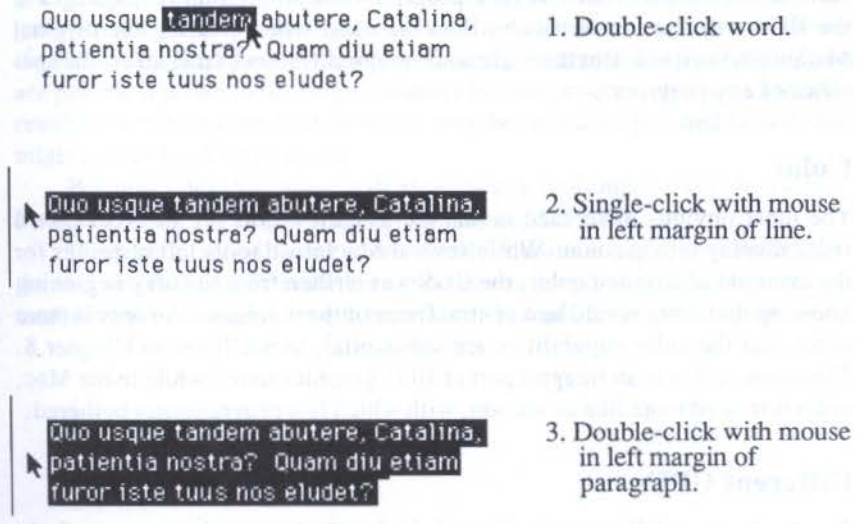


Figure 4-3. Logically extending the *User Interface Guidelines*.

For example, in Microsoft Word for the Macintosh, you can select a word in a document (to tell the program which word is to be deleted or underlined, for example) by double-clicking the mouse pointer anywhere in the word. This is consistent with the original Macintosh text editing guidelines. In fact, all of the Macintosh text editing guidelines are observed in Word. But the Microsoft designers went a couple steps further to simplify the selection of entire lines of text and whole paragraphs. To select a line of text in Word, you click the mouse pointer anywhere in the left margin next to the line you wish to select. To select a paragraph, you double-click with the mouse pointer in the left margin anywhere along that paragraph.

These are two logical extensions of the Macintosh guidelines. Both demonstrate that the guidelines are hardly as restrictive as some critics imply.

The *User Interface Guidelines* for the Apple IIGS are patterned after the Macintosh guidelines, since one of Apple's goals is to establish a user interface family look to all products in its line.

MACINTOSH AND IIGS TOOLS

Creators of the Apple IIGS tools had a significant advantage over their Macintosh counterparts of the early 1980s: they had the Mac tools to start with. In many cases, there is a strong resemblance between the tools in both machines. In fact, the similarities will help many Macintosh commercial software developers translate their programs and programming languages to the IIGS with far less difficulty than they had when creating the original Macintosh versions. But there are some major differences that affect the specifics of any program.

Color

The most obvious difference is that the graphics tools on the IIGS take a color display into account. While several Macintosh tools left openings for the eventual addition of color, the IIGS was written from the very beginning knowing that color would be a central focus of the machine. Not only is there color, but the color capabilities are substantial, as we'll see in Chapter 8. Therefore, color is an integral part of IIGS graphics tools, while in the Mac, color is treated more like an add-on, with which few programmers bothered.

Different CPUs

A second major difference in the tools is that the two machines use entirely different microprocessors. The 68000 has an internal architecture of 32-

bit-wide paths, while the 65816 has a 16-bit wide internal architecture. This difference affects the way tools manipulate some numbers that go along with them. Assembly language programmers in particular will be affected by this difference if they are familiar with Macintosh toolbox operations.

Memory Management

Finally, each machine manages its memory differently from the other. The Macintosh, for example, places its stack at the top of available memory, growing downward as the stack fills up. The IIGS, as we've seen, places the stack at the top of bank $00, which is near the bottom of memory.

The rest of available memory in the Mac is treated as one large, contiguous block of memory. It's so big in relation to the rest of the Mac memory map, that it is called the *heap* in all the technical documentation. Free memory on the IIGS is grouped into 64-kilobyte banks, and the allocation of space in those banks is a concern of the programmer and the memory management tools built into the toolbox.

TOOLBOX AND SKILL

It's true that the toolbox will offer you substantial help in designing the look and operation of your IIGS programs. But the tools will only be helpers; they won't be writing your program for you. You must still bring to your programs the planning and knowledge about your chosen programming language to put all the pieces together. Just the way a power drill doesn't know if you have marked the hole position correctly, neither will a tool know if you are putting it to use in the proper manner in your program. In both cases, the resulting product your first time out may be less than you had hoped, and might, indeed, collapse in use.

But don't despair. Your skill at using any tool improves with practice. So it will when you begin programming with IIGS tools.

From here, we will peek inside the IIGS toolbox to get acquainted with the tool sets that will be available to you as you program. You'll also see how to make a programming language pick up the tools and put them to work.

CHAPTER 5

Opening the Toolbox

If this chapter were in a woodworking book, it would illustrate the overall layout of an expansive woodworking tool chest as you open the lid. We'd show you where the various groups of tools were located and describe how to select a tool. We wouldn't show you necessarily how to use a particular tool, but give you an overview of the process of using those tools in whatever kind of work you're involved with. We'll be doing all of this, but focusing on the toolbox of prewritten routines available to programmers of the Apple IIGS.

TOOLBOX ORGANIZATION

A number of tools are built into the IIGS's ROM and the rest, called *RAM tools*, arrive on a ProDOS 16 start-up disk, located in a disk subdirectory called TOOLS. The location of a particular tool will probably change over the life of the Apple IIGS. As the machine matures, Apple will surely produce updates to the RAM tools and perhaps the ROM. Just as the Macintosh was upgraded from a 64K ROM to a 128K ROM in 1986, so too might the Apple IIGS be upgraded from a 128K ROM to a 256K ROM in the future. When that happens, more of the toolbox will likely be incorporated into ROM. Memory-map locations of tools will be entirely different from what they are today. Fortunately such changes won't affect your programs

because one of the tools, called the Tool Locator, points your program to the right tool, to matter where it comes from or what part of the memory map it's in. Therefore, it should be of little consequence whether a particular tool set arrives on your machine in ROM or on disk to be loaded into RAM. Hereafter we will consider the toolbox as if it were a single source, as you should.

A *tool set* consists of functionally related routines. Some tool sets have the name "manager" tacked on, largely as a carryover from the Mac programming environment. A tool set and manager are one and the same.

The organization of tool sets is primarily for the convenience of us humans, who need to examine supporting documentation for a particular programming operation. For example, if you wanted to know more about the details of creating a window on the screen, you would narrow your search through the reams of Apple IIGS technical documentation by focusing on the section covering the Window Manager. Inside the toolbox (in memory), individual tools are simply stacked atop one another. Routines from the same tool set may be in adjacent areas of the memory map, but they don't have to be.

TOOLBOX ROAD MAP

It's difficult to list the tool sets without appearing to assign a specific order to them. As you will learn in subsequent chapters, many tool sets rely on others either directly — a tool in one set may automatically call a tool in another set — or indirectly — one tool may require that a tool in another tool set be in use prior to execution, because your program must use tools in both sets. All this is a preface to saying that the following description of key tool sets should not imply any rigid hierarchy in the IIGS toolbox. Tool sets that have the most impact on others are the Tool Locator, the Memory Manager, and QuickDraw II. Those are described first. To master the toolbox, however, you will have to study each tool and tool set on its own, including ones not detailed in this book.

Tool Locator

Most applications programmers will not come into direct contact with the Tool Locator, since its main job is to do a lot of dirty work behind the scenes for the programmer. This is the mechanism that finds the location in memory of a toolbox routine that your program needs. It gets an assist from the system the IIGS designers established for numbering major tool sets and each tool therein. For example, when your program wishes to use a toolbox routine, the Tool Locator automatically looks up the actual address of that

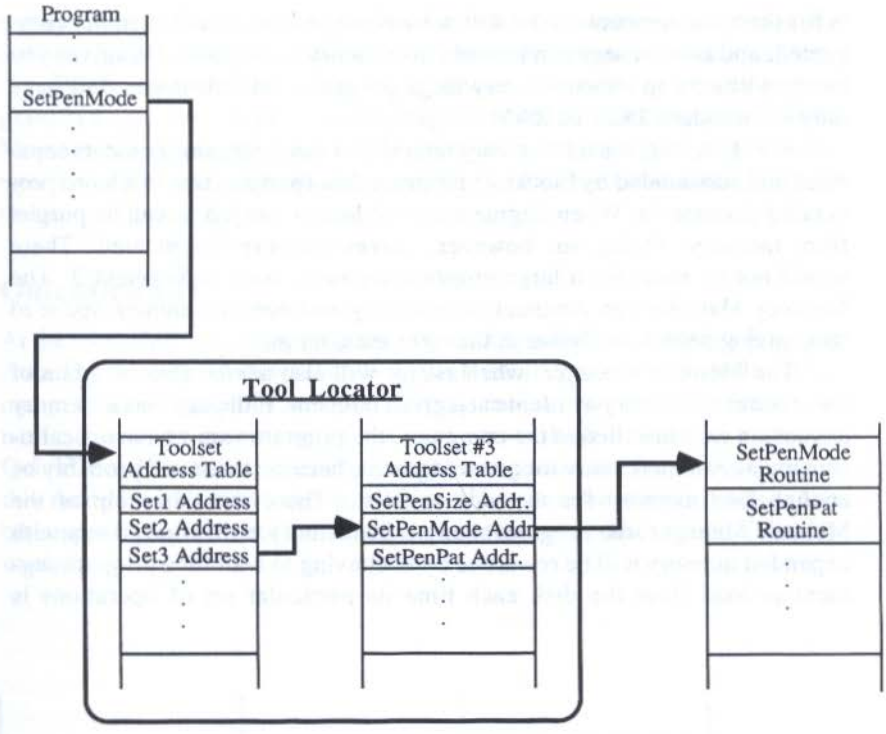


Figure 5-1. Tool Locator mechanism.

tool by first looking in a table of pointers to tool sets, and then in a table of pointers to tools in that particular tool set.

The Tool Locator also allows experienced programmers to develop their own tools or their own versions of existing tools and have the program summon those new tools in place of the built-in IIGS tools.

Memory Manager

Toolbox programs on the IIGS require a considerable amount of memory management. A program must request an allocation of memory before loading in its remaining portions. It should also deallocate that memory when it is no longer in use. Although this may seem like a burden, it allows a great deal of program design flexibility. If you divide your program into several modules or segments, the program can load in only the segment(s) needed for a particular operation the user is performing. When that operation

is finished, the segment can be withdrawn (*purged*), available memory compacted, and another segment loaded into memory in its place. This gives you the opportunity to shoehorn very large programs into machines that have only the standard 256K of RAM.

In Figure 5-2, the left memory map shows two program segments separated and surrounded by blocks of program data (perhaps text in a word processing document). When Segment 2 is no longer needed, it can be purged from memory. Doing so, however, leaves memory fragmented. There would not be room for a large program segment, such as Segment 3. The Memory Manager can compact the memory and open up enough space to tack on Segment 3, as shown in the right memory map.

The Memory Manager, when asked, will also advise your program of the amount of memory available at a given moment. If the user has a memory expansion card installed in the computer, the program won't have to deallocate memory to make way for a new segment, because there will probably be enough free memory for the new segment. Thus, with the help of the Memory Manager, the program changes its memory utilization. Users with expanded memory will be rewarded by not having to wait for a program segment to load from the disk each time its particular set of operations is

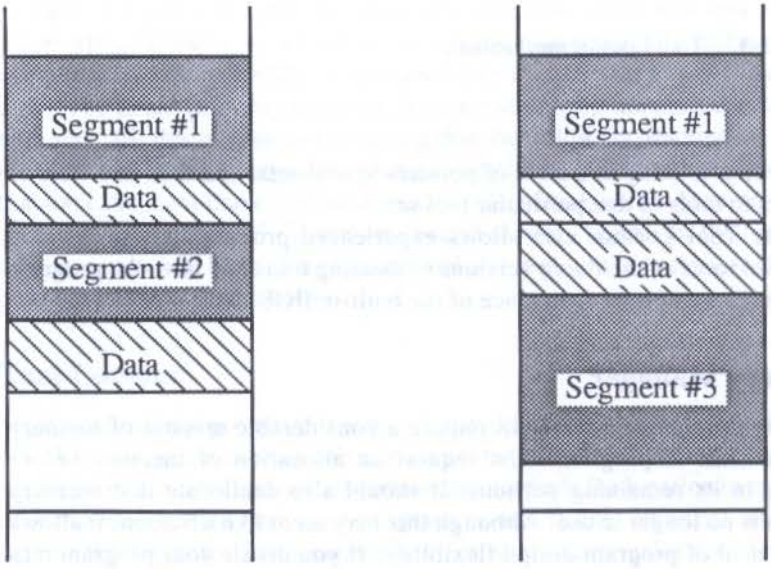


Figure 5-2. Purging and compacting memory makes room for a new segment.

required by the user. Once Segments 2 and 3 are loaded into memory, they will stay there as long as there is no further competition for memory space.

While you will have to know the Memory Manager quite well to program in assembler, a high-level language automatically performs most basic Memory Manager calls for you. You'll still have to be aware of this manager's abilities and requirements, though, for more sophisticated memory tasks.

QuickDraw II

At the root of all video display output of the Apple IIGS in a toolbox program is the QuickDraw II tool set. While Apple II emulation on the IIGS allows for many different graphics and text display modes, the standard output for native mode programming is the new super high-resolution graphics. QuickDraw II contains the routines that manage text and graphics display in this mode (strictly speaking, it is all graphics, because text is displayed as bit-mapped graphics, not built-in text characters as on other Apple II display modes).

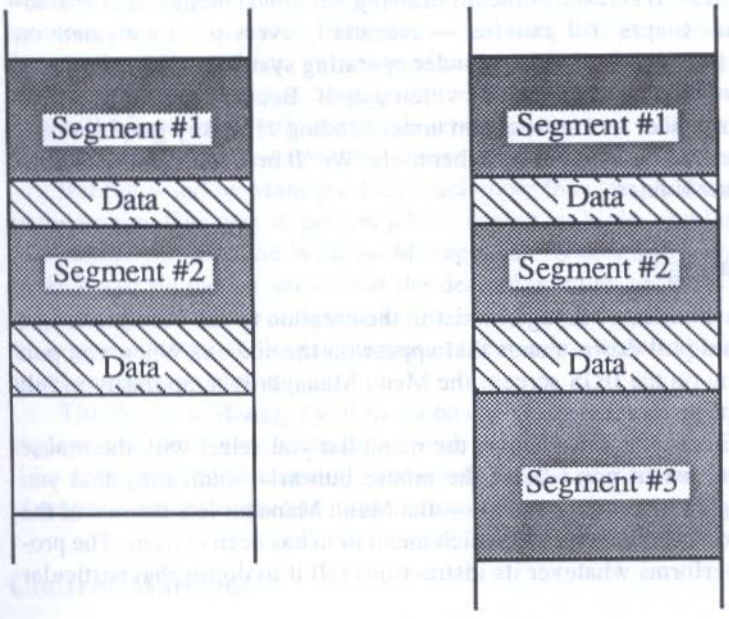


Figure 5-3. Expanded memory lets more of the program stay in RAM.

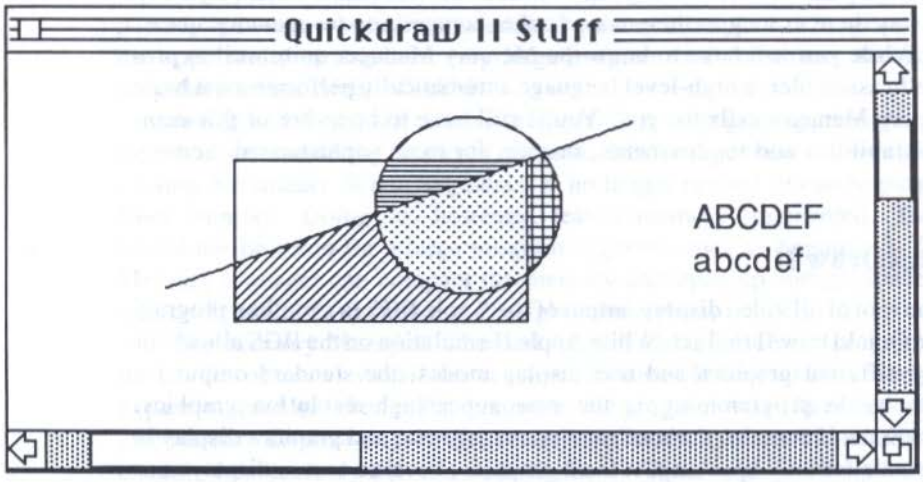


Figure 5-4. Typical QuickDraw II prowess.

QuickDraw II is responsible for drawing windows, menus, text characters, graphics shapes, fill patterns — essentially every picture element on the screen. Even the ProDOS 16 Finder operating system makes calls to the QuickDraw II tool set to create its video output. Because QuickDraw II is used by many other tool sets, a firm understanding of its key graphics concepts is essential for using many other tools. We 'll be looking more closely at QuickDraw II later.

Menu Manager

Routines in the Menu Manager assist in the creation of the Macintosh-like menu bar and pull-down menus that appear on the screen. When you pull down a menu on the IIGS screen, the Menu Manager is temporarily in full control of the program.

It monitors which item down the menu list you select with the mouse pointer. The instant you release the mouse button — indicating that you are choosing an item on the menu — the Menu Manager lets the rest of the program know which menu and which menu item has been chosen. The program then performs whatever its instructions tell it to do for that particular choice.

The Menu Manager is one of the tool sets that relies on QuickDraw II and benefits from QuickDraw's color routines.

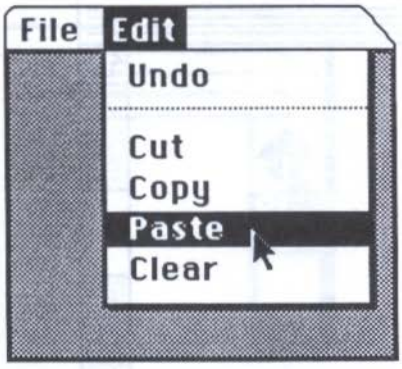


Figure 5-5. Example of Menu Manager output.

Window Manager

Action in most of your programs will take place in one or more on-screen windows. Even if the screen you ultimately generate does not look like the windows you see in the Finder, as far as the IIGS is concerned, it will be a window. Creating a window of any kind — from a simple full-screen blank area to a complex one with a title bar, scroll bars, and other features — is the responsibility of the Window Manager. When your program gives instructions to the Window Manager to create a window, the Window Manager will adhere to specifications in a set of *parameters*, which will instruct the Window Manager on the exact characteristics of the window.

But the Window Manager does much more than simply draw windows on the screen. One very important job is keeping multiple, overlapping windows under control. The Window Manager assists in sensing where mouse clicks occur on the screen so that the desired window becomes the active window (the one on top of the stack of windows, as it were). It then performs the important task of filling in the part of the window that had been obscured by other windows atop it.

The Window Manager will likely be one of the most important tool sets to play a visible role in the design of your programs. It relies on QuickDraw II for drawing window elements, and it relies on the next tool set, the Control Manager, for scroll bars and other features.

Control Manager

A *control* in a IIGS application can take the form of an on-screen *button* that you "press" with the mouse pointer, window *scroll bars*, a *check box*, which

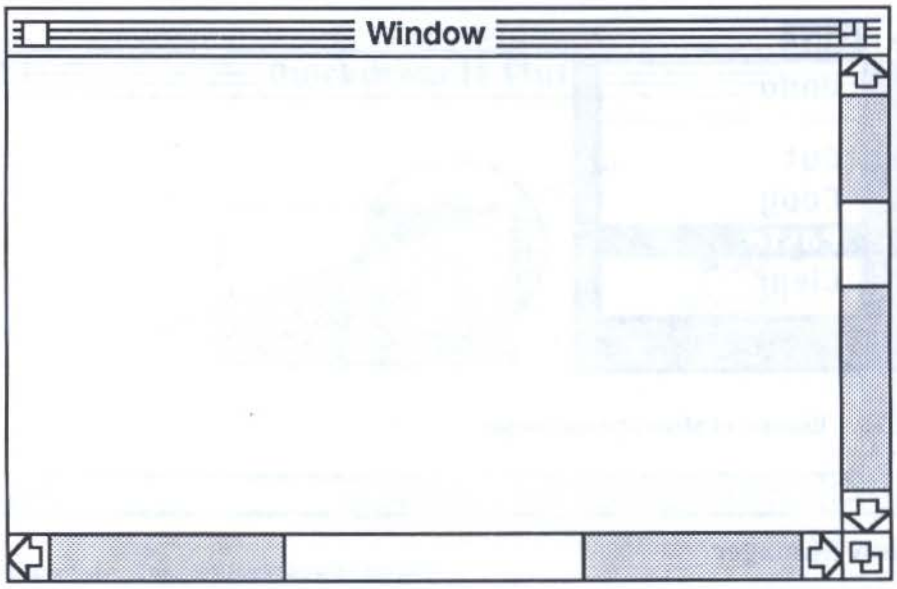


Figure 5-6. A typical window drawn by the Window Manager.

allows you to select one or more options from a list, and other on-screen devices whose activation with the mouse pointer produce clearly defined actions in the program. All of these IIGS screen objects are created and managed by the Control Manager.

Since controls are graphics objects that are placed in a window, the Control Manager works in concert with both QuickDraw II and the Window Manager. For instance, when you adjust the scroll bar in a window, the Window Manager temporarily passes program execution to the Control Manager, whose job it is to observe where you adjust the scroll bar on the screen.

The IIGS toolbox comes with several predefined controls including buttons, radio buttons, check boxes, and scroll bars (detailed in Chapter 12). You won't be confined to these controls only, because the Control Manager assists in the creation of custom controls, which can take on many different forms, such as temperature gauges, sound level meters, and so on.

Event Manager

We will be discussing the concept of *events* in Chapter 7, because it is critical to the organization of your IIGS programs. But for now, we can say that

Figure 5-7. Part of a window's scroll bar control.

every press of the keyboard, every press of the mouse button, is called an event. An event usually causes the program to perform a particular operation as a result of that event. Riding herd over these events is the Event Manager. Its role in your programs will be demonstrated in Chapter 9.

Sound Manager

The Apple IIGS, of course, can emulate the single-tone sounds that the Apple II family's internal speaker produces. But the IIGS also includes a powerful sound generator circuit created by Ensoniq, called the Digital Oscillator Chip (DOC). This chip, along with 64 kilobytes of RAM dedicated to the sound circuitry and two other chips, give the IIGS remarkable sound capabilities for a personal computer (see Figure 5-8).

The DOC chip includes 32 oscillators (tone generators). One of the oscillators is turned into a special clock that the sound circuitry has to itself. Common practice is to pair oscillators to produce a high quality tone for music. That leaves enough oscillators for 15 independent voices. The Macintosh, by comparison, has only 4 voices.

To gain access to these wonderful sound abilities, you use the Sound Manager.

Dialog Manager

According to the Apple *User Interface Guidelines*, a program designer can obtain information from a program user by way of a device called a *dialog box*. A dialog box essentially asks you questions, and you supply answers — you and the program carry on a kind of dialog. For example, if you want to

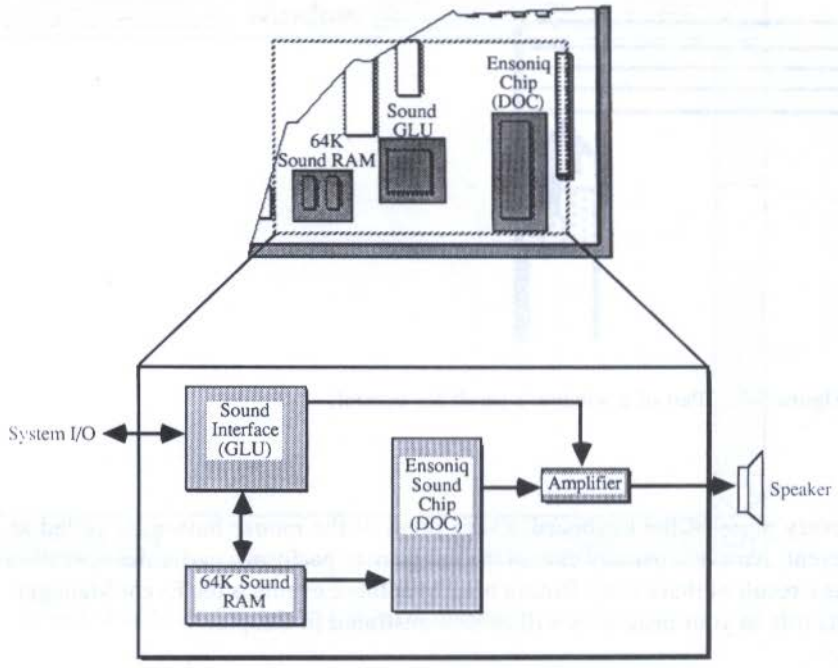


Figure 5-8. Apple IIGS sound circuitry.

establish a number of settings for the way a printed page should look — paper size, margins, text of a header or footer, and so on — you select a menu choice called "Page Setup" or similar. When you choose that menu item, a new window appears atop the current work area and requests information, such as a click of a radio button signifying the paper size from a list of three possible sizes. The new window that prompts for this information is called a dialog box.

The Dialog Manager handles many routines that create dialog boxes. This tool set relies on tools from many other sets, although most of that reliance exists "behind the scenes." For instance, a dialog box is a window, so the Dialog Manager calls many of the tools provided by the Window Manager. A dialog box also frequently contains one or more controls — radio buttons, OK buttons, check boxes — so that it is no stranger to the Control Manager. If the dialog box has a text entry box in it, then it calls upon the tools built into the Line Editor tool set (see below). Dialog Manager tools make many of these external calls on their own, letting you accomplish more with your dialog boxes with fewer steps in your programming.

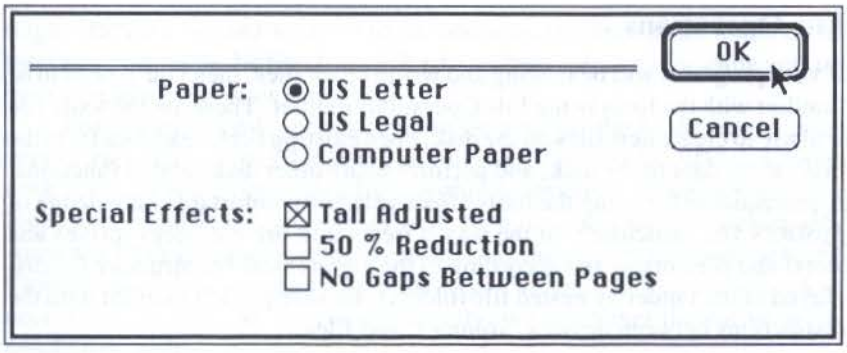


Figure 5-9. Example of a Page Setup dialog box.

Line Editor

The *User Interface Guidelines* are quite clear on the ways IIGS programs are to behave when editing text in a window or a text box. For example, you should be able to select text by dragging the text insertion pointer across the desired characters. Selected text is then displayed in an inverse highlighted fashion.

From there you can cut or copy the selected text into a scratchpad area of memory called the *scrap*. Later, you can place the text insertion pointer anywhere in the text and paste the contents of the scrap into the text. Text in its window should also *wrap* such that words are not broken at the end of a line.

Controlling all this text manipulation is the Line Editor tool set. Line Editor routines are often called as the result of menu choices (e.g., Cut, Copy, and Paste). They are also called automatically by the Dialog Manager, as we saw above. The Line Editor, itself, is not concerned with the text font or the font size. That is left up to QuickDraw II, upon which the Line Editor relies heavily. And even if your program is entirely graphics oriented, such as a game program, you may still need to invoke the Line Editor at the beginning of the program if desk accessories require Line Editor tool functions.

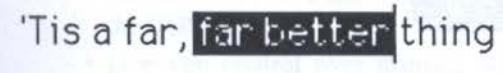


Figure 5-10. Selected text is highlighted.

File Operations

If your programs will be reading and writing disk files, then you'll need to be familiar with the tools in the File Operations tool set. These are the tools you will use to create new files on the disk, open existing files, read data from the disk, write data to the disk, and perform many other disk-related functions. A prerequisite for using the tools effectively is a comfortable knowledge of ProDOS 16, particularly in the way it treats disk drive devices (drives and slots) and files organized according to the hierarchical file structure (as displayed in the Finder by nested file folders). Be sure you are familiar with the distinctions between devices, volumes, and files.

Desk Accessory Manager

A *desk accessory* is usually a program of relatively small code length that can run atop a main application program. Examples of popular desk accessories are an alarm clock, a calculator, and a note pad.

Desk accessories are often designed to take the place of physical desk accessories a user might keep on his or her desk. But a desk accessory can also be a stripped-down version of a larger application program. For example, a desk accessory program that behaves like a small spreadsheet program

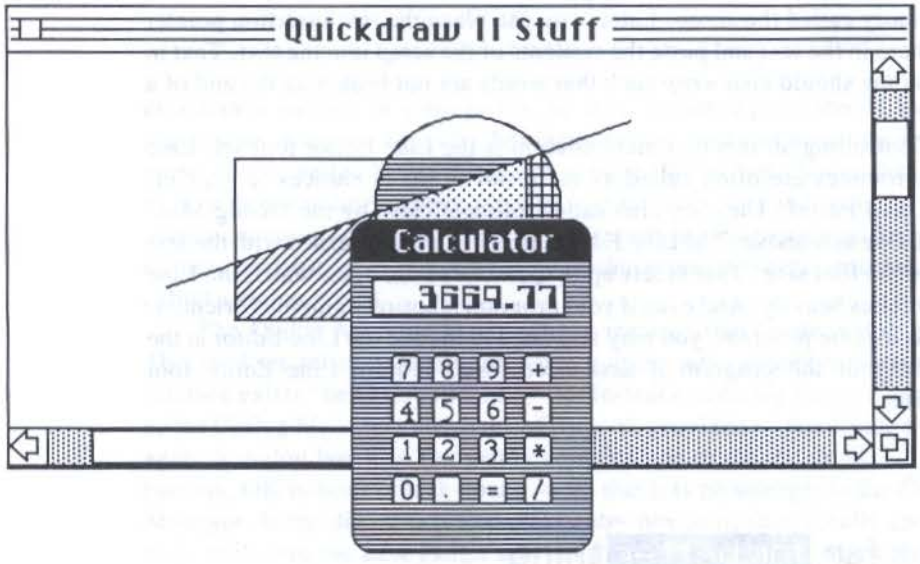


Figure 5-11. Calculator desk accessory atop an application window.

might feature a limited set of built-in functions and have strict limits on its total size, yet it may be fine for quick spreadsheet calculations while you are busy in a word processing program.

Two types of desk accessories can be used on the IIGS: *classic desk accessories* (also called *CDAs*) and *new desk accessories* (or *NDAs*). Classic desk accessories can be called from within programs running in emulation mode and, usually, native mode as well. New desk accessories operate only when the IIGS is running in native mode. If you've experimented with the Control Panel desk accessory on a IIGS, you've experienced a classic desk accessory, since it sets system configurations from either the native or the emulation mode.

Keeping your program and a desk acessory alive at the same time falls under the job description of the Desk Accessory Manager. Among its jobs is to check whether a desk accessory you are about to call will work in the program environment you are working in. If your program is to be receptive to running desk accessories atop it, the program must make provisions for the Desk Accessory Manager's tools.

SANE

The term *SANE* is Apple's acronym for Standard Apple Numerics Environment. These tools consist of built-in routines for various math functions, including floating-point math operations (addition, subtraction, multiplication, division, square root), logs, exponentials, trig functions, time-and-money calculations, random number generation, and many more. They can be found in the Macintosh toolbox as well. SANE has been fully documented and is part of the *Apple Toolbox Reference* series (see Appendix C).

Other Tools

While the above tool sets are the ones that get most of the headlines in IIGS toolbox documentation, many more tools are available to ease the programmer's task. Among the miscellaneous tools you may find helpful are those that:

- access information stored in the battery-backed-up RAM connected with the real-time clock circuitry of the IIGS.
- allow you to retrieve clock (time and date) information for inclusion into your programs.
- let you move information in and out of peripheral cards plugged into the IIGS's slots.
- give you control over printing with a variety of output devices (grouped together as the Print Manager).

- access and control information copied into the area of memory called *desk scrap* (sometimes called a *clipboard*) for retrieval or storage on disk (the Scrap Manager).

TOOL SET INTERDEPENDENCIES

We've noted above several cases in which one tool set relied on routines in other tool sets, often making calls to those other tool sets automatically without intervention from the programmer. After a while, these interrelationships will become second nature to you. But for now they may seem like a tangled web of threads running through the IIGS toolbox.

At great risk, we will attempt to diagram the relationships of the major tool sets as described above. One risk is that newcomers will consider these relationships to be a rigid structure when in fact the toolbox relationships established by Apple's designers are flexible enough to sustain many modifications by experienced programmers. Another risk is that the following diagram will by necessity be an oversimplification of the threads running through the toolbox. With those warnings in mind, we offer a hierarchy of tool sets. Those at the bottom form the foundation upon which higher-level tool sets rely.

INCORPORATING TOOL SETS

We must now discuss the link between your program and the tools. We'll be talking in generic terms because the specifics of incorporating tool sets into

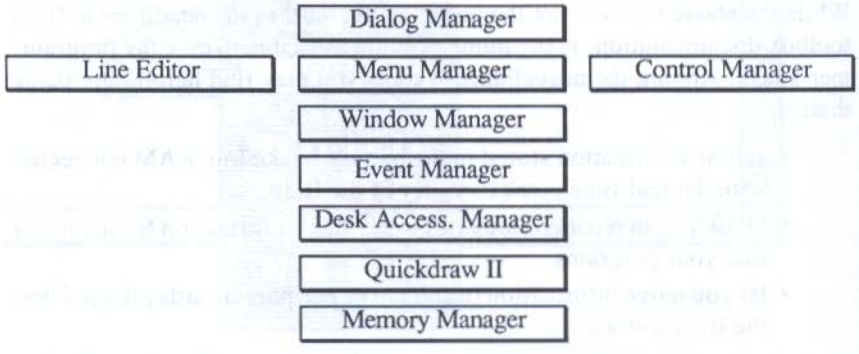


Figure 5-12. Tool set hierarchy.

programs varies slightly from language to language. Fortunately, the concepts are similar, especially among the languages in the Apple IIGS Programmers Workshop.

A IIGS language usually comes with two sets of toolbox-related files. One set is source code, the other object code.

The source code files contain many predefined variables and data structures carrying readily identifiable names that you can begin using in your own source code. These will save you from declaring the same toolbox variables over and over in each application. Moreover, you are assured that they are done correctly.

You can use these predefined variables in your source code listing provided you instruct the compiler to incorporate the external source files into your source code at compile time. The procedure for doing this is placing instructions for the compiler (*compiler directives*) at the top of your source code listing to *include* or *use* as many of those source files as are needed. The files are typically grouped according to tool set, making it easy to specify those files to be merged into your program.

The object code files supplied with the language contain the actual routines that make toolbox calls possible in your program — something the core compiler does not furnish. Therefore, the linker will link your program's object code with as many tool set object modules as you direct in the command to start the linker. The result will be a load file that makes the appropriate calls to the Tool Locator each time a tool is requested as the program runs.

It should be made clear that the "include" or "use" instructions are simply assembler or compiler directives. They do not represent the calls to the toolbox routines while the program runs. Actual calls to the toolbox are placed throughout the program as needed.

CALLING A TOOL FUNCTION

Apple has documented the toolbox calls in full detail in a two-volume set called *Apple IIGS Toolbox Reference*. A call to a toolbox routine, as listed in these references, looks like any statement that might be a part of a programming language vocabulary. Here are some examples of tool calls you might make to the Window Manager:

NewWindow
CloseWindow
GetFrameColor
SetWTitle
SelectWindow

Most of these statements are in plain language, although occasionally a tool call will be abbreviated. SetWTitle, for example, is short for "Set Window Title." Notice, however, that all tool calls are single words. This is more for the convenience of the compiler, since compilers find it easier to recognize single-word commands than those consisting of multiple words.

Most languages try to adhere to the vocabulary of tool calls as defined in Apple's reference material. That's not always the case. In fact, you may encounter languages, particularly assembly language, that have different ways of making tool calls. Instead of using the tool call vocabulary as is, the assembler may require you to precede the call with an underscore character, like this:

```
_NewWindow
```

The underscore is for the convenience of the assembler: it recognizes any word beginning with an underline as being a toolbox call. You may also find languages that use slightly different words for some toolbox calls. When this happens, the new vocabulary words are close enough for you to make an immediate connection between the new words and the ones defined in Apple's programmer documentation.

Jumping to the Toolbox

In case you're wondering what happens inside the computer when you make a toolbox call, here is a synopsis of the procedure.

Typically, your program will be following a list of instructions that you write (although converted into machine language). The instruction pointer will be wildly directing the microprocessor to follow instructions from your program loaded in the perhaps tens of thousands of RAM addresses. When the microprocessor encounters a toolbox call, the instruction pointer jumps to the address of the toolbox routine (perhaps in ROM). As soon as the toolbox routine is completed, the instruction pointer returns to its jumping-off spot in RAM and continues working its way through your program instructions.

If you've had experience programming in any language, you will recognize this methodology as a simple subroutine from your main program. In this case, however, you don't have to write the subroutines, since they have already been designed and optimized for you. Nor do they take up any disk space in your finished program file.

PASSING PARAMETERS

Toolbox calls are occasionally self-contained, action-oriented functions, such as the one named HidePen. When you issue this tool call, it unilaterally

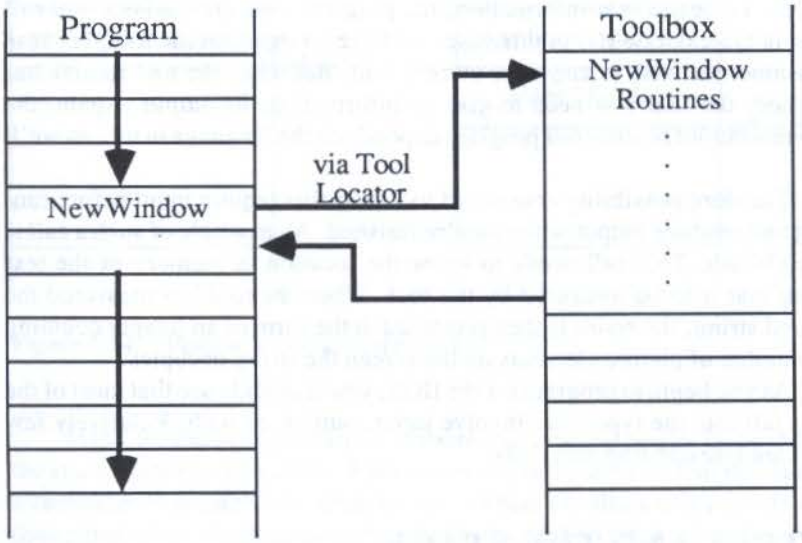


Figure 5-13. Inside a toolbox call.

turns off the drawing pen on the screen. When you issue that statement, the tool simply does its action and returns control back to the program — the tool has nothing to report back to the program. But the vast majority of toolbox routines fall into one of three categories:

1. They require input.
2. They generate output.
3. They require input and generate output.

Let's look at an example for each, using calls that affect the display of text in a window.

To change the font of a patch of selected text, the program would have to call the SetFont toolbox call (in QuickDraw II). Of course, just calling SetFont would tell the computer nothing, since somehow we need to convey the particular font we wish to set. That information is considered *input* to a toolbox call. The way information is passed to the toolbox varies with the language in which you're programming, as we'll see later in this chapter. But for now, suffice it to say that this tool requires we submit a handle to the information in memory that contains the characteristics of the font we wish to use.

The opposite occurs when our program needs to know what the current font is. To obtain that information, the program uses the GetFont tool call (also in QuickDraw II). In this case, we have no input for the tool, because it assumes we wish to know the current font. But when the tool routine has finished, the tool will need to give us information: its *output*. Again, the way this output reaches our program depends on the language in use, as we'll see.

The third possibility consists of tool calls that require input before running and produce output when they're finished. An example of such a call is StringWidth. This call needs to know the location in memory of the text *string* that is to be measured by the tool. When the tool has measured the desired string, the result is then produced in the form of an integer counting the number of picture elements on the screen the string occupies.

As you begin to program for the IIGS, you'll quickly see that most of the tools fall into the types that involve input, output, or both. Relatively few tools are freestanding functions.

PARAMETERS AND THE STACK

The best way to illustrate the way parameters are passed to and from toolbox routines is to examine what happens to the stack during a toolbox call. Only assembly language programmers will have to bother with direct stack manipulations. High-level programmers will have the impression of using other means of passing parameters; in reality, the load file generated by a high-level compiler and linker will be using stack mechanics, just like the assembly language programmer. Everyone, therefore, can benefit from this explanation.

Input Parameters

When we called SetFont, above, we had to pass the handle to the font we wished to become the current font. To do this with the stack, an assembly language program would first push the handle onto the stack.

As you recall in our discussion about the way the stack grows down and the contents of the stack pointer (SP) decrements when an item is pushed onto the stack, the illustration makes perfect sense.

Once the parameter is on the stack, the program can call the toolbox routine. When the routine runs, it automatically looks to the stack for the information it needs — a handle in this case — and pops it from the stack without any intervention from the assembly language program. Once the routine is finished, control of the program returns to the assembly instructions, and the stack is returned to its previous status.

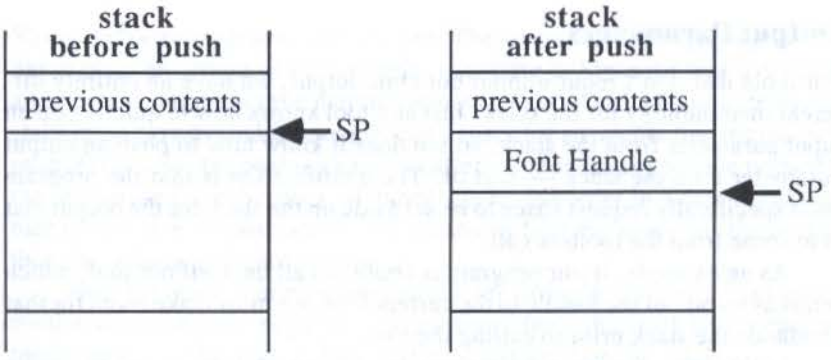


Figure 5-14. Pushing a handle on the stack.

Many toolbox calls require that more than one parameter be pushed on the stack before being called. Parameters can be of unequal lengths, such as a sequence of pointers and table arrays. When a toolbox call expects multiple parameters, those parameters must be pushed on the stack in the proper order so that the tool will pop them in the right order. For example, if you push an integer and a pointer onto the stack in that order, the tool must expect to pop a pointer and an integer from the stack — the reverse order in which they were pushed onto the stack. If the tool expects a 2-byte integer and instead pulls half of a 4-byte pointer, then the tool will surely fail and cause a system error. The order of multiple parameters for each toolbox call is detailed in the *Apple IIGS Toolbox Reference* manuals. Observe parameter order religiously.

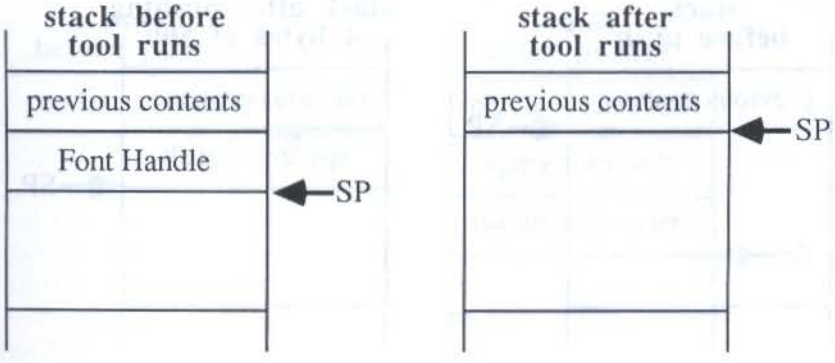


Figure 5-15. The tool pops the handle from the stack.

Output Parameters

For tools that don't require input but emit output, we have an entirely different methodology for the stack. Just as a tool knows how to quietly pop an input parameter from the stack, so too does it know how to push an output parameter onto the stack — sort of. The qualification is that the program must specifically request space to be set aside on the stack for the output that is to come from the toolbox call.

As an example, if our program is about to call the GetFont tool, which sends as its output the handle to the current font, we must make room for that handle on the stack prior to calling the tool.

In this case, we must make room for 4 bytes of data, since the handle coming back will be 4 bytes long. The empty space usually consists of 0s. The importance of this procedure is that the stack pointer must decrement so that the tool won't overwrite important stack data with the output. If multiple parameters are to be output by a tool, then enough space must be reserved on the stack for them all.

When a tool supplies output as its result, the tool is said to *return* a particular kind of data. In good programmer's jargon, GetFont returns a handle to the current font. Get used to hearing "return" as a way of identifying a tool's output.

Input and Output

Stack manipulation for a tool that both takes input parameters and returns output parameters is only slightly more complicated in that it combines the actions of the two individual actions. To demonstrate, we'll use the

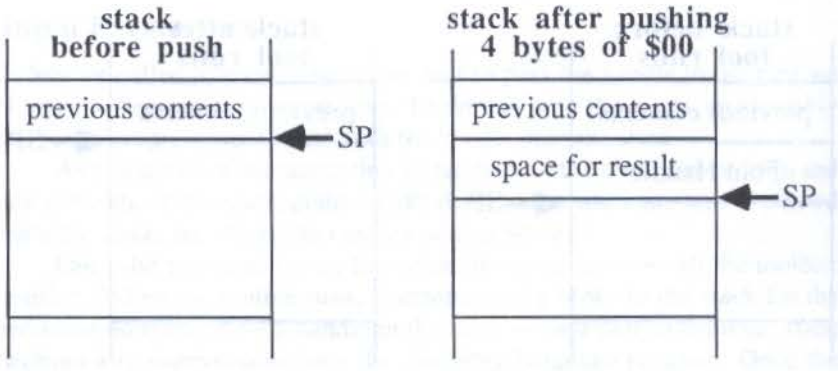


Figure 5-16. Preparing the stack for output.

StringWidth tool call described earlier. The assembly language programmer has to plan the actions of the tool prior to calling it. Since the tool will return an integer representing the picture element (pixel) width of a text string, the stack must have available enough empty space for an integer that the tool will push. Of course, the pushing happens after the tool has popped the pointer to the text string from the stack. In other words, the order in which the assembler program must push space and pointer is first the space for the output, then the pointer to the text string.

Then the assembler program can call the StringWidth tool. The toolbox routine automatically pops the string pointer from the stack and writes the resulting text width integer into the space left for it on the stack.

Again, by preparing the stack prior to a tool call, the assembly language programmer (or a high-level language program under direction of the compiler) can lay in any number of blank spaces and input parameters for complex tools that have multiple input and output parameters.

High-Level Parameters

Although a high-level compiled language usually doesn't bother with stack manipulation, you will still have to furnish input parameters and know how to obtain output parameters after IIGS toolbox calls. Pascal and C are remarkably similar to each other in working with toolbox routines that accept and return parameters. Therefore, we'll generalize here a bit to give you an overview of the mechanics of handling parameters in these languages.

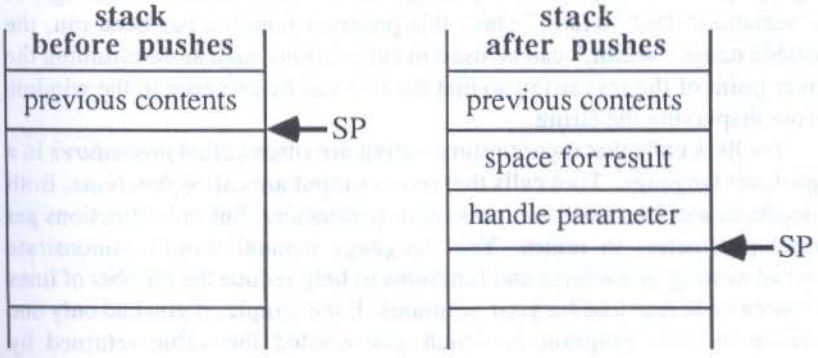


Figure 5-17. Pushing space and a parameter onto the stack.

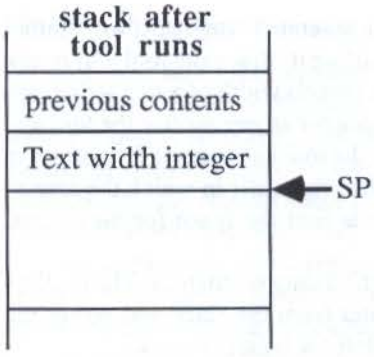


Figure 5-18. Toolbox routine leaves only its output on the stack.

Both languages have a syntax that lets the programmer assign the value returned by the tool to a variable name. Input parameters are attached to the toolbox call by placing them in parentheses immediately following the call in the program listing. Except for minor variations in punctuation, the following listing shows how the StringWidth toolbox call might look inside a high-level language listing. The call will measure the width of a text string whose location in memory has already been identified as a pointer called "txt" earlier in the program:

```
width = StringWidth(txt);
```

Behind the scenes, the pointer (named "txt") is placed on the stack, the StringWidth routine called, and the width integer placed on the stack. The language goes one step further by assigning the value of the width integer to the variable called "width." Once this program function has been run, the variable name, "width," can be used in calculations such as determining the center point of the text string so that the text can be centered in the window before displaying the string.

Toolbox calls that do not return output are often called *procedures* in a high-level language. Tool calls that return output are called *functions*. Both procedures and functions can pass input parameters, but only functions get output parameters in return. Your language manual should demonstrate ways of *nesting* procedures and functions to help reduce the number of lines of source code required for your programs. For example, if you had only one instance in your program in which you needed the value returned by StringWidth for use as an input parameter in a different toolbox call, you could skip the step of defining the "width" variable. Instead, use the

StringWidth(txt) function itself as an input parameter. Its returned value (the width) will be passed directly to the other call.

In the next chapter, we'll dig deeper into the parameters passed in toolbox calls, while demonstrating a key concept for IIGS programming: the record.

CHAPTER 6

Understanding Records

If you have programmed graphics on an Apple II, you probably noticed that as soon as you give the command that draws a graphics shape on the screen, your program loses contact with the object. For example, to move a shape from one location to another, your program must erase the existing shape on the screen and then redraw the shape at other coordinates. This means that you must supply the coordinates, size, color, and other parameters for the shape each time you draw it. Even the command for erasing the shape requires coordinate and size parameters so that the program will rub out the picture elements on the screen precisely where the shape is drawn.

Even if you have not programmed graphics before, you can imagine how tedious it can become to continually specify coordinates, size, and other parameters each time you wish to draw or move an object on the screen. Now imagine a graphically based environment such as the IIGS, and you can see that keeping track of multiple, overlapping windows, for instance, could be a nightmare if your program has to reinvent the wheel each time a window is brought forward as the active window. Fortunately, the IIGS toolbox programming environment greatly simplifies the maintenance of windows and other programming items. Specifications about these items are stored in memory as a list of parameters, a list commonly called a *record*.

RECORD BASICS

The concept of records will likely be radically different from anything you've encountered in non-Macintosh-like programming. Yet once you understand the way they work, much of IIGS programming should be easier to grasp.

Let's use a hypothetical window record as an example of how records are created and how important they are in a program.

To create a new window for the screen, you must predefine a large number of parameters that the Window Manager will assign to a given window. The specifications include, among others:

Coordinates of the top left corner of the window

Coordinates of the bottom right corner of the window

Coordinates of the top left and bottom right corners of the active work area inside the window

Pointer to the memory location containing the text of the window's title

Components of the frame the window should have

There are many more features to a IIGS window, but we'll save those for our in-depth examination of the Window Manager, in Chapter 10. For now, the above parameters specify enough information for us to visualize the results of various values we assign to window parameters.

If we were to take the information about a window and store it in a protected area of memory — one that won't get overwritten by another program segment — then our program will have a fixed place to retrieve information about our window anytime during execution of the program. The list of window parameters in memory might begin as shown in Figure 6-1.

To make a change to the parameter list once a window has been created, the program does not fiddle with list items directly. Instead, the toolbox provides many routines that adjust whatever parameter we wish. For example, if we wish to rename a window, we can make a different toolbox call that reaches into the window's feature list and adjusts the pointer reference so that it points to the memory location storing the new title's text. Figure 6-2 shows the situation before and after the toolbox call that changes a window's title.

Calling this parameter list a "record" helps us visualize the way information about an object is stored in memory. It follows naturally that a record, such as a window record, maintains an inventory of the window's features and settings at any given moment. In fact, even if the user closes the window by clicking the close box on the title bar, the window record will

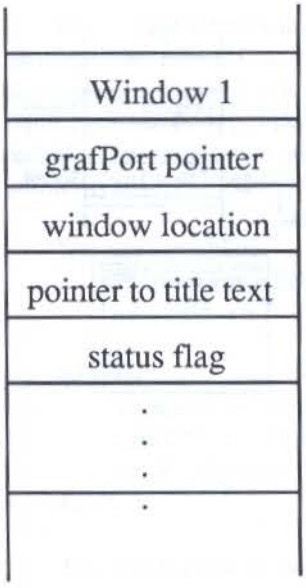


Figure 6-1. Hypothetical window record in memory.

not necessarily be erased from memory. With the record still intact in memory, the user can reopen the window (by giving the appropriate menu command) and watch the window appear in the same size and location, and displaying the exact same content as the window had when it was closed.

Records as Snapshots

The above example demonstrates how a program can manipulate a record's contents to effect a change. A record, depending on the tool set managing it, can also report current conditions. We'll see later on, for example, that one often-called record logs whether the screen cursor was in a menu bar or in a window when the mouse button was pressed. The toolbox monitors what's going on and automatically adjusts the content of the record to reflect the present state of affairs. The program, then, can read the contents of the record and make decisions based on those contents.

Record Pointers

Records in the IIGS vary in length, ranging from only a handful of bytes to hundreds of bytes for complex objects. Most of the time, when you initiate a

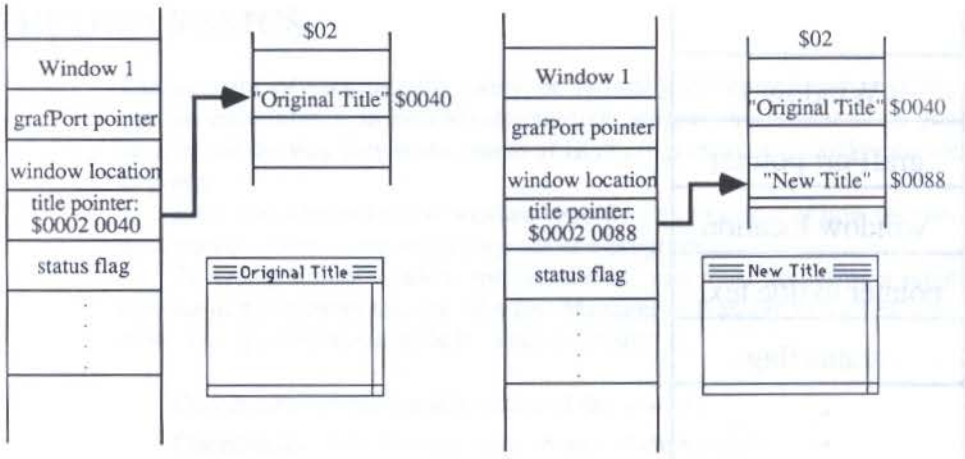


Figure 6-2. Changing a window title parameter.

new object, such as a new window or menu, the toolbox routine you use to create the object also finds and reserves room in memory for the record and may even plug in values for a standard, or *default*, record. The act of creating such a record often returns a pointer to the beginning of the record.

In a high-level language program, the toolbox call that creates the new object is usually written as a function that returns a value which is assigned to a variable name (see Chapter 5). Therefore, if you create a window, the function might look like this:

```
Window1 = NewWindow (window1Data)
```

"Window1" is a variable that, after the call, contains a pointer to the window record created with the statement. If the program needs to create a second window, a second statement could be added to the program:

```
Window2 = NewWindow (window2Data)
```

with "Window2" now capable of standing in for the pointer to the second window's record. This is convenient because other toolbox calls that inspect or modify the contents of a record ask that you pass the pointer to the record you identify as the one to be worked on — such as changing the title of Window1 or resizing Window2. Therefore, you get a comfortable feeling of working with objects according to readily identifiable names you assign to their pointer variables. Let's see how this works in real life.

One of the criteria in a window record is a pointer to the location in memory where the text of the title of the window rests. If you wish to change the name of a window, you use the SetWTitle toolbox call. This call has two input parameters: a pointer to the title text, and a pointer to the window's record. The program would have to (1) declare the named variables as pointer types, (2) call NewWindow, and (3) call SetWTitle.

```
Declare variables:
    Window1 is a POINTER
    newTitle is a POINTER

BEGIN
    Window1 = NewWindow(Window1Data)
    newTitle = "Worksheet 1"
    SetWTitle (newTitle, Window1)
END.
```

The fact that these two variables are pointers, while of crucial importance to the SetWTitle call, becomes largely hidden to us once they've been declared as pointers. Instead, we conceive of the pointer variable names as standing for the actual objects (the title text and window). This is an excellent example of how a high-level language can disguise much of the "dirty work" that must be addressed directly in assembly language.

GETTING AND SETTING DATA

When you program in a record-intensive environment, such as the IIGS toolbox, your program must often dig into a record to retrieve the current state of the object specified by that record. Conversely, the program will often have to change a specification about an object by writing new information into specific parts of the record. It is certainly more cost-effective (in terms of programming and execution time) to read and write only the desired specification(s) from a record than either "thumbing through" a record to reach the item you want or to rewrite an entire record when only one specification changes. Fortunately, any IIGS object that maintains a record also has toolbox calls handy that read and write individual specifications to the record. A tool that reads information from a record begins with the word *Get*, while a tool that writes information into a record begins with the word *Set*.

We saw an example of Get and Set tools in the last chapter when demonstrating the way the stack operates while passing parameters to a tool. The GetFont and SetFont tools were actually reading and writing information — font handles, you'll recall — in a record associated with the current

window (technically speaking, font handles are part of the GrafPort record, a component of the window record).

Get and Set tools are often used closely together. Before changing a parameter in a record, your program will probably first perform a Get to see what that current parameter setting is. If it is not the desired setting, then the program issues a Set command to make the change. But if the existing setting is fine, then the program can skip the Set routine, thus speeding its way on to the next operation.

PRIVATE DATA

Records, especially ones such as window records, allow you to create multiple windows without any fear that parameters for one might get mixed up with the others. For example, if your program must display two windows of entirely different appearance — one may be a window with scroll bars and title bar; the other may be a smaller, plain window — your program will create a separate window record for each when you create the windows. Adjusting the location or size of one of the windows will affect the record of only the one window undergoing adjustment. Parameters for the other window are not in any danger of being accidentally adjusted.

Information contained in records is called *private data* because no other object shares or is necessarily aware of the data for any other object.

DATA TYPES

As you begin studying the toolbox calls and records in the *Apple IIGS Toolbox Reference* volumes, you will observe that parameters are often referred to by their *data type*. Pascal and C programmers will already be comfortable with the idea of data types, because both languages are strongly based on the concept.

For those who don't have that background, a data type is a declaration or statement that a piece of data is going to be a certain kind of number, like an integer (whole number within a specified range), or a string (a group of text characters). By specifying the data types for all toolbox call parameters and records (detailed in the *Toolbox Reference* volumes), Apple helps us use these functions in our programs and tells us how much memory space each parameter occupies.

High-level programmers traditionally think in terms of data types, while assembly language programmers concern themselves with the size of each piece of data. Let's look at the data types and sizes each kind of programmer will encounter, explaining the terminology along the way.

Fixed Length Data

High-level programmers of the IIGS rarely deal with data that is smaller than an *integer*. According to convention, an integer is 2 bytes long. Because of its 16-bit binary length, it can represent any number from −32768 to +32767. Since the 65816 microprocessor handles information in 16-bit-wide paths, the integer is generally the smallest data type that will be specified for a toolbox call parameter.

There are exceptions, however. Occasionally a parameter can be specified by as little as 1 *bit* or 1 *nibble* (4 bits). When this happens, several small parameters are often packed together to make up a 16-bit value, thus keeping the 2-byte width of information intact. When there is a single byte-length parameter, it will be padded with 0s to fill up the 2-byte space.

Assembly-language programmers call a 2-byte collection of data a *word*. Therefore, sometimes you'll see reference to a word data type. "Word" in a 16-bit environment such as the IIGS simply means that the data is 2 bytes long.

When the information to be passed to a tool requires more than 2 bytes, such as a pointer to a memory address (which needs actually 4 bytes — 2 for the address within that bank, 1 for the bank number, and 1 to fill out the data to an even word length), the data type called into action is the *LongInt*, which stands for *Long Integer*. A LongInt is 4 bytes long. In assembly language, a 4-byte space for data is called, simply, a *long* (and sometimes a *long word*).

Data containing a memory address, as noted earlier, must be expressed within 4 bytes. Addresses are either pointers or handles (pointers to pointers). Just as you assign a pointer to a LongInt (4-byte) variable, so, too, do you assign a handle to a LongInt. More than likely, you won't come in direct contact with the actual value, but let the variable name carry the "baggage" of the handle value. When it comes time to pass that handle to a tool call, you'll just plug it into the high-level language function as we demonstrated for the pointer to a window record, earlier.

Booleans

A common Pascal data type is called a *Boolean*, named after the nineteenth-century mathematician George Boole (see Appendix A for further information about Boolean arithmetic). A IIGS tool requires or issues a Boolean value when it is looking for "yes" or "no" kind of information. By convention, a "yes" is signified by any number other than 0, a "no" is signified by 0. In IIGS programming, a Boolean is 2 bytes long. The common convention is to indicate a "yes" Boolean by $FFFF (clearly non-zero), and a "no" Boolean by $0000.

Variable Length Data

The IIGS toolbox generates a number of variable length *data structures* in memory. A record is a good example of such a data structure. But records from different tool sets are of different lengths, because each tool set has its own list of parameters. Since the toolbox allows you access to key parameters within a record by way of its many routines, you won't be attempting direct access to items within a record. At most, you'll need only the pointer to the record to accomplish any manipulation of parameters inside an existing record. Consequently the exact byte count of a record won't be of importance unless you're short on memory allocation for a new record.

There is, however, one variable length data structure that you may be examining one byte at a time. Called a *string*, it can contain any kind of textual material, from words in a text box to the short title of a window. When a string data structure is part of a record, the usual connection is via a pointer to the string data in memory. The actual construction of a string in memory varies with the type of strings you use, either a Pascal or C string. Your language manual will guide you in the proper form.

Manipulating information inside a string data structure may be necessary, depending on the program. For example, a portion of a program may convert the first letter of each word in the string to a capital letter. For your program to perform this operation, it must work its way through the string, looking for space characters, testing the character after each space, and subtracting 32 from the ASCII value of lowercase letters to make them uppercase (see the ASCII chart in Appendix B for the reason behind the subtraction). Of course, Apple has provided a tool that helps in searching through the string data structure, but your program must guide it each step of the way as it thumbs through the structure. It's not the same as a tool call that modifies a set location in a record.

Custom Data Types

Throughout the *Apple IIGS Toolbox Reference* manuals you will see data types, particularly in QuickDraw II, that seem to be graphically oriented data types, such as *point* and *rect*. Data types with these names don't occur naturally in Pascal or C, but they can be added to programs in either language (using the *type* facility in Pascal; *typedef* in C) for convenience. Your high-level language compiler will probably include these type declarations in the supplemental source code files associated with QuickDraw II.

We'll examine these custom data types in more detail in Chapter 8, but a sample is in order here. A point on the screen is determined by its horizontal and vertical coordinates — the number of picture elements across

and down from the upper left corner of the screen. The horizontal and vertical components of the coordinate can be each measured by an integer. But if we need to refer to coordinate points often enough, it becomes easier to refer to those two integers as a "point" whose data component is 4 bytes long.

Going one step further, if a rectangle is a common object — as it is in IIGS windowing — then we can establish a data type, called *rect*, which has as its data the coordinate points of the rectangle's upper left and bottom right corners. Those two points, contained within 8 bytes of data, supply all the information necessary to establish the location and size of a rectangle on the screen. Hence, one of the components of a window can be an 8-byte chunk of data type RECT that supplies important information about the appearance of that window.

You'll encounter similar custom data types in the IIGS toolbox reference material. Also, don't be afraid to define your own data types in programs when they will aid you in keeping a large number of items and concepts straight.

Before jumping into the key tool sets, we have one last concept to discuss: event-driven programs. This will be the heart of your programs if you wish them to follow Apple's *User Interface Guidelines*.

CHAPTER 7

The Main Event

So far, we've been looking at some of the components that go into an Apple IIGS toolbox-based program. Quite likely, you haven't been able to see how these pieces go together. That's what we'll be doing in this chapter: seeing the main structure of a IIGS program. At the same time, you'll be introduced to some program organization concepts that professional programmers use to make their jobs easier.

NONEVENTS

There's a strong likelihood that if you've done any programming in the past, it has been along straight procedural lines. By that we mean that you write a program from beginning to end in precisely the same order that things take place on the screen. For example, if the program is an arithmetic drill for elementary school students, the program may start with an opening menu of levels of complexity of the problems the student wishes to work on.

The program essentially stops when it displays the menu. It waits for the user to type a letter or number to indicate the menu choice. In BASIC, for example, the INPUT statement literally halts program execution in its tracks, waiting ever patiently for someone to type a character and the Return key.

Once a choice is made, the program jumps to the part in the program listing that contains the arithmetic problems at the level selected by the stu-

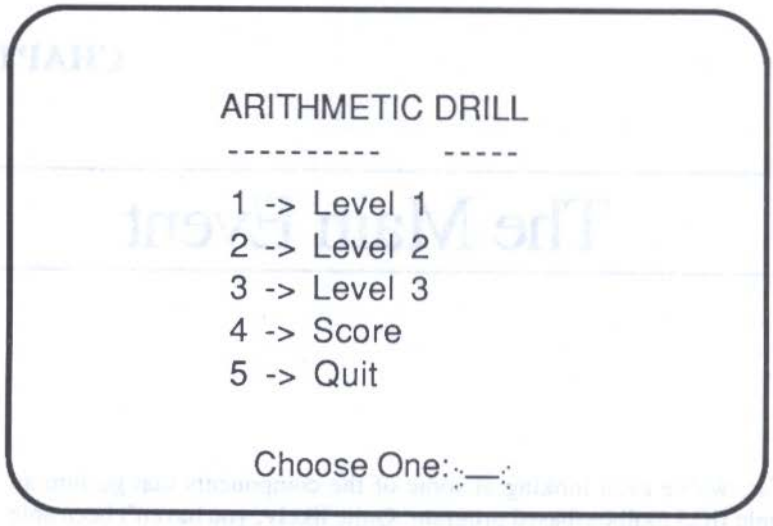


Figure 7-1. Procedural program menu.

dent. After, say, ten problems, the program halts again as another INPUT statement waits for the student to type in the response to a question such as, "Do you wish more? (Y/N)." Questions such as this are actually tiny menus, here with two possible choices, Y for Yes and N for No.

So it goes throughout the program. Sections of action are punctuated by pauses for menu selections. In block form, the program's code might look something like Figure 7-2.

A more sophisticated program, along the lines of a word processor or a spreadsheet (like Multiplan or Lotus 1-2-3), is less procedural in nature, but removes the user from direct access of commands by one or more steps. For example, in the Multiplan user interface on the Apple II, the IBM PC, and other computers (but not the Macintosh), you have free reign over entering data in the spreadsheet's cells by pressing any number or letter keys as well as arrow keys and Return. But to gain access to the program's built-in commands, you must press the Escape key. This action takes you one step away from the cell mode and puts you into a command mode. A selection of menu items appears at the bottom of the screen, and you use the keyboard keys to make your menu selections. In block form, this kind of program looks like the illustration in Figure 7-3.

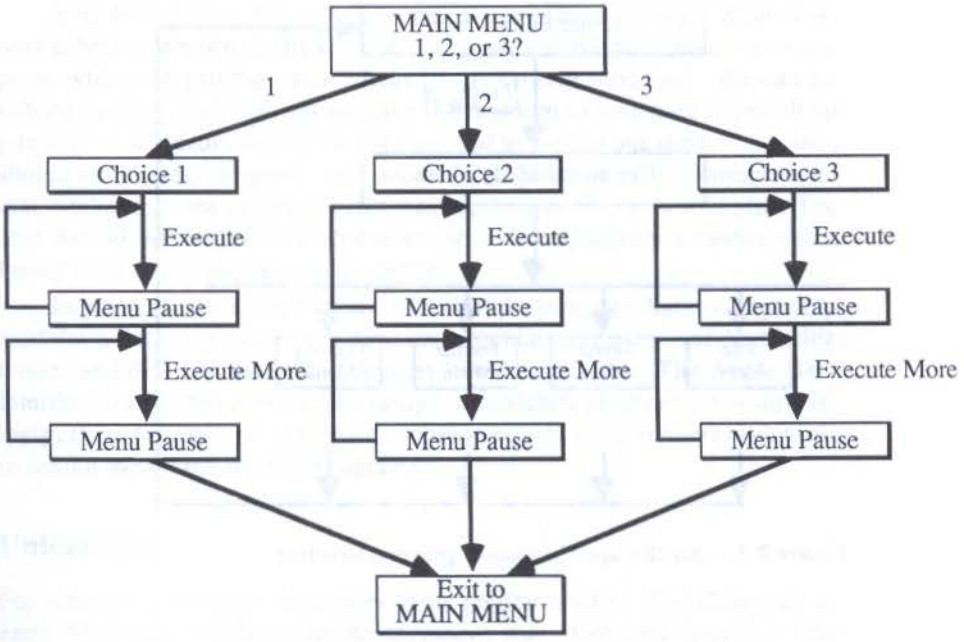


Figure 7-2. Procedural program overview.

MODALITY

If you study the above application structures, you'll observe that each restricts the user to certain *modes* of operation at various times. When the arithmetic program is presenting problems on the screen, the student may be locked into going through all ten problems in a set before exiting the current difficulty level. When one of the menus appears, particularly one of the submenus (the "Y/N" kind), the program offers a substantially restricted list of alternatives. To quit the program entirely (without turning off the computer), the student may have to work his or her way back through multiple menu levels until reaching the hallowed Main Menu, which finally offers a Quit option.

In the Multiplan kind of program, the modality is much more obvious, because you are either in data entry mode or in command mode. When you are in the latter, the command menu at the bottom of the screen is active, and you can select items there by moving the highlight bar to the desired word (using the arrow keys) or typing the first letter of the desired command word. You couldn't indicate a command while in data entry mode, because the pro-

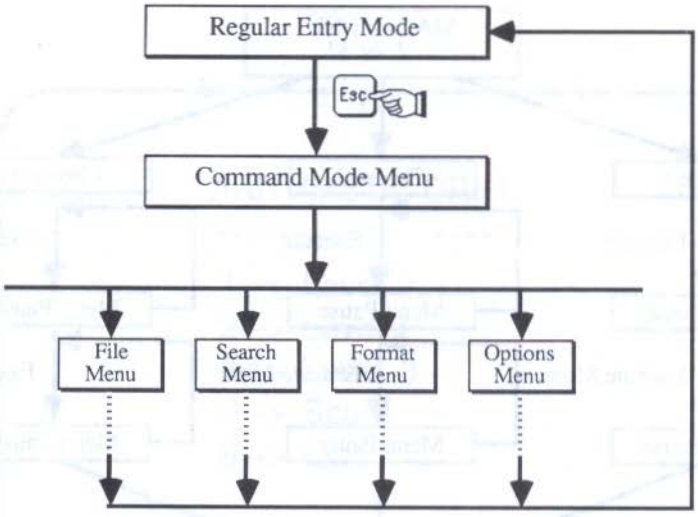


Figure 7-3. An "Escape-Command" program structure.

gram interprets the arrow key as a cell pointer mover and letter keys as characters worthy of entry into spreadsheet cells.

No Modes

From a user's point of view, modes can be very distracting. In the arithmetic program, for instance, a student intent on running through twenty-five problems at a particular level is interrupted after each set of ten with a question about solving more problems. And then, the student has no choice but to solve problems in multiples of ten. As we've also seen, quitting the program requires remembering the steps needed to backtrack to the Main Menu. The thought process is frequently disturbed, as the user puts aside arithmetic solving skills while focusing on the details of program operation.

A Multiplan-like modality suffers from a similar distraction. If you're busy entering information into a spreadsheet and wish to change the alignment of a cell's contents (i.e., change a label cell from its natural left-aligned format to centered), you must take your mind off the spreadsheet and its contents by taking a giant step away from the spreadsheet and going into command mode. After you've made the adjustment, then you must make sure you are back in data entry mode — otherwise your key presses will be interpreted as commands.

You should strive for modelessness in your program design. Modelessness is best characterized by the freedom to use any program function at any point while the program runs. Real life is largely modeless. We can be talking on the telephone, yet we have the freedom to jot a note or punch up numbers on a calculator. We're not confined to "telephone mode." So, too, should a modeless program offer access to all possible menu commands so that it takes but one action to print, save, paste, or change a font size. The user should not be forced to focus attention on a program's modes while losing track of the application's content.

In practice, it may be impossible to build a program that is completely modeless, but it is possible to reduce the most blatant instances of distracting modes and disguise the rest in devices like dialog boxes. The Apple User Interface Guidelines promote the design of modeless programs. Not surprisingly, then, the IIGS programmer's toolbox equips programmers with tools to design modes out of their programs.

Unlearning

For many programmers, this takes some getting used to. The slow start of early Macintosh applications development was attributed largely to the rethinking required to design a program in this user interface environment. In fact, in some editions of the Mac programmer's bible, *Inside Macintosh*, a paragraph heading in the first chapter reads, "Everything You Know Is Wrong." In that paragraph is another statement conveying the meaning of that heading: "You'll probably find that many of your preconceptions about how to write applications don't apply here." Since the IIGS programmer's toolbox and user interface are patterned after the Mac's, those same words apply here.

So if modes are "out," what in the toolbox helps you design modelessness into your programs? It's something called an *event*.

FROM MODE TO EVENT

Everything that exists as input to the computer — a press of a key, a mouse button press, even a character coming in through a serial port — is called an event. Events have many characteristics. In fact, you might consider events as objects because the toolbox Event Manager generates an event record in memory that contains all the attributes of an event. Your program, then, looks into the event record, decides what kind of event it is, and acts accordingly.

Additionally, the Event Manager can keep track of a series of events that happen too quickly for the computer to handle all at once. The list of events is kept in a section of memory called the *event queue*. For example, if you want to close two overlapping windows that are open on the screen, you can quickly click the Close boxes in the two windows. You might click the box on the second window before the program has finished removing the first window from the screen. Both clicks of the mouse go into the event queue. The program immediately takes the first mouse click from the queue (popping it from the stack, if you will). As soon as the program closes the first window and activates the second, it *polls* the event queue to see if anything is in there. In this case, the second mouse click will be there. Pulling this event from the queue, the program closes the second window. The user benefits because he or she doesn't have to wait for the screen action to catch up with two closely spaced mouse actions.

THE EVENT LOOP

Building the event mechanism into a program entails a program structure that may be entirely new to experienced programmers. Actually, if you haven't programmed a computer before, you'll have an easier time understanding and applying the structure required for an *event-driven* program.

The central, "living" section of an event-driven program is called the *event loop*. It consists of two types of instructions: (1) a function that reads the event queue to see what kind of event has taken place, and (2) several statements that test the event pulled from the queue to determine specifically what kind of event it is and what the program should do next for that particular kind of event.

```
If there has been an event,

  - was it a key press?
      (if so, display the character)

  - was it a mouse button press?
      (if so, where was it?
         if in a menu, pull down the menu;
         if in a window, make it the active window;
         if in a window's scroll bar, scroll accordingly)

Go back to the beginning of the loop until "Quit" has been
 selected from the menu.
```

Figure 7-4. Event loop structure.

Instructions for each event type are located elsewhere in the program code as stand-alone subroutines. When execution branches to one of these routines, the action takes place (it could be a routine responding to a mouse selection in a menu, a press of a keyboard key, or a click of the mouse on a scroll bar). When the routine is completed, execution returns to the event loop in the location from which it branched in the first place. At the end of the loop section, program execution returns to the beginning of the loop, where, usually, the event queue is polled once more.

Therefore, when a program doesn't appear to be doing anything, it is actually racing through the event loop, waiting for some kind of event to take place. If the event is a keystroke, for example, a word processing program might jump to a subroutine that instructs the computer to display the character on the screen and store the character in memory as part of the document being built.

Every key press, then, is an event. That means, of course, that event polling and subroutines must take place at a very fast pace for someone capable of typing 100 words per minute to keep from typing faster than the computer can accept events. At 100 word per minute, for example, the machine must handle 10 complete events (including their subroutines) each second.

Of course, there is no guarantee that any program will automatically accommodate 10 events per second. It is very possible to design a cumbersome event loop and keystroke subroutine that would not let a typist get much past 50 wpm before the event queue overloads and the program loses characters. That's where experience and good program design come into play — something you'll acquire in time.

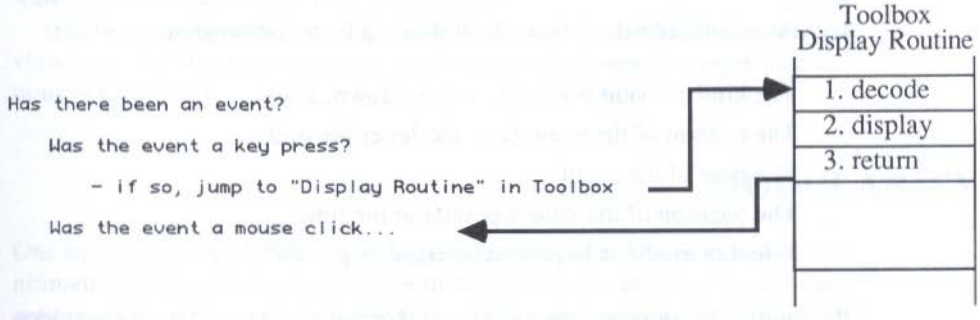


Figure 7-5. The event loop branches to subroutines.

EVENT PROGRAM STRUCTURE

There is no rigid structure for an event-driven program. About the only requirement is that the program begin with the necessary initialization calls to "wake up" all the tool sets that your program will be using.

Next, depending on the content of your program, the program should perform any other program-specific initializations, such as creating an opening window and menus. This initialization section should include all setup procedures that your time-critical event subroutines will need. Perhaps you will use this opportunity to create window or control records in advance for objects that won't show up right away (although this may not be necessary). Program execution should proceed from the end of these program initializations directly to the event loop.

Even though execution jumps to the event loop, the loop does not have to physically follow the initializations in the source code listing. Pascal, C, and assembly language let you define the actions for each event subroutine anywhere in the program, assigning a readily identifiable name to each action. Consequently, you can define your subroutines in the middle of the code and place your event loop — the main program — at the end of the code. Or you can place the event loop after the initializations and put all the subroutines at the end of the program. Figure 7-6 demonstrates two ways you can structure an event-driven application.

EVENT DECISIONS

The event loop mechanism really owes its power to the event record, which the Event Manager automatically creates in memory when an event takes place. If multiple events are stacked in the event queue, only one event — the next one to be acted upon — has its information posted to the event record.

An event record contains the following information about an event:

The kind of event it is (e.g., mouse down, key)

The content of the event (e.g. the letter pressed)

The time of the event

The location of the mouse pointer at the time

Whether modifier keys were pressed (e.g., Shift key)

By reading one or more items of an event record, statements in an event loop can test any detail about an event. For example, if the event was a mouse-down event, then the event loop might branch to a subroutine that handles

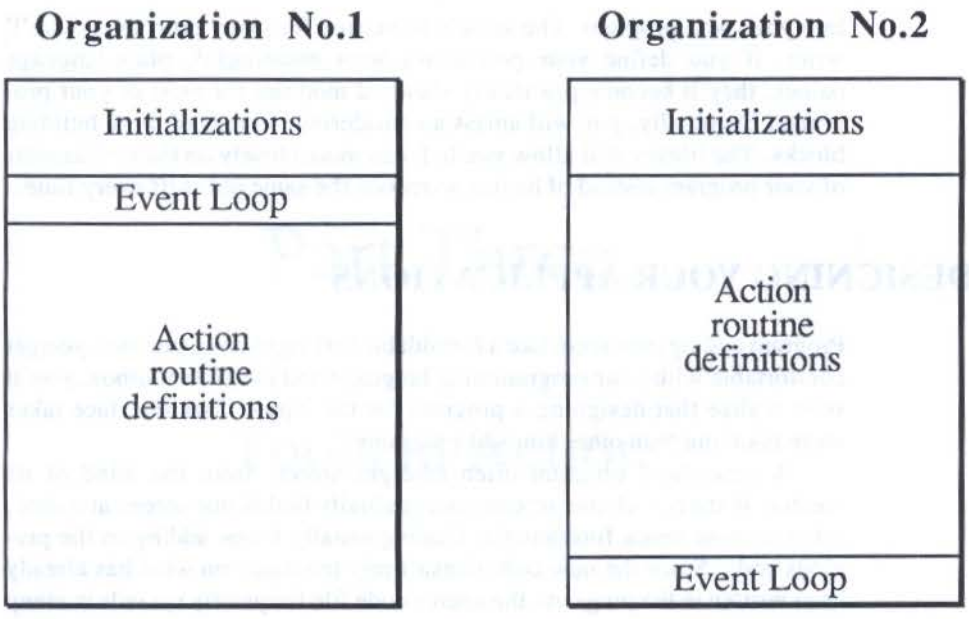


Figure 7-6. Two possible program structures.

all mouse-down events. That routine may subsequently check to see if this mouse-down event's time was within a specifically defined interval from the last mousedown event, and if the mouse pointer was located inside a particular coordinate range on the screen. If so, then it performs an action that was programmed as an action for double-clicking on a particular icon. After execution of that action, the program zips back to the event loop, where it waits to test the next event that comes along.

We're looking at the event loop and its actions from a highly superficial view here. We'll get into much greater detail on the Event Manager's operations in Chapter 9.

MODULARITY

One beneficial result of the event-driven program structure is that you will ultimately program your application in modular form. In other words, each action — the action instructions for a Copy menu command, for example — will be its own module, written in your source code program. You'll be able to take that very same module and, without blinking, incorporate it

into your next program. The same will be true for many procedures you'll write. If you define your procedures with meaningful, plain-language names, they'll become practically standard modules for most of your programs. Eventually, you will amass a considerable library of these building blocks. The library will allow you to focus more closely on the new aspects of your program instead of having to recode the same old stuff every time.

DESIGNING YOUR APPLICATIONS

Program coding may seem like a formidable task right now, but once you get comfortable with your programming language and the IIGS toolbox, you'll soon realize that designing a program for the Apple User Interface takes more planning than other kinds of programs.

A procedural program often emerges slowly from the mind of its creator. It starts with one screen, and gradually builds one screen at a time, often without much forethought. Coding usually keeps adding to the previous code. Since the new code doesn't rely too much on what has already been written in the program, the source code file frequently sprawls in many directions at once.

Planning an event-driven application, however, will put an immediate end to procedural thinking. Instead of designing screens punctuated by menu choices, think of your program in terms of event actions — menu choices, mouse clicks, key strokes. By defining the actions in advance, you often find refreshing ways of portraying tired topics. That's really what the User Interface Guidelines set out to promote.

Remember, too, that with the prospect of high-resolution, color graphics, you can turn windows — which take on their own physical entities in your program — into metaphors of real-world objects, such as accounting ledgers, music machine control panels, and so on. Don't just limit yourself to the bland windows or screens of yesteryear. Think visually. Turn your ideas into colorful pictures accompanied by inspiring sound. Recreate images from real life on the screen.

This kind of thinking has brought us many wonderful software products on the Macintosh. It will happen on the IIGS just as easily, with the added enticement of color and superior sound.

Part Three

Tools in Action

CHAPTER 8

QuickDraw II

We begin our exploration of the most important tool sets with QuickDraw II, the tool set that governs the Apple IIGS's native mode screen display. Although this tool set's name might imply that you'll use it only for graphics — drawing circles, squares, squiggly lines, and so on — QuickDraw II governs the display of everything on the screen. Windows, menus, controls, and text all rely on QuickDraw II to do the actual "drawing" on the screen.

As a programmer, you will come into direct contact with QuickDraw II via its own tools only as often as the kind of programs you write need them. Programs that have the user draw or paint on the screen will use QuickDraw II actively. Animation programs will also make many QuickDraw II calls. Text-oriented programs, however, will have little direct contact with QuickDraw II calls. They'll be using QuickDraw II plenty, though. It's just that the calls will be made by other tool sets, such as the Window Manager and text tools. Your understanding of these "higher level" calls may require a thorough knowledge of what QuickDraw II is doing and requires.

Therefore, no matter what kind of toolbox programming you intend to do on your IIGS, QuickDraw II is the place to start examining tools in detail. Many concepts here will be crucial in the design of your programs, as indicated by QuickDraw II's relatively low position in the tower of toolset building blocks illustrated in Figure 5-12, on page 80.

QUICKDRAW II VS. QUICKDRAW

The conceptual framework for QuickDraw II came from the original QuickDraw, masterminded by Bill Atkinson for the Macintosh. Coupled with the speedy processing of the Mac's 68000 32-bit microprocessor, the original QuickDraw presented programmers with a large library of tools that simplified the potentially enormous task of writing routines for animation graphics, finely detailed graphics images, text in clearly distinguishable display typeface styles, and the common denominator user interface features of windows, scroll bars, menus, and the rest.

QuickDraw II does the same for the Apple IIGS. While many of the toolbox calls are the same and behave the same in both tool sets, QuickDraw II differs from its forebear in some important matters.

Color

The most obvious, of course, is that the Apple IIGS has been a color computer from the very beginning, while the first QuickDraw focused primarily on monochrome (black and white). The original QuickDraw does have some color facilities, as *Inside Macintosh* will tell you, but since the original hardware was not set up for color video output, this color facility is rudimentary compared to the system built into the IIGS. QuickDraw II, as we'll see later in this chapter, has a sophisticated mechanism for generating a screenful of colors unmatched by most computers in its range.

Screen Resolution

Less obvious differences between the two QuickDraws include the difference in screen resolution between the Macintosh and the Apple IIGS super-high high-resolution mode. Also, owing to the way Apple II video monitors usually create images on the screen, individual picture elements are conceived of as rounded, while the Macintosh screen makes square elements. The Mac's square elements and monochrome screen are responsible for the crisp images displayed on Macintosh monitors. Still, with a quality RGB monitor, the Apple IIGS is quite capable of producing sharp color pictures, especially in its 640 × 200 display mode (i.e., 640 dots horizontally, 200 dots vertically).

The remaining differences between QuickDraw and QuickDraw II are of little significance. QuickDraw II fares well in a head-to-head competition with QuickDraw, so don't feel that you're using a lesser tool than what's on the Macintosh. In some ways, it's even more powerful.

Now that we've acknowledged the differences between the two QuickDraws, we will hereafter be referring exclusively to QuickDraw II as

implemented in the Apple IIGS toolbox. For convenience, we will refer to this tool set as, simply, QuickDraw.

GRAPHING COORDINATES

You can think of QuickDraw drawing images in a two-dimensional coordinate plane. A location in the plane is denoted by a coordinate point consisting of numbers representing horizontal and vertical measures in each direction — just like a coordinate point on a geographical map. The unit of measure is the *picture element*, or *pixel*. A pixel in the coordinate plane is identical to a dot of an image on the video screen. Therefore, a filled square image measuring 10 pixels vertically and horizontally would look like a small square on the video monitor, consisting of 100 dots (a box of 10 by 10 dots).

You might conclude incorrectly that since drawing occurs in this coordinate plane, the plane exists somewhere in the computer's memory. Nothing could be further from the truth. The plane exists merely as a conceptual drawing space, as a convenience for us when we visualize graphics images and the physical space they would occupy if the images were real objects.

It's true that some images, when drawn on the screen by QuickDraw, occupy an area of memory that contains information about each pixel in that image as shown in Figure 8-1.

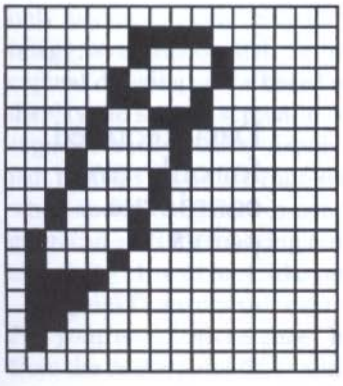


Figure 8-1. A screen image requires data in memory for each pixel, whether black or white.

QuickDraw, however, also manages to draw other types of images without taking up nearly the amount of memory that their on-screen image might imply. For example, a rectangle image occupies little more memory than is needed to specify its two basic coordinates — locations of its top left and bottom right corners — no matter how large the rectangle is.

We'll have more to say about these two ways of drawing images later.

Drawing Space

QuickDraw II designers have established a very specific way of visualizing the range of coordinate measurements in the conceptual drawing space (Figure 8-3). The space is a square of 32K (32768 to be precise) pixels on a side. Coordinate 0,0 is situated at the centerpoint of the drawing space. Numbers increase along the horizontal axis from left to right; numbers increase along the vertical axis from top to bottom. Note that the vertical axis orientation — from top to bottom — is directly opposite from what you may have learned in geography or geometry.

This means that coordinates in the lower right quadrant are the only ones whose vertical and horizontal components are both positive numbers. In other quadrants, one or both components are negative.

We must emphasize that the conceptual drawing space is not real memory or disk space. It is large enough for a programmer to map images that are far larger than a screen can hold. If, for example, you are designing an extensive background scene for a program, you may want it to be larger than one screen so that the user can scroll around a scene that may be the size

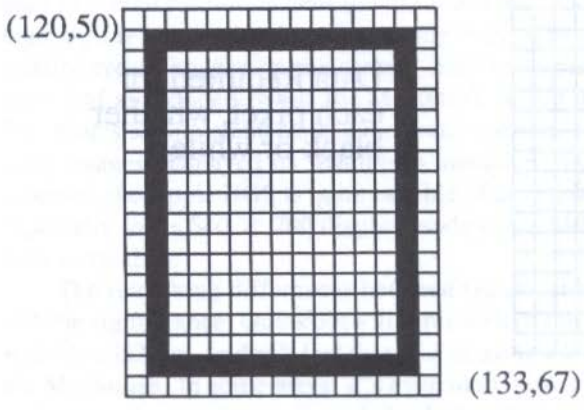


Figure 8-2. A rectangle is defined by only two coordinate points.

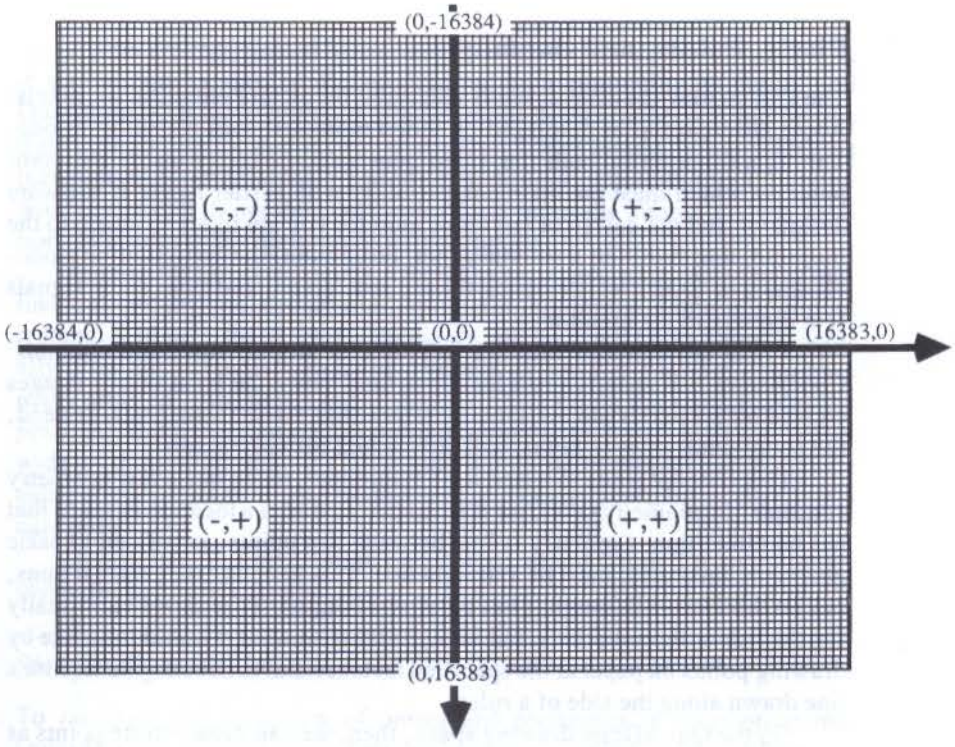


Figure 8-3. The QuickDraw conceptual drawing space.

of several screens. Not all of the image may be in memory at one time — only as much as can be displayed in one screen may be in memory. But the image, as stored on disk, is "mapped" to, say, a four-screen area in the conceptual drawing space based on its coordinates. When the program needs to display the second screenful of the scene, it can load in the necessary image segment from disk and display it. If the computer has enough memory installed, perhaps the entire four-screen image can be loaded into memory. Then, when another segment is needed in response to a user's scrolling, the program can shift the viewing area to another part of the picture with no noticeable delay. It will look like one seamless graphic image to the user.

QuickDraw doesn't care where in the drawing space you plant images. For your own convenience, however, perform your first QuickDraw calls from the centerpoint (0,0) and work in the lower right quadrant until you get a feel for using this drawing space. In that quadrant alone, there's room for an image over 2000 screens large in the densest graphics mode — 64 megabytes of uncondensed picture information.

Pixels, Points, and Rectangles

The QuickDraw drawing space is measured by picture elements — pixels. Apple IIGS pixels, as displayed on video monitors, look round or oblong. This is a function of both the video generation circuitry inside the computer — which must maintain compatibility with earlier Apple II software design — and the design of the video monitor you are likely to attach to the machine, whether it be a television set, composite monitor (color or monochrome), or RGB (which stands for the red, green, and blue color signals sent directly to the monitor).

Pixels are bunched together in perfectly aligned rows and columns. Coordinates that specify pixel locations (and hence the locations of images created by the pixels) do not actually refer to the pixels themselves. Instead, the coordinates refer to *points* that are located between pixels.

This concept tends to sound a bit theoretical, much like a few geometry concepts. In geometry class, we learned that a point is a location in space that has no dimension. Similarly, a line between two points, as defined in basic geometric theorems, must be a straight line. This line, like the points it joins, has no thickness (although it has a definite length). That is, we can't really see the line or its end points, but we can visualize what it would look like by drawing points on paper at the specified distance and connecting them with a line drawn along the side of a ruler.

On the QuickDraw drawing space, then, we can demonstrate points as existing between pixels as in Figure 8-4.

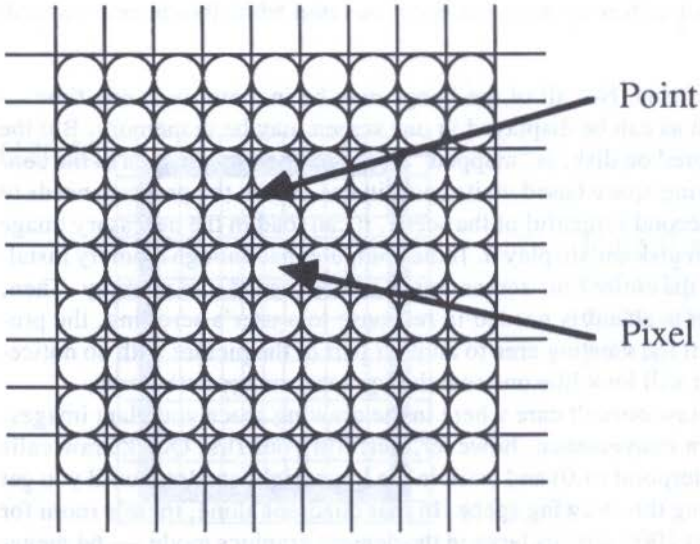


Figure 8-4. Relationship between point and pixel.

Points are represented by intersections of row and column lines. Now, if we wished to specify the coordinates of a rectangle, we would tell QuickDraw that the rectangle's two definition points (upper left and bottom right corners) are located at two points. QuickDraw would then consider the rectangle to be an imaginary rectangle with sides of no thickness. In other words, just specifying the coordinates for a rectangle simply assigns locations in space for the dimensions of a rectangle.

For the sake of consistency, QuickDraw considers the rectangular outline defined by the coordinates to be the outermost extension of that rectangle. If we want to see the rectangle, we must tell QuickDraw to actually draw the rectangle with its *pen*. A pen, as you'll see later on, can have many different attributes. One of those attributes is its thickness (called the *pen size*). If the thickness of the pen is assigned to be 1 pixel high and 1 pixel wide — 1 pixel period — QuickDraw would draw the defined rectangle using a row of pixels *inside* the theoretical rectangle defined by the coordinate points.

A pen size of 2 pixels high and 2 wide would fill in 2 rows of pixels inside the theoretical rectangle.

Simple Data Structures

To facilitate the definition of points and rectangles in your programs, QuickDraw readily acknowledges data structures called *Point* and *Rect*.

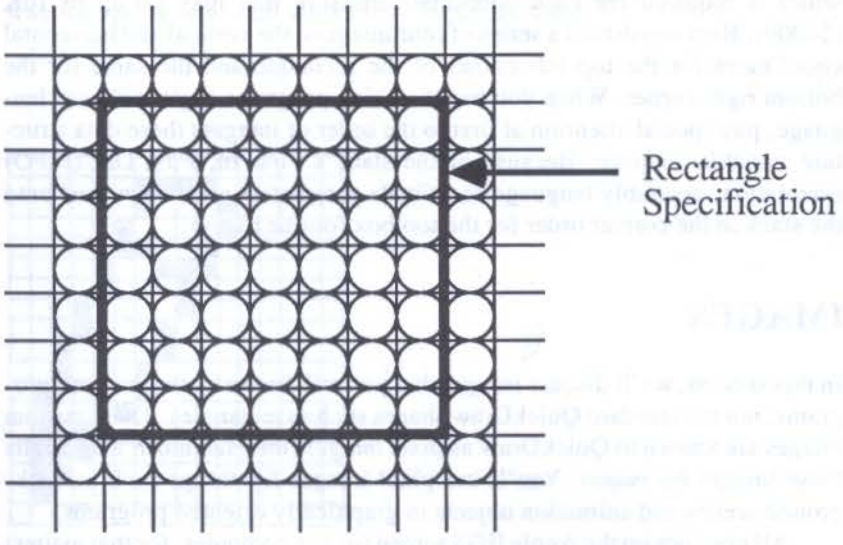


Figure 8-5. A rectangle's coordinates merely indicate its location and size.

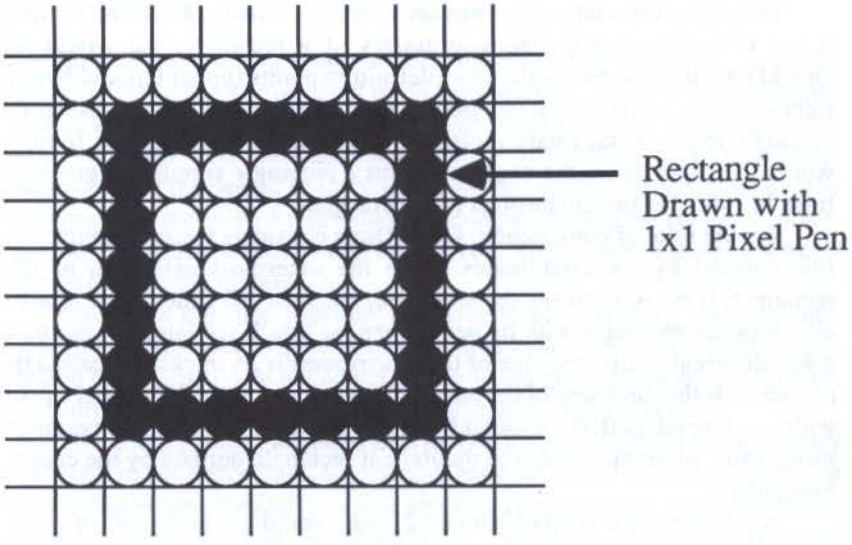


Figure 8-6. Drawing occurs inside the rectangle specification.

The Point data structure consists of two integers, one each for the vertical and horizontal coordinates in the drawing space. An integer (2 bytes wide) is required for each coordinate measure that may go up to 16K ($4000). Rect consists of a series of four integers: the vertical and horizontal coordinates for the top left corner of the rectangle and the same for the bottom right corner. When you begin writing programs in your chosen language, pay special attention at first to the order of integers these data structure variables require. Because of the stack's First In, First Out (FIFO) orientation, assembly language coordinate parameters must be pushed onto the stack in the correct order for the toolbox routine to pop.

PIXEL IMAGES

In this section, we'll discuss images that you will be designing for your programs, not the standard QuickDraw shapes such as rectangles. These custom images are known to QuickDraw as *pixel images* (the Macintosh world calls these images *bit maps*). You'll use pixel images for things such as background scenes and animation objects in graphically oriented programs.

All graphics on the Apple IIGS screen (or any computer, for that matter) consist of carefully designed patterns of pixels. In the monochromatic

Macintosh environment, pixels are either "on" or "off" — either black or white. When pixels are small enough, the pattern of on and off pixels can become a recognizable image.

Pixels, then, are like a mosaic of uniformly sized pieces that blend together to form your pictures. When designing a picture, however, you must work on these pictures under a microscrope, because you must program each pixel. You may have the benefit of a graphics program that provides painting and drawing tools such as those in MacPaint or MousePaint. Such a program may allow you to save pictures in such a manner that your own programs will be able to load and use them without pixel-by-pixel programming. Still, it's unlikely you'll escape some image programming in a graphics program.

Pictures as Numbers

Before you can design a picture and store it in memory (and on disk) for your program to use, you need to know in what form QuickDraw II expects to find your picture. We're dealing with a computer that, at its basic level, knows how to work only with numbers, not pictures or words. Therefore, a picture as we know it must be turned into numbers for the machine to toss around memory and display on the screen as colored dots.

As a stepping stone to this understanding, we'll look at how the monochromatic Macintosh performs this picture-to-number-to-picture conversion. The concepts are a bit easier to grasp in monochrome, and will pave the way for seeing how it all works in color a little later.

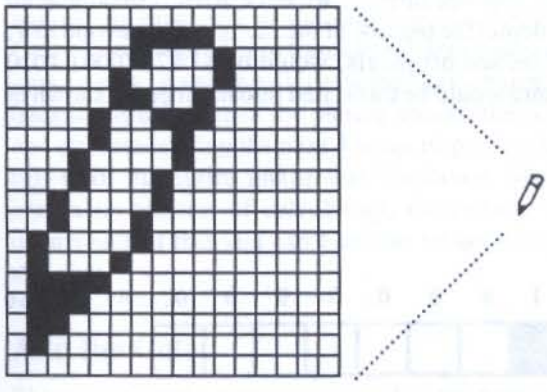


Figure 8-7. A Macintosh image enlarged to see each pixel and in its normal size.

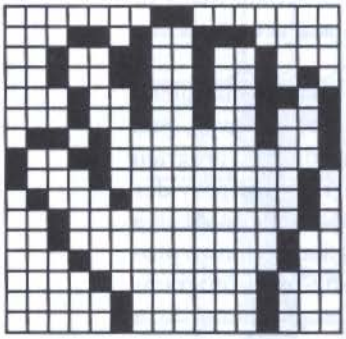


Figure 8-8. Enlarged pixel image of a hand.

Look at the pixel image of the hand in Figure 8-8.

The image is contained in a pixel area consisting of 16 rows of 16 pixels each. Now study the topmost row closely. You'll see that it contains pixels in the pattern shown in Figure 8-9.

The row of boxes may remind you of the binary boxes of earlier chapters. In this case, however, empty (white) boxes are equivalent to 0s, while filled (black) boxes are equivalent to 1s. Taking eight boxes, that is, pixels, as a group, the content of those boxes can be represented by a number whose binary equivalent has 0s standing in for white pixels, 1s for black pixels.

In the first row of the hand picture, then, the leftmost group of 8 pixels can be communicated to the computer by the binary number 0000 0001, which is $01. The rightmost group can be portrayed by binary 1000 0000, or $80. Combining the 2 bytes into one integer, we have $0180 containing all the information we need to depict the top row of the picture. The second row, a more complex pattern of on and off pixels, would be $1A70 (0001 1010 0111 0000). The entire picture would be translated into numbers as shown in Figure 8-10.

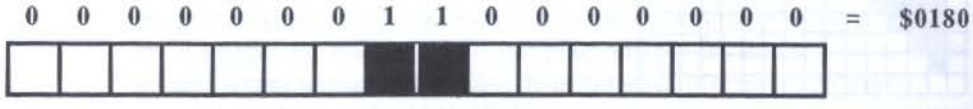


Figure 8-9. The top row of pixels from the hand image.

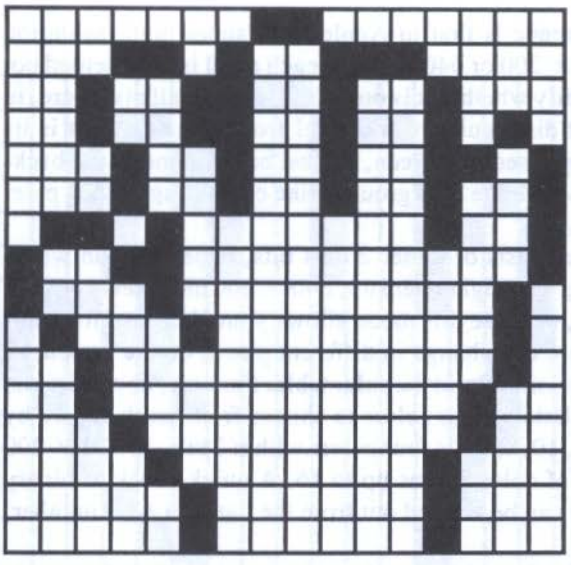


Figure 8-10. The hand image and its numeric equivalents.

This pixel image would be stored in memory as a long sequence of the numbers shown in the illustration. We must also pass another important parameter as part of this image's definition: the width of the pixel image. Width is specified as an integer and denotes the number of bytes wide any row (sometimes called a *slice*) of the image will be — 4 bytes in the above monochromatic illustration. With this figure at hand, QuickDraw II knows how many of the hex values from the list of pixel image numbers are to be applied to each row of the drawing. To display the hand on the screen, QuickDraw would begin translating pixel image values into on and off pixels along a row. As soon as it drew the first 4 bytes' worth of data, it would zip back to the left edge of the picture area on the screen one line below the first and start translating the next 4 bytes of pixel values. It would continue with this until the entire image was displayed. While this may sound like an enormous amount of calculating, converting, and gnashing going on, it occurs so fast that you can't see the image being drawn — at least not one this size.

Add the Color

Things get a little more complex when the pixel image is to carry color information with it, but the concept is the same as for the Macintosh monochrome

bitmap. The main difference is that in Apple IIGS super high-resolution color modes (either 320 × 200 or 640 × 200), each pixel is represented not by just a single bit to signify whether it is on or off. Instead, all pixels are, in a sense, "on." But what distinguishes one pixel from one next to it is its color. If a pixel is not supposed to be seen, it must be the same as the background color so it blends with the background, like camouflage. Each pixel has a color.

Color information consists of either 2 or 4 bits, depending on which color mode you are using. The higher density, 640 × 200 mode lets you pick a color using only 2 bits, because this mode allows your choice of four colors. What those colors are can change in different places on the screen, as we'll see later in our discussion about the color table. For now, however, just think of there being a palette of four colors to choose from, each known by its binary number 00, 01, 10, and 11: four colors within 2 bits. In 320 × 200 mode, however, choice of color zooms up to 16. A quick check of binary math tells us that a color can be singled out from the list by a 4-bit number, from 0000 to 1111.

Going back to the hand pixel image, but this time in 320 × 200 mode Apple IIGS color, if the background color happens to be color 0100 and the outline of the hand is to be color 0000, the first 8 pixels of the top row of the image would be

0100 0100 0100 0100 0100 0100 0100 0000

or $44, $44, $44, $40, while the first 8 pixels of the second row from the top would be

0100 0100 0100 0000 0000 0100 0000 0100

or $44, $40, $04, $04. The hexadecimal values for the entire hand pixel image are shown in Figure 8-11.

You can see that a color image requires more memory to store and display a pixel image than a monochrome-based picture. By placing the color information with each pixel, the programmer ends up with excellent control over the application of a diverse color spectrum on each image.

Color Image Width

A QuickDraw II pixel image must also be stored with its width factor so that QuickDraw II knows when to start applying numbers from its long data list to pixels in the next row. But an important factor to remember is that *the width must be a multiple of 8 bytes*. That means that in a 320 × 200 mode pixel image, whose pixels are 1 nibble in length, the minimum width of an

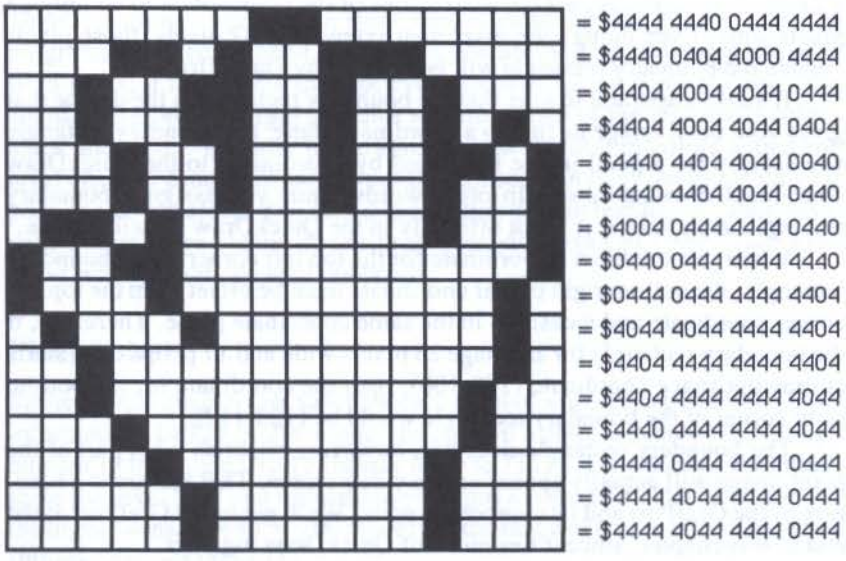


Figure 8-11. Pixel image data in 320 × 200 mode.

image is 16 pixels. It also means that all pixel images must be stored with widths in multiples of 16 pixels.

Fortunately, this does not mean that all pixel images *as displayed on the screen* must be an even multiple of 16 pixels. If that were true, and you had an active image that was, say, 23 pixels wide, the remaining 9 pixels to the right of the image would have to be assigned some color. But then, if the image moved to another part of the screen where images other than the background color were being displayed, the background color of the rightmost 9 pixels of the image would cover the existing image on the screen.

The BoundsRect

What saves us from that potential disaster is yet another pixel image parameter, called *boundsRect*, which is attached to information stored about each pixel image. BoundsRect is short for *boundary rectangle*, and it specifies the rectangular part of the stored image that is to be available for display on the screen. The boundsRect does not necessarily determine where or even whether the image will actually appear on the screen or whether a QuickDraw tool can alter that image, but the boundary rectangle does mask

out extraneous pixels to the right of the desired width. Therefore, if you have a 23-pixel-wide image, set the dimensions of the boundsRect to be only 23 pixels wide. Even though the pixel map extends for 32 pixels, those pixels outside the boundary rectangle will be ignored by QuickDraw.

It's also important to note that the boundary rectangle is the device that gives your pixel image its link to a coordinate plane: the boundary rectangle you specify for a pixel image is defined by coordinates to the QuickDraw conceptual drawing space. In other words, until you assign a boundary rectangle to an image, it is not officially in the QuickDraw drawing space.

When you establish a coordinate for the top left corner of the boundary rectangle, the bottom right corner coordinate must be offset from the top left corner coordinate and measured in the same coordinate plane. Therefore, if the boundary rectangle for an image 23 pixels wide and 17 pixels deep starts at drawing space coordinate (100,100), then the coordinate for the bottom right corner of the boundary rectangle would be (123,117).

The boundary rectangle does not, however, establish what part of the pixel image will actually appear on the video screen. That feature is a function of the GrafPort and its component parts. We'll get to the GrafPort at the end of this chapter, since it ties much of QuickDraw together.

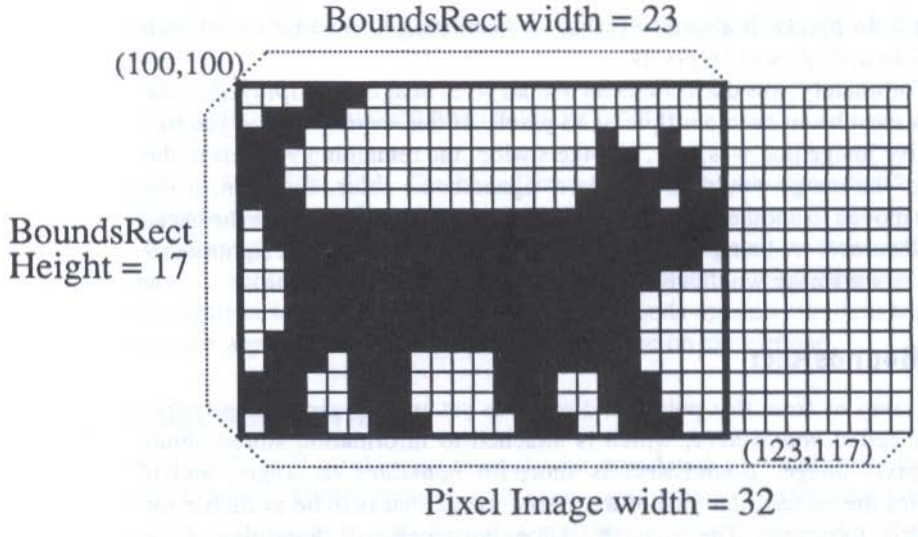


Figure 8-12. BoundsRect imposes a coordinate plane on an image and hides extraneous pixels.

COLORS

There's nothing engraved on stone tablets dictating that your Apple IIGS programs must be colorful. In fact, if you are writing programs for other IIGS users, it is possible that not everyone will have a color monitor. You could, therefore, program graphics screens in two of the machine's colors: black and white. But why not reward those with color monitors with a colorful display? Since you have to program in color anyway, be imaginative in your choice of colors. You may discover that certain features or functions in your program are clearer to the user when displayed in color. Your extra effort will probably be well rewarded.

Pixel Color

Earlier in this chapter, we saw how each pixel of a pixel image contains color information. In 320 × 200 mode, the color of a pixel is denoted by a unique number in the range 0 to 15 (binary 0000 to 1111). This number essentially turns the pixel "on" and immediately assigns a color to it. The same goes for the higher density, 640 × 200 mode, but the range of colors is limited to four (binary numbers 00 to 11).

It's important to understand, however, that the color number you assign to a pixel in a pixel image does not stand for a particular color directly. That is, color number 4 is not always blue. What that number refers to is a position in a look-up table of colors stored in memory: a *color table*. If blue happens to be the color in position 4 of the color table when the image is drawn on the screen, then it will appear as blue. By changing the colors assigned to a color table, we can control the precise shading and hue of a pixel to be any one of 4,096 different colors.

The Standard Color Table (320 Mode)

If you make no changes to the contents of the color table when working in QuickDraw, you will have at your disposal the *standard color table*, which has two lists of entries, one for each resolution mode, 320 or 640. The 320 mode standard color table looks like this:

Pixel Value	*Color Name*	*Color Value*
0	Black	000
1	Dark Gray	777
2	Brown	841
3	Purple	72C
4	Blue	00F
5	Dark Green	080

Pixel Value	*Color Name*	*Color Value*
6	Orange	F70
7	Red	D00
8	Flesh	FA9
9	Yellow	FF0
10	Green	0E0
11	Light Blue	4DF
12	Lilac	DAF
13	Periwinkle Blue	78F
14	Light Gray	CCC
15	White	FFF

The Pixel Value column is the number you assign to a pixel in a pixel image. In 320 mode, for instance, a pixel value of 4 (0100) is blue in the standard color table, just as we showed in Figure 8-11.

The Color Value column should be quite revealing if you look at the pattern of hex values assigned to the basic colors red, green, and blue, and to both black and white. First of all, you should immediately recognize that it takes 3 nibbles of information to convey a particular color in the table. Values in the range $000 to $FFF can represent any one of 4,096 numbers — the number of colors available in the IIGS palette. Now, looking at the color value for blue (4 in the table), you'll see that its value is $00F, meaning the rightmost nibble is topped out at $F, while the other two are 0. The green selected for the standard table has a high value, $E, in the middle digit, while the others are 0. And red has a high value in the leftmost digit, while the other two are 0. You may discern a pattern: each of the 3 nibbles controls the amount of red, green, and blue color in a particular shade. Black, which is the total absence of color, comes in as $000; white, which is a combination of all three basic colors has a value of $FFF. The other colors consist of proportions of two or more colors, like mixing paint.

The Standard Color Table (640 Mode)

You might expect that because a 640 mode pixel value covers a range of only 4 (00 to 11), the 640 color table would be only 4 entries long. Fortunately, that's not the case. If it were, we would have only two colors other than black and white to display at one time (black and white must usually be present for drawing crisp text, lines, and other standard user interface images).

While a pixel in 640 mode can be programmed in only one of four possible colors, the human eye and the color video screen can play tricks with our color perception. The result, as Apple engineers discovered, is that you can achieve additional colors by placing two differently-colored pixels next to each other. To the eye, the pixels blend to form the color average of the

two pixel colors. For example, an area of alternating red and yellow pixels looks like orange. This perceived mixing of colors on the screen is called *dithering*.

To facilitate dithering, QuickDraw has established a 640 mode standard color table that has four *mini-palettes*, each containing the four values any pixel may choose.

Pixel Value	*Name*	*Master Color*
0	Black	000
1	Red	F00
2	Green	0F0
3	White	FFF
4	Black	000
5	Blue	00F
6	Yellow	FF0
7	White	FFF
8	Black	000
9	Red	F00
A	Green	0F0
B	White	FFF
C	Black	000
D	Blue	00F
E	Yellow	FF0
F	White	FFF

QuickDraw assigns four horizontally adjacent pixels to the full color table, such that each pixel in the group has its own mini-palette. The order in which pixels are assigned to mini-palettes is not particularly intuitive; pixel 0 uses mini-palette 2; pixel 1 uses mini-palette 3; pixel 2 uses mini-palette 0; pixel 3 uses mini-palette 2. Therefore, if you want four pixels in a row to produce what to our eye looks like orange, you would assign the pixel values as 01 11 01 11, which QuickDraw would interpret as the 9th, 14th, 1st, and 6th entries, respectively, on the 640 color table.

Custom Color Tables

You can also make your own color tables. To store a color in a new color table, you must specify the color value as a word-length (2-byte) number. An extra nibble must be added to round out the 3-nibble color values. Values

for each nibble must also be laid out in the proper order for QuickDraw to assign the values to the correct basic colors. A color word is shown in Figure 8-13.

A color table, then, consists of a list of sixteen color values. By simple arithmetic, an entire color table takes up only 32 bytes of memory. You can adjust the contents of a color table for all sixteen color values at once (SetColorTable call) or a single entry in the table (SetColorEntry). Input parameters for the latter's call include the number of the table, the number of the entry (corresponding to the left column in the tables above), and the word-length color value to go in that entry.

Multiple Color Tables

QuickDraw is ready to accommodate rather creative colorists, because you can set up as many as sixteen color tables in memory at one time. You assign a number (from $00 to $0F) to each table when you create it and load it with your color values.

Having multiple tables resident allows you to perform sophisticated color graphics effects with a minimum of memory and code manipulation. Let's say your program features an animated, flying eagle and, in the lower left corner of the screen, a glowing hot ember. Animation consists of twelve different pixel images depicting the eagle in its various wing-flapping stages of flight. What you want to display is that as the eagle nears the ember, the eagle's colors change from a predominantly golden brown to a hot orange.

The twelve pixel images of the eagle have color values assigned to each of their pixels. But instead of changing the pixel color values of the images as the bird nears the hot ember, simply change the color table that the images use. The pixel values remain the same in all pixel images, but the values refer to different colors on the second color table. In the first table, a pixel value of 2 may indicate a golden brown; in the second table, a pixel value of 2 is a reddish orange. As the eagle retreats from the burning ember, the program reverts to the starting color table, returning the eagle to its original brown color.

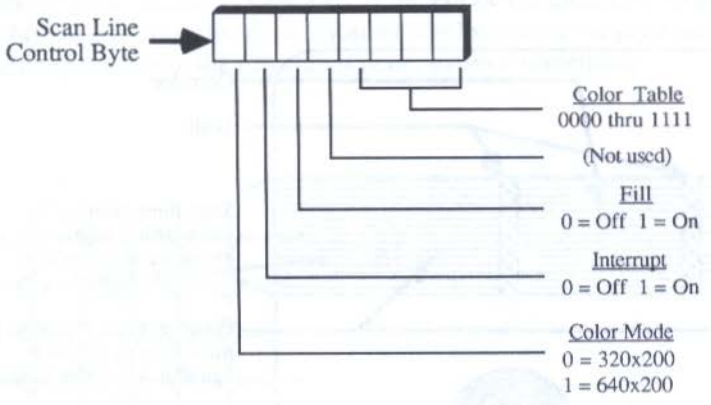
Figure 8-13. A color word.

Scan-Line Control Bytes

You can extend the color capabilities of the IIGS even further by assigning different color tables to different parts of the screen. You can, if you desire, assign a different color table (i.e., one of the sixteen possible tables in memory) to a different *scan line*. A scan line is the same as a row of pixels on the screen. In other words, you can have one color table apply to scan line 1 on the screen, another table apply to scan line 2, and so on for up to sixteen scan lines. It's highly unlikely you'll need to do this for your applications, but you should be aware of the principle behind it.

What determines the color table being used by any scan line is a byte of information called the *Scan Line Control Byte*, or *SCB*. The SCB contains several flags as well as a 4-bit group that signifies which of the sixteen color tables to use. The byte is illustrated in Figure 8-14.

Generally, the entire screen (or window) will use the same color table and, therefore, be governed by the same SCB. Unless you're doing sophisticated graphics or animation, you will probably use only the standard color table, which is referred to by the default SCB color table of zero. But if you need a region of the screen to have a different color table, you can assign SCBs to specific scan lines with the SetSCB QuickDraw call. As parameters you pass the number of the scan line(s) to be changed (any of 200 in either



SCB 10100011 ($A3) = Color mode 640x200; Interrupt off; Fill on; Color Table 3

SCB 00101110 ($2E) = Color mode 320x200; Interrupt off; Fill on; Color Table 14

Figure 8-14. A scan line control byte.

320 or 640 mode) and the actual SCB that is to apply to that scan line. If you need twenty contiguous lines on the screen that need a change in color tables, then you'd set up a finite loop that changes the SCB for those scan lines.

Let's look at an example that should go a long way to explain how color tables and SCBs work. The example will be an animation sequence in which an onscreen character, controlled by the mouse, changes color whenever it passes through a raybeam corridor.

Let's also assume that the 320-mode standard color table is in effect, except when the character is in the corridor. The screen's background color is blue and the character's regular colors are red and yellow (standard color table pixel values 7 and 9, respectively). In other words, background pixels are set to 4 and the character's pixels are in a pattern of 7s and 9s that give the character its facial features. What we want to do is change his colors to a purple and brown when he is in the corridor space on the screen.

The program must create a second color table that assigns new color values to pixel values 7 and 9. Then the program will track the location of the character so that when it is in the region of the corridor, the scan lines encompassing that region change their color table to the second table. When this happens, entire horizontal rows of pixels change to the new color table (see Figure 8-15). Colors you don't want to change, such as the background color to the left and right of the corridor, must have the same color values in both

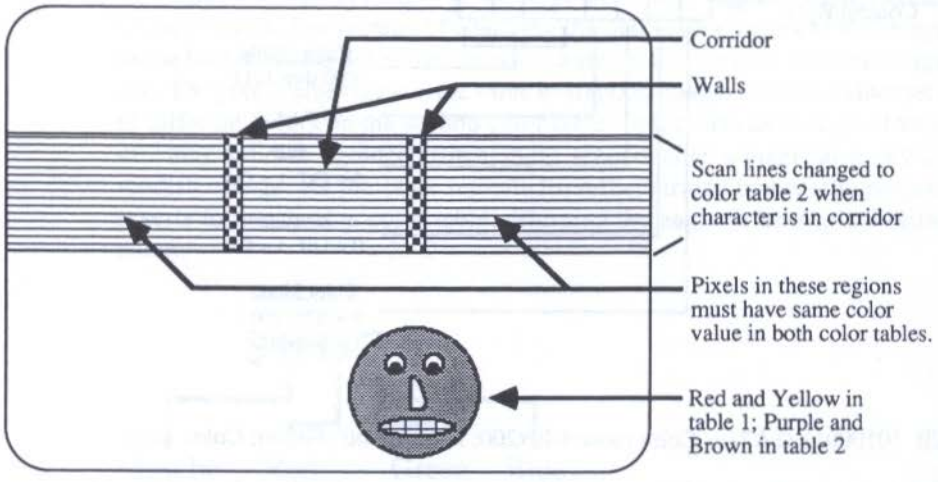


Figure 8-15. When the character enters the corridor, SCBs for the middle lines change color table.

tables (pixel value 4 would be blue in both tables). Only those pixel values with different colors in them (7 and 9 in this example) will change. As soon as the program detects that the image is outside the corridor, it switches the affected scan lines back to the first color table so that the character can travel in its original colors anywhere outside the corridor region.

THE PEN

All drawing by QuickDraw, including the drawing of text on the screen, is done with an imaginary *pen*. A pen has several controllable properties, which, collectively, are called the PenState. The properties and their QuickDraw names are:

Property	*QuickDraw Name*
pen location	PnLoc
pen size	PnSize
pen mode	PnMode
pen pattern	PnPat
pen mask	PnMask

The PnLoc is a coordinate within the boundary rectangle — the rectangle that assigns a coordinate system to a pixel image. Like the imaginary geometric point, the point indicated by the PnLoc component does not mean that the pen is necessarily visible. Rather, the location is simply a point in the drawing space where the pen stands ready to draw something.

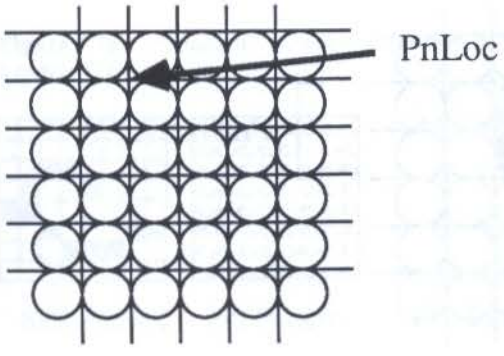


Figure 8-16. Pen location.

PnSize is defined by a point-like data structure — with a horizontal and vertical component. The components of the data structure are the number of pixels in each of those directions that the pen will be changing whenever it draws. A pen size can be square, like the most common PnSize, (1,1), or it can be rectangular, such as 3 pixels wide and 2 high. The point of reference of measurement of the height and width sizes is PnLoc. Width is measured to the right of PnLoc; Height is measured down from PnLoc.

QuickDraw recognizes eight different pen modes, each with a unique integer number. Their names and numbers are as follows:

Pen Mode	*Integer*
COPY	$0000
notCOPY	$8000
OR	$0001
notOR	$8001
XOR	$0002
notXOR	$8002
BIC	$0003
notBIC	$8003

These pen modes affect how a pen's pixels and its pixel pattern (described next) affect pixels already in an image when the pen writes over them. The most common mode is the default mode, COPY, in which every pixel of a pen completely overwrites anything in the pixel image. Other modes, how-

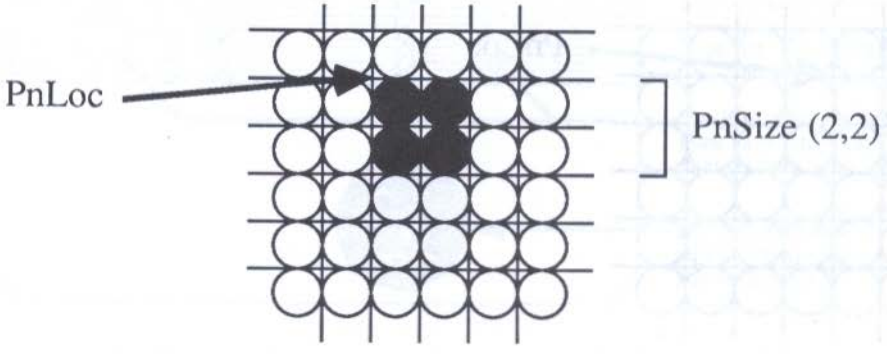


Figure 8-17. Pen Size of (2,2).

ever, offer different combinations of effects, depending on the binary number representing a particular pixel (4-bit numbers for 320 mode, 2-bit numbers for 640 mode). XOR (exclusive OR), for example, changes the bits of existing pixel values to their opposites if all bits in the pen are 1. BIC (Bit Clear), on the other hand, changes existing 1 pixel image values to 0s when overwritten by a pen pixel value consisting of all 1s. Pen modes (also called Transfer Modes) are worth understanding because they can be helpful in creating pleasing graphics effects. Discussion of their precise possibilities, however, is better left to more advanced programming guides.

Pen Patterns

A pen can have not only size and location, but a pattern consisting of colored pixels. It takes a square of 64 pixels (8 by 8) to establish a pattern. Once a pattern is defined, it repeats throughout the drawing space wherever the pen draws. Consequently, if you wish to design an interesting fill pattern to cover a large screen area, you must design it via the 8-by-8-pixel pattern.

Typically, you will assign intricate patterns to a pen only when it is to fill a relatively large screen area. Since it often takes several contiguous rep-

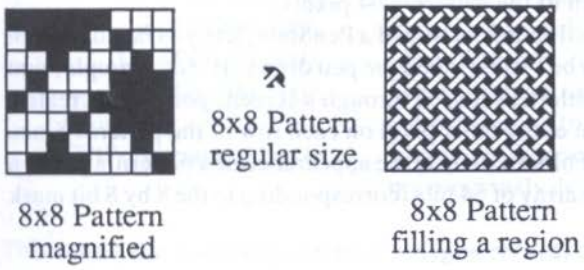


Figure 8-18. A pattern's pixel representation, the same pattern in real size, and as a repeated fill pattern.

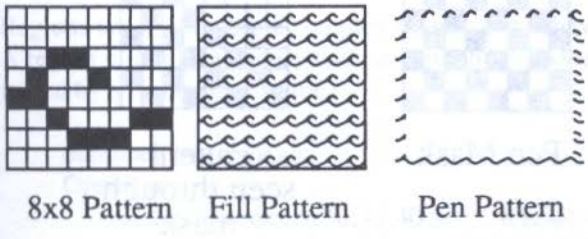


Figure 8-19. Wave pattern works well as a fill pattern, but not as a pattern for a small pen size.

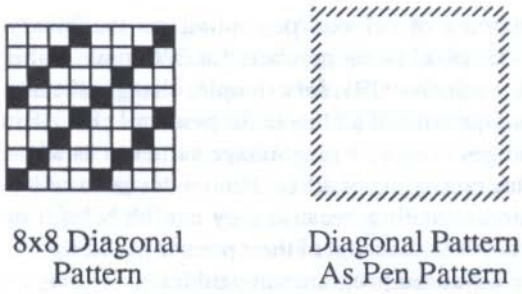


Figure 8-20. Some patterns work well in small pen sizes.

etitions of a pattern for it to be recognizable, you are not likely to assign a pattern to a pen that will simply draw a 1- or 2-pixel-wide outline to a box.

Still, with careful pattern design, it is possible to use a pattern and a small size pen to good effect. A pattern of diagonal lines, for example, will give a box outline the appearance of a dotted line.

PnPats are defined in memory as lists of 64 nibbles (320 mode) or 64 two-bit numbers (640 mode), each nibble or two-bit number signifying a color table entry for each of the pattern's 64 pixels.

A pen mask, the final characteristic of a PenState, lets you decide which pixels of a pattern are to be visible when the pen draws. If, for example, you wish a pen to draw its pattern as if seen through a screen, you might create a pen mask that blocks out every other pixel on each row of the pattern. Since a mask simply admits or blocks (0 or 1) the appearance of a pattern's pixel, a PnMask is defined by an array of 64 bits (corresponding to the 8 by 8 bit mask pattern).

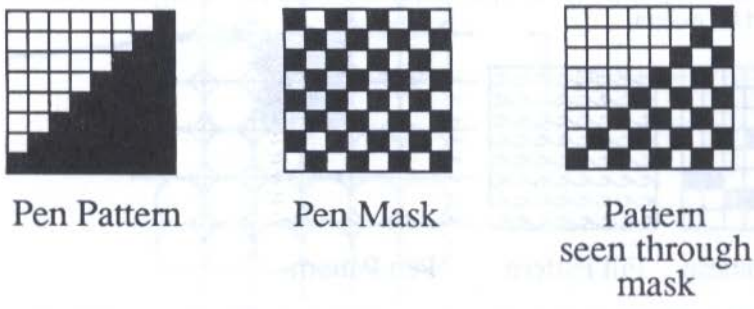


Figure 8-21. The pen mask blocks pixels from the underlying pattern.

IRREGULAR SHAPES

We saw earlier that a pure rectangle can be defined with very little data using QuickDraw's built-in rectangle routines. This compares favorably with a pixel image of a rectangle, which must have each pixel of the rectangle specified by 2 or 4 bits, depending on graphics mode. Once the dimensions of a rectangle are specified, the rectangle's outline can be drawn with the FrameRect call, and its interior space can be filled with a pattern by way of the FillRect call. Only a handful of bytes are affected, no matter how enormous the rectangle. We can define more complex, irregular shapes with similar frugality using QuickDraw's Polygon and Region families of tools.

Polygons

Polygons (shapes with many straight-edge sides) are defined by coordinates of the shape's outline. In a high-level language coding of a triangle, for instance, you would start the coordinate definition procedure with the OpenPoly call, then move the pen's coordinates from point to point, and finally issue the ClosePoly call.

```
triangle = OpenPoly;
    MoveTo(100,100);        {move pen to starting point}
    LineTo(200,100);        {draw line from 100,100 to 200,100}
    LineTo(150,200);        {draw line from 200,100 to 150,200}
    LineTo(100,100);        {draw line back to starting point}
ClosePoly;                  {stop gathering triangle data}
FillPoly(triangle,bricks);  {draw "triangle" on the screen with
                             my previously defined brick pattern}
```

This series automatically creates a polygon record, complete with information such as the dimensions of the rectangle that contains the polygon.

Regions

Regions are slightly more complex shapes, but behave much like polygons. A region might be a shape, such as a dumbbell, which has two oval shapes joined by a rectangular shape.

```
dumbbell = NewRgn;
OpenRgn;
    SetRect(temprect,30,30,40,60);  {create a temporary rectangle
                                     called temprect at these
                                     coordinates — left weight}
```

```
FrameOval(temprect);                {draw an oval within
                                     temprect's boundaries
                                     (Figure 8-22)}
SetRect(temprect,90,30,100,60);     {create a second temporary
                                     rectangle, use the name
                                     temprect again — right
                                     weight}
FrameOval(temprect);                {draw this rectangle as an
                                     oval}
SetRect(temprect,40,40,90,50);      {create the bar's rectangle
                                     coordinates (Figure 8-23)}
FrameRect(temprect);                {draw the bar (Figure 8-24)}
CloseRgn(dumbbell);                 {that's all the data; save it
                                     in a region record called
                                     dumbbell}
FillRgn(dumbbell,black);            {fill the area defined in the
                                     dumbbell record with the all-
                                     black pattern}
```

All in all, this region is far less memory-hungry than its pixel image equivalent. To change its color, we could refill the region with a pattern defined as the desired color.

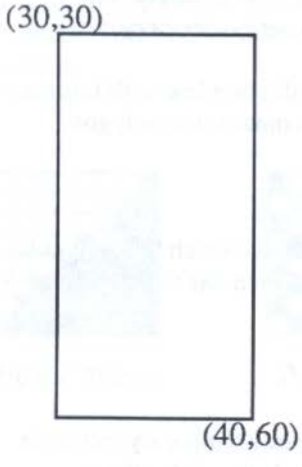


Figure 8-22. FrameOval draws an oval within the rectangle specification.

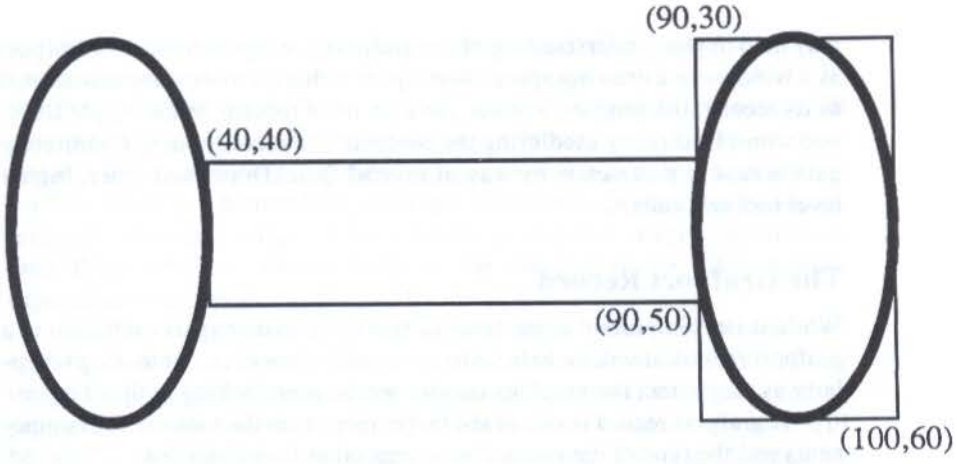


Figure 8-23. The crossbar rectangle's specification is added.

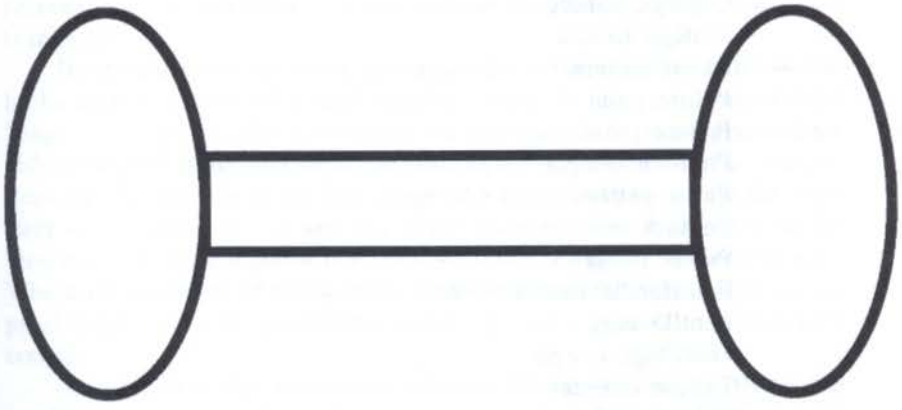


Figure 8-24. FrameRect draws the crossbar.

THE GRAFPORT

Now that we've gone through the major graphics entities of QuickDraw, it's time to pull everything together and see how these elements work inside a toolbox IIGS program. The unifying element is called a *grafport*.

A grafport is a self-contained drawing environment that exerts control over graphics and text drawn in its drawing space. As you'll see in Chapter 10, a screen window is built upon a foundation created by a grafport, so it

may help in your understanding of the grafport concept to think of a grafport as a window to a drawing space. A grafport's characteristics are maintained in its record, the *grafport record*. As with most records in the Apple IIGS, you won't be directly modifying the contents of a record, but will indirectly gain access to parameters by way of myriad QuickDraw (and other, higher level tool set) calls.

The Grafport Record

While it isn't important at the level of this book to tear apart each item in a grafport record, it will be helpful to get an overview of its contents, particularly as they affect the graphics entities we've been looking at thus far.

A grafport record is one of the larger records in the toolbox. Its components and the type of data each component takes looks like this:

```
PortInfo: LocInfo
PortRect: rect
ClipRgn: handle
VisRgn: handle
BkPat: pattern
PnLoc: point
PnSize: point
PnMode: integer
PnPat: pattern
PnMask: mask
PnVis: integer
FontHandle: handle
FontID: long
FontFlags: integer
TxSize: integer
TxFace: style
TxMode: integer
SpExtra: fixed
ChExtra: fixed
FGColor: integer
BGColor: integer
PicSave: handle
RgnSave: handle
PolySave: handle
GrafProcs: pointer
ArcRot: integer
UserField: long
SysField: long
```

PortInfo consists of a compact data structure (only 10 bytes) that establishes a great deal about the grafport's environment. One byte is the SCB, which sets the port's color table, its color mode (320 or 640), and other SCB parameters. A pointer in PortInfo links the grafport to a pixel image (if there is one), while an integer specifies the image's width in bytes (as described earlier). Finally, a boundsRect rectangle in PortInfo specifies the boundary rectangle of a pixel image, if there is one. If no pixel image is pointed to, then BoundsRect is automatically set to the size of the IIGS screen, depending on the mode established by the SCB.

A grafport's portRect, the second item in its record, is an important specification. We said earlier that assigning a boundary rectangle to a pixel image simply imposed a coordinate system on the image, and did nothing to promote its display on the screen. That's because an image of any kind will be active — visible on the screen and subject to drawing changes by QuickDraw calls — only if the image or any part of it falls within the edges of the portRect. If you've been following the logic so far, you'll recognize that for an image to be active it must be within the boundsRect and portRect rectangles — within an intersection of these two rectangles. An illustration is in order.

Imagine that you are using a program that creates a screen window that looks onto a portion of a large graphics area. As you scroll around the "page," you can see only a portion of the entire page. You can type or draw only in the part of the document that you can see in the window. The boundsRect may be the size of the full image or a section of image, but the portRect — the one you can see and make changes in — is the size of the viewing area of the window (i.e., not including the scroll bars or title bar). Where the portRect of the window intersects with the boundsRect of the pixel image, you can view the document, type text, or draw to your heart's content.

As a result of this interaction between boundsRect and portRect, the portRect is a separate item in the grafport record. As you change the size of a window, the size of the portRect changes with it, while the boundsRect will probably never change as long as the image doesn't grow or shrink from its original proportions.

ClipRgn, short for *clipping region*, won't be adjusted in every grafport, but can be helpful in many graphics-oriented situations. A clipping region lets you restrict the area inside a portRect in which the user (and, hence, QuickDraw) can change a pixel image. It is actually a third layer to the boundsRect-portRect intersection mechanism. But with clipRgn, the third layer can be a region of nonrectangular proportions. Imagine creating a view through a hypergalactic telescope in which the surrounding area is a never-changing celestial background. But inside the circular telescope view you have an active view of two Martians playing Crazy Eights at the edge of a

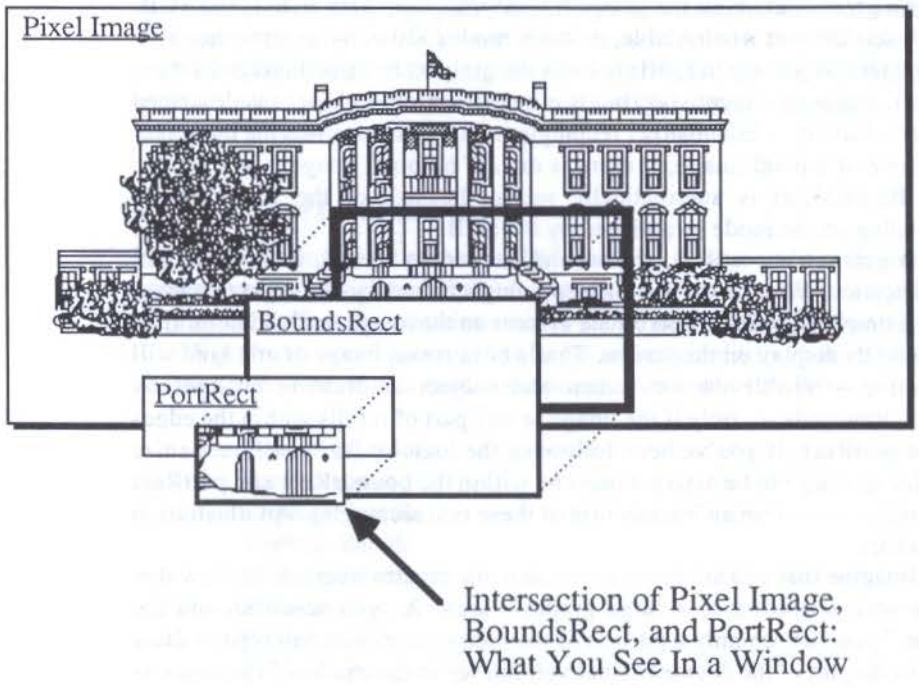


Figure 8-25. Relationships among pixel image, boundsRect, and portRect.

crater. To program one Martian getting up in a huff and walking out of view of the telescope, you could animate his movements so that his image inches its way out of the clipping region. You won't have to draw special animation versions of a partial Martian as he walks off the edge of view. Instead, any part of the Martian image that steps out of the clipRgn will not be displayed on the screen, even though it is within the limits of the portRect.

If you don't specify a clipRgn when creating a new grafport, it automatically sets itself to coordinates of the full size of the IIGS screen. In the case of the telescope example, you might start the program with the clipRgn being set to its default size (i.e., identical to portRect). A menu item, "Telescope," will temporarily store the original clipRgn and reset the region to the circle for the close-up graphics. At the end of the telescope session, another menu choice restores the clipRgn to its original, full window size, and the original background scene is redrawn to cover the image displayed in the telescope's hole.

Although the visRgn item in the grafport record looks like something that specifies a "visible region," this region is generally under the control of the Window Manager. The visRgn corresponds to that region of the grafport that you can see without obstruction. If a second window overlaps the first window, the visRgn of the first window is that area not covered by the second window. The grafport record keeps track of this information so that if your program attempts to draw in the first, underlying window, it won't accidentally extend its drawing image into the region occupied by the overlapping window — it will draw only in the visRgn. As soon as the first window comes fully into view (as when it is made the active, top window), the visRgn assumes the same coordinates as the portRect.

Apple IIGS grafports can be automatically filled with a particular pattern — perhaps a gray shaded pattern like the one on many Macintosh screens — by assigning a pattern design to the bkPat item in the grafport record. This item can be adjusted in the course of a program with the SetBackPat QuickDraw call. Another call, SetSolidBackPat, adjusts the background pattern to a solid pattern with a color you specify from the grafport's color table.

You should recognize the next five items down the grafport record: PnLoc, PnSize, PnMode, PnPat, and PnMask. These are items associated with the Pen State, as we discussed earlier. PnVis stores the current visibility status of the pen. This factor is adjusted by HidePen and ShowPen calls.

The next eight items track the current state of fonts and text parameters currently on hand in the grafport. That is, these items establish and record fonts and text display characteristics for the next display of a keyboard press (or text coming in from a serial port).

FGColor and BGColor indicate the foreground and background colors respectively.

The balance of items in the grafport record are primarily repositories for information about the status of various operations during the running of a program. You probably won't get involved with them, at least not in your first programs, unless you need to take a "snapshot" of the information before performing some task that will alter these items in the record. By temporarily saving the values, you can restore them later to reconstruct the record.

MULTIPLE GRAFPORTS

One of the beauties of working with grafports is that they make dealing with multiple windows almost child's play. That's because the data in a grafport is private data that belongs only to that grafport. When you open a second window and grafport on the screen, all the information contained in the

first grafport's record is untouched. Therefore, if the pen in the first grafport is at location 300, – 72 and set to a diagonal pattern with a pen size of 3-by-3, you can activate the second grafport and make any adjustments you want to its pen. When you reactivate the first grafport, its Pen State (and everything else in the record) is exactly the way you left it. There's no guessing or reconstruction of "the way things were." They never changed.

QuickDraw has a built-in mechanism that prevents you from getting into trouble by accidentally adjusting the record of one grafport when you really mean to adjust a second one. For most QuickDraw calls, you issue the same QuickDraw instruction, regardless of the grafport to which you intend the call to apply. What governs the recipient is *which grafport is the current port*. Therefore, before changing the Pen State of a grafport called "secondport," you would issue the SetPort (secondport) QuickDraw call (the precise syntax will differ from language to language). Thereafter, any Pen State calls apply only to the pen in secondport.

It's vital that you understand the intent of the grafport record. It exists in the Apple IIGS primarily as a log or roster of the current state of all adjustable criteria in a grafport. You can draw a frame according to a rectangle's coordinates with a pen size of 1,1. Later you can reset the pen size (SetPenSize) to 2,2, for example, and frame a different rectangle with a pen twice as thick as the first rectangle's frame. Once the pixel image of the first rectangle is drawn on the grafport, it disconnects itself from the pen. Changing the pen size will not affect any item that has already been drawn by it. A change in pen specification will, however, affect any new drawing you make — until you change the pen once more.

There may be times in your program when you wish to "remember" one or more grafport settings for later restoration. To store those settings away safely — remember, once a record item changes, it knows not what it was before — you will use the GET call for those settings (e.g., GetPenSize) and assign those values to a variable (e.g., oldSize = GetPenSize) for later recall and SETting (e.g., SetPenSize). The grafport record will be the main storehouse for grafport characteristics in your program. Use its resources often.

CURSORS

While we're in QuickDraw, we'll introduce you to manipulating the cursor's design.

Cursor manipulation in the Apple IIGS toolbox is more flexible than in the Macintosh. For example, cursors can be of virtually any size on the IIGS, whereas they are restricted to 16-by-16-pixel measure on the Mac.

You create a cursor by assembling data in the following order:

```
CursorHeight: integer
CursorWidth: integer
CursorImage: [array 1..CursorHeight,1..CursorWidth] of word
CursorMask: [array 1..CursorHeight,1..CursorWidth] of word
HotSpotY: integer
HotSpotX: integer
```

CursorHeight is the number of rows ("slices") the cursor and/or its mask image will require. CursorWidth is the number of pixels needed to define the horizontal dimension of the cursor or mask image. Importantly, the last word of each slice must be $0000, so you'll have to take this extra space into account when assigning the CursorWidth. CursorImage and CursorMask are the actual data points for each image (described below). And the HotSpot designations are the coordinates *within the cursor height/width map* that act as the "tip of the arrow," so to speak. A cursor's hotspot is the pixel on the cursor that the Event Manager will understand to be the mouse location when you press the mouse button.

Cursor Image and Mask

Design of a cursor is done on a pixel-by-pixel basis in two layers. The first layer is the *cursor image*, which represents the actual cursor design you will see when the cursor is atop any nonblack color. The second layer is called the *cursor mask*.

The map for an arrow cursor image is shown in Figure 8-26. Notice that (1) you can use the map as a way of organizing your actual data values, and (2) the last 2 bytes (1 word) are 0.

Since all cursors are black, each pixel is assigned either a zero (white) or nonzero (black) value. In the above image grid, the nonzero value is $F.

The cursor mask acts somewhat differently than a pattern mask. Its common use is to provide a kind of invisible outline to the cursor so that when it is atop a color object, you can still make out the edges of the cursor image. Yet, where the cursor image is white, the mask must allow the underlying screen image to show through.

Data for a mask is lined up the same way as the cursor image. For convenience in seeing the effect of the cursor mask in our illustration, the "F" pixel markers corresponding to the cursor image are shown in lowercase, while the markers for mask pixels above "0" cursor image pixels are in uppercase.

```
0000000000000000
0F0000000000000
0FF000000000000
0FFF00000000000
0FFFF0000000000
0FFFFF000000000
0FFFFFF00000000
0FFFFFFF0000000
0FF0FF000000000
00000FF000000000
0000000000000000
```

Cursor Image Data

Figure 8-26. Cursor image data.

Onscreen Cursor

On a white background, the cursor looks like the arrow cursor image. Where a mask pixel is filled but its corresponding image cursor pixel is white, the

```
FF00000000000000
FfF0000000000000
FffF000000000000
FfffF00000000000
FffffF0000000000
FfffffF000000000
FffffffF00000000
FfffffffF0000000
FffFffFF00000000
FFF0FffF00000000
00000FFF00000000
```

Cursor Mask Data

Figure 8-27. Cursor mask data.

resulting display at that pixel is white. Atop a color background, therefore, the cursor mask turns some of the background's color pixels into white, making the outline of the arrow stand out, even in an all-black area.

When the cursor image straddles color and white areas, the mask appears to work only on the color area, turning selected pixels to white to help the cursor image stand out in the sea of color. And where both the cursor image and mask image are 0, the underlying image's color pixels show through without any interference.

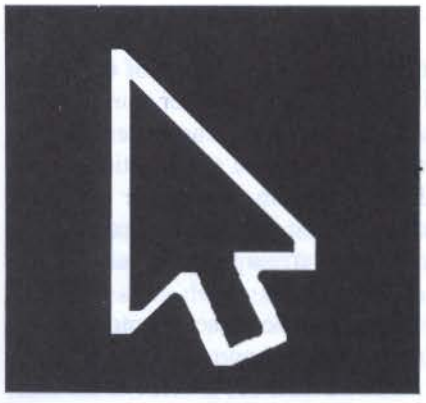


Figure 8-28. The cursor mask turns color backgrounds to white along the cursor's edge.

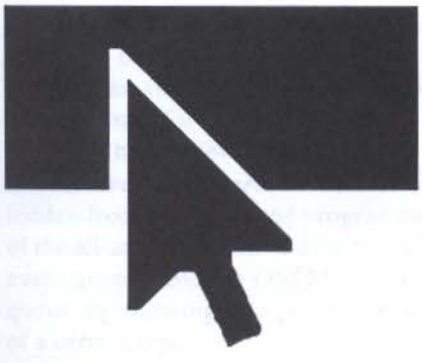


Figure 8-29. Where cursor and mask pixels are 0, the background image shows through.

Refer to the following table to understand how the cursor and mask images work together.

Cursor	*Mask*	*Background*	*Resulting Pixel on the Screen*
F	F	color	black
F	F	white	black
0	F	color	white
0	F	white	white
0	0	any	same color

Multiple Cursors

The cursor we've been showing here happens to be the design of the cursor QuickDraw automatically provides if you supply no other cursor parameters. An application program can easily have multiple cursor designs predefined and then called by QuickDraw depending on the location of the mouse pointer. For example, in a painting program, the cursor may be a pencil atop the drawing surface. But when you move the cursor to the menubar, the cursor switches to the traditional arrow to aid in selection of menu choices.

The mechanism that triggers the change of cursor based on mouse location is the Event Manager. This is only one of the important tasks we'll see this tool set doing in the next chapter. In the meantime, if some of the concepts in this QuickDraw slipped past your total understanding, take a few moments to review the material you're unclear about. Pay particular attention to grafport concepts. We'll see them later.

CHAPTER 9

The Event Manager

We had a brief introduction to the concepts behind event-driven programs in Chapter 7. There we said that the Event Manager provided tools to test for the presence of events — mouse clicks, key presses, and so on — so that the program can branch to predefined routines, or actions, in response to a user event. In this chapter, we'll look more closely at the event mechanism. Along the way we will also be introducing you to window and menu concepts that will be more fully described in their respective tool sets' chapters.

TWO EVENT MANAGERS

The Event Manager is one toolset, but it has two categories of functions: hardware-oriented and application-oriented. The distinction may not be perfectly clear, since applications obviously rely on hardware such as the mouse and keyboard for input.

The hardware-oriented part of the tool set is called the *Operating System Event Manager*, or *OSEM*. Most of what occurs in this tool set is hidden from both user and programmer. For example, the OSEM takes care of the all-important job of detecting hardware events and posting them to the event queue. Another OSEM tool does wholesale maintenance of the event queue, by allowing your program to clear the queue of all events or all events of a certain type.

As a programmer, you will be dealing more with the other part of the Event Manager, the *Toolbox Event Manager*, or *TBEM*. TBEM calls will be the focus of your program's main event loop. For example, the TBEM picks up events from the event queue, records the location of the mouse pointer at any instant, reports the condition of the mouse button, limits user input to certain kinds of events (e.g., temporarily disabling the keyboard when you want only mouse input), and other operations.

Details about each event are maintained in a place in memory called the *event record*. Every event, whether it is one that is stored temporarily in the event queue or one that does not get placed in the queue, has its own event record. We'll examine the event record more closely later in the chapter, but for the moment we'll consider one event record item: the *event type* that caused the event record to be created in the first place.

EVENT TYPES

Since an event-driven application must branch in response to user or system input, it must be able to distinguish one type of event from another. Indeed, the Event Manager keeps track of such information, classifying each event according to its type. There aren't many types to remember, yet you'll see that they have great power over your applications.

Mouse Events

Every time a user presses the mouse button in an event-driven application, that action creates an event record. Actually, the mouse button is the source of two different types of events: *mouse-down* and *mouse-up* events. The distinction is very important, because an application may require the user to hold the mouse button down while dragging a screen object from one location to another. The event loop must know that the mouse button is pressed, is still held down by the user, and eventually released so that any screen updating can be performed.

The Event Manager also tracks the location of the mouse pointer — the hot spot of the cursor — at every instant. When a mouse-down or mouse-up event occurs, the location of the mouse pointer at that instant is written into that event's record.

Even when a mouse-down or mouse-up event is not occurring, your application can retrieve the coordinates of the mouse pointer from the TBEM with the Event Manager call, GetMouse. Coordinates resulting from the GetMouse call are *global*, which means they are coordinates of the entire screen, with no regard to grafport coordinates. You'd most likely put a GetMouse call inside the event loop, if you want to test for the location of the

mouse pointer during each cycle through the event loop. If the pointer falls within a particular region of the screen (a short test in the event loop can determine this), the program might branch to a quick procedure that summons a different cursor record, thereby instantaneously changing the cursor image, or perhaps changing an SCB.

Mouse events are not strictly limited to the mouse as a handheld input device. Event records automatically track which of two possible buttons have been pressed when the event took place. Of course, the standard IIGS mouse has only one button, but some joystick controllers have two. The Event Manager recognizes these buttons as button 0 and button 1.

Keyboard Events

Whenever you press a character key on the keyboard (including the numeric keypad section), that action creates a *key-down event*. Notice we specify *character keys* — the keys that would generate identifiable characters on the screen if you were to type them in, say, a word processing program. Other keys — Shift, Caps Lock, Control, Option, and Open-Apple — are called *modifier keys*, and they do not generate key-down events by themselves.

You normally press a modifier key simultaneously with a character key to effect an action other than the normal key press. For instance, the Shift key causes the capital letter of that character to be passed to the system and screen; the Open-Apple modifier may be programmed to signify a keyboard shortcut for a pull-down menu action, such as Open-Apple-S to save a file. Whenever a key-down event occurs, the status of each modifier key is logged into the event record. Therefore, your application's event loop will be able to test for the presence of a modifier when the user pressed a key.

Another kind of keyboard event is called an *auto-key event*, which occurs whenever the user presses a character key and holds it down until the key begins sending repeated characters. The time delay before a key begins repeating is controlled by the Key Delay setting in the Control Panel desk-accessory. Moreover, the speed at which a key already in auto-key mode issues repeated characters to the system is also controlled in the Control Panel.

Window Events

Unlike mouse and keyboard events, which a user generates by some direct physical action on the computer's input hardware, a window event is generated by the Window Manager. In all fairness, the ultimate cause behind a window event is probably a human physical action, such as clicking the mouse pointer inside a partially covered window to make that window

active. But when you click the mouse button, the Event Manager has no notion that the mouse is located in a particular grafport. Yes, it knows the coordinates of the mouse pointer, but it takes a call to the Window Manager to interpret those coordinates of the mouse-down event as occurring inside the portRect of a particular grafport. When the Window Manager makes that window the active window with the SelectWindow call, it also generates an *activate event* and its corresponding event record.

After an activate event, particularly if the window that was just made the active window had been previously overlapped by another window, the Event Manager will likely generate an *update event* and its corresponding event record. The presence of an update event should signal your application that it needs to redraw one or more windows on the screen. The Window Manager will handle the actual redrawing, but it takes its cue from the Event Manager.

Neither activate nor update window events are placed in the event queue. Otherwise, they are just like any other kind of event.

Your program should include a test for the presence of an update event in each pass through the event loop. This will keep your windows fully drawn and "up-to-date" whenever you resize, move, or change the order of overlapping windows.

The Switch Event

Just as the Window Manager generates window events, so too does the Control Manager generate *switch events*. "Switching" here means switching from one application to another. Although not directly supported with toolbox calls at this time, application switching is a convenience that will probably become a standard feature of the Apple IIGS applications environment, just like Andy Hertzfeld's Macintosh *Switcher* became an accepted standard for that computer. In anticipation of the future, the IIGS Event Manager has reserved an event type for switching.

This is another one of those indirect events, because a switch event is ultimately the result of a press of the mouse button (sometimes a key press) when the pointer is in the screen region of a switch control. The Event Manager initially simply detects a mouse-down event occurring at a point on the screen. The Window Manager and Control Manager work together to determine if the mouse-down event occurred in the switch control. If so, the program can post a switch event to the event queue — a case where the application momentarily takes control of the event queue (via the Operating System Event Manager tool call, PostEvent) to force an event. After that, the TBEM can test for the presence of a switch event and branch program execution to the routines that do necessary housekeeping (screen updating, file manipulation, storage of various screen states) before switching to a different application loaded in another section of memory.

Device Driver Events

A *device driver* is software that allows your IIGS to communicate with external devices such as printers, modems, digitizer pads, and the like. If your programs will be using other than standard devices, you will probably have to write the drivers. Input from these devices must be handled as events, just like the keyboard and mouse inputs are. Your program will have to post these events as *device driver events* to the queue using the OSEM PostEvent call. Then your main event loop can test for the presence of a device driver event and branch accordingly.

Application Events

The TBEM has accommodations for up to four user-specified events, which will be unique to your application. As with device driver events, your program is responsible for posting the events to the event queue when they occur.

Desk Accessory Events

A special event type, called a *desk accessory event*, is generated whenever the Control-Open Apple-Escape three-key sequence is pressed to invoke the classic desk accessory menu (the one from which you select the Control Panel). This special event is supplied as a convenience for your program to detect the call to classic desk accessories, which require a dramatic change in the computer's operating modes. By checking specifically for this three-key sequence, the Event Manager bypasses the need to continually test key-down events for this unique sequence and possibly slowing down the regular key-down event actions. Instead, your event loop simply checks for the presence of a desk accessory event during each pass through the loop. When this event is pulled from the queue, the program can branch to the routines that prepare your application (and especially memory management) for the change in environments to the classic desk accessory display.

The Null Event

When you don't touch the mouse or the keyboard, and when no device driver is sending data to the computer, the event queue is empty. In the meantime, your event loop is cycling like mad, asking (*polling*) the event queue for whatever is there. As long as no events are waiting to be polled, the Event Manager will tell your application that there is a *null event* — nothing — pending. A critical event loop call (GetNextEvent) has been designed to test for the presence of a null event quickly so the loop can start over as soon as possible.

EVENT PRIORITIES

The TBEM recognizes that some events are more important to an application than others. As a result, it has divided event types into four ranks of importance. An event with a higher priority than others will be presented to an application first, even if it actually occurred more recently than an event that has been in the queue for some time (we're talking tens or hundreds of milliseconds here, but that could be a long time to an event loop polling the event queue thousands of times a second). Establishing priorities is particularly important when events are generated by other tool sets, which work much faster than our fingers do on the mouse or keyboard.

If the Event Manager held a fully loaded event queue and had every type of nonqueue event pending, it would offer events to your application in the following order:

1. Activate events
2. Switch events
3. Mouse-down, mouse-up, key-down, auto-key, device driver, application-defined, and desk accessory events in the same order they were posted to the event queue (that is, First In, First Out)
4. Update events (in order from the frontmost to rearmost window in a multiwindow screen display)
5. Null event

Let's see how the Event Manager handles each level of event priority.

Top on the priority list is the activate event. You'll recall that this event type is not placed on the event queue. When your program polls the Event Manager, the TBEM looks for a pending activate event before it looks into the event queue. If an activate event is available, the TBEM will pass it along to your application before anything else.

If an activate event is not present, the TBEM looks once more outside the event queue, this time for a switch event. If a switch event is ready to be passed to your application, the TBEM makes a short detour to pass along any update events that may be pending (even though they are normally lower on the priority list). This assures that all windows are updated *before* the application switches to another program (and will be fully updated and ready when you switch back to it). As soon as the TBEM works its way through all pending update events in this detour, it finally passes the switch event to your application.

If the TBEM, when polled by your application, finds no pending activate or switch events, then it looks into the event queue and pulls off the oldest event in the queue to give to your application. Events in the event

queue are generally those that are directly controlled by user input. If the event queue should fill up before your application has a chance to retrieve events, the TBEM automatically starts "scrolling" old events out of the queue — they disappear into thin air. A poorly designed program, for instance, may not poll the Event Manager frequently enough to keep the event queue from spilling over. A fast typist may appear to outrun the speed of the computer, when in fact quick fingers are outrunning the program's ability to retrieve events from the queue owing to a ponderously implemented event loop.

Once the TBEM clears the event queue, it looks to see if any update events are pending. By placing the normal update event (i.e., not the one forced prior to a switch event) near the bottom of event priorities, window updates, which tend to trigger relatively slow window-updating routines, don't get in the way of important user input events. For example, if your program displays three overlapping windows on the screen, each of which is affected by the movement of the topmost window, three update events will be generated: one for each window. The Event Manager will order these three events so that the frontmost window's update event is handed to your program first. If, while the front window is redrawing its content, a key or mouse button is pressed, that higher priority event will be passed to your application before the second and third windows are updated. The program will seem to respond much more quickly to user input than if it had to wait for all three windows to redraw themselves. Additionally, if the event is a mouse-down event in a different window, the other windows not yet updated will wait for the mouse action to finish and be updated only once to reflect the overlapping order of windows as the result of the mouse-down event.

At the bottom of the list is the null event. It merely indicates that there are absolutely no events pending — in the queue or otherwise — at that instant.

EVENT RECORDS

Links between the Event Manager and your application are established via event records — specifications about each event on or off the queue. When an event occurs, the Event Manager takes a "snapshot" of several indicators and stores those readings in that event's record. Each event record consists of five items:

what:	word	(event code)
message:	long	(event message)

when:	long	(clock ticks since system last started)
where:	point	(global mouse location)
modifiers:	word	(modifier flags)

An event record contains all the information your application's event loop needs to branch to previously defined procedures that make things happen in the program — the actions.

Most compilers and assemblers you will use to program the IIGS will have predefined variables (or variable suffixes) for each of these event items, thus saving you from explicit variable declarations for these items that will occur in practically every event-driven application you write. A typical variable name for the *what* event record item would be *event.what*.

Now let's look at each event record item, paying particular attention to the way the Event Manager translates some of this information into numeric codes that you will use to perform event loop tests. As we go through these items, try to look at them from the point of view of your application: You poll the Event Manager for an event to process; in return you receive a "dossier" about the first event to be processed; you must make a decision about what to do next based on the information in the dossier.

What — Event Codes

First on the event record is a number identifying which of sixteen possible event types this event is. The codes and their events are as follows:

Code	*Event Type*
0	null
1	mouse-down
2	mouse-up
3	key-down
4	undefined
5	auto-key
6	update
7	undefined
8	activate
9	switch
10	desk accessory
11	device driver
12	application-defined
13	application-defined
14	application-defined
15	application-defined

Again, owing to the frequency with which an Apple IIGS event-driven application will be testing for the presence of these events, most toolbox programming languages provide source code for predeclared variables (also called *identifiers*) that correspond to these event codes, just as they provide prefabricated data structures for event record items. Therefore, the event loop would likely consist of a series of CASE statements in a structure such as the following:

```
REPEAT

IF GetNextEvent(everyEvent,eventRecord)
  THEN
    CASE eventRecord.what OF
    mousedown:
        {branch to mousedown routine}
    keydown:
        {branch to keydown routine}
    activateEvent:
        {branch to activate event routine}
    updateEvent:
        {branch to update event routine}
    END;

UNTIL quit;

END.
```

In this sample, the event loop (between the REPEAT and UNTIL statements) first tests to see if the TBEM has an event of any kind (signified by the everyEvent parameter, discussed below) ready to hand over. If the TBEM returns an event other than a null event, the Event Manager hands the record of the event to the variable named eventRecord (it could be any name). The following CASE statement tests the "what" item in the event record (eventRecord.what). If the event code number is the same as assigned to mouseDown, then the program branches temporarily to the procedure elsewhere in the program's code that processes that event. If the event code is something else, then the program "falls through" to the next test, keyDown. If the event is a null event, the IF GetNextEvent test fails and falls through all the way to the bottom of the REPEAT routine. Until the value of "quit" is a logical TRUE (something the Quit choice in a menu could assign, as we'll see in Chapter 11, "The Menu Manager"), the loop will repeat itself over and over.

Message — Event Message

An *event message* is additional information your application needs from some of the event types coming its way. Not all event types need messages, because their event type is enough to help your application make its branching decision. But most types have additional information associated with them. They are:

Event Type	*Event Message*
Key-down	ASCII character code
Auto-key	ASCII character code
Activate	Pointer to window generating event
Update	Pointer to window needing redrawing
Mouse-down	Button number (0 or 1)
Mouse-up	Button number (0 or 1)
Device Driver	Defined by device driver software
Application	Defined by the application
Switch	Undefined
Desk Accessory	Undefined
Null	Undefined

Your language will ease the passing of messages from the event record to your application with the help of a predefined event message variable suffix, making your variable look something like eventRecord.message. In the case of a key-down event, for instance, eventRecord.message will be equal to the ASCII character that was pressed on the keyboard. Your program will then probably pass the value to the LineEdit tool for display on the screen and storage in a text buffer part of memory.

When — Timer Ticks

While the internal, battery-backed-up clock in your Apple IIGS keeps track of the current date and time without your having to update the numbers each time you turn on the computer, it also measures the number of ticks — sixtieths of a second — that have elapsed since you last rebooted the machine. This tick count is what the Event Manager uses to log the "time" of an event. If you need this information from the event record, it is readily available with the eventRecord.where variable. You can also retrieve the current tick count from the system with the Event Manager call, TickCount.

The tick count should not be considered "gospel" as far as time is concerned. Its counting may be temporarily interrupted during program execution (owing to some system-level calls), throwing the counter off a bit (unlike the system clock, which stays on track). To play it safe, use the tick

counts only to determine the relative precedence of events or the elapsed time between two closely spaced events, such as mouse clicks for a double-click.

Where — Mouse Location

As noted earlier, the event record holds the global coordinates of the mouse pointer at the instant an event occurs. For processing some mousedown events, subsequent procedures automatically convert the global coordinates to the coordinates of the grafport in which the event occurred. But others, notably Control Manager calls, work only with local, grafport coordinates.

The problem centers around the fact that a window has its own coordinate plane that is not related to the screen's global coordinates. For example, if you open a new window, the top left corner of its portRect will likely be (0,0). As you drag the window around the screen, the grafport for that window recognizes that starting point as (0,0), no matter where on the global screen that corner appears. Fortunately, QuickDraw II provides a conversion tool, GlobalToLocal, which converts a point from its global screen coordinates to the coordinates of the active window. From there, the Control Manager can see if the mouse-down even occurred in an area that indicates immediate action is needed.

Modifiers — Modifier Flags

Bringing up the rear of the event record is a 2-byte modifier flag. Several bits in this flag record the status of modifier keys and certain other modifying conditions when the event that generated this record occurs. It is up to your application, then, to test for the presence of these flags if you are looking for specific modifiers. For example, you might want to intercept key-down events that are modified by the Option key to signal a menu choice from the keyboard.

Modifier flags are distributed along the integer as shown in Figure 9-1.

The KeyPad bit will be set to 1 if a key-down event was the press of a key in the numeric keypad at the right of your IIGS keyboard. ControlKey, OptionKey, CapsLock, ShiftKey, and AppleKey behave similarly: their bits will be set to 1 when in the keydown position at the time of the event. One or more of these modifier keys can be pressed when pressing a character key or a mouse button, so several bits can be set in the modifier flag for an event. Bits for Btn0State and Btn1State, the conditions of buttons 0 and 1 (0 for the single button mouse) behave opposite their keydown companions. When a button bit is set to 1, it means that the button was in the *up* position at the instant of the event. Therefore, a mousedown event will have the Btn0State flag set to 0, while the Btn1State bit will be one. Lastly, ChangeFlag and

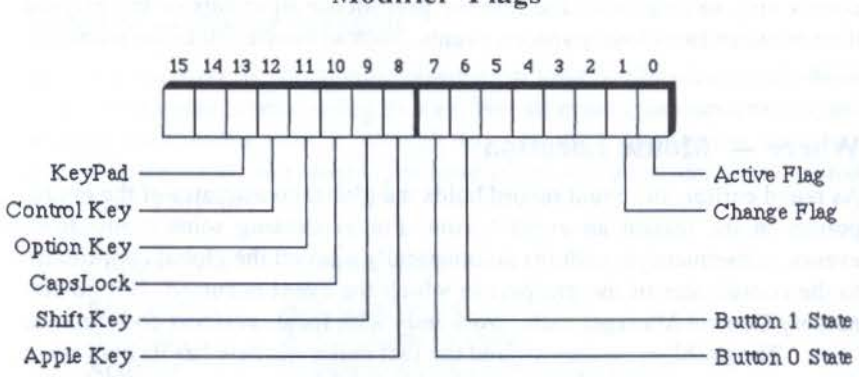


Figure 9-1. Event Record modifier flags.

ActiveFlag bits are helpful in performing actions resulting from activate actions. See the *Apple IIGS Toolbox Reference* for further details.

As with other items in the event record, the modifier item will likely be a predefined variable name in your programming language (something like eventRecord.modifiers). You'll be able to test the value of that variable against the decimal or hex equivalent of the modifier flag you wish to locate.

MASKING EVENTS

Occasionally, your applications will not require the Event Manager to proffer every kind of event that it is capable of recording. For example, a completely mouse-driven application may wish to essentially turn off the keyboard as an input device. Or perhaps you don't want the program to recognize auto-key events because the rapid input they provide will overload your application. To cut down on the number of events passed to your application from the Event Manager, you can *mask* those event types you don't need by setting bit flags in the *event mask*.

The event mask is a 16-bit integer, 13 bits of which control which event type(s) you wish to mask. The numbers of the controlling bits are the same as the event code numbers for the event types we saw earlier. In their mask integer form, the event types are shown in Figure 9-2.

A 1 in a bit location means that the mask will pass that event type; a 0 will prevent events of that type from being passed to your application.

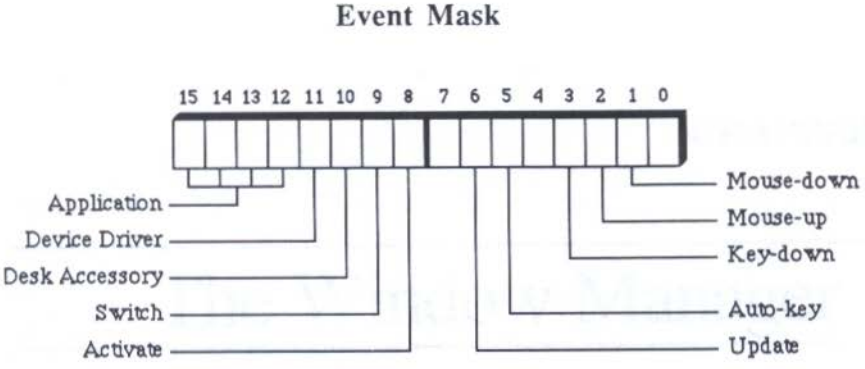


Figure 9-2. Event mask flags.

The mask integer is one of the input parameters of the GetNextEvent TBEM call. Your programming language may have a predefined constant (something like everyEvent) that you can plug into that parameter slot in the GetNextEvent call. It will tell the TBEM to pass all events. More event-limiting masks may be predefined, or you'll have to define them yourself in the application.

It's important to realize that the event mask does not prevent events from loading up the queue. The mask merely specifies which events will pass to the application. If you wish to prevent events from even reaching the event queue or the Event Manager's attention (for nonqueue events), you can use the Operating System Event Manager call that sets the *system event mask* to accept only those events you wish to process. Something to watch out for, though, is that if you turn off a particular kind of event with the system event mask, desk accessories won't be able to pass or receive masked events either.

In general, use an event mask with GetNextEvent whenever you can logically do so. Although masked events still go on the queue, the Event Manager discards unwanted events when the GetNextEvent routine runs. A mask may also speed your event loop by restricting forays through CASE structures to only those events your program recognizes.

Now that we've seen the basic workings of the Event Manager, we can proceed to the three tool sets that work with event records. We'll start with the Window Manager.

CHAPTER 10

The Window Manager

In the eyes of the user, most of the "action" in a toolbox-created Apple IIGS program will appear to take place in one or more onscreen windows. Through the window the user will be able to see some or all of the action area of a program. In text-oriented programs, the window will be the display and editing space for the document's contents. If the document contains more text than can fit comfortably within view of a window, you can *scroll* around the document, as if adjusting the large document beneath an opaque layer that has one transparent rectangle cut out; you keep adjusting the position of the underlying document until the desired area is visible through that cutout. The same is true for a graphics document that is larger than the screen. Yet you may design a graphics-oriented program that has but one steady background that will be seen through a window that is exactly one screenful in size.

Windows are relatively complex objects in the Apple IIGS toolbox world. Part of that complexity comes from the multiplicity of elements that a window comprises — graphics elements obvious to the user and additional toolbox elements solely under the care of the programmer. Another part comes from the responsibilities programmers have of upholding the user interface conventions for windows. For example, guidelines call for a window to become the active window when the user presses the mouse button with the mouse pointer anywhere in the window. This activation procedure is not completely automatic. It's true that the Event Manager and

Window Manager provide the mechanisms for all the actions needed to activate a window, but your program must coordinate these actions.

The IIGS Window Manager has a powerful tool, called the TaskMaster, which greatly simplifies many of the user interface concerns of programming windows. To use the TaskMaster correctly, however, you should understand the "manual" operations it replaces. We'll get to the TaskMaster, but only after a firm grounding in the lower level calls in the Window Manager.

WINDOW CONCEPTS

A good place to start is defining the many terms that apply to windows and window management. Consistency in terminology is important so that if you need to ask questions of more experienced programmers, you'll be asking the correct questions.

Desktops and Windows

When you start up your Apple IIGS with ProDOS 16 and the Finder, the computer presents you with a screen containing several icons and a menu bar. The entire screen work space is called the *desktop*. Your program, too, has a desktop. It is just like a totally blank desk surface. It has no feature of its own except for pattern and color, like a rosewood grain on a real desktop. Only by placing objects on the desktop, and perhaps moving them around on the desktop, do you make it useful. One such object might be an icon or a menu bar.

The object we're concerned with here, however, is a window. A window will offer your program a way of displaying information — text, graphics, or both — to its user. Extending the desktop metaphor an additional step, a window lets us see the contents of a document just as we can see what's written on a piece of paper resting on a real desktop. Multiple-page documents, such as a ten-page report, however, are not placed on the IIGS desktop as a stack of pages. Instead, the pages are positioned end to end, with the window being used as a viewer to a portion of the document.

The Apple IIGS also allows more than one window to be on the desktop at a time. Windows can overlap each other, but you see only that part of each window that has no obstruction between it and your eyes. The desktop metaphor of overlapping pages holds true.

Standard Windows

You can design a window to be of any size — even larger than the screen — and virtually any shape (although shapes other than rectangle variants are

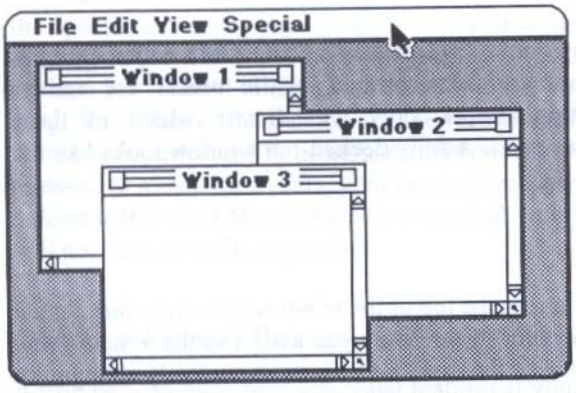


Figure 10-1. Overlapping windows on a desktop.

difficult to produce). To help newcomers overcome the complexity of designing a window from scratch, the Window Manager provides two predefined window frames, called *document* and *alert* windows.

Of these two, you will work predominantly with the document window. Alert windows are created by the Dialog Manager, and don't offer the design element flexibility of the document window.

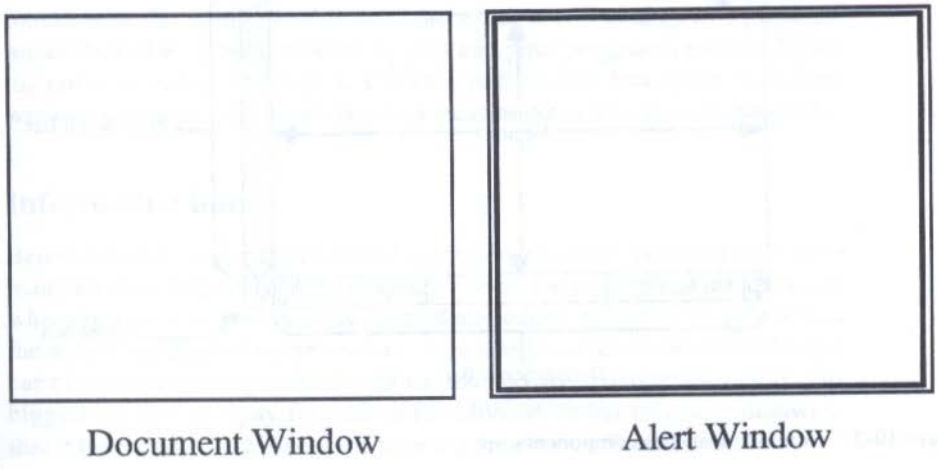


Figure 10-2. Predefined windows.

WINDOW COMPONENTS

To the standard document window you can add a variety of standard components. Some components are informational, while others are action-oriented. Your application's work flow will dictate which of these components your windows need. A fully decked-out window looks like the one in Figure 10-3.

Title Bar

Extending the full length across the top of the window is the *title bar*. Aside from its clear duty as conveyor of the name you assign to the window (the name may also be the name of the disk file document currently showing in the window), its four horizontal lines provide an important visual clue about the window. Whenever the window is active, the four lines will be visible; the instant the window becomes inactive, the lines disappear. When several windows are on the desktop, these horizontal lines show you immediately which window is the active one of the bunch. Fortunately, the Window Manager handles the erasing and drawing of these title bar lines automatically when your program deactivates one window and activates another.

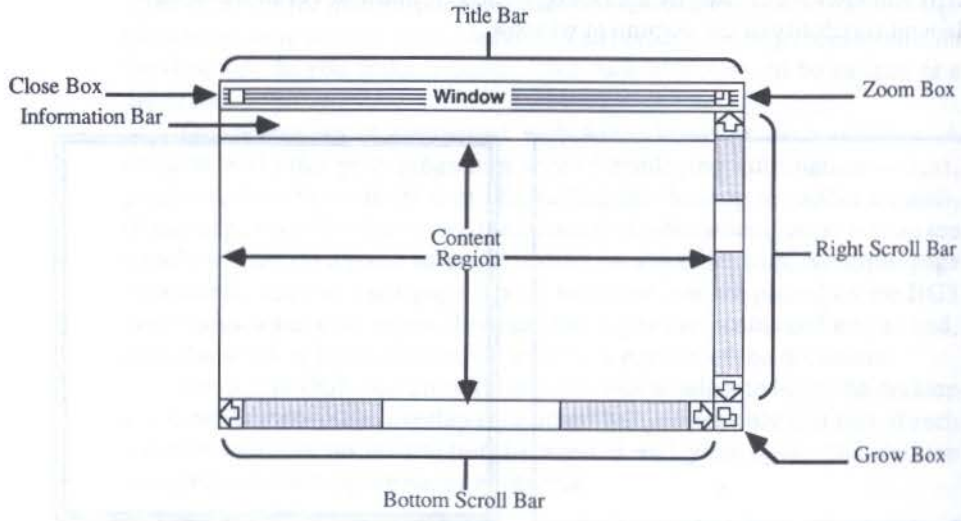


Figure 10-3. A window and its components.

The title bar claims another important function for a window. If one of the specifications for your window is that it is *movable*, then the user can drag the entire window around the desktop by placing the mouse pointer in the title bar (but not in either the close or zoom boxes), clicking, and dragging. An outline of the window will be visible, indicating where the window would be repositioned if the mouse button were released. When the button is released (a mouse-up event), your program must reposition the window and redraw it (the TaskMaster helps a great deal with this). The window's size will not change in this operation.

Close and Zoom Boxes

A title bar can have two additional features if your program requires them. The most common is the one at the left edge of the title bar: the *close box*. If you specify that your standard document window is to have a close box (assuming you have already specified a title bar), the Window Manager will place the close box along that edge. The Window Manager has only one location for the close box so that your program users will instinctively know how to close a window in precisely the same way they close windows they see in the ProDOS 16 Finder. When a user presses the mouse button with the pointer inside the close box, your program will branch to any of several possible actions, one of which should be to remove the window from the desktop. The window may still remain an object in memory (described later) for fast reopening later in the program, but a close box should remove it from the desktop to be consistent with the *User Interface Guidelines*.

The second title bar feature is called a *zoom box*. When a user clicks the mouse with the pointer in this box, the window will change to a predetermined size (i.e., predetermined by you and your program), usually filling the entire screen or close to it. Clicking in the zoom box again causes the window to resize itself to the size and location from which it last zoomed.

Information Bar

Below the title bar is a region called the *information bar*. In this space can go many kinds of helpers for your program's user. For example, it may be a line where spreadsheet formulas are entered and edited before being placed into the actual spreadsheet in the window. You may place icons there that the user can click to perform specified tasks without having to pull down menus. The biggest penalty you pay for adding an information bar to your windows is that it takes space away from the viewing area inside the window.

Scroll Bars

Scroll bars can be added as necessary. If you wish to restrain the document from growing any wider than the width of one screen, you can prevent horizontal scrolling by omitting the *bottom scroll bar* and placing only a *right scroll bar* in the window. When you add a scroll bar to a window's definition, the Window Manager automatically places the desired scroll bar in its appropriate place in the window. Similarly, if you resize the window, the Window Manager adjusts the scroll bar sizing for you. But specifying what a click in each part of a scroll bar does for the user is your responsibility inside the program.

Grow Box

At the lower right corner of the window is the optional *grow box*. Dragging the mouse pointer in the grow box allows the user to adjust the location of the lower right corner of the window to see more or less of the window's document. The Window Manager (particularly with the help of the TaskMaster call) will help your application handle the resizing routines that result from movement of the grow box. You may have applications, or special windows in those applications, however, that will not need resizing for any reason. For example, you may use a fixed-size window to display a reference table on the screen. If you don't want to encourage users to resize a window, leave the grow box out of the window's specifications.

Content Region

The dominant part of a window is the area in which information is displayed: the *content region*. If we design two windows of the same frame size, one with scroll bars and one without, the one without scroll bars has a larger content region. Even though a document may be many times larger than the rectangular area that provides the window to it, the content region is strictly that area we can "see through."

Since each window component is a distinct entity, you can mix and match elements as you please. Here are examples of some of the combinations you can use in a standard document window.

THE PROGRAMMER'S WINDOW

So far, we've been talking about window components that program users see and recognize. Now we'll peel away the user layer to reveal some underlying machinery inside a window.

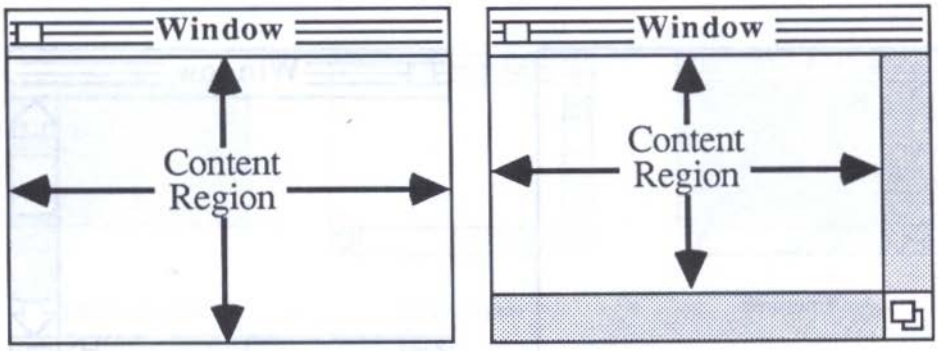


Figure 10-4. Content regions of two windows with different components.

Common Regions

Every Window Manager window is defined by two regions (among other specifications, to be sure): the *content region* and the *frame region*. We've already seen where the content region is: in QuickDraw terms, the content region is the portRect of the window's grafport. The frame region is the *outline* of the complete window — the outermost reaches of the window, including controls, title bars, and so on. Together, the content region and frame region define the window's *structure region*. Your programming concerns, however, will focus on the content and frame regions.

Optional Regions

Depending on what extra features you add to a plain document window — a title bar, a grow box, and so on — your window will automatically gain up to four additional regions that the Window Manager will work with. Those regions and their corresponding window components are:

Region	*Component*
go-away	close box
drag	title bar
grow	grow box
zoom	zoom box

Notice that when we introduced these regions by their component names, they were specifically *not* a part of the content region. That's true of their corresponding regions as well. All four of these regions are within the window's frame region.

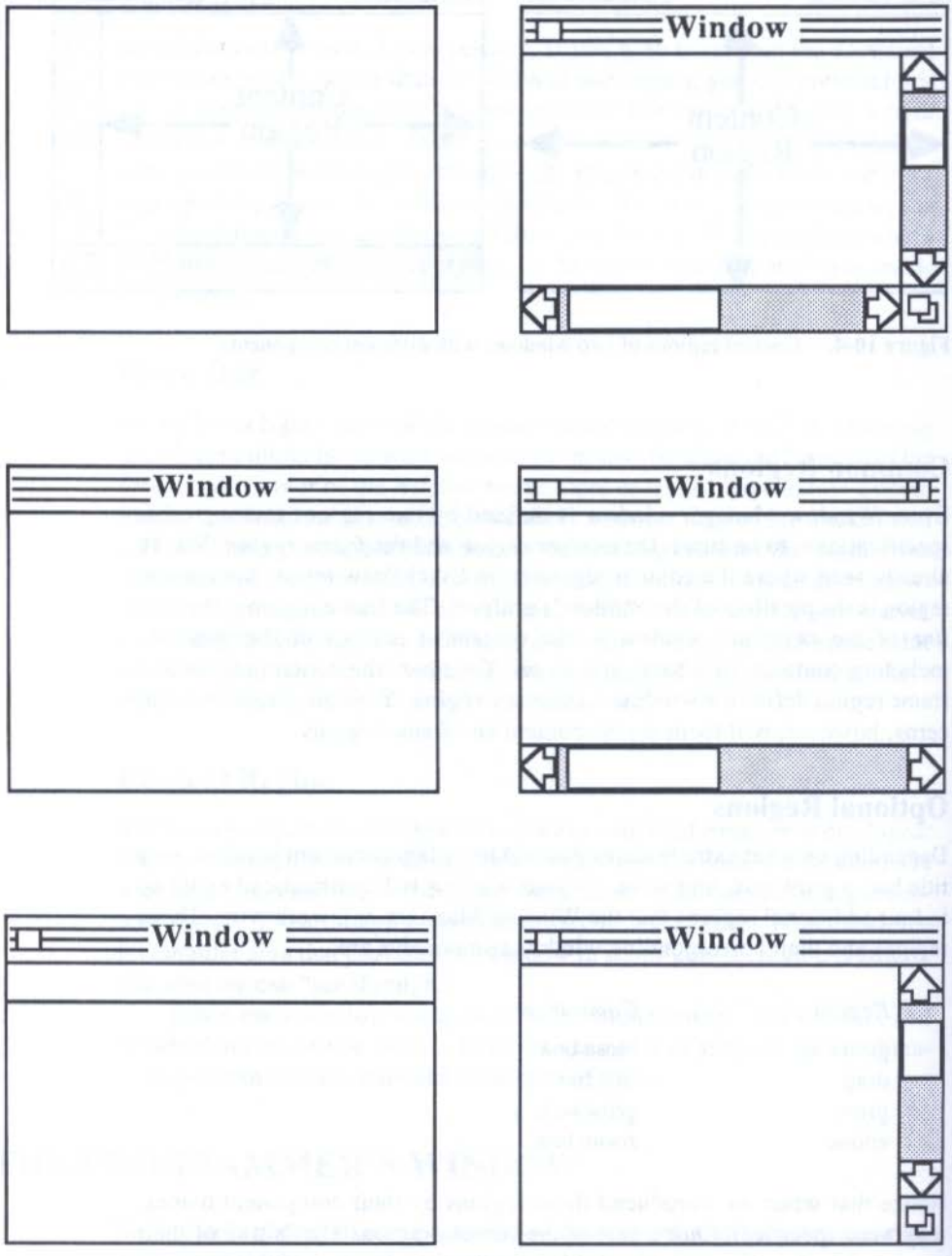


Figure 10-5. Some document window possibilities.

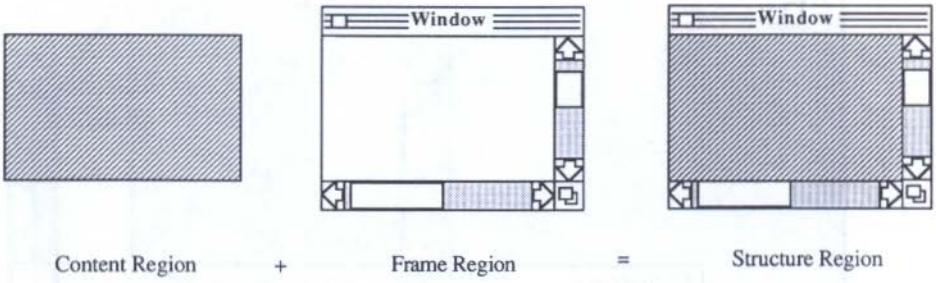


Figure 10-6. Relationship among content, frame, and structure regions.

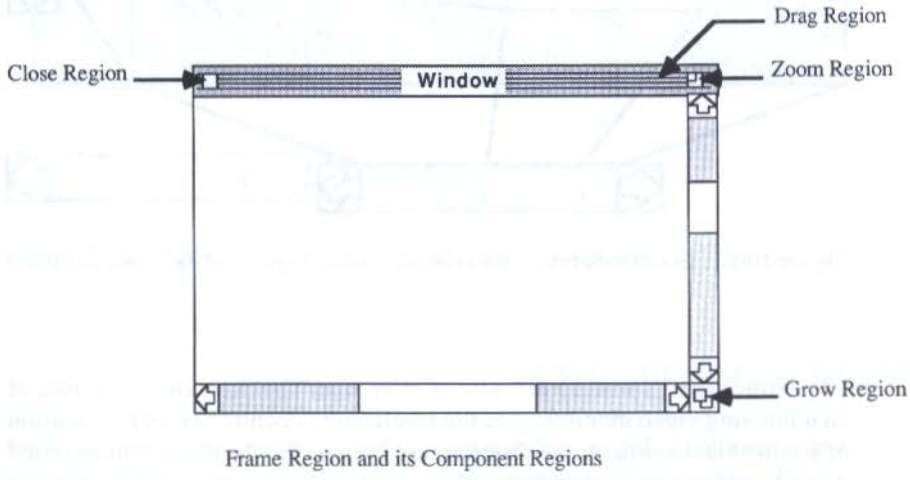


Figure 10-7. Window frame region and its component regions.

SCROLL BARS AND REGIONS

The precise workings of scroll bars will be covered in Chapter 12, "The Control Manager," but since scroll bars are often a significant part of your program and its windows, we'll discuss what these controls mean to a window, its regions, and the document.

From the user's perspective, the purpose of scroll bars, of course, is to bring parts of a large document into view of the window. From the programmer's perspective, however, scroll bars allow you to adjust the location of the content region to other parts of the *data area* — the extent of the information you wish to show.

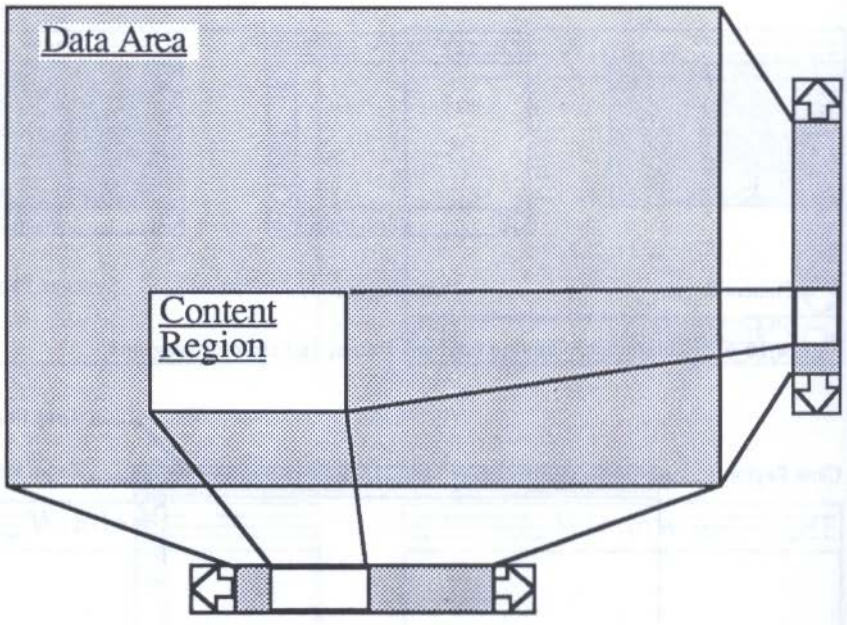


Figure 10-8. Scroll bars reveal the relative content region and data area sizes.

Notice the distinction in frame of reference: The user, sitting in front of an unmoving video monitor, sees the scroll bars as controllers of the location of a movable document; the programmer knows that the document is nailed down to a QuickDraw coordinate plane, and the scroll bars move the content region around the plane.

Scroll bars generated by the Apple IIGS Control Manager offer more feedback to the user than the standard scroll bars of the Macintosh Control Manager. The difference is that the IIGS scroll bar gives the user a visual clue to the *proportion* of the entire document he or she is viewing through the content region at any instant. Think of the gray area of a scroll bar as representing a scale of the entire measure of the data area in that dimension, either horizontally or vertically. The white box — the *thumb* — of the scroll bar, then, represents the size of the content region in that dimension *relative to the size of the data area*. Therefore, if you look at a scroll bar and find that the right scroll bar has a small thumb, it means that the content region is viewing only a small portion of a long, vertical document. A large thumb on the bottom scroll bar means that the content region is showing nearly the complete width of the data area, perhaps indicating that there is little need, if any, to scroll the document horizontally.

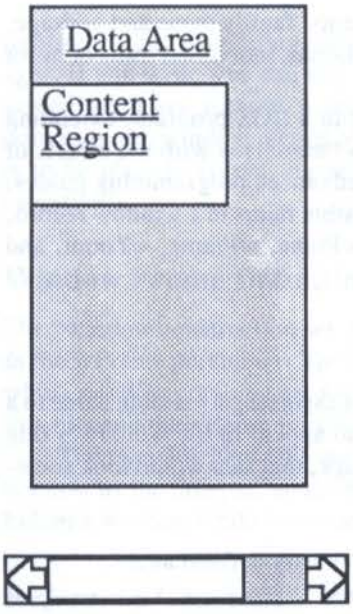


Figure 10-9. Scroll bars in a predominantly vertical document.

THE WINDOW RECORD

Each window you create will have its own window record. That means that all specifications for one window are stored separately from another. Therefore, if you give one window a set of visual characteristics, including a special set of components, a particular screen location and size, the information will be stored in that window's record and will not be adjusted as you manipulate other windows on the screen. Moreover, with a window's specifications safely stored in its record, you can remove the window from the screen (make it invisible), perform all kinds of other tasks with other windows, and later make the original window visible again. It will appear on the screen exactly the same way it did before it was hidden.

When the Window Manager creates a new window (as the result of the NewWindow toolbox call), it opens a new QuickDraw grafport (in fact, it specifically makes an OpenPort call behind the scenes), which becomes one of the items in a window record. All characteristics of a grafport, such as color tables, fonts, and pen states, then become part of the new window. Experienced programmers call this relationship between window and grafport *inheritance* in that a window inherits all the characteristics of a graf-

port, the way a newborn baby inherits its parents' family name and heritage, even though the child will grow up with additional, individual traits of his or her own.

A window record is the longest record in a IIGS program, extending over 300 bytes. Several parameters concern themselves with the effects of scrolling — a subject we'll leave for more advanced programming guides. But to give you a taste of the more readily visible items in a window record, we'll examine parameters labeled wTitle, wPlane, wFrame, wZoom, and wRefCon. Let's define each of these.

Window Title

Specifying a title for the window is as easy as assigning a variable name to a pointer indicating the actual text you wish to appear in the window's title bar. Depending on your programming language, that step would look something like this:

```
newTitle = "Window 3" {newTitle previously defined as a
                       pointer variable}
```

You will then plug newTitle into the NewWindow call's parameter list (see below).

The Definition Procedure

Toolbox programming frequently uses *definition procedures* as shortcuts to predefined screen objects. For example, the window record's wDefProc item is a number corresponding to predefined window types. In the Macintosh Window Manager, there are several different predefined window types, whereas in the IIGS Window Manager, there are only two, the document and alert windows. Since alert windows should be reserved for the Dialog Manager, you are left with the document window, which is specified in wDefProc by a 0. If you create your own custom-designed windows, you will assign each one a wDefProc identification number that you can pass as a parameter to a NewWindow parameter list, summoning that window type instead of the standard document window.

Window Order

You will normally want a new window to be topmost on the stack of windows currently open on the screen. To make sure that happens, you should specify a −1 as the wPlane parameter. The Window Manager takes over from there, deactivating the current active window, creating the new window, high-

lighting it, and generating all necessary activate events to the Event Manager. If you wish, you can slip a new window elsewhere in the stack of on-screen windows without making it the active window. Simply pass the pointer to the window record of the window behind which the new window is to be placed. To put the new window at the very bottom of the stack, specify a 0.

Window Frame Definition

The "window construction set" feeling of the Window Manager comes to life in the wFrame parameter. The flags you set in the wFrame word determine which window components are added to the window's frame. The controlling bits and their items are shown in Figure 10-10.

The movable item tells the Window Manager to allow a user to drag the window by the title bar, while the visible item acts as a switch for showing or hiding a window while its record is in memory.

Reference Constant

A program with the potential for many windows may need to keep track of those windows by a serial number or some other identifying number. The wRefCon parameter lets you establish a reference constant for a window. A practical use for this feature would be to generate new window titles consisting of the word "Window" and a number of the current window since you started the application (e.g., "Window 5"). By retrieving the wRefCon value of the last window opened, you can increment the value by 1, assign it to the next window, and incorporate the number into the new window's title.

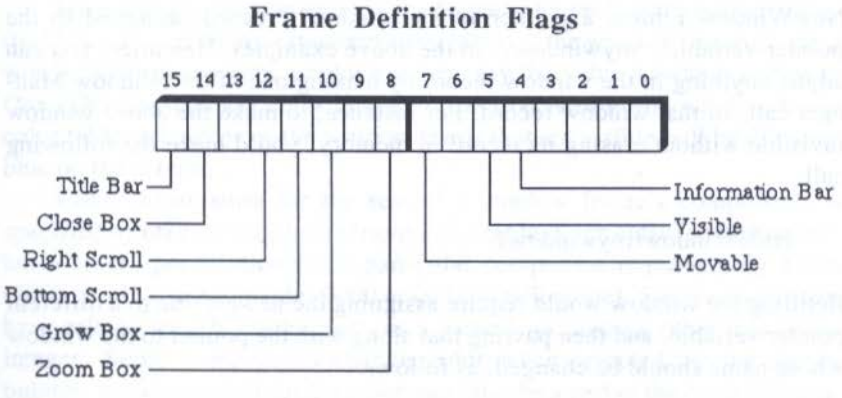


Figure 10-10. Window frame definition flags.

Full Size Window

The last parameter, wZoom, is the size of the rectangle the window's content region will become when a user clicks on the window's zoom box. If this parameter is set to 0, the window's content region will zoom to fill most of the screen (but not cover the menu bar).

CREATING A NEW WINDOW

Before your program actually creates a window, you first establish a list of specifications about the window. The list is rather long — 24 items — and items must be listed in a very specific order. Fortunately, many items can be set to zero, which tells the Window Manager to use standard values when creating the window. Data for this *parameter list* will be in the form of a *record* (or *structure*, depending on the language). Then you will call the NewWindow function, passing as an input parameter a pointer to the start of that list. The Window Manager reads that data and creates a window object out of it. NewWindow returns a pointer to the window record, which you can assign to a pointer-declared variable. The steps are sketched below.

```
Declare variables...
    windowData : POINTER
    myWindow : POINTER
BEGIN {window creation}
    windowData = struct (wFlag value, wTitle pointer, wRefCon...)
    mywindow = NewWindow(windowData)
END.
```

NewWindow returns a pointer to the window's record (assigned to the pointer variable, "mywindow," in the above example). Hereafter, you can adjust anything in the window record by making one of the Window Manager calls to that window record. For instance, to make the above window invisible without erasing its record in memory, you'd make the following call:

```
HideWindow(mywindow)
```

Retitling the window would require assigning the newest title to a different pointer variable, and then passing that along with the pointer to the window whose name should be changed, as follows:

```
NewestTitle = "Harvey's Window"
SetWTitle(mywindow,NewestTitle)
```

NewWindow does not draw the window on the screen. Its job is simply to create the window record. To draw myWindow on the screen, the program would call

```
ShowWindow(myWindow)
```

to finish the window creation job.

WINDOW FRAME COLORS

In our QuickDraw discussions, we went through the way colors can be applied to images in grafports. The Window Manager really doesn't care what colors QuickDraw is producing for your pixel images or other objects displayed in the grafport (i.e., as seen in its content region), but it is concerned about the colors of window parts — the frame, the title bar, the grow box, and so on. These colors can be set for each window and passed as a part of the window parameter list or adjusted later in the window record with the SetFrameColor call.

Frame Color Table

Terminology for frame colors may get a little confusing at first because it includes references to a window frame color table that is an entirely different concept from the QuickDraw color table. The window frame color table is not a list of pixel color values but, rather, a list of colors that apply to specific frame components. A pointer to the entire table is then included in the window parameter list or passed as the parameter of the SetFrameColor call.

For example, the first entry in the frame color table is the color number (0–15 from the grafport's standard color table) of the window frame. In other words, when you assign a color number, say the value 4 (which is blue in QuickDraw's standard color table), to the first item in the window frame color table, all pieces of the window frame that are visible will be drawn in blue on the screen.

Color information for the rest of a window frame's components is specified by other entries in the frame color table. Each entry is 16 bits wide, but a color specification for a particular component requires only 4 bits (0000 to 1111 in 320 mode, 0000 to 0011 in 640 mode). Some components have related specifications, such as a pattern, occupying other bits of the integer. Some components change color when selected by the mouse pointer, so this second four-bit color must also be a part of the color information for those items. Still other bits are unused, but are present just the same to fill out the space.

byte	Frame Component
0	Frame Color
2	Title Color
4	Title Bar Color
6	Grow Box Color
8	Information Bar Color

Figure 10-11. Window frame color table.

The window frame color table, which consists of 5 two-byte entries, is set up in the order shown in Figure 10-11.

Frame color applies to the lines that define the window's frame outline as well as the *outlines* for the close box, the zoom box, the grow box, and the information bar (Figure 10-12).

The title color entry sets many items. The lowest nibble controls the color of the title text and the interior colors of the close and zoom boxes. A separate color for title text in an inactive window is specified in the next higher nibble. The third nibble affects the color of the title bar background when the window is inactive (Figure 10-13).

Title bar color controls both the pattern and the color of the title bar graphics (Figure 10-14). Title bar patterns can be either the familiar four horizontal lines, a dither pattern, or solid, as defined by the high byte of the title bar color integer.

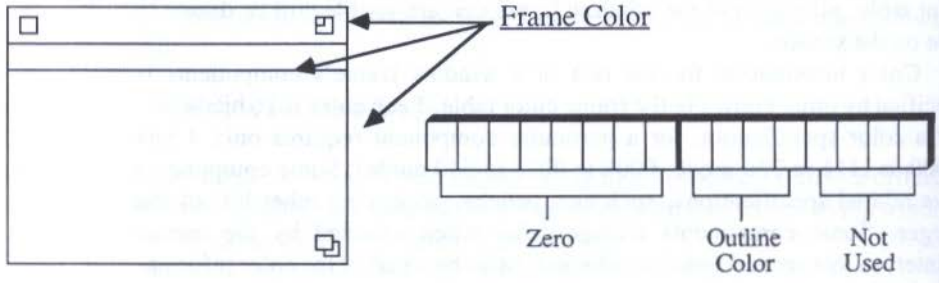


Figure 10-12. Frame color.

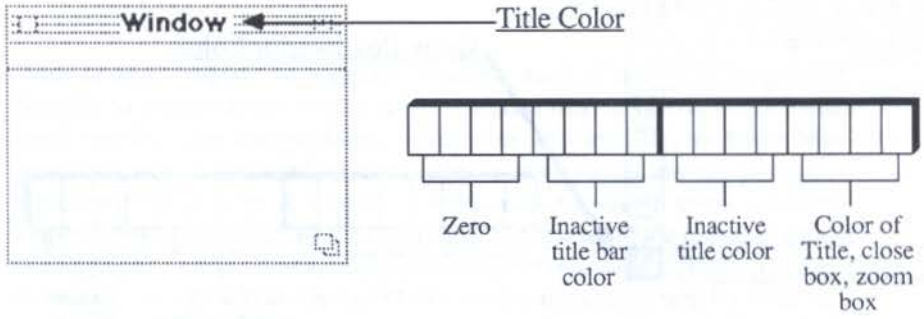


Figure 10-13. Title color.

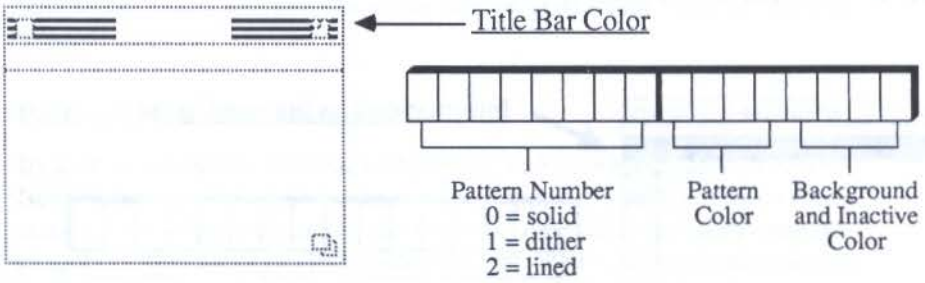


Figure 10-14. Title bar color.

The grow box has its own color item in the window frame color table (Figure 10-15). The lowest nibble in the color integer controls the *interior color* of the box when selected. The next highest nibble controls the interior color when not selected. Recall that the outline for this box is under the control of the frame color item.

If your program uses an information bar, its *interior color* is controlled by the last integer in the window frame color table (Figure 10-16).

UPDATING WINDOWS

Very few applications will escape the need for window updating. Even if the program has only one window (so there's no threat of a second applications window overlapping it), there will likely be desk accessory windows (called *system windows*) or alert windows temporarily covering the main window. When these overlapping windows close, they will leave blank holes in your main window. Your application must *update* those blank regions.

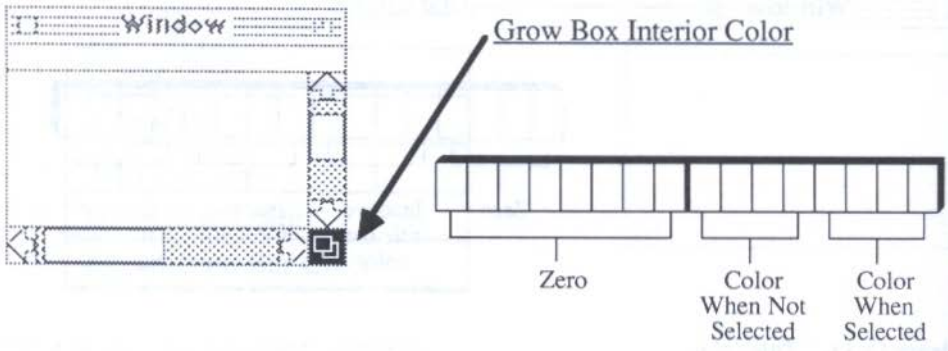


Figure 10-15. Grow box region color.

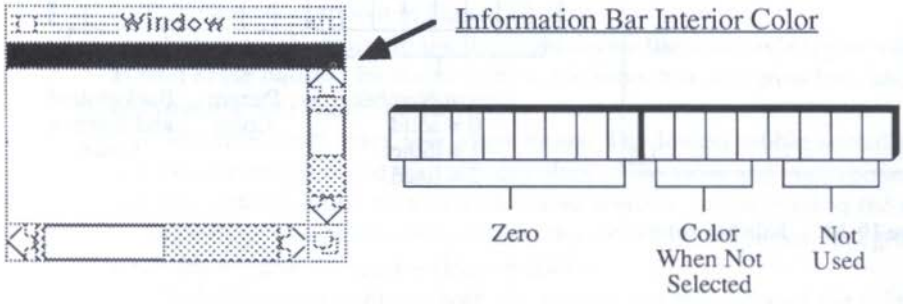


Figure 10-16. Information bar interior color.

The Window Manager and the Event Manager work together in alerting your program of the need for a window update and in performing the actual updating. When the closure or movement of an overlapping window exposes a blank region of a window, the Window Manager posts an update event to the Event Manager. Your application's event loop should be continually testing for the presence of an update event. If the program finds an update event, then it should branch to a redrawing routine that your window record points to. Redrawing by way of the Window Manager sets into motion a number of complex, yet automatic, procedures to make sure that only the affected region(s) of a window is(are) redrawn. This speeds up the perceived redrawing time of the window because the entire content region doesn't need to be drawn.

WINDOWS AND EVENTS

With so many regions in a typical window, each of which is to react differently to mouse-down events, there is a great deal of interaction between the Event Manager and the Window Manager. In particular, the Event Manager will trigger the need to check the location of the mouse pointer — which window region it was in — when a mouse-down event occurred. Depending on which region the cursor's hot spot was in, the program should branch to the routine that actually performs the onscreen action in response to that mouse click. Your program can do this manually, step by step, or if you're not doing anything too out of the ordinary, it can call upon the TaskMaster, which greatly reduces your need for managing every step of the way. But since the TaskMaster doesn't do it all (at least not in its current rendition), you'll still need to know the "old-fashioned" way of handling window events.

Polling the Event Manager

By now you should be familiar with polling the Event Manager with the GetNextEvent call. When it has an event to report, it proffers that event's record, from which you can extract the event type (Event.what in some languages). If that event is a mouse-down event, then your application should make further determinations as to the location of the mouse pointer at the time of the event.

The mechanism for that task is the FindWindow call. This tool takes the global screen coordinates of the mouse-down event (the "where" in the event record) and calculates (1) the window in which the event actually occurred, and (2) the region of that window in which the mouse-down event occurred. In other words, the first job FindWindow does is look up the global coordinates of the mouse pointer (as on a map) and see which window is under that pixel. The results of that search is a pointer to that window's record. Most languages will assign that pointer to a variable named "whichWindow" or similar. Once that is established, you will use whichWindow to help you perform one of several possible operations, as we'll see in a moment.

The other information that FindWindow calculates is the region in which the user clicked the mouse pointer. This information comes back as a constant from a table of predefined locations:

Constant	*Location*	*Meaning*
0	wNoHit	Event did not occur in the window
16	wInDesk	On the desktop
17	wInMenuBar	On the system menu bar
18	wInSysWindow	In a system window (e.g., desk accessory)

Constant	*Location*	*Meaning*
19	wInContent	In the window's content region
20	wInDrag	In the window's title bar
21	wInGrow	In the window's size box
22	wInGoAway	In the window's close box
23	wInZoom	In the window's zoom box
24	wInInfo	In the window's information bar
25	wInVScroll	In the window's right scroll bar
26	wInHScroll	In the window's bottom scroll bar
27	wInFrame	In the window, but in none of the above regions

These constants make it relatively simple to build a series of CASE statements within the event loop in a high-level language to test for the event taking place in any of these regions. You won't need to test for all of them — just the ones your windows and your application are concerned about. Depending on your language of choice, the structure might look something like this:

```
REPEAT
  IF GetNextEvent(everyEvent,thisEvent)
    THEN
      CASE thisEvent.what OF
      mouseDown:
          CASE FindWindow(thisEvent.where, whichWindow) OF
          wInMenuBar:
            {code to handle menus};
          wInDrag:
            {code to drag window};
          wInZoom:
            {code to zoom window};
            ...
          END; {finish mouseDown tests}
          ...
          END; {finish Event CASE routines}
UNTIL done;
END.
```

Thanks to the built-in constants of the language and their readily identifiable names, you don't even come into direct contact with the constant values for the various regions. Assembly languages for the IIGS have macros that perform many of these tasks as well, but you will have to move the values into registers, perform Boolean arithmetic on the values, and as a result of a test

between the constant value and the Boolean answer, either branch to the window routine defined elsewhere in the code or "fall through" to the next test. The precise implementation may be different, but the structure is largely identical to its high-level language counterparts.

Window Events

Several Window Manager calls are to be summoned as the result of Event Manager tests. For instance, when the event loop determines that a mouse-down event occurred in the drag region of a window, the next procedure should be the Window Manager's DragWindow call. Typically, your event loop will make the following Window Manager calls, based on the results of mouse-down region tests (i.e., after it has performed a FindWindow to determine both the target window and the region in that window):

DragWindow
GrowWindow
TrackGoAway
TrackZoom
SelectWindow

A few other calls are also involved, but we'll leave that up to the *Apple IIGS Toolbox Reference* to provide you with the details. It is a good idea to become acquainted with these items. Then you'll appreciate how much work the TaskMaster can do for you.

THE TASKMASTER

Realizing that a high percentage of event-driven programs perform identical key-down and mouse-down event tests, the Apple IIGS toolbox designers consolidated a number of Event Manager and Window Manager calls into a single call, TaskMaster. It is a built-in toolbox subroutine that makes a number of tool calls on its own. In addition to relieving the programmer of producing many tedious lines of code, it performs several important tasks with only two input parameters, while returning a single parameter that your application can use most efficiently for additional event loop tests.

Calling TaskMaster

TaskMaster call goes into the event loop in place of GetNextEvent. Actually, TaskMaster calls GetNextEvent as one of its first tasks. Input parameters to TaskMaster are just like the ones handed to GetNextEvent: an

event mask and a pointer to task record. Therefore, you can set an event mask if you like, using the same event mask constant your language provides (such as "everyEvent"). For the task record pointer, in a high-level language, you can simply define a task record pointer variable with an identifiable name, such as "myTask." When TaskMaster is finished, it returns a Task Code, which we'll discuss in a moment.

The Task Record

When TaskMaster calls GetNextEvent, the Event Manager hands over the record of the event having the highest priority at that instant (as it continually polls the system while the event loop cycles madly). That event record, of course, contains the five event record items that every event record has. The entire record reaches TaskMaster intact. TaskMaster, however, adds two extra items to the record: TaskData and TaskMask. TaskData is used by some of the internal calls that TaskData makes, particularly dealing with menus. The "data" that this record item tracks are the menu and menu item chosen by the user (more about this in Chapter 11). TaskMask allows you to tell TaskMaster which kinds of events it should not process, in case you wish to process a particular event differently from TaskMaster's usual handling. You set one or more of the thirteen bits (in a 32-bit LONG), each of which controls a single TaskMaster internal function.

A task record, therefore, looks like this:

what:	WORD
message:	LONG
when:	LONG
where:	LONG
modifiers:	WORD
TaskData:	LONG
TaskMask:	LONG

These record items behave just as event record items do. Therefore, when TaskMaster finds out from GetNextEvent that a mouse-down event has occurred, for instance, it knows to perform its magic by performing its major mouse-down event tasks: (1) calling FindWindow, and (2) performing the action appropriate to a mouse-down event in a particular region of the window.

Open-Ended

Now, you might imagine that as "smart" as TaskMaster is, there is only so much it can assume about your application. A hazard with this powerful a

routine is that it might do too much, thus preventing the programmer from exercising his or her own creativity.

Fortunately, TaskMaster doesn't try to be all things to all programs. The tool leaves enough information around for your event loop to use for further tests not covered by TaskMaster. For example, if TaskMaster polls the Event Manager and receives a mouse-down event in the content region of a window, TaskMaster will first make sure the window clicked upon is the active window. Then, even though TaskMaster is through with its work, it leaves wInContent as the TaskCode output parameter, and leaves the pointer to the window in the Message field of the task record. Now your event loop can perform further procedures based on the knowledge that the event took place in a window's content region. The program may, for instance, change the cursor from an arrow to a text insertion pointer because you intend to type text into that window.

As another example of TaskMaster passing through information for further program execution, if TaskMaster finds that a mouse-down event did not take place in the system menu bar or in the window's drag region, close box, zoom box, grow box, or either scroll bar, then it returns the window pointer from the FindWindow call it made inside the macro. Even though the call could find no action to perform, it still did some work along the way, easing further event loop tests you may wish to make.

To give you an idea of what TaskMaster call does for your application, here is a schematic of several internal Window Manager calls it makes:

```
mouseDown Event:
   Call FindWindow

If FindWindow says event was in:

      wInDrag:
            If TaskMast bit #6 = 0:
                  Exit and return TaskCode = wInDrag
            If command key not down and window inactive:
                  CallSelectWindow
            Call DragWindow
            Return TaskCode = inNull..

      wInContent:
            If TaskMast bit #7 =0:
                  Exit and return TaskCode =wInContent
            If window is inactive:
                  Call SelectWindow
                  Return TaskCode = InNull
```

```
                Else:
                        TaskRec Message field = window pointer
                        Return TaskCode = wInContent
        wInZoom:
                If TaskMast bit #8 = 0:
                        Exit and return TaskCode = wInZoom
                Call TrackZoom
                If TrackZoom returns TRUE:
                        Call ZoomWindow
                Exit and return TaskCode = InNull
Else:
        Return TaskCode = FindWindow value
        TaskRec Message field = whichWindow's pointer
```

Since TaskMaster checks only for key-down, mouse-down, and update events, it automatically passes through the event code from an event record as the TaskCode if no such events are being processed. The TaskMaster, then, takes the place of a great deal of the event loop, but not the entire loop.

TaskMaster's Future

Apple encourages you to use the TaskMaster call in your event loops for a reason that should appeal to most programmers. If, in the future, Apple enhances the TaskMaster call in a new release of the Apple IIGS Toolbox, then the routine may do some additional window and menu management for your application without your doing any additional coding.

It is unlikely that your old program would become unusable with the new system, since Apple tries to make upgraded ROMs and software tools compatible with earlier versions. But an enhanced TaskMaster may make a number of standard housekeeping jobs both faster and more compact in terms of program size. That should be a good incentive to use it whenever appropriate to your application — which may be all the time.

We'll see TaskMaster coming around once more in the next chapter, as we look inside the Menu Manager.

CHAPTER 11

The Menu Manager

Pull-down menus figure prominently in Apple's User Interface Guidelines for both the Macintosh and the Apple IIGS. We saw quite clearly in Chapter 7 that a pull-down menu system built into an event-driven program gives the user options during the running of a program that may not have existed in other program environments. Menus, as generated and handled by the Apple IIGS Menu Manager, are summoned without a change in modes — the user is at once in command mode and data entry modes. Choices of both menu category and menu item are done with one swift mouse motion. All in all, it is an elegant system, and the IIGS Menu Manager facilitates its use by handling the complex graphics for your application automatically.

MENU CONCEPTS

You have probably enjoyed working with pull-down menus on the IIGS already, but you may not have realized how much detail has gone into the menu mechanism and the way users interact with menus. As a programmer, you'll have to be very much aware of these concepts, because you should know how every part of your program functions, including those parts under total control of the toolbox. In this first section, however, we'll be speaking in user terms — what the user experiences when using menus.

The System Menu Bar

Applications using pull-down menus display the names of one or more menus in the *system menu bar*, which is drawn across the screen at the very top of the video area. The IIGS ProDOS 16 Finder, for example, displays a system menu bar when the operating system is loaded. If an applications program will be using a system menu bar, it will be in precisely the same location on the screen, although the names of various menus may be quite different.

Names of menus are called *menu titles*. A menu title should provide a clue to the kinds of operations provided by *menu items* in its hidden menu.

A special graphics character you can use in a menu title is in the shape of an apple. This character is traditionally used as the menu title for the leftmost menu in a system menu bar. Items in this menu are predominantly desk accessories, as well as an "About..." item. This latter item provides an avenue for the program writer to identify the program, its origin, its copyrights (if any), or other information the author thinks the user should know. The About... item usually generates a dialog box on the screen displaying the author's information.

Choosing a Menu and Item

When the user presses the mouse button with the cursor atop a menu title, the title's text changes to its inverse color and a list of menu items drops down in

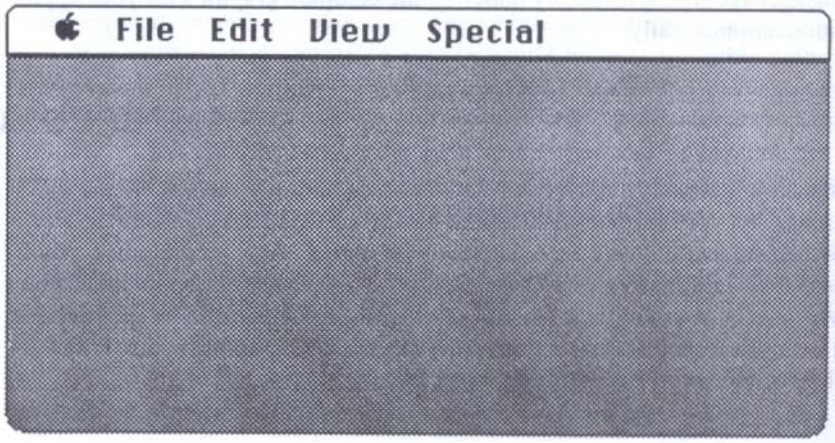


Figure 11-1. A system menu bar.

a small window below the menu title. As long as the user holds the mouse button down, the menu will be visible. Dragging the cursor into the window of menu items causes item names touched by the cursor to change to their inverse colors — become highlighted.

If the cursor passes beyond the edge of the item list, the menu will remain pulled down, but no item will be highlighted. You can drag the cursor back into the menu and make a menu choice.

To choose a menu item, touch the menu item text with the cursor while pressing the mouse button so that the menu item is highlighted. Then release the mouse button to choose the item.

(There's a fine point of terminology here, but one worthy of note. When a menu item is highlighted, it is said to be *selected*. But the instant you release the mouse button to set the action in motion, you have *chosen* the menu item.)

When you have chosen an item, the menu item will blink a couple of times, the menu list will disappear, and the action indicated by that menu choice will take place. While the action is in progress, the menu item's title will remain highlighted. This gives a visual clue that the program is still churning away, even if there is no other indication — changes on the screen or disk activity — of action taking place. As soon as the action is completed, the menu item returns to its normal color.

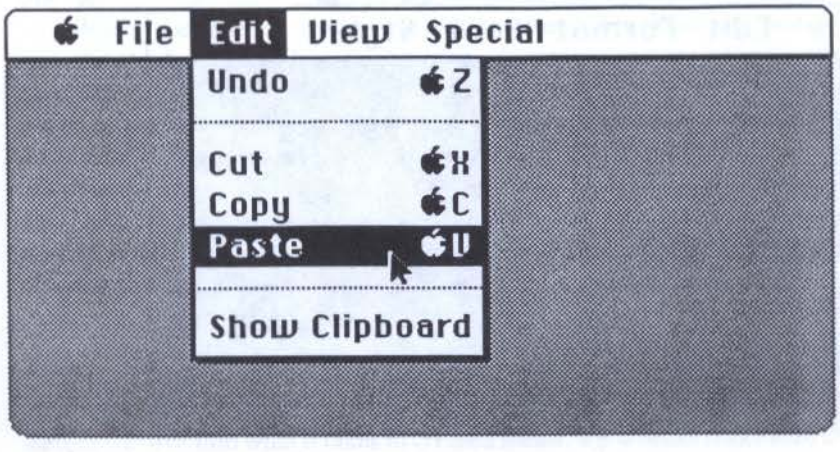


Figure 11-2. A pull-down menu with an item selected.

Enabled and Disabled Menus

Menu titles can be *disabled*, which means that you'll be able to pull down the menu, but you won't be able to choose any of its menu items. You can spot a disabled menu title by its shaded appearance in the menu bar.

When a menu title is disabled, so are all the items in its menu. As you drag the cursor across disabled items, they will not become highlighted.

You may also find cases when a particular item in an active menu is disabled under certain circumstances in a program.

This usually means that the operation indicated by the menu item would not make sense at that point in the program. For instance, if you have not made a change to a document, there would be no need for an Undo menu item in the Edit menu. The instant you press a key to type a character, the Undo menu item can be enabled.

MENUS FOR PROGRAMMERS

So much for the user's perspective. Let's look at the mechanisms we've been describing so far and find out how much the Menu Manager does for you and how much your application will have to do.

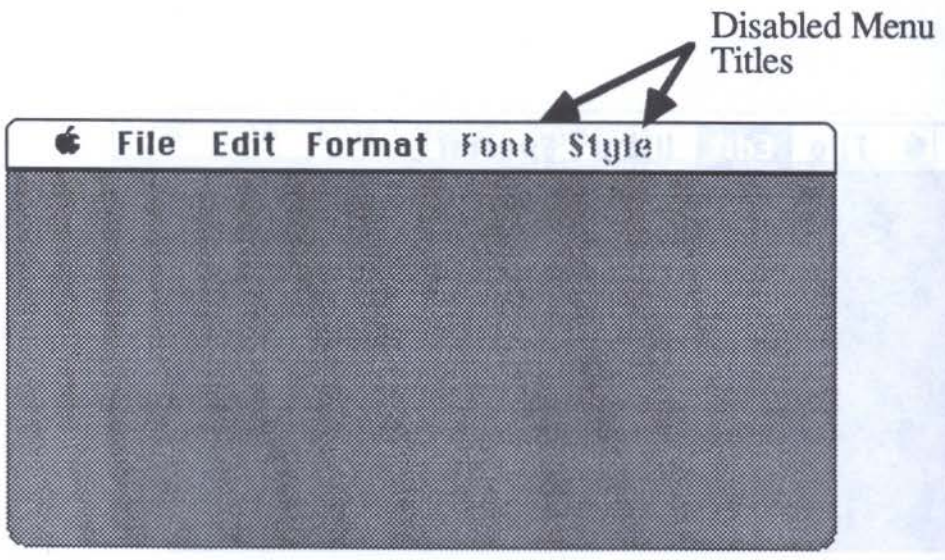


Figure 11-3. Enabled and disabled menu titles.

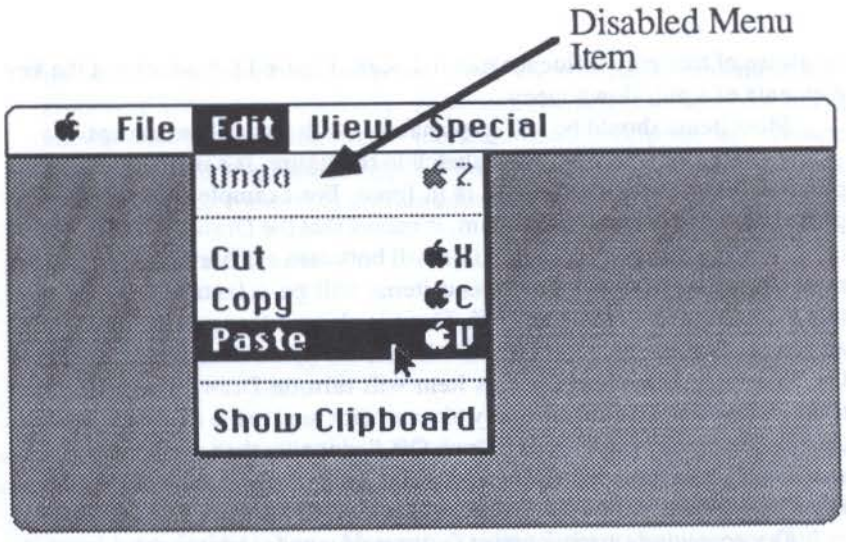


Figure 11-4. A menu with one disabled item.

For one thing, you'll have to supply the Menu Manager with the text characters that are included in the menu titles and menu items. Menu items will let you specify a number of extra parameters, as we'll see a little later.

Getting the Menu Manager to display a pull-down menu takes an easy interaction with the Event Manager. Once the Menu Manager discovers that a mouse-down event has occurred in the system menu bar, it takes over, pulling down the menu to which the cursor is pointing, highlighting the menu title, and highlighting each menu item as the cursor drags down the list.

When you release the mouse button, the Menu Manager instantly records which menu and menu item you chose (each has an identification number). It is your program's job to branch to the action routine that corresponds to the particular ID numbers recorded by the Menu Manager. In the meantime, the Menu Manager erases the pull-down menu from the screen, calls for a redrawing of the window(s) beneath the now-departed menu window, and keeps the menu title text in reverse. As soon as your menu action routine is complete, it must make the call that returns the menu title to its original state, indicating the program is ready for more input.

MENU TERMINOLOGY

Before we dive into what it takes to create a menu, we'd better make sure that menu terminology is clear. There are also several interface items you should

be aware of that may influence menu design. Figure 11-5 points out the key elements of a pull-down menu.

Most items should be self-explanatory, with a couple of exceptions.

A mark, indicated by a crosshatch in the figure, is a method of alerting the user that a particular feature is in force. For example, when a mark is placed next to the Draw menu item, it means that the Draw feature is turned on. Choosing this item a second time will both turn off Draw and remove the mark from the window. Few menu items will be a feature "switch" like this — turning an item on and off. There is also another way to handle such an action. For example, when Draw is off, the menu item can read, "Draw On," meaning that choosing this item will turn on Draw. The action that turns on Draw and would normally place a mark next to the item can, instead, change the menu item to read "Draw Off," meaning that choosing the item again will turn Draw off. Either method is acceptable under the User Interface Guidelines.

Key commands are important features of menus, because they provide experienced users of your application a way of calling menu items without reaching for the mouse. Instead, they would hold down the Open-Apple key and the character key indicated by the key command in the menu. Key commands are listed in the menu as a way of alerting users that a keyboard shortcut exists for the menu item. If the user grows tired of pulling down the

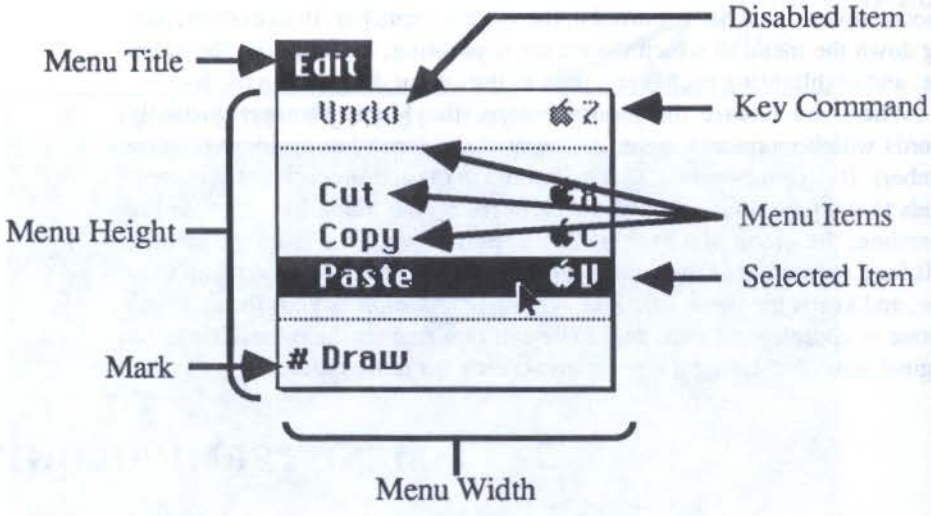


Figure 11-5. A menu and its components.

menu for a frequently used action, he or she will likely remember the key command and use it exclusively. Keyboard characters selected for key commands should be mnemonic — the letter should help the user remember what command it is, such as Apple-S for Save. The exceptions to this rule are four editing key commands that are common to the majority of applications:

Key Command	*Edit Action*
Z	Undo
X	Cut
C	Copy
V	Paste

Dividing lines should be used sparingly and intelligently. They can group together logical items in a menu. Often, two related menus can be combined into one menu by placing items from both menus in one list separated by a dividing line. Also use dividing lines to help a user find menu items quickly in a long list of largely unrelated items. Again, group logical items together as best as possible.

CREATING MENUS

Each menu title and its related menu items are considered a single *menu* on the system menu bar. To put together a system menu bar with several menus in it, you begin by creating each menu individually. The basic procedure for creating a menu bar is to (1) define the textual content and characteristics of each menu, (2) turn that definition into a menu that the Menu Manager will recognize, (3) insert each menu into the system menu bar, and (4) draw the system menu bar, which will display the full set of defined menu titles across the top of the screen.

To define the content and characteristics of a menu, you put together a *menu/item line list* for each menu.

The Menu/Item Line List

The list consists of the actual words that are to appear as the menu title and its items, as well as one or more special characters that the Menu Manager needs for the processing of menu commands and the display of things like key commands, menu marks, and so on. Importantly, the list must be put together in a strict format.

The format calls for the menu title to be preceded by two identifying characters, such as >>. Menu items in the list are preceded by a pair of different characters, such as hyphen. The end of the list must contain a single

character that is different from the item characters — you could use a period, for example, or even the same character as used for the menu title character. This last character tells eventually the Menu Manager that there are no more items for this particular menu. These leading characters behave like compiler punctuation marks.

A menu/item line list must also assign identification numbers (IDs) to each menu title and item in the list. Your program will use these numbers to identify which menu or item has been chosen by the user. Consequently, the IDs must be different numbers for each item in the list. No hard and fast rules apply to ID numbering conventions, but Apple suggests that menu titles be numbered sequentially, starting with 1; menu items should also be sequential, but starting with 256.

Menu IDs are added to each item in the menu/item line list by tacking on a backslash (\), the letter N (indicating a decimal number), and the number. The backslash character separates the actual text string that will appear in the menu from special characters that furnish additional information about a menu title or item (more on this in a moment).

Taking all this into account, a program with two menu simple menus in it would have two menu/item line lists:

```
>>File\N1
--Open\N256
--Save\N257
```

```
>>Edit\N2
--Cut\N258
--Copy\N259
--Paste\N260
```

A pointer to each menu/item line list's text is then passed as the parameter to the NewMenu toolbox call. That is, you make the NewMenu call twice in the above example. The NewMenu function returns a handle to the menu record. Note that this function does not produce any image on the screen — it simply creates the menu in memory. For each menu, you then call InsertMenu, with the menu handle as one of the call's input parameters. This call, too, deals with menus in memory only, not on the screen. Finally, call DrawMenuBar to display the menus on the screen, as shown in Figure 11-6.

The two menus, when pulled down by the mouse, are shown in Figure 11-7.

Menu Modifiers

Earlier we saw that menu items can be disabled, marked, and chosen from the keyboard. The way all of this is passed along to the Menu Manager for a new menu is by way of the menu item line list. Any menu item to which you wish to add one or more attributes needs to be sent to the Menu Manager with appropriate modifiers, sometimes called *special characters*. To alert the

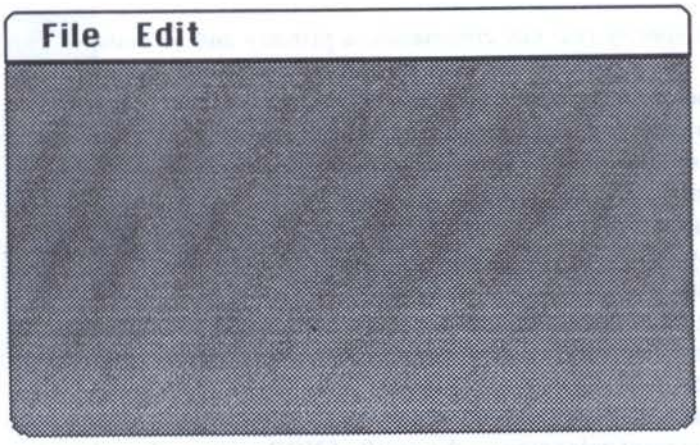


Figure 11-6. File and Edit menu titles in a system menu bar.

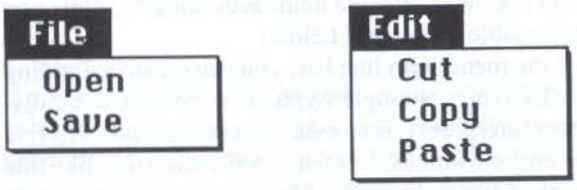


Figure 11-7. File and Edit menu items.

Menu Manager that a menu title or item will have some special attributes coming its way, add the special characters to the text string anywhere to the right of the backslash. Possible modifiers are:

*Aa	where "A" is the primary character to be used as key command, "a" the secondary
Ca	where "a" is the character to be used as an item mark
B	boldface menu item text
I	italicize menu item text
U	underline menu item text
V	insert dividing line after this item without adding an extra menu item
D	disable menu item text
X	use color replace highlighting

All of the above modifiers can apply to menu items; menu titles, however, can be affected only by D and X.

You can specify two key commands, a primary and secondary. This feature is in the toolbox mostly for the convenience of programs using keyboards and character sets in countries other than the United States. You don't have to specify a secondary character, but you must provide a blank space as a place holder for the secondary character in the menu/item line list. Only the primary character is displayed in the menu, so if you wish the key command to appear in uppercase in the menu but allow the user to access the menu item by typing a lowercase letter with the Open-Apple key, then specify both letters, with the uppercase character as the primary one.

With regard to the highlighting special character, if you don't specify an X character in a menu/item line then the Menu Manager uses XOR highlighting, which inverts the colors of the menu text and its background when an item is selected. For black and white menus, XOR highlighting works fine. But if you want color menus, the result of XOR math on the colors you choose may produce unreadable results. Therefore, the X special character alerts the Menu Manager to highlight selected items according to colors you plug into the menu bar color table (described below).

As another feature of the menu/item line list, you can create a dividing line between menu items by typing a single hyphen as the text to be displayed. The Menu Manager interprets this to mean you wish a dividing line to be displayed across the entire width of the menu. A dividing line like this must have its own ID number and must be disabled. As an alternative to a standalone dividing line, you can underline a menu item entry by appending the V special character. The underline does not add any space between menu items, as does the standalone dividing line. Your choice between underline or dividing line will depend on your personal taste in menu design.

To demonstrate the effects of special characters on a menu/item line list, we'll show you a list from a hypothetical menu and, in Figure 11-8, the resulting menu.

```
>>Text
--Undo\N270D
---\N271D
--Left\N272C#*L1
--Centered\N273*Cc
--Right\N274V*Rr
--Bold\N275B*Bb
--Italic\N276I*Ii
```

Incidentally, you can create your own Apple menu title by placing the @ symbol as the lone character in the Title entry in a menu string. If you place other characters in that string with the @ sign, the actual @ character will be displayed instead.

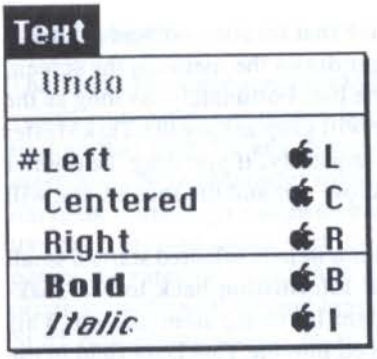


Figure 11-8. Sample menu.

MENU COLORS

If you want to add color to your menus, the Menu Manager has a menu bar color table, a pointer to which is part of the menu bar record. The table behaves much like the window frame color table in that it contains pixel values that refer to the QuickDraw standard color table. Three 16-bit integers control settings for menu text and background in both highlighted and unhighlighted conditions, plus the color of the outline box of a pulled-down menu. Notice that the colors apply to the entire menu bar.

MENUS AND EVENTS

The TaskMaster call simplifies the linkage between the event and menu mechanisms. Its basic job is threefold: (1) to identify when a mouse-down event occurs in the system menu bar; (2) to wake up the Menu Manager so that it will display the appropriate pull-down menu; and (3) pass along the number of the menu item actually chosen by a release of the mouse button.

Mouse Events

We saw in the last chapter how the TaskMaster automatically interprets the location of a mouse-down event. When the event occurs in the system menu bar (wInMenuBar), the TaskMaster calls the Menu Manager's workhorse routine, MenuSelect. As long as the mouse button is held down, MenuSelect has complete control over your program. Its initial job is to determine from information held in the task record where on the screen the mouse pointer is.

Then it figures out the menu title with which that location coincides. If it is indeed over a menu title, the Menu Manager draws the menu on the screen, allowing you to drag the pointer through the list. Fortunately, as long as the mouse button is down, the Menu Manager will keep asking the TaskMaster for the location of the mouse pointer. Consequently, if you drag the pointer to another menu title, the first menu will close up, and the second one will drop down.

Releasing the mouse button while a menu item is selected starts a small chain reaction. MenuSelect passes critical information back to the TaskMaster: the menu title ID and the menu item ID of the item chosen. This information is compacted together and placed into the TaskData field in the task record. Then the TaskMaster bails out, passing wInMenuBar as the TaskCode your application can test further. From there, your application can perform a CASE procedure to test the low-order word to uncover which menu item was chosen. Your program will then branch to a predefined action for that menu choice.

For a menu bar with two menus in it (each with a few menu items), the structure of the menu decisions would look like this in a high-level language:

```
PROCEDURE DoMenuStuff (TaskData);

  theItem = LoWord(TaskData);
  CASE theItem OF
                {File Menu items}
      openCommand: {branch to file open routine};
      saveCommand: {branch to file save routine};
      quitCommand: {branch to quit routine};
                {Edit Menu items}
      undoCommand: {branch to undo routine};
      cutCommand: {branch to cut routine};
      copyCommand: {branch to copy routine};
      pasteCommand: {branch to paste routine};

  END; {theItem CASE}
END; {DoMenuStuff PROCEDURE}
```

Routines that make up the menu item actions can be located within the CASE structure or defined elsewhere in the program as separate procedures.

One other item that should be added to the above listing is the HiliteMenu call. Inserted as the last call in the entire DoMenuStuff procedure, this Menu Manager routine turns off the highlighting of the menu title that MenuSelect highlighted. It provides feedback to the user that the event loop is again looking for something to do.

Key Events

The Menu Manager and TaskMaster also accommodate the keyboard equivalent commands for menus quite handily. The TaskMaster has a built-in routine that tests for key-down events. It passes the key character and its mask to the Menu Manager routine called MenuKey, which checks the menu list in the current system menu bar for a match, provided the Open-Apple key has been pressed along with the character. If there is a match, MenuKey passes the same information to the TaskMaster that MenuSelect does. Your program will use the same branching routines (analogous to the DoMenuStuff procedure, above) as if the menu choice had been made with the mouse.

CHANGING MENUS MIDSTREAM

There is no problem changing the selection of menu titles in the menu bar or menu items in a given menu while in the middle of a program. If the adjustment is a small one, you have several Menu Manager calls that help insert and delete menus and menu items. Insertions are generally done by specifying the ID number of the menu title or item that you want the new one placed after. Similarly, deleting menus and menu items is done by passing the appropriate IDs to the calls that remove items.

Just changing the content of a menu record will not automatically make the menus reflect that new content. You'll have to draw the menu bar each time you make a change to it (using DrawMenuBar). Neglecting to do so may cause your choices to produce results other than what you had planned.

Additionally, there may be times in your program when you wish to change the modifier of a menu item. Placing and removing a mark or enabling and disabling an item are the most compelling reasons. All specifications about a menu item can be adjusted with calls such as DisableItem, SetItemMark, SetItemStyle, and SetItemFlag. Consult the *Apple IIGS Toolbox Reference* for listings of calls and parameters that adjust these settings. They're simple operations you should be ready to incorporate into your programs.

Now that we've been through QuickDraw, the Event Manager, the Window Manager, and the Menu Manager, we've already covered the lion's share of the IIGS toolbox that affects the way users will interact with your native mode programs. We'll look at one more area, though, the Control Manager, to acquaint you with some additional terminology and give you more to think about in designing your programs.

CHAPTER 12

The Control Manager

Menus aren't the only screen objects that influence program actions or change program settings. We also have *controls*, run by the Control Manager. For the programmer, this tool set performs many of the same kinds of jobs as the Menu Manager does, but for onscreen controls. Your program will summon the Control Manager to do its low-level work, such as displaying controls and observing how a control was changed by the user. Then the Control Manager will pass along its results to your program for further action based on those results.

Most standard control functions that you will encounter in your early exposure to IIGS programming are handled by other managers. The Window Manager and the Dialog Manager, in particular, take care of numerous Control Manager calls for the program. Therefore, we will introduce you only to key concepts about controls in this volume.

CONTROL TYPES

The Control Manager comes equipped to help us create any of four standard control types: buttons, check boxes, radio buttons, and scroll bars.

Buttons

According to the User Interface Guidelines, a click of the mouse pointer on a button causes an immediate action. In other words, a button control (and

the window it appears in) usually disappears immediately after clicking it with the mouse. The name of the action is indicated by a text word located inside the rounded rectangle button outline prescribed by the Control Manager.

Check Boxes

A check box consists of a small square whose center is either blank (off) or is marked with an X (on). Clicking the box with the mouse *toggles* the setting in the box from one to the other and back again. Text, which the Control Manager places immediately to the right of the box, should indicate what action will be in effect when the check box is on. Check boxes are used primarily as a way of turning on or off a feature that will affect some future action, such as indicating in a print dialog box that you wish to print a background pattern along with the text in the window.

Radio Buttons

When you wish the user to make one selection from a list of two or more possible items, radio buttons are in order. Found in groups of two or more, these buttons and their actions in your program should be arranged so that when the user clicks on one button in a group (and it becomes highlighted), any previously highlighted radio button becomes unhighlighted. This will give the user the feel of pressing the pushbuttons on a car radio — only one can be in effect at a time.

Figure 12-1. Control Manager buttons.

Figure 12-2. Control Manager check boxes.

Figure 12-3. Control Manager radio buttons.

Scroll Bars

Falling under the broad Control Manager category of *dials*, scroll bars can be used for scrolling documents in a window as well as for adjusting numeric setting from a possible range of settings. Whereas other types of controls are essentially on/off indicators, scroll bars (and other dials) provide control over quantities, whether the subject is the measure along a lengthy document or a numeric setting that affects some other actions, such as adjusting a volume control. A scroll bar or other dial can also be used as a way of visually communicating a value from the program to the user, such as the relative amount of free space on a disk.

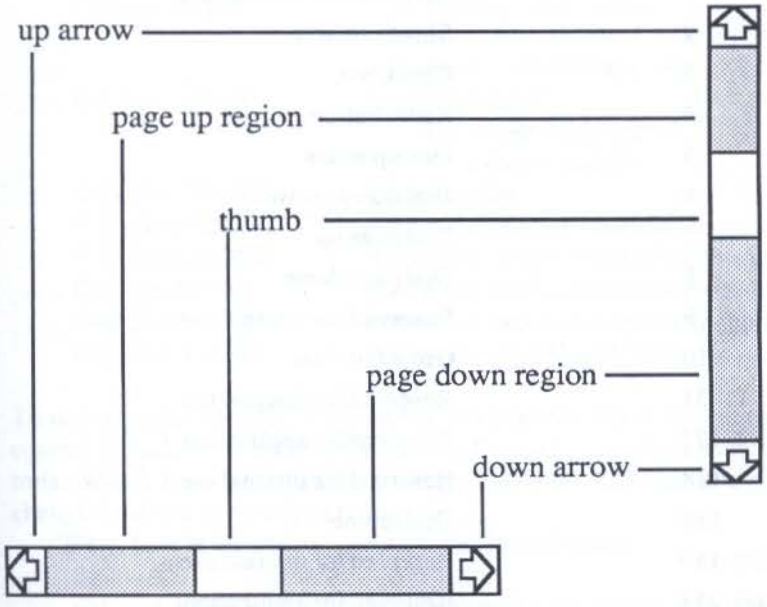


Figure 12-4. Scroll bars parts.

Scroll Bar Components

Scroll bars have several *parts*, each of which the Control Manager recognizes as a distinct unit. Since your program may need to respond to mouse actions in these parts, you should be aware of their names, as illustrated in Figure 12-4.

Clicking in one of the arrows should cause the screen to scroll in the smallest logical unit in the direction of an arrow — one line of a text document, for instance. Clicks in page regions usually cause the screen to scroll one screenful in the direction relative to the thumb. The size of the thumb represents how much of the entire document the content region of the window is showing (see Chapter 10 for more details).

Each control type or part of complex controls in scroll boxes is known to the Control Manager by a *part code*. For example, if an application features a scroll bar inside the content window (separate from scroll bars on the window frame), the Control Manager will let the program know over which part of the control a mouse-down event occurred. The part codes are as follows:

Part No.	*Control or Part*
0	No part
1	Reserved for internal use
2	Simple button
3	Check box
4	Radio button
5	Dial up arrow
6	Dial down arrow
7	Dial page up
8	Dial page down
9	Reserved for internal use
10	Grow box icon
11–31	Reserved for internal use
32–127	Reserved for application
128	Reserved for internal use
129	Dial thumb
130–159	Reserved for internal use
160–253	Reserved for application
254–255	Reserved for internal use

As you might guess from the large number of part codes that are reserved for applications, the Control Manager allows you to design your own controls. This isn't something you're likely to do in one of your first programs, but the capability is there when you're ready. You'll be able to design rather complex controls, perhaps with several unique parts.

CONTROL RECORDS

While a control record functions much like any other toolbox record, it differs in that some of the items in the record mean different things depending on the type of control the record describes. A button control, for example, needs a place for the text that appears inside the button; a scroll bar has no text, but must contain information about the data area size and content region size so that the Control Manager can draw a properly proportioned thumb.

It may be convenient to think of all control records as having the same "front end" to their records — specifications they all share in common. This front end can be characterized by the following generic control record:

CtlNext: LONG	Handle to next control (0 = last control)
CtlOwner: LONG	Pointer to window holding the control
CtlRect: RECT	Local coordinates of enclosing rectangle
CtlFlag: BYTE	Control specifications flag: bit 7 0 = visible; 1 = invisible bits 6-0 control definitions
CtlHilite: BYTE	Highlighted part 0 = no part highlighted 255 = control inactive
CtlValue: WORD	Current value
CtlProc: LONG	Address of control's definition procedure
CtlAction: LONG	Address of control's default action procedure
CtlData: LONG	Data requested by definition procedure
CtlRefCon: LONG	Reference constant assigned by application
CtlColor: LONG	Pointer to control's color table

To demonstrate how the different built-in controls affect the contents of a control record, we'll present the records for each of the four standard controls. Watch how record items CtlFlag, CtlValue, CtlData, and CtlColor change from one record type to another.

First, here is the control record for a simple button:

CtlNext: LONG	Handle to next control (0 = last control)
CtlOwner: LONG	Pointer to window holding the control
CtlRect: RECT	Button's RECT

CtlFlag: WORD	Control specifications flag: bits 7-2 not used bits 1-0 0 = single round corner outline 1 = bold round corner outline 2 = single square corner outline 3 = single square corner outline with drop shadow
CtlHilite: WORD	Highlighted part 0 = no part highlighted 2 = button highlighted 255 = control inactive
CtlValue: WORD	Always 0
CtlProc: LONG	$00000000
CtlData: LONG	Pointer to title string
CtlRefCon: LONG	Reference constant assigned by application
CtlColor: LONG	Pointer to control's color table color 1 = outline color when normal color 2 = interior color when normal color 3 = interior color when selected color 4 = text color when normal color 5 = text color when selected color 6 = special highlight color color 7 = thick outline color

Bit 0 of the CtlFlag, when set to 1, has the Control Manager draw a heavy line around the button. This, in line with the User Interface Guidelines, should indicate to the user that a press of the Return key will be the same as clicking that button with the mouse button. This flag setting affects only the drawing of the button on the screen. Your application will have to handle the Return key key-down event separately, passing that event along to the control action that would occur when the button is clicked by the mouse.

Defining the button's RECT, as in the CtlRect record item, must take into account the length of the text string that will be the title of the button. If CtlRect is not wide enough for all of the button's title, the text will be cut short.

The color table referred to in CtlColor is a component color table like the Window Manager's color table. The color numbers you apply to each component refer to colors in QuickDraw's standard color table. A 0 in this record item will cause the default color table to be applied to the control.

Next comes the check box control record:

CtlNext: LONG	Handle to next control (0 = last control)
CtlOwner: LONG	Pointer to window holding the control
CtlRect: RECT	RECT of box and title

CtlFlag: WORD	Control specifications flag: bit 7 0 = visible; 1 = invisible bits 6-0 not used
CtlHilite: WORD	Highlighted part 0 = no part highlighted 3 = check box highlighted 255 = control inactive
CtlValue: WORD	0 = not checked; nonzero = checked
CtlProc: LONG	$02000000
CtlData: LONG	Pointer to title string
CtlRefCon: LONG	Reference constant assigned by application
CtlColor: LONG	Pointer to control's color table color 1 = outline color when normal color 2 = interior color when normal color 3 = interior color when selected color 4 = text color color 5 = color of the X

Notice that the CtlRect is a rectangle that includes the space for the control's text area. That means that users don't have to be so precise in their mouse pointing to check or uncheck a text box. A click anywhere in the box or on the title will call the Control Manager.

Similar to the check box record is the radio button control record.

CtlNext: LONG	Handle to next control (0 = last control)
CtlOwner: LONG	Pointer to window holding the control
CtlRect: RECT	RECT of button and title
CtlFlag: WORD	Control specifications flag: bit 7 0 = visible; 1 = invisible bits 6-0 family number
CtlHilite: WORD	Highlighted part 0 = no part highlighted 4 = radio button highlighted 255 = control inactive
CtlValue: WORD	0 = off; nonzero = on
CtlProc: LONG	$04000000
CtlData: LONG	Pointer to title string
CtlRefCon: LONG	Reference constant assigned by application
CtlColor: LONG	Pointer to control's color table color 1 = outline color when normal color 2 = interior color when normal color 3 = interior color when selected color 4 = text color color 5 = color of the dot

We get a big difference, however, in the scroll bar control record.

CtlNext: LONG	Handle to next control (0 = last control)
CtlOwner: LONG	Pointer to window holding the control
CtlRect: RECT	RECT of entire scroll bar
CtlFlag: WORD	Control specifications flag: bit 7 0 = visible; 1 = invisible bits 6-5 not used bit 4 0 = vertical scroll bar 1 = horizontal scroll bar bit 3 1 = right arrow bit 2 1 = left arrow bit 1 1 = down arrow bit 0 1 = up arrow
CtlHilite: WORD	Highlighted part 0 = no part highlighted 5 = up arrow highlighted 6 = down arrow highlighted 255 = control inactive
CtlValue: WORD	Current value, between 0 and data size minus view size
CtlProc: LONG	$06000000
CtlData: LONG	Low-order WORD = view size High-order WORD = data size
CtlRefCon: LONG	Reference constant assigned by application
CtlColor: LONG	Pointer to control's color table color 1 = outline color color 2 = arrow color when normal color 3 = arrow color when selected color 4 = arrow box interior color color 5 = thumb interior color when normal color 6 = thumb interior color when selected color 7 = page region color color 8 = inactive color
Thumb: RECT	Thumb's rectangle
PageRegion: RECT	Rectangle of thumb's slider region

This record appends additional record items that hold the size of the thumb and the extent of the area the thumb can be dragged in.

CONTROLS AND EVENTS

Since TaskMaster automatically handles mouse-down events in window frame controls (i.e., window scroll bars), we'll concentrate here on the steps required to link the Event Manager, the Control Manager, and your application together when the user clicks on a control in a window's content region.

Assuming the program is polling the Event Manager with TaskMaster (which calls GetNextEvent), TaskMaster will report a mouse-down event (in the "what" field of the task record) as well as the pointer to the window in which the event occurred (in the message field of the task record). With this information you can call FindControl.

Among the input parameters passed to FindControl are the coordinates of the mouse pointer at the time of the mouse-down event. FindControl sets the whichControl parameter to the control handle. Then call the TrackControl tool, which works very much like MenuSelect. It takes control of the computer as long as the mouse button is still pressed. For example, in a scroll bar, TrackControl automatically lets you drag an outline of the thumb up and down the control bar, drawing it in the desired place when you release the mouse button. When the mouse button is released, TrackControl passes an output parameter specifying the code number of the part just chosen by the user. Armed with this information, your application can execute a CASE structure to test for the control and part number, branching to your predefined routines in response to the actions indicated by the control settings.

TrackControl does one more important task for scroll bars. When you drag the thumb, TrackControl not only handles the display part of the scroll bar, but it also updates the control value (CtlValue) in the control's record, relative to the minimum and maximum settings you assign to the scroll bar. This action does not do the actual scrolling of the document or adjust the display of a value controlled by the scroll bar — your application must read what the new value is (GetCtlValue) and update either the scrolling or on-screen value as adjusted by the drag of the thumb.

Resizing a window that displays a control in its content region puts a little burden on your program to resize the control as well. Unlike the Window Manager, which automatically resizes its frame-based scroll bars when the window grows or shrinks, the Control Manager does not keep as watchful an eye on controls in the content region. If the window with such a control is growable, your program may have to do some fancy calculations about the new size of a window to both move and size the control so that it fits in the new window size. Consequently, you may wish to limit content region controls to windows that do not include grow and zoom boxes.

A good strategy to learn the ins and outs of the Control Manager is to work with controls managed by other managers, notably the Window Manager for scroll bars and the Dialog Manager for simple buttons. Once you understand the basics, work your way up to radio buttons, check boxes, and free-standing scroll bars.

CHAPTER 13

Where Do We Go from Here?

If you've been following Parts Two and Three of this book from beginning to end, you'll realize that while we have covered a great deal of ground, there are still several tool sets we haven't discussed, such as text, dialog, and sound tools. But we have shown you enough to start thinking about how an Apple IIGS application works from the programmer's view, and also how an application should work from the user's point of view.

WHERE WE ARE

It should be quite clear by now that programming with the toolbox is more difficult than programming on the old Apple II ever was. The difficulty comes not from the complexity of the tools — they make a number of tasks rather easy — but from the much higher level of human-to-machine interactivity of today's software design. Sure, you can still write programs for the IIGS in emulation mode, or even in native mode without all the fancy menus, windows, and controls. But that wouldn't be striving for the state of the programming art.

Commercial software reaching store shelves today is far more sophisticated in user interface and raw computing power than it was only a couple of years ago. Even public domain and inexpensive "user-supported" software is outclassing expensive commercial products of a few years ago.

Everyone's expectations about what a computer program should do and look like are rising quickly. Computer users won't have the patience for old-fashioned, unfriendly-looking programs.

The ante is higher. Your programming skills must match it if you want to get into the game.

In this book, we've stripped the Apple IIGS toolbox of the complexities that frighten away first-timers. We reached the heart of key issues about program structure and accessing the tools from a programming language. But this is only one part of the learning process.

NEXT

The next step is to dig into the programming language of your choice. Since you now know where you're heading, it should be easier to think ahead to applying acquired language skills to actual applications. As you're learning the language or relearning the IIGS version of an old favorite, you won't feel as if you're learning language concepts in a total vacuum, even though the language's manuals may not be able to put it all together for you until the very end. You will be approaching the language with a well-defined sense of purpose.

Every language will come with at least one sample program, hopefully one that provides sample source code examples for the major IIGS tool sets already tested and running. Study these samples. Make sure you comprehend the structure of the entire source code listing, paying particular attention to the way parameters are passed to tools and how their output parameters are retrieved for later use.

After you've mastered the sample program, make a copy of the source code and start modifying parameters one at a time. Assemble or compile the listing to see how the parameter change affected the program. Experiment with only one or two parameters at first so you can see clearly whether your anticipated changes actually occurred in the compiled version. Do obvious things first, such as changing the size of a newly created window or removing a zoom box from a window frame definition.

The real test of your understanding of the language and the toolbox will be to recreate on your own a program like the sample program. Type in an entire source code listing from scratch. Decide what you want it to do — perhaps a variation of the sample program. Then write the program's source code. Compile it, link it, and if there were no errors in the compilation or linkage stages, run the program. Even if errors pop up, don't despair. Errors are simply reminders that you forgot something that the language needed.

By the time you code your first solo program of any length, you should begin to feel comfortable with programming in this new and perhaps strange

environment. Before heading out on your own, gather a library of reference books, especially those from Apple listed in Appendix C. The *Toolbox Reference* volumes, in particular, will prove to be invaluable sources for tool call information.

Also try to locate a local Apple user group, if you don't already belong to one. Often all it takes is a brief conversation with someone else who may have struggled with and solved the same problem you're having. There is so much to learn about the toolbox, that you may know parts of it better than others in the group. They, in turn, may know more about other areas that can help fill the gaps in your knowledge. User groups were at the foundation of the personal computer industry, where enthusiasts could share accomplishments and problems with each other. The viability of the user group as a support mechanism for programmers is very much alive today. Take advantage of it if you can.

TRAPS

A potential pitfall for programmers meeting the event-driven and toolbox world for the first time is that it is easy to focus on the wrong end of the programming task first. You've seen and perhaps marveled at the basic structure of a toolbox program and can't wait to apply it to a program of some kind. But that may be the wrong approach. Instead of looking at the tools and asking, "What can I build with these?" forget about the tools. Instead, look at the user interface, as exemplified in the ProDOS 16 Finder and any Macintosh application and ask, "What do I want my IIGS to do?"

Dream about the application first — what real-world situation it may simulate, how the mouse and keyboard input will flow smoothly for the user, how it can make a normally dull task fun — *then* figure out what tools you'll use to accomplish those goals. If you begin by focusing on the tools, you run the risk of constricting creative program design ideas to your current level of expertise with the tools. By imagining creative applications, you will force yourself to dig deeper into the toolbox to uncover tools you might otherwise overlook. That's an excellent way to grow as an Apple IIGS programmer.

Appendixes

APPENDIX A

A Short Course in Hexadecimal and Binary Math

Our first counting encounters as children were in the decimal numbering system. That may be fine for humans, but computers at their lowest levels respond to binary representations of the signals running through the microchips. If, as programmers, we are to communicate effectively with the computer, we will have to learn its way of counting. Therefore, it is vital to understand not only the binary numbering system, but a more convenient system for conveying binary information: the hexadecimal numbering system.

BACK TO SCHOOL

As children, we learned first to count from one to ten. When we reached ten, numbers increased from single to dual digits, and the rightmost digits started counting over again, from zero to nine. We soon learned that numbers, as we knew them, had columns that represented ones, tens, hundreds, thousands, and so on. If we were looking at a three-digit number, and a 7 was in the leftmost column, that meant that the number would be at least 700. Each additional digit to the left indicated another factor of ten had been added to the number. Therefore, a 7 in the fourth column meant that the number

would be ten times 700, or 7000. This dependence on the factor of ten is the reason for the numbering system's name, decimal ("deci" is derived from "deka," the Greek word for "ten"). We also call it *base 10*.

From Deci to Hexadeci

Now imagine a numbering system that allows you to signify more than ten numbers in a single digit. In such a system, you start counting up from zero as in the decimal system. But when you reach nine, the next number won't leap to a two-digit number. Instead, letters, signifying the values equivalent to ten and up, appear in the single column number. After nine, therefore, would come the letter A. Following A comes B, and so on until you reach the letter F, which is equivalent to the decimal number fifteen. The next number in this new series then becomes 10 — not the ten we're accustomed to seeing, but the value one-zero in the *hexadecimal* numbering system. "Hexadeci" is also derived from the Greek, this time for the number sixteen. In other words, instead of columns labeled "ones, tens, hundreds," and so on, they are labeled "ones, sixteens, two-hundred-fifty-sixes," and so on. Each column increases by a factor of sixteen, and values in this system are called *base 16* values.

And Then to Binary

At the other extreme is the binary numbering system, which has at most, one of two different numbers — zero and one — in any column. Columns increase to the left by a factor of two, making the column labels "ones, twos, fours, eights, sixteens, thirty-twos," and so on. As you might guess, we're in *base 2* territory.

Comparing binary to hexadecimal numbers, it takes a five-digit binary number to indicate the value for sixteen (10000), but only a two-digit number in hexadecimal (F0).

SIGNIFICANCE

All three numbering systems share a feature that is hardly apparent until you begin working with computer data. A digit at the rightmost end of a number is said to be the *least significant digit* because its impact on a number's value is usually small compared against those to the left. The digit at the far left of the number (regardless of its length) is called the *most significant digit*. A change to the most significant digit has the greatest effect on the value of a number.

For example, in the decimal number 1005, the most significant digit is the 1, standing in the thousands column, while the least significant digit is the 5 in the ones column. If you increase the 1 by only 1, you nearly double the value of the number; if you raise the 5 by 4 to a 9, the value increases only slightly.

In computer numbers, which are predominantly hex and binary numbers, bytes of data are often grouped together as *integers* (2 bytes), *long integers* (4 bytes), and others (1 byte is represented by 8 binary digits or its corresponding 2-digit hexadecimal number). When a number represents the data in a 2-byte integer, one byte will be the least significant byte, and the other the most significant. The order of these bytes is the same as the numbers we just discussed. Therefore, in the hex integer $A478, $A4 is the most significant byte, $78 the least significant byte.

When describing data in multiple-byte structures, such as integers and long integers, it is very important to respect all significant digits that the computer expects. For example, if the data consists of 12 bits of information (in the form of three 4-bit binary groups), you must specify the number with a leading zero in its hexadecimal representation. Therefore, if the information component of an integer can be represented by 4A3, the data must be entered in the program as 04A3, the leading zero satisfying the computer's need for a full integer. In other words, all four hex digits are significant in an integer data structure.

NOTATIONS

When discussing numbers from more than one numbering system, things can get a bit confusing, especially when decimal and hexadecimal numbers are tossed about in the same sentence. To help reduce confusion, several notation conventions are in common use today.

The notation you'll see most of the time is one that distinguishes a hexadecimal number from other numbers. A hexadecimal number will be preceded by a dollar sign ($), as in $20.

Binary numbers do not have a preceding character like the hexadecimal's dollar sign. But in computers, binary numbers are usually grouped in eights (or sometimes fours), and are generally recognizable by their lack of digits other than 1s and 0s.

Another convention you may see (although not in this book) is to explicitly note the number base of the number in a parenthetical subscript after the number. Therefore, the hex number $20 may also be written 20_{16}. If you use this notation, it is assumed that you will use it for all numbers, including decimal numbers.

HEX AND BINARY RELATIONSHIPS

Study the following table of decimal, hex, and binary numbers.

Decimal	*Binary*	*Hexadecimal*
0	0000 0000	0
1	0000 0001	1
2	0000 0010	2
3	0000 0011	3
4	0000 0100	4
5	0000 0101	5
6	0000 0110	6
7	0000 0111	7
8	0000 1000	8
9	0000 1001	9
10	0000 1010	A
11	0000 1011	B
12	0000 1100	C
13	0000 1101	D
14	0000 1110	E
15	0000 1111	F
16	0001 0000	10
17	0001 0001	11
18	0001 0010	12
19	0001 0011	13
20	0001 0100	14

Notice in particular the relationship between hexadecimal and binary numbers at the critical juncture where single-digit hex numbers become two-digit numbers. This relationship will help you to convert binary numbers to hexadecimal and vice versa.

CONVERSIONS

Programming a computer in assembly language or a high-level language such as C and Pascal often requires the conversion of numbers from one base to another. Hex and binary numbers are relatively simple owing to their special arithmetic relationships. Decimal numbers, however, are not so easy.

Hex-to-Binary and Back

Because a single hex digit represents the values zero to fifteen, and since those same values are encompassed in the binary numbers 0000 to 1111, you can convert hex to binary and vice versa by simply dividing a long number into four-digit binary or single-digit hex numbers and then stringing the resulting conversions together.

Using the table presented above, you can look up a binary nibble or individual hex number to find the corresponding conversion. Let's say you wish to convert the binary number 0011 1101 into hexadecimal. First look up the most significant nibble in the table, where you'll find the hex number $3. Binary 1101 corresponds to $D. Joining these two hex digits in the same order as their binary counterparts produces the hex value $3D.

Conversely, if the hex number is $1A0B, look up each single hex digit in the table and reassemble the binary equivalents in the same order: 0001 1010 0000 1011.

Decimal Conversions

Converting numbers to or from decimal is more difficult, because neither binary nor hex numbering systems have much in common with the decimal system to facilitate conversion. While manual conversion is possible, most programmers invest in programmers' calculators that do conversions at the press of a few keys.

Two popular models near most programmers' sides are the Texas Instruments TI-Programmer and the Hewlett-Packard HP-16C. Both models were designed for programmers and the kinds of math or conversion problems they encounter. Between the two models, the Texas Instruments is easier to use because its features have been pared down to bare essentials. Binary-to-hex conversions have to be done somewhat manually, since the calculator display does not show binary numbers. On the calculator's keypad overlay are printed the four-digit binary equivalents of the sixteen hex digits. To convert from hex to binary, you essentially use the keypad as a lookup table, like the table presented above.

The Hewlett-Packard model is more expensive and is loaded with features you may not need, at least in your early programming days. This calculator, however, does display binary numbers, and provides a respectible amount of programming ability itself. It's not the kind of calculator you're likely to pick up and start using without studying the manual a bit, but it does offer a great amount of flexibility in its binary math functions.

BOOLEAN ARITHMETIC

Assembly language programmers in particular will be faced with performing many tests inside their programs for the presence of a particular number in one of the microprocessor's registers. Since these numbers are stored in the chip in binary format, the tests are performed in binary.

The common way to test the presence or absence of a particular bit in a long series of bits (in a byte or two, for instance) is to perform *Boolean* algebra on the entire string of bits. Based on fundamental work performed by English mathematician George Boole (1815–1864), Boolean algebra as used today in computer math is a convenient way to dissect the bit contents of a byte or more of data.

Boolean math looks upon the 1 and 0 not as numeric values, but as equivalent to logical TRUE and FALSE, respectively. Therefore, in comparing the third digits of two binary numbers, the result of Boolean addition is 1 only if the digits in both numbers are also 1. In other words, the result is TRUE if both numbers show a TRUE in those slots. This operation is usually called an AND operation: TRUE AND TRUE yields a TRUE; TRUE AND FALSE yields a FALSE, because one of the two values was not TRUE.

```
      TRUE                 TRUE
AND   TRUE           AND   FALSE
-----------          -----------
TRUE                 FALSE
```

As applied to a long binary number, Boolean logic works one bit at a time. The result of an AND is not the sum of the two binary numbers, but the logical result of their respective TRUE and FALSE values. For example, if we have a binary number consisting of all 1s (1111 1111), we can test for the presence of a 1 in the least significant digit by performing an AND operation with the number 0000 0001. The result is 0000 0001, because the only location that is TRUE on both binary numbers is the least significant digit.

```
      1111 1111
AND   0000 0001
---------------
0000 0001
```

We can use a Boolean to test the presence of more than one bit at a time. For instance, in the following example, three bits are tested, only two of which test as being present in the topmost number.

```
      1011 0010
AND   1001 1000
-------------
      1001 0000
```

Now, just because the Boolean tests are performed on the binary numbers, that doesn't mean that your program is limited to using the binary figures. In the above example, you can just as easily write the problem using hexadecimal numbers.

```
      $B2
AND   $98
-------
      $90
```

The microprocessor is still doing the Boolean in binary. But by doing the work in hexadecimal, you are relieved from risking errors in typing correct binary numbers.

Boolean Variables

You will also encounter a variable type called a Boolean. This is not strictly connected with Boolean algebra. In the case of a Boolean variable, the computer is looking for (or perhaps sending back to your program) a value corresponding either to a TRUE or a FALSE. FALSE is always represented by a 0. That 0 may be in the form of a full 2-byte integer ($00), so be sure to include all significant digits. Computers usually recognize a TRUE as any nonzero number. It could be a simple $01 or, more often, the hex value $FF.

APPENDIX B

ASCII Table

While American Standard for Information Interchange (ASCII) codes are understood by all personal computers when they communicate with each other, some computers, notably the Apple II series, store characters internally using a slightly different version of the ASCII code. The order of characters is the same for both versions, but the actual code numbers are different.

When you place text characters in your program, you usually do so with the actual letters and numbers, rather than their ASCII values. You should have reference to these values, however, in case your program needs to send commands to a printer or when you debug your program.

The following table presents both standard and Apple II ASCII codes. Decimal and hexadecimal values for both types appear in their respective columns.

	ASCII		Apple ASCII	
Character	Decimal	Hexa-decimal	Decimal	Hexa-decimal
NUL	0	00	128	80
SOH	1	01	129	81
STX	2	02	130	82
ETX	3	03	131	83
EOT	4	04	132	84
ENQ	5	05	133	85
ACK	6	06	134	86
BEL	7	07	135	87

	ASCII		Apple ASCII	
Character	Decimal	Hexa-decimal	Decimal	Hexa-decimal
BS	8	08	136	88
HT	9	09	137	89
LF	10	0A	138	8A
VT	11	0B	139	8B
FF	12	0C	140	8C
CR	13	0D	141	8D
SO	14	0E	142	8E
SI	15	0F	143	8F
DLE	16	10	144	90
DC1	17	11	145	91
DC2	18	12	146	92
DC3	19	13	147	93
DC4	20	14	148	94
NAK	21	15	149	95
SYN	22	16	150	96
ETB	23	17	151	97
CAN	24	18	152	98
EM	25	19	153	99
SUB	26	1A	154	9A
ESC	27	1B	155	9B
FS	28	1C	156	9C
GS	29	1D	157	9D
RS	30	1E	158	9E
US	31	1F	159	9F
Space	32	20	160	A0
!	33	21	161	A1
"	34	22	162	A2
#	35	23	163	A3
$	36	24	164	A4
%	37	25	165	A5
&	38	26	166	A6
'	39	27	167	A7
(	40	28	168	A8
)	41	29	169	A9
*	42	2A	170	AA
+	43	2B	171	AB
,	44	2C	172	AC
-	45	2D	173	AD
.	46	2E	174	AE
/	47	2F	175	AF

Character	ASCII Decimal	ASCII Hexa-decimal	Apple ASCII Decimal	Apple ASCII Hexa-decimal
0	48	30	176	B0
1	49	31	177	B1
2	50	32	178	B2
3	51	33	179	B3
4	52	34	180	B4
5	53	35	181	B5
6	54	36	182	B6
7	55	37	183	B7
8	56	38	184	B8
9	57	39	185	B9
:	58	3A	186	BA
;	59	3B	187	BB
<	60	3C	188	BC
=	61	3D	189	BD
>	62	3E	190	BE
?	63	3F	191	BF
@	64	40	192	C0
A	65	41	193	C1
B	66	42	194	C2
C	67	43	195	C3
D	68	44	196	C4
E	69	45	197	C5
F	70	46	198	C6
G	71	47	199	C7
H	72	48	200	C8
I	73	49	201	C9
J	74	4A	202	CA
K	75	4B	203	CB
L	76	4C	204	CC
M	77	4D	205	CD
N	78	4E	206	CE
O	79	4F	207	CF
P	80	50	208	D0
Q	81	51	209	D1
R	82	52	210	D2
S	83	53	211	D3
T	84	54	212	D4
U	85	55	213	D5
V	86	56	214	D6
W	87	57	215	D7

	ASCII		Apple ASCII	
Character	Decimal	Hexa-decimal	Decimal	Hexa-decimal
X	88	58	216	D8
Y	89	59	217	D9
Z	90	5A	218	DA
[	91	5B	219	DB
\	92	5C	220	DC
]	93	5D	221	DD
^	94	5E	222	DE
_	95	5F	223	DF
`	96	60	224	E0
a	97	61	225	E1
b	98	62	226	E2
c	99	63	227	E3
d	100	64	228	E4
e	101	65	229	E5
f	102	66	230	E6
g	103	67	231	E7
h	104	68	232	E8
i	105	69	233	E9
j	106	6A	234	EA
k	107	6B	235	EB
l	108	6C	236	EC
m	109	6D	237	ED
n	110	6E	238	EE
o	111	6F	239	EF
p	112	70	240	F0
q	113	71	241	F1
r	114	72	242	F2
s	115	73	243	F3
t	116	74	244	F4
u	117	75	245	F5
v	118	76	246	F6
w	119	77	247	F7
x	120	78	248	F8
y	121	79	249	F9
z	122	7A	250	FA
{	123	7B	251	FB
¦	124	7C	252	FC
}	125	7D	253	FD
~	126	7E	254	FE
DELETE	127	7F	255	FF

At the beginning of the table are thirty-two special characters, which owe their heritage to the old clackety teletype machine days. A number of these signals are still in use today for controlling printers and modems and for other external communications jobs that personal computers perform. Only a handful of them are commonly used, but we provide a list of all their meanings.

Character	Meaning
NUL	null character
SOH	start of heading
STX	start of text
ETX	end of text
EOT	end of transmission
ENQ	inquiry
ACK	acknowledge
BEL	bell (beep)
BS	backspace
HT	horizontal tab
LF	line feed
VT	vertical tab
FF	form feed
CR	carriage return
SO	shift out
SI	shift in
DLE	data link escape
DC1	device control 1
DC2	device control 2
DC3	device control 3
DC4	device control 4
NAK	negative acknowledge
SYN	synchronous idle
ETB	end of transmission block
CAN	cancel
EM	end of medium
SUB	substitute
ESC	escape
FS	file separator
GS	group separator
RS	record separator
US	unit separator

APPENDIX C

For Further Reading

Apple Computer produces a series of programmer's guides to toolbox programming on the Apple IIGS. A serious programmer will have most of these volumes at hand when designing applications programs. All volumes are published by Addison-Wesley. The exact publishing schedule of the series was not available as this book goes to press.

Technical Introduction to the Apple IIGS
Programmer's Introduction to the Apple IIGS
Apple IIGS Hardware Reference
Apple IIGS Firmware Reference
Apple IIGS Toolbox Reference: Volume I
Apple IIGs Toolbox Reference: Volume II
Apple IIGs ProDOS 16 Reference

If you are just starting to build your reference library, then start with *Programmer's Introduction* and the two *Toolbox Reference* volumes. These will carry on from the knowledge gained in this book.

Additionally, Apple will be publishing Programmer's Workshops for assembly language, C, and Pascal. These are designed primarily for serious program developers, and include editor, compiler (or assembler), and linker, as well as macro libraries that allow you make calls to the entire

toolbox from any of those languages. All three workshop compilers produce object modules in the standard format that facilitates linking modules from two or three languages into a single load file. Contact Apple Computer directly for more information about the Apple IIGS Programmer's Workshops.

Apple Computer Inc.
20525 Mariani Ave.
Cupertino, CA 95014

Glossary

activate event: A special type of event that signals your program that a particular window has been made the active window.

active window: The window on the desktop into which typed text or mouse-controlled drawing will appear — usually denoted by horizontal lines in the title bar and active scroll bars if so equipped.

alert window: A window drawn by the Menu Manager with a double border and generated primarily by the Dialog Manager to request further input from the user or to signal an error condition.

allocate: To reserve an area of memory for a collection of data.

application event: An event type reserved for use by programmers building applications with nonstandard events.

ASCII: The American Standard Code for Information Interchange, a table of values assigned to each letter, numeral, punctuation mark, and certain control characters.

auto-key event: An event signifying that the user has pressed a key and held it down until the Repeat Delay time has expired, causing the key to type multiple characters.

base: The numbering system (e.g., binary, decimal, hexadecimal) to which a number belongs; in math, signified as a subscript in parentheses, such as 4333(16); in computers, generally signified by the construction of the number, such as two groups of four binary numbers, a hexadecimal preceded by a dollar sign ($), and no extra markings for a decimal number.

bit: the smallest unit of information inside a computer, commonly signified with either a 1 or 0.

bit map: Macintosh terminology for a pixel map.

boundary rectangle: The rectangular coordinates that both define the extent of a pixel image and impose a coordinate system on the image.

button: A control whose click by the mouse pointer usually produces immediate action.

byte: A group of 8 bits in memory.

check box: A control consisting of a small square with text (its title) to its immediate right; clicking the mouse pointer on this control causes an "X" to fill the box, meaning that the particular feature has been selected.

choose: To indicate a particular option on a pull-down menu.

classic desk accessory (CDA): a desk accessory that can run either in native mode or emulation mode.

clip: To restrict any drawing operation to fall inside a particular boundary; any drawing outside that boundary is not displayed.

clipping region: An area on the screen to which any drawing operation will be clipped.

close box: A small box at the left edge of a document window's title bar; clicking the mouse pointer here should remove the window from view.

close region: The area in the window frame that is to be clicked by the mouse to remove a window from view; in standard document windows, the region is inside the outline of the close box.

color table: One of a possible sixteen lookup tables in memory that lists four (640 × 200 mode) or 16 (320 × 200 mode) color values, which are accessed by their respective number down the table.

color value: A 12-bit designation (in a 16-bit integer) of the intensity of red, green, or blue in a particular color in the color table; a color value of $0F00 indicates a maximum of red and absence of green or blue.

compaction: The act of squeezing together all segments of movable data in memory to make room for additional segments.

content region: The area of a window in which drawing actions (graphics and text) take place.

control: A screen object whose manipulation by the mouse influences the display of information in a window or operation of the program with respect to that window.

control record: A data structure consisting of specifications for a control.

cursor: A small iconic screen image appearing in black only that shows the user where the mouse pointer is located; the image can change in different regions of the screen.

default: The setting of a parameter or series of parameters that the Toolbox makes unless other values are specifically defined.

desk accessory: A small applications program whose window overlaps the underlying application's window; can be either a classic or new desk accessory.

desktop: The background of the active screen area upon which are overlaid the menu bar, windows, icons, and other objects.

dial: A control, often designed as an onscreen metaphor for an analog control from the real world, such as a slider or meter dial; used to make quantitative adjustments in a program.

dialog box: A window of the alert type that prompts the user for additional information before proceding with program execution.

disabled: A menu title, menu item, or control that is not functional at a particular stage in the program.

DOC: The Digital Oscillator Chip, created by Ensoniq, at the core of the Apple IIGS sound circuitry.

document window: The standard window type drawn with a single-pixel-wide outline; a title bar, scroll bars, and other elements may be added as desired.

double click: Two clicks of the mouse in rapid succession.

drag: To reposition an object on the screen by placing the mouse pointer on that object, pressing and holding down the mouse button, rolling the mouse around its work surface, and releasing the mouse button to plant the screen object in its new position.

drag region: A region in a window (usually on the title bar) in which the mouse pointer must be placed before the user can drag the window.

emulation mode: The operational mode in which the Apple IIGS behaves likes an enhanced Apple IIe or Apple IIc.

enabled: A menu title, menu item, or control that responds to mouse action.

event: The report of an occurrence, such as a press of a keyboard key or mouse button, that a program uses to branch to an appropriate series of action instructions to effect a response to that occurrence.

event mask: An integer that filters certain kinds of events from reporting their occurrence to a program.

event message: A part of the event record that contains additional information regarding an event.

event queue: A section of memory devoted to temporarily storing events in a first-in, first-out order until the program fetches them.

event record: A data structure containing specifications about an event.

event type: An identifying code specifying the nature of the event occurrence.

fill pattern: An 8-by-8-pixel image repeated over and over that can be used to color some or all of a grafport.

Finder: The toolbox-based operating system extension of ProDOS 16 that facilitates file operations by the use of icons and pull-down menus.

global coordinates: The coordinate system assigned to the visible display area of an Apple IIGS screen when programming with the toolbox; the top left corner is assigned the point (0,0).

GLU: An integrated circuit that performs many miscellaneous functions on a circuit board, acting as the "glue" that ties other major chips together.

grafport: A drawing environment consisting of a coordinate system and many specifications (e.g., pen size, background color, and text font) managed by QuickDraw.

handle: A pointer to a master pointer, which, in turn, points to a place in memory that may move during execution of a program.

high-level language: A programming language, such as C and Pascal, that generally insulates the programmer from the computer's architecture.

highlight: To display a menu title, menu item, or control in a color opposite its normal color to indicate that it is selected.

hot spot: The single coordinate point in a cursor image that coincides with the location of the mouse pointer on the screen.

K: The abbreviation for kilobyte and kilobit.

key-down event: An event signifying that a keyboard key has been pressed.

kilobit: A unit of measure for memory chips, equaling 1024 bits.

kilobyte: A unit of measure of computer memory and disk drive capacity, equaling 1024 bytes.

library: A collection of prewritten routines that can be merged into a program.

local coordinates: A coordinate system imposed on a window by QuickDraw II, totally independent from the global coordinates of the screen; local coordinates do not change as the window is dragged across the screen.

long integer: A data type available in most programming languages consisting of 4 bytes (32 bits) of information.

mark: A character that can be programmed to appear next to a menu item that turns on a mode or other operation.

master pointer: A pointer, in a fixed memory location, that keeps track of movable blocks of memory.

megabyte: A unit of computer memory or disk drive capacity equaling 1,048,576 bytes.

menu: A list of options from which the user can choose.

menu bar: An area extending across the top of the screen containing the titles of available menus.

menu item: A single choice within a list of choices in a pull-down menu.

menu string: A text string in memory containing information about titles, items, and item modifiers for a given menu bar.

menu title: A word or color patch in a menu bar indicating that a menu of related items can be pulled down with the mouse.

modifier key: One of several noncharacter keys on the keyboard that influences the meaning of a character key when both are pressed simultaneously.

mouse-down event: A report that the user has pressed the mouse button.

mouse-up event: A report that the user has released the mouse button.

native mode: The operational mode of the 65816 microprocessor in which the chip manages information internally in 16-bit wide paths.

new desk accessory (NDA): A desk accessory program that can function only atop native mode programs.

null event: An event signifying no event has taken place.

object module: A disk file containing the compiled version of a program.

owning window: The window in which a control is drawn.

part code: An integer signifying a component of a control.

pen state: A list of specifications about the pen in a grafport; includes information about the pen's coordinate location, size, pattern, and transfer mode.

pixel: A single dot on the video monitor.

pixel image: A graphics picture consisting of a rectangular grid of colored pixels.

pixel value: the 2-bit (in 640 mode) or 4-bit (in 320 mode) representation of a pixel's color on the screen.

point: A location in a QuickDraw II coordinate plane signified by a horizontal and vertical coordinate.

pointer: A data type that holds the address of a location in memory.

polygon: A shape defined by enclosing straight lines.

pop: To remove an item from a stack.

port: In QuickDraw II, short for grafport.

port rectangle: A rectangle defining what portion of a pixel image the grafport may draw into.

push: To add an item to a stack.

radio button: A control, usually in groups of two or more, consisting of a small circle and text running to its right; clicking on this control usually de-selects others in the group.

RAM: The acronym for random access memory.

RAM tools: The toolbox routines contained on the ProDOS system disk and loaded into RAM when needed.

random access memory: The type of memory inside a computer, more accurately called read/write memory, that allows information to be written to it and read from it.

read-only memory: The type of memory that can only be read; the Apple IIGS ROM contains many of the toolbox routines.

rectangle: A shape definition as defined by two coordinates, the upper left and lower right corners of the area.

region: An area in a grafport of any shape or of multiple shapes.

register: A temporary storage area inside a microprocessor; some registers have only one function, while others are general-purpose.

relocatable: The Macintosh equivalent of movable.

ROM: The acronym for read-only memory.

ROM tools: The toolbox routines embedded into the ROM chip.

SANE: The Standard Apple Numeric Environment, a preprogrammed arithmetic environment built into the toolbox.

65816: The part number of the microprocessor chip at the core of the Apple IIGS.

size box: A small box at the lower right corner of a document window that can be dragged to adjust the size of the window.

size region: The area of a document window frame inside a size box that responds to the dragging of the mouse for adjusting the size of a window.

stack: An area in memory that is used as temporary storage space for information that must be passed to and from toolbox routines.

stack pointer: A 65816 register that always contains the memory address of the top of the stack.

standard color table: The color table in force if no other color table is specified by a program.

string: A series of text characters.

structure region: The area occupied by both a window's content region and window frame.

toolbox: Collectively, the preprogrammed routines both built into the Apple IIGS ROM and supplied in the ProDOS 16 system disk.

tool locator: Toolbox routines that assist a program in loading specific tool sets into memory for the program to use.

top of the stack: The open end of the stack to which items are pushed and from which items are popped.

transfer mode: A way of specifying how pixels in a pen and in an existing pixel image are to combine when overlapped.

update: To redraw that part of a window's content region that has been exposed by the adjustment of window locations on the screen.

update region: The area of a content region exposed by the adjustment of window locations and requiring redrawing to fill in the blank space.

window: An object on the screen in which text and graphics information is displayed.

window event: An event signifying that some action has occurred that affects the display of one or more windows on the screen.

window frame: The overlaying border of a window, often consisting of a title bar, scroll bars, and other components.

word: A group of 16 bits of information.

Index

ABOUT THE AUTHOR

Photo by Linda Racine.

Danny Goodman has been showing people productive ways to use personal computers since the late 1970s. He is Contributing Editor to *Macworld* and *PC World* and is the author of nine microcomputer books, including Bantam's *The Idea Book for the Apple II*. One of his books, *Going Places With the New Apple IIc*, was a computer book best-seller in the United States and has been translated into French, Italian, German, and Dutch for European IIc fans. Danny's articles interpreting computer and consumer electronics technologies also appear in many general interest magazines, including *Playboy* and airline magazines of the East/West Network.

Open The Apple IIGS Toolbox...

...and you'll uncover nearly 600 assembly language subroutines that will give your programs the professional look and feel.

Programming the Apple IIGS with the help of the toolbox makes it easy to add onscreen windows, pull down menus, colorful animation, and wonderfully detailed graphics in super high-resolution video modes. But designing programs around its Macintosh-like user interface is quite different from programming in other environments.

THE APPLE IIGS TOOLBOX REVEALED introduces you to crucial concepts before you start programming—concepts that your programming language and other reference guides assume you already know.

- **For Programming Newcomers**: We'll take you inside the computer and some of its circuits; you'll learn about programming languages and how to choose one that's best for you.
- **For Experienced Apple II Programmers**: We'll show you how the toolbox works and how to incorporate its routines into C, Pascal, and assembly language programs; you'll learn how to design event-driven programs around Apple's User Interface Guidelines.
- **For Macintosh Programmers**: We'll demonstrate how the Apple IIGS toolbox differs from the Mac's; you'll learn how to adapt your programs to the machine's 4096 colors.

This is the book to read *before* you start programming. With an understanding of the concepts presented here, you'll be on your way to Apple IIGS toolbox programming much more quickly and efficiently.

Danny Goodman has been showing people productive ways to use personal computers since the late 1970s. He is contributing editor to *Macworld* and *PC World*, and is author of *The Idea Book for Your Apple II*.

34360

0 76783 02195

N 0-553-34360-2>21195

34360-2 ■ IN U.S. $21.95 (IN CANADA $25.95) ■ BANTAM COMPUTER BOOKS