HOWARD W. SAMS & COMPANY 22599

Programming the Apple IIGS™

in C and Assembly Language

Mark Andrews

Programming the Apple IIGS™

in C and Assembly Language

Programming the Apple IIGS™

in C and Assembly Language

Mark Andrews
with
Michael Halpin

HOWARD W. SAMS & COMPANY
A Division of Macmillan, Inc.
4300 West 62nd Street
Indianapolis, Indiana 46268 USA

FIRST EDITION
FIRST PRINTING—1987

International Standard Book Number: 0–672–22599–9
Library of Congress Catalog Card Number: 87–62537

Acquisitions Editor: *Greg Michael*
Manuscript Editor: *Susan Pink Bussiere, Techright*
Technical Reviewer: *Eagle I. Berns*
Designer: *T. R. Emrick*
Cover Art: *Ric Harbin*
Compositor: *J. Jarrett Engineering, Inc.*

Printed in the United States of America

Trademark Acknowledgments

Contents

Introduction

The Apple IIGS is two computers in one, and this book is about both of them. It's also about the two most powerful programming languages for the Apple IIGS: assembly language and C.

Apple calls the IIGS a two-in-one computer because it runs most software written for earlier Apple IIs, yet offers today's computer user a host of brand new Macintosh-like features—plus full color—at an Apple II price.

This book is a two-in-one book, twice over; it teaches you how to program the IIGS in both of its operating modes—8-bit emulation mode and 16-bit native mode—and it teaches you to do that in two languages—assembly language and C.

If you want to learn to program both of the computers built into the IIGS—in C, assembly language, or both—this is the book you are looking for.

In plain English, and with the help of many, many figures and tables, this book introduces you to the IIGS from the ground up: how it's laid out, how its microprocessor works, and how it is different from—and similar to—other computers in the Apple II family. After that ground has been covered, you learn how to start programming the Apple IIGS in assembly language and C.

What This Book Can Do for You

If you've written programs in BASIC, Pascal, or any other programming language, this book is all you need to start programming the Apple IIGS in assembly language. If you're an experienced assembly language programmer, you can learn how to expand your knowledge to include all the new and special features of the Apple IIGS . If you're primarily a C programmer, you can learn how to deal with all the IIGS's new features in programs written in C.

This book is also an asset to assembly language programmers who would like to start saving time by including C routines in their programs and to C programmers who would like to streamline and speed up portions of their programs by learning some assembly language. If either of these possibilities appeals to you, you'll be happy to learn that the software development system used to write the programs in this book, the Apple Programmer's Workshop (APW), makes it easy to combine routines written in assembly language and C—and this book teaches you how.

What You Can Find in These Pages

As you read this book, and type and run the many example programs, you may notice that

- Unlike many books on C and assembly language programming, it is written in English, not computerese, and is designed for people who want to learn to program, not just for professional programmers and engineers (though some of them will find it useful, too).
- It includes a complete course on how to use the Apple IIGS Toolbox, a set of built-in assembly language subroutines that distinguish the IIGS from all previous Apple IIs. The Toolbox is what provides the IIGS with such spectacular graphics features as windows, pull-down menus, icons, and mouse-controlled commands. This book teaches you how to use most of the tools in the Toolbox, in both C and assembly language.
- It is packed with what almost every computer book could use more of: type-and-run programs that do far more than illustrate the points being discussed. They are designed to put the IIGS through its paces as you learn how it works. When you finish this book, these programs form a useful library of commonly used Apple IIGS routines.

What You Can Learn

By the time you finish this book, you'll also know how to

- Program the Apple IIGS's 65C816 chip in assembly language, in both its 8-bit emulation mode and its 16-bit native mode. Part 1 covers the fundamentals of Apple IIGS programming. Most of the programs in this segment are written in emulation mode. In part 2, you can pull out all the stops and learn how to program the IIGS in its full 16-bit native mode.

- Write text-based programs using the Toolbox's Text Tool Set and write super high-resolution graphics programs using QuickDraw II—a IIGS tool set that you use to design text screens, pictures, and even printed documents with a palette of 4,096 colors and a screen resolution of up to 640-by-200 pixels.
- Equip your programs with eye-catching graphics features such as pull-down menus, multiple windows, icons, and the dialog boxes that serve as communication windows between the user and the IIGS.
- Write sound tracks for your programs using the IIGS's 15-voice, 32-oscillator sound and music synthesizer—the most powerful sound system in any computer in the IIGS class.

You learn how to do all of this—and much, much more—in both C and assembly language.

What You Need

To use this book, you need an Apple IIGS with at least two 3.5-inch disk drives, a monochrome or color monitor, and at least 512K of extra memory. A hard disk, a 1-megabyte RAM disk, and at least another 512K of extra memory are highly recommended.

As you advance in your knowledge of IIGS programming, a few books besides this one might come in handy. Two works that every serious IIGS programmer should own are the *Apple IIGS Toolbox Reference* and the *Apple IIGS ProDOS 16 Reference*, both written at Apple and published by Addison-Wesley. The *Apple IIGS Toolbox Reference* is a particularly important work because it explains exactly how to use every tool in the IIGS Toolbox in programs written in both assembly language and C.

Three other books that are required reading for IIGS programmers are the *Apple IIGS Programmer's Workshop Reference*, the *Apple IIGS Programmer's Workshop Assembler Reference*, and the *Apple IIGS Programmer's Workshop C Reference*, which were also written at Apple and published by Addison-Wesley. Many other books that you might find useful or interesting are listed in the Bibliography.

Ready, Set, Go!

If you've read this far, it's a safe bet that you're at least a little bit interested in learning how to program the Apple IIGS in C, assembly language, or both. There's no better time to begin than right now. So turn the page and start from the top—with chapter 1.

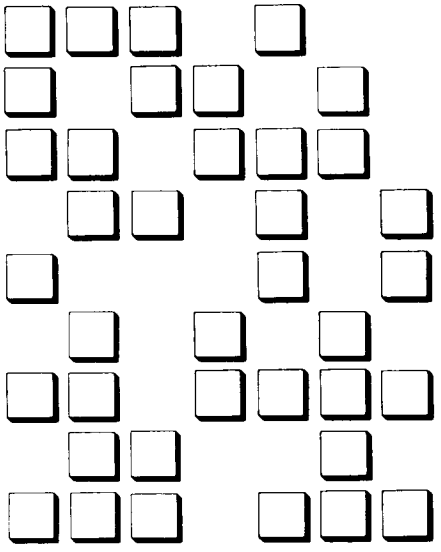
Acknowledgments

Many thanks to Eagle I. Berns, Steve Glass, Loretta Barnard, Kevin Armstrong, Brent Olson, David D. Good, Greg Borovsky, Eric Ford, Anil Gursahani, Ray Hughes, Brian Hurley, Dennis Kudo, and Alireza Latifi, all of Apple. Without their help and patience, this book could not have been written.

To Swami Muktananda

PART 1

Fundamentals of Apple IIGS Programming



CHAPTER 1

Introducing the Apple IIGS

The Apple II for the Rest of Us

What do you get when you cross an Apple Macintosh with an Apple II? When hardware engineers at Apple Computer attempted that feat, they came up with the Apple IIGS—a remarkable new personal computer that offers Macintosh-like features at an Apple II price, with super high-resolution graphics and spectacular sound thrown in as part of the bargain.

An Apple II—Plus!

The specifications of the Apple IIGS are not quite the same as those of the Apple Macintosh. For example, the IIGS uses a 65C816 microprocessor, but Macintosh computers are built around chips of the 68000 family. Also, the IIGS has a different type of screen display. The IIGS generates a color video display with a screen resolution of either 320-by-200 pixels or 640-by-200 pixels, depending on the graphics mode. The Macintosh Plus and the Mac SE produce black-and-white displays that measure 512-by-342 pixels. Table 1–1 lists the most important specifications of the Apple IIGS.

There are other differences between the IIGS and the Macintosh. One difference, immediately apparent to a potential computer purchaser, is that a Mac, even a low-end model, is considerably more expensive than a IIGS.

Table 1–1
Apple IIGS Specifications

Feature	Specifications	Comments
CPU	65C816	16-bit microprocessor with 24-bit (16 MHz) addressing capability. 6502 and 65C02 compatible.
Operating speeds	2.8 MHz and 1 MHz	Selectable dual operating speeds provide compatibility with earlier Apple IIs.
Memory capacity	256K RAM, 128K ROM	RAM expandable to 8.25 megabytes. One megabyte of memory available for ROM expansion.
Desktop user interface	Mouse, windows, pull-down menus	Macintosh-like programming and user environment.
Mouse	Two button	Connects with IIGS by ADB (Apple Desktop Bus) cable.
Toolbox	In RAM and ROM	Toolbox contains more than 800 prewritten routines that can be used in application programs.
Keyboard	78 keys	Detached keyboard has built-in numeric keypad and can be used to type in foreign languages.
Monitor outputs	RGB and NTST	Can be used with analog RGB monitor, composite monitor, or TV (with modulator adaptor).
Text modes	40 column and 80 column	Text modes measure 40 columns by 24 lines and 80 columns by 24 lines. Border, foreground colors, and background colors are user-selectable.
Graphics modes	Apple II modes and super high-resolution mode	All Apple IIc and IIe graphics modes, plus super high-resolution mode.
Resolution	320-by-200 pixels, 640-by-200 pixels	Two screen resolutions offered in super high-resolution mode.
Colors	4,096	4,096-color palette available in super high-resolution mode; 16 or more colors can be displayed simultaneously.
Sound	32-oscillator synthesizer	Ensoniq synthesizer supports 15 independent voices. Sound chip includes 64K of dedicated RAM for storing sound patterns.
Enhanced monitor	Built into ROM	Handles 24-bit addresses. Includes mini-assembler and I/O routines. Can perform hex math.
BASIC	Applesoft	Enhanced BASIC interpreter built into ROM.
Control panel	Built-in desk accessory	Can be used to set display parameters, slot and port use, operating speed, RAMdisk, and disk drives.
Clock	Built in	Provides time and date.
Serial ports	Two built-in serial ports	Support modems, printers, and AppleTalk. Serial card can also be installed.
AppleTalk	Uses one serial port	AppleTalk can be used with either serial port. No peripheral card required.
Disk port	Disk I/O port uses custom IC	Up to six disk drives can be supported by built-in port, or plug-in cards, or both.

Table 1–1 (cont.)

Feature	Specifications	Comments
Hard disk	Optional	Hard disk 20SC can be connected with SCSI interface card.
Expansion slots	Seven slots for plug-in cards	Supports plug-in cards as well as built-in ports.
Game I/O	External and internal	External 9-pin jack, internal 16-pin socket. ADB (Apple Desktop Bus) connector also available for game controllers.
Operating system	ProDOS 16, ProDOS 8, DOS	Designed to use ProDOS 16, but also compatible with ProDOS 8 and DOS.
Interrupts	Fully supported	Built-in interrupt handler. Vertical blank interrupts, scan line interrupts, mouse and sound interrupts, and many other kinds of interrupts are supported.

Another difference, not quite so obvious but as important from a programmer's point of view, is that the Mac and the IIGS don't "speak" the same machine language. The Mac has a 32-bit microprocessor designed to be programmed in 68000 assembly language. The main microprocessor in the IIGS, the 65C816, is a 16-bit successor to the 8-bit 6502 and 65C02 chips in older Apple IIs. (The difference between an 8-bit chip and a 16-bit chip is covered in chapter 5.) Furthermore, the memory of the Macintosh is laid out as one continuous bank, but the memory map of the IIGS is broken into 64K banks, like the memory map of an Apple IIc or an expanded Apple IIe. The memory architecture of the Apple IIGS is covered in chapter 4.

Because of the Apple IIGS's 6502-family microprocessor, color display, IIc and IIe compatibility, and Apple II heritage, it is actually related more closely to earlier members of the Apple II than to the Mac (although it is something of a Mac lookalike). Nontheless, the IIGS is much more than just a souped-up Apple II.

"Like Janus, the god of doorways," one Apple spokesman explained, "the IIGS looks in two directions." First, he pointed out, the computer looks toward the future: "With its many high-performance features—such as its improved color display, advanced sound system, 16-bit processor, and larger memory, it makes it possible for more powerful programs to be designed." But, he emphasized, it also "looks back on the past. Because it also has the features of earlier members of the Apple II family, it can run most of the vast library of software that was written for its predecessors, such as the Apple IIc and the Apple IIe."

The IIGS, in its forward-looking stance, is a new breed of Apple II, operated in a Macintosh-like desktop environment—complete with a super high-resolution screen, icons, pull-down menus, desk accessories, and a mouse. To make life easier for the programmer who wants to use these new features, the IIGS comes with a fully equipped Toolbox—an enormous library of prewritten routines that are easily incorporated into user-written programs. With the Toolbox, programmers working in high-level languages such as C,

assembly language, Pascal, and even BASIC can make use of windows, menus, icons, and the rest of the IIGS desktop environment without writing the code from scratch. With the help of the Toolbox, it is easier to write sophisticated, eye-catching programs for the IIGS than it is to write simpler programs for earlier Apple IIs.

The main features of the IIGS Toolbox are described in detail in part 2, which begins with chapter 7. Important tools in the Toolbox are covered individually, beginning in chapter 7.

Memory Magic

Of all the remarkable features of the IIGS, the one probably most welcome to programmers is the IIGS's prodigious memory capacity. The computer comes with 256K of RAM and 128K of ROM—a far bigger supply of memory than the 128K of RAM and 32K of ROM built into its most recent predecessor, the Apple IIc. You can expand the generous amount of RAM supplied with the IIGS to as much as 8.25 megabytes with the simple addition of a plug-in card.

24-Bit Addressing

The huge memory capacity of the IIGS is made possible by the addressing capabilities of its 65C816 microprocessor. As you will see in chapter 4, the 65C816 has 24-bit addressing capability, giving it a total memory space of 16 megabytes. Of this total, 8.25 megabytes are available for RAM expansion and 1 megabyte is available for ROM expansion.

The memory of the IIGS is mapped out in detail in chapter 4. In chapter 6, which is devoted to the addressing modes of the IIGS, you'll see how the IIGS addresses memory.

The Apple IIGS as an Apple II

Because the IIGS is compatible with earlier Apple IIs, its memory layout can be used in two ways: in a mode that emulates earlier Apple IIs or in a mode that takes full advantage of the computer's memory capacity. When the IIGS is in Apple II emulation mode, only 128K of memory is used, and that 128K is laid out like the main and auxiliary memory banks of a IIc or IIe. Figure 1–1 is a map that shows how the memory of the Apple IIGS is organized when it is operated in Apple II emulation mode.

The IIGS in Native Mode

When the IIGS is in native mode, another 128K of RAM and a full 128K of ROM are added, along with whatever additional memory is installed. All this added memory is available for use in application programs, except for a few areas in low memory claimed by ROM addresses, operating system RAM, sound and video RAM, and system I/O memory. Figure 1–2 is a map that shows the memory architecture of a IIGS system running in 16-bit native mode.

The Memory Manager

One new feature of the IIGS is that all memory-related operations can be handled by a special tool called the Memory Manager. The Memory Manager is active when the IIGS is booted and, from that moment on, is in complete control of the computer's memory. It can allocate, deallocate, and compact

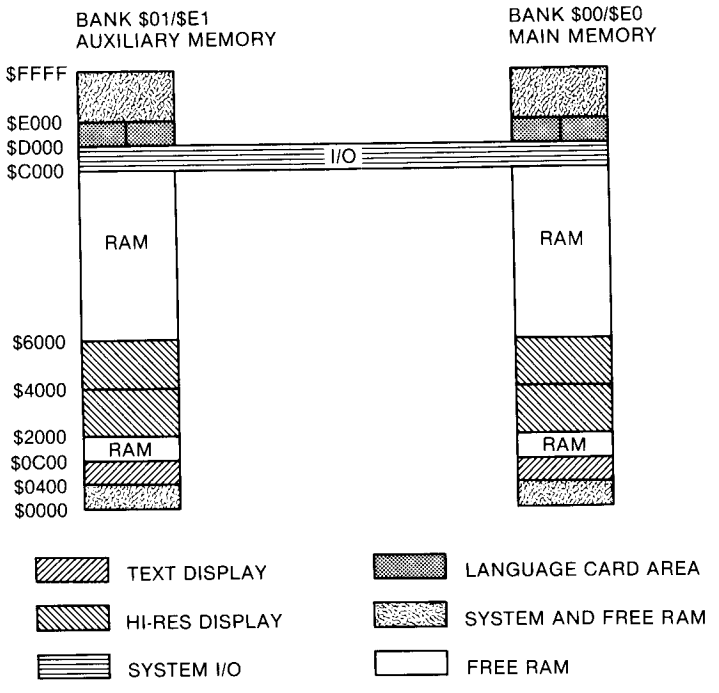


Figure 1–1
Memory map of the IIGS in IIc/IIe emulation mode

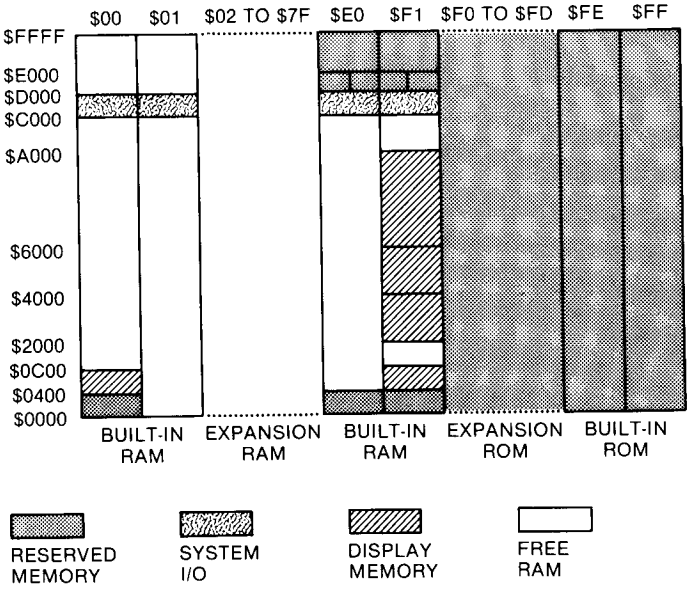


Figure 1–2
Memory map of the IIGS in 16-bit native mode

memory while application programs are running, taking most of the burden of memory management off the programmer. The memory architecture of the IIGS and the role of the Memory Manager are discussed in more detail in chapter 4.

Faster than a Speeding Apple II

In addition to a larger memory capacity, the IIGS runs faster than earlier members of the Apple II family. The IIGS's 65C816 processor operates at 2.8 MHz, almost three times as fast as the 1 MHz speed of the 6502 and 65C02 chips in the IIe and IIc. But the 65C816 can also be set to run at the same speed as a 6502 or 65C02. Because of this dual-speed capability, the IIGS can run most of the vast library of software for earlier Apples. You can experiment with operating speeds. Many programs designed for earlier Apples can be run on a IIGS at either the 1 MHz speed they were designed for or the IIGS's native clock speed of 2.8 MHz. This can add new challenges to arcade-style games designed for earlier Apples. On a IIGS, some games can be accelerated to almost three times their speed on earlier Apple IIs.

Besides the 65C816 chip's faster speed and expanded memory addressing capability, it has a bigger and more powerful set of internal registers. Its accumulator, X register, and Y register are expanded from 8 bits to 16 bits. It also has three new registers: an 8-bit data bank register, an 8-bit program bank register, and a 16-bit direct page register. Other features of the 65C816 include 11 new addressing modes and 36 new assembly language instructions, for a total of 24 addressing modes and a total vocabulary of 91 assembly language mnemonics. These new features are examined in chapter 5.

GS: Graphics and Sound

The IIGS has many other special features. Two attributes are so important that the computer was named after them: the *g* in IIGS stands for *graphics* and the *s* stands for *sound*. So let's pause for a closer look at the graphics capabilities of the IIGS and a brief glance at the IIGS world of sound.

IIGS Graphics

The IIGS can handle both text modes and all three graphics modes of its most recent predecessors, the IIc and the IIe. Like the IIc and the IIe, the IIGS has two text modes. It can produce a 40-column, 24-line text screen, which is displayed on an ordinary television screen, or an 80-column, 24-line text screen, which requires a high-resolution color or monochrome monitor. The IIGS's three graphics modes are like those in the IIc and the IIe: a low-resolution mode, a high-resolution mode, and a double high-resolution graphics mode with a 16-color palette and a screen display 560 dots wide by 192 dots high.

But these three graphics modes—designed for earlier Apples and built into the IIGS primarily for compatibility—are not the modes for which the Apple IIGS is named. Besides the three graphics modes in the IIc and the expanded IIe, the IIGS has two new graphics modes called super high-

resolution modes. One of these, 320 mode, has a screen display that measures 320 dots wide by 200 dots high. The other, 640 mode, has a 640-by-200 dot display. In super high-resolution graphics mode, a palette of 4,096 colors is available, and up to 16 colors—or even more, with interrupts—can be displayed simultaneously.

Both of the graphics modes native to the IIGS are produced by a large-scale integrated (LSI) video chip called the *video graphics controller,* or VGC. The VGC can generate 4,096 colors and, with video interrupts, can simultaneously display up to 256 colors on the screen. Without using interrupts or other special techniques, the VGC can display up to 16 colors at a time in 320 mode and up to 6 colors at a time (including black and white) in 640 mode. With a color-interleaving system called *dithering,* a 640-mode screen, like a 320-mode screen, can display up to 16 colors at a time. More details about IIGS graphics—and a collection of type-and-run graphics programs—are presented in chapter 8.

IIGS Sound

In addition to spectacular graphics, the IIGS has sensational sound. Computer critics have raved that the IIGS has the finest sound system of any computer in its class.

The IIGS owes its sonic superiority to a 15-voice, 32-oscillator integrated circuit called the *digital oscillator chip,* or DOC. The DOC is manufactured by Ensoniq and used in their line of professional sound synthesizers. The chip has 64K of independent RAM and can generate waveforms from digital samples stored on a disk and loaded into its memory. So it can produce multivoice music and other kinds of complex sounds without tying up the IIGS's main microprocessor.

The IIGS sound system includes another custom chip called a *general logic unit,* or GLU. The GLU chip is a system interface with the DOC. This enables the IIGS to produce sound in two ways: with its DOC chip or with a simple, switch-controlled circuit that produces notes, tones, and beeps in the manner of earlier Apple IIs.

The IIGS sound system, like most of the computer's other new features, is designed to be programmed with the help of the IIGS Toolbox. The sound-producing capabilities of the Apple IIGS are described in more detail in chapter 13.

A Closer Look at the Toolbox

In the earliest models of the IIGS, parts of the Toolbox were built into ROM and parts were included on a system disk. In later models, as the design of the Toolbox became more solid, tools originally included on the system disk were moved into ROM. From a programmer's point of view, it ordinarily doesn't matter whether a given IIGS tool is built into ROM or provided on a system disk and loaded into RAM when needed (except that tools in ROM load and work faster). That's because the Toolbox includes a special tool-finding and tool-loading program called the Tool Locator. The Tool Locator

can automatically find any tool—in ROM or RAM—and then load that tool into memory.

After a tool is found and loaded by the Tool Locator, it can be incorporated into an application program by calling an assembly language macro—if the program is written in assembly language. C programs call Toolbox functions using standard C calling functions.

The Apple IIGS Programmer's Workshop (APW), the software package used to write and assemble the assembly language programs in this book, comes with a library of macros that make it easy to include Toolbox macros in application programs. There's more about macros in chapters 3 and 7. The APW C compiler, which was used to write and compile the C programs in this book, has an interface library that allows Toolbox functions to be incorporated into C programs. There's more about that in chapter 3.

The APW assembler is introduced in chapter 2, and the APW C compiler makes its first appearance in chapter 3. Most of the assembly language programs in part 2 contain calls to APW Toolbox macros. Most of the C programs use Toolbox functions in the APW C interface library.

Opening the Toolbox

The Apple IIGS Toolbox contains a large assortment of useful prewritten routines. Five of these tools are of primary importance. Apple refers to them as the "Big Five." These five major tools are

- The Tool Locator. Details about the Tool Locator are presented in chapters 3 and 7.
- The Memory Manager. The Memory Manager is covered in more detail in chapter 7.
- QuickDraw II, which handles graphics and drawing routines. QuickDraw II, modeled after the QuickDraw tool set for the Apple Macintosh Toolbox, is examined in chapter 8.
- The Event Manager, which handles mouse operations and determines what the IIGS does in response to various moving and clicking operations that involve the mouse. The Event Manager is covered in chapter 7.
- The Miscellaneous Tool Set, which—despite its unimportant-sounding name—is vital to the operation of the IIGS. The Miscellaneous Tool Set handles low-level mouse operations, firmware interrupt operations, access to the RAM that is backed up by the built-in battery, reading and setting the computer's built-in clock, and many other important functions. The Miscellaneous Tool Set contains so many different kinds of tools that it is not covered in a chapter of its own, but is referred to as required in part 2.

The other tools in the IIGS Toolbox are

- The Menu Manager, which is used to create and control pull-down menus. The Menu Manager is the subject of chapter 7.
- The Window Manager, which takes care of the document and picture windows displayed by application programs. With the help of the Window Manager, you can place multiple windows on the screen. You can also scroll, shrink, expand, and drag windows, and place windows in front of and behind other windows on the screen. You get a close look at the Window Manager in chapter 10.
- The Dialog Manager, which handles alert dialogs—text windows that warn of impending danger—and boxes that let you choose functions by activating controls (such as scroll bars and pushbuttons) displayed on the screen. The Dialog Manager is examined in chapter 11.
- The Control Manager, which handles scroll bars, buttons, and all other kinds of onscreen controls used by tools such as the Window Manager and the Dialog Manager.
- The Font Manager, which controls the selection, loading, styling, displaying, and printing of character fonts.
- The LineEdit Tool Set, which handles keyboard text input when the IIGS is in super high-resolution graphics mode.
- The Text Tool Set, which handles keyboard text input when the IIGS is in 40-column or 80-column text mode. The Text Tool Set is introduced in chapter 3.
- The Scrap Manager, which manages cut-and-paste operations.
- The Standard File Operations Tool Set, which works with ProDOS 16 to create dialog windows that load and save disk files. The Standard File Operations Tool Set and ProDOS 16 are covered in chapter 12.
- The List Manager, which handles lists displayed on the screen when the IIGS is in super high-resolution display mode. The List Manager is used by higher-level tool sets such as the Standard File Tool Set and the Font Manager. It is also available for use by application programs.
- The Print Manager, which interfaces the IIGS to a variety of printers, including dot-matrix graphics printers such as the ImageWriter and laser printers such as the LaserWriter.
- QuickDraw Auxiliary, which adds some tools—and more graphics power—to QuickDraw II.
- The Integer Math Tool Set, which can make life easier for the designer of mathematically oriented programs. With the help of the Integer Math Tool Set, a program can easily handle mathematic operations ranging from simple integer addition to complex trigonometric functions.

- The Standard Apple Numerics Environment (SANE), which includes a library of more advanced arithmetic and mathematic operations.
- The Sound Tool Set, which controls both the old-fashioned switch-style sound system of the IIGS and the computer's newer super-sophisticated digital oscillator chip (DOC) sound synthesizer. Instructions for programming the Sound Tool Set, and some type-and-run routines that put it through its paces, are presented in chapter 13.
- The Desk Manager, which controls the operation of desk accessories—mini-applications that can be run at any time without interfering with application programs.
- The Scheduler, which delays the activation of desk accessories and other applications until the resources they need are available.
- The Apple Desktop Bus (ADB), a tool for connecting input devices such as the keyboard, the mouse, graphics tablets, and game controllers to the Apple IIGS.

The disk operating system used by the IIGS is ProDOS 16. ProDOS 16 is a 16-bit descendent of ProDOS 8, the IIc and IIe operating system. The IIGS can run programs written under ProDOS 8, ProDOS 16, and even Apple DOS, the operating system that preceded ProDOS 8. To help programmers use ProDOS effectively, the IIGS Toolbox includes a Standard File Manager, which is covered in chapter 12.

What Happens When You Turn It On

When you turn on the IIGS and boot the system disk, the first thing you see depends upon how much memory your IIGS has. If it has 512K of memory or more, you'll see the IIGS Finder—a screen patterned after the opening screen of the Apple Macintosh, but displayed in full color. If your IIGS has less than 512K of memory, the startup screen will be a Program Launcher—a plainer looking display that does not have all the features of the Finder, but does allow you to select and run programs with a mouse. If you have 512K of memory and still see a Launcher display, your system disk is not a Finder disk, which now comes with every Apple IIGS, but a Launcher disk, which was packed with the first IIGS computers and is now outdated. Early IIGS disks were missing some tools, had bugs in others, and thus won't work with some of the programs in this book. So, if you have a Launcher disk instead of a Finder disk, please see your Apple dealer. Figure 1–3 is an illustration of the Finder disk's screen display. On the opening screen of the Finder disk, the Apple IIGS displays icons, or small pictures, representing various components in the system. On the Finder screen, each 3.5-inch disk in a disk drive is represented by an icon that looks like a 3.5-inch disk. If your system includes a hard disk, a RAM card, or a hard disk drive, those are represented by icons too.

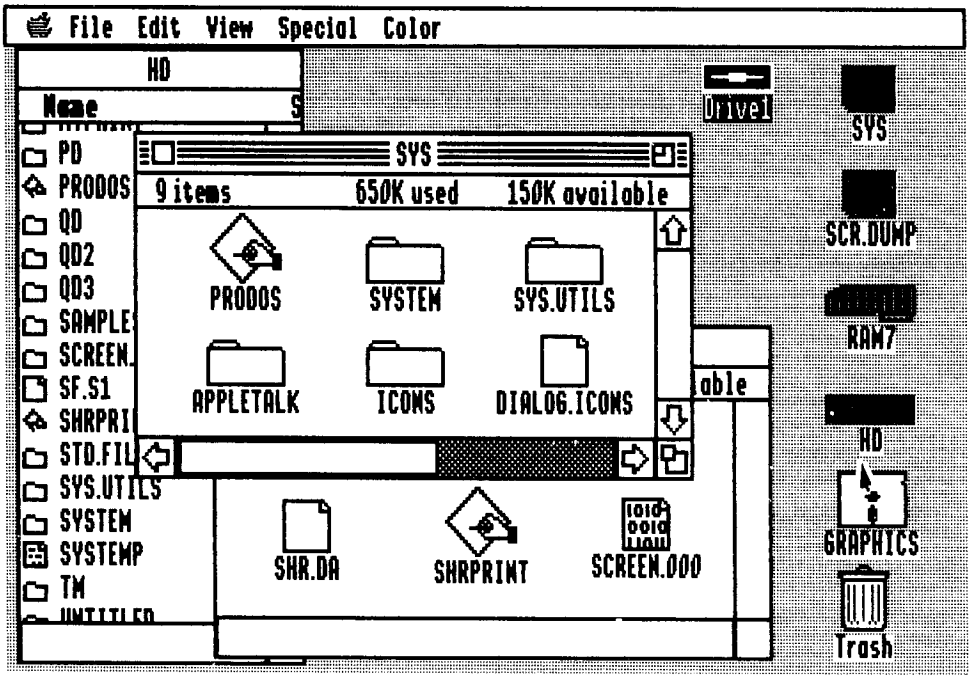


Figure 1–3
Finder disk screen display

From the IIGS Finder disk, you can load, or launch, any executable program stored on a disk. For example, you can use the Launcher to load the APW assembler-editor system, the APW C compiler, or programs you have created using the APW system.

The User Environment

Much has been written and said about the new era in personal computing that began with the introduction of the Apple Macintosh. By offering the personal computer user a new type of user environment—featuring such innovations as windows, pull-down menus, icons, and the mouse—the Apple Macintosh started such a revolution in desktop computing that even IBM was finally forced to incorporate Mac-like features in its personal computer line.

The secret behind the success of the Macintosh—and the IIGS—is *event-driven programming*. In the pre-Macintosh era, computers were designed to operate under a system called *sequential programming*. If pre-Mac computers were difficult to understand and easy to hate, it was largely because of the sequential design of their programs. When a program is written in a sequential fashion, it presents the user with an onscreen prompt and expects the user to type in something. If the user types in a response that the computer considers acceptable, the computer goes to another part of the program it is running—

that is, into another mode. At that point, the user might be presented with another menu, forcing a choice that puts the program into still another mode.

To get from one kind of operation to another, the user of a sequentially designed program usually has to move up or down through a hierarchy of menus, often having to pass through one mode to get to another. This approach puts the computer in charge of the user and often makes the user feel subservient, intimidated, and even angry at the machine.

Event-driven programming, in the hands of a skilled programmer, can reverse this scenario and make the computer the servant of the user. The main characteristic of an event-driven program is that it is modeless. When an event-driven program is executed, the computer can do just about anything the program allows at just about any time, without the user having to switch modes or move through a heirarchy of menus.

The IIGS—with its pull-down menus, windows, and icons—is very much at home with modeless, event-driven programs. In a typical IIGS program, you are first presented with a menu. With the help of a mouse, you can then select a menu option. If you make a mistake while runnning an event-driven program, the program (if it is well-written) courteously indicates the mistake and suggests an alternate approach. This style of programming makes you the master and the computer the servant—which, of course, is the way things ought to be.

So it is not difficult to see why computers programmed in the old-fashioned sequential style have been the targets of so much wrath and why event-driven computers like the Mac have become so popular—among program designers and users. All the programs in part 2 are event-driven programs, and more about event-driven programming is presented in chapter 9.

To support event-driven programming, a computer needs a host of features that were unavailable in the computers of yesteryear. The IIGS, like the Macintosh, has all the features needed to make event-driven programming possible: windows, pull-down menus, icons, dialog windows that enable the user to communicate with the computer, and the mouse. Because of these features, the "feel" of the IIGS is similar to the feel of the Mac—although a few features of the venerable Apple II line have also been thrown in so that the computer's Apple II heritage is not forgotten.

The goal of this book is to help you learn to program the IIGS in the way it was meant to be programmed—using its mouse-controlled, event-driven, user environment. You'll do that using both assembly language, which is fast but not easy to master, and C, which is a little slower (though still light-years ahead of BASIC) but considerably easier to learn and quite a bit easier to manage.

In this chapter, you looked at the Apple IIGS, some of its principal features, and its most important programming tool, the IIGS Toolbox. In chapter 2, you start programming the IIGS in assembly language. In chapter 3, you start writing some C programs.

CHAPTER 2

Programming the IIGS in Assembly Language

Using the APW Assembler

If you've written assembly language programs for an Apple II, but haven't done any assembly language programming for the Apple IIGS, you're in for a big surprise. Programs written for the IIGS run faster, offer more sophisticated graphics and sound capabilities, and—best of all, from a programmer's point of view—can use more than 800 prewritten routines built into the Apple IIGS Toolbox. Some of the tools in the IIGS Toolbox are built into ROM and others are loaded into RAM when you boot the computer's system disk. But they're all available for use at any time in application programs.

The APW Assembler-Editor

The Apple IIGS Programmer's Workshop (APW), which was used to write most of the assembly language programs in this book, comes with a library of macros that make it quite easy to use the IIGS Toolbox in user-written programs. APW was created by the Byte Works Inc., a small company in Albuquerque, New Mexico, and is marketed by Apple. It is the first assembler-editor package offered solely for the Apple IIGS, and it is designed with all the IIGS's advanced features in mind.

The APW Package

Apple calls the APW package "a development environment for the Apple IIGS computer." It contains

- A shell that enables the IIGS programmer to run programs and use many useful file management and utility functions.
- An editor that can be used to write assembly language programs, C programs, executable shell files (exec files), and text files.
- An assembler that converts, or assembles, assembly language programs into machine language programs.
- A linker that converts machine code files produced by the APW assembler or C compiler into load files—files the IIGS system loader can load into memory. Briefly, here's how the linker works. When a program is written using the APW assembler or the APW C compiler, it is stored in memory in a format called *object module format,* or OMF. Before an OMF file can be executed, however, it must be linked, or converted into a format that the system loader can load into memory. The process of converting OMF files into linked files, or loadable and executable files, is the job of the APW linker. To create a linked file, the linker resolves external references (references in one program segment to routines or data in another). The linker then creates relocation dictionaries that the system loader uses to relocate code as needed when it is loaded into memory.
- A generous selection of utility programs that perform many functions. These programs format disks, copy files and disks, catalog disk directories, assemble and link assembly language programs, disassemble machine code and display it as source code, display the contents of memory, and much more. (It is beyond the scope of this book to examine the APW system's utility programs in detail.)
- An optional C compiler that converts, or compiles, C programs into executable machine language programs.
- An optional debugger that helps programmers correct assembly language programs.

A Warning

Before we go into any more detail about the APW development system, it should be pointed out that the version of the system available at this writing may not be exactly the same as the one you're using. The APW development system evolved from the ORCA/M assembler, which was designed long before the advent of the Apple IIGS, and the evolution of the APW system is still continuing. When this book was written, APW was a text-oriented system that did not use the sophisticated graphics or event-driven programming capabilities of the Apple IIGS. By the time you read this, APW may have evolved into a super high-resolution program with windows, pull-down menus, and mouse controls. If that's the kind of APW system you have, some of the information in the following paragraphs won't apply because

Using the APW Shell

When you use the APW system to write an assembly language program, the system's shell provides the interface that allows you to execute APW commands and programs. When you are writing a program, for example, you can activate the APW editor and assembler by typing shell commands. You can also use the shell to perform such tasks as copying files, deleting files, and listing directories. More ways to use the shell as an assembly language programming tool are described in the *Apple IIGS Programmer's Workshop Assembler Reference*, written by the folks at Apple and published by Addison-Wesley.

Getting Started

There's no such thing as a standard IIGS configuration, and APW systems can also be different (a system designed for assembly language programmers will include a machine language assembler, one intended for C programmers will include a C compiler, and still other systems could include both an assembler and a C compiler).

Ordinarily, an APW system designed for assembly language programming will include two disks: one labeled /APW and the other labeled /APWU (for APW utilities). A C-based package will generally include one disk labeled /APW and one labeled /APWC.

In this chapter, we devote our attention primarily to APW systems designed around the APW assembler. Specific tips on installing and operating C-based systems are provided in chapter 3.

To simplify the installation of the APW development system, the designers of the system have placed a utility program called INSTALL on the APW disk. For owners of hard disks, a utility called HDINSTALL is provided.

It's easy to install an APW package on an Apple IIGS system. First, you should back up your original APW disks and put them in a safe spot. Then, if you are using a floppy disk system, place the copy of your /APW disk in one drive and a blank formatted disk in another. If you have a hard disk system, you can use APW's HDINSTALL program to install APW on your hard disk.

If you have a floppy disk system, you can install APW by simply booting APW from your master disk copy and typing a command like this following APW's # prompt:

```
install /apw /[name of your disk]
```

If all has gone well, the APW system will install itself on your blank formatted disk. When installation of your /APW disk is complete, you should see a prompt on the screen telling you that it is now time to install your /APWU disk. You can then remove the /APW disk, insert your /APWU disk, and type the command `install /APWU`. Your disks will start to spin again, and when everything is finished, you will have an installed copy of APW, complete on a single disk.

APW's HDINSTALL program works in a similar way, except that the program is installed in a hard disk directory instead of on an individual floppy.

What the APW System Contains

When you have the APW system installed on a disk—either hard or floppy—a catalog of the system will reveal that it contains the following files:

- A directory titled SYSTEM. This directory contains the APW program and text editor, which you will use to write your source code programs; a LOGIN file, which takes over when APW is booted and can configure APW to your individual Apple IIGS system; a SYSHELP file, which you can use to obtain information about any shell command by simply typing the word HELP followed by the actual command; and a few other files used by the APW system.
- A LANGUAGES directory, which includes the APW assembler (or, if you have a C-based system, your C compiler). The LANGUAGES directory also includes a file called LINKED that is used link object code programs after they have been assembled.
- A LIBRARIES file, which contains a subdirectory called AINCLUDE. In the AINCLUDE directory, you will find a collection of files divided into two categories. About half the files begin with the prefix E16, and the other half start with the prefix M16.

  The files that begin with M16 are APW macros: short, prewritten assembly language source files that you can incorporate easily into application programs. The files that begin with E16 are equate listings: source code files that define constants often used in Apple IIGS programs. After you learn how to use the equate files in the AINCLUDE library, they can be very useful in assembly language programs.
- In a C-based APW system, C libraries are also included in the LIBRARIES directory.
- A UTILITIES directory, which contains many important APW utilities. These include MACGEN, which is used to include APW macros in application programs; MAKELIB, which can be used to convert application programs into libraries so that they can be accessed more rapidly; and DEBUG, which can be used to run APW's optional assembly language debugger.
- APW.SYS16, the main APW program.

Using the APW System

After you set up the APW system, you can boot it by itself, from your IIGS finder disk, or from a hard disk, depending upon your preference and the configuration of your IIGS system. No matter how you launch APW, the first thing you'll see after APW goes into action is a screen heading that looks something like this:

```
Apple IIGS Programmer's Workshop
Copyright Byte Works, Inc. 1980–1986
Copyright Apple Computer, Inc. 1986
All Rights Reserved
```

A few lines below this display is a number sign prompt followed by a cursor:

```
#_
```

When this prompt appears on the screen, APW is installed and operating, and the computer is in the APW shell's command line mode. If you're using a pair of 3.5-inch drives and don't have a hard disk drive, you may have to do a little prefix changing; that is, you may have to direct APW to read your data disk by using the APW shell's `prefix` command. The `prefix` command can be followed by a full or partial pathname, like this:

```
prefix /MYVOLUME
```

or by a device number with a period in front of it, like this:

```
prefix .D2
```

More details on the use of the `prefix` command are in the *Apple IIGS Programmer's Workshop Reference*, the *Apple IIGS Programmer's Workshop Assembler Reference*, and the *Apple IIGS ProDOS 16 Reference* (all were prepared by Apple and published by Addison-Wesley).

The APW Editor

After APW is up and running, and the prefix of your data disk is set, it's easy to activate the APW editor. Just tell APW you want to edit a file and enter the name of the file. For example, type this line following APW's # prompt (don't type the prompt, just the two words that follow it):

```
#edit ZIP.SRC
```

This line tells APW you want to start editing a file named ZIP.SRC. Although the SRC suffix is not required, it is often used to distinguish source code files (assembly language programs) from object code files (machine language programs). The convention in this book is to give source code programs the SRC suffix and to assign no suffix to machine language programs.

When you type a command line using the format `edit filename`, APW looks on your data disk for a file with the name you have provided. If it can find one, it displays the file on the screen so you can edit it. If there is no file on the disk with that name, APW goes into editor mode and presents

a blank screen—blank, that is, except for a ruler line at the bottom. Then you can write a new program that will have the filename you have chosen.

This is a good time to install and load APW and type the command line `edit ZIP.SRC`, if you haven't done so already. Then you'll be ready to type, assemble, and execute the ZIP.SRC program, which appears in listing 2–1. If you're familiar with the adventures of a certain pinhead cartoon character, you'll understand how the program got its name.

Listing 2–1
ZIP.SRC program

```
*
*   ZIP.SRC
* A program that asks an important question
*
        KEEP ZIP
        LIST ON

Zippy   START
        phk                             ; make program bank
        plb                             ; and data bank the same

        pea testmsg|-16                 ; push msg bank on stack
        pea testmsg                     ; push msg address on stack

        ldx #$200C                      ; put tool no. in x reg
        jsl $E10000                     ; long jump to tool dispatcher
        rtl                             ; long return

testmsg dc c'Are we programming yet?',h'00'

        END
```

The ZIP.SRC program is written in the Apple IIGS's 16-bit native mode. It doesn't use the Memory Manager or some of the other advanced features of the IIGS, but it is a native mode program.

In a few moments you'll examine the ZIP.SRC program line by line. First, though, let's take a close look at the APW editor, so you can see how it works and how it is used in assembly language programming.

If you've programmed an Apple II or another microcomputer using other kinds of assembly language editors, one of the first things you may notice about the ZIP.SRC program is that it has no line numbers. The APW editor doesn't need them. Line numbers date back to the days of line-oriented editors, when programs were corrected a line at a time and lines were referred to by their line numbers. The APW editor doesn't have any use for line numbers because it is a screen-oriented editor, with a cursor that you move with arrow keys and cut-and-paste functions, which allow large blocks of text—not just

individual lines—to be copied, deleted, and moved. The APW editor operates similar to a full-featured word processor and is a remarkably sophisticated program editing system.

No Origin Directive

If you're an old hand at Apple II assembly language programming, another odd fact you may notice about listing 2–1 is that it has no origin directive. Almost every program ever written for a pre-GS Apple II begins with an origin directive, usually abbreviated `ORG`, that tells the assembler (and the programmer) where to load the program into memory. The APW assembler has an `ORG` directive and can use it to assemble programs designed to run in the Apple IIGS's 8-bit emulation mode. But Apple strongly advises that you not use the origin directive in programs written in native mode. When you write a native mode program for the IIGS, Apple suggests that you let the Memory Manager make all decisions about where to place programs in memory. If you ignore that advice and insist on placing programs in specific locations by using origin directives, you may interfere with the Memory Manager's operations and clobber other programs resident in memory.

Control Commands

Before you start typing the ZIP.SRC program, you may want to practice typing on the empty screen that appears before you now. As noted, you can use the arrow keys to move the cursor around the screen. You can also move the cursor using the spacebar, the Delete key, the Tab key, and the Return key, just as you would with a word processor.

To move the cursor more than one line up or down at a time, or to move it right or left more than one word at a time, hold down the ⍁ key on your keyboard while you press an arrow key. Pressing ⍁-Right arrow or ⍁-Left arrow moves the cursor right or left a word at a time. Pressing ⍁-Up arrow or ⍁-Down arrow moves the cursor to the top or bottom of your screen.

You can move the cursor to the beginning of a line by typing ⍁-< and to the end of a line by typing ⍁->. ⍁-1 moves the cursor to the top of a file, ⍁-9 moves the cursor to the bottom of a file, and ⍁-2 through ⍁-8 move the cursor to various points in-between.

Typing Control-T or ⍁-T deletes a line of text; typing Control-Z or ⍁-Z restores it. Control-W or ⍁-W deletes a word. Control-Z or ⍁-Z restores the last word deleted, if what you last deleted is a word and not a line.

To delete a block of text, press Control-X or ⍁-X and then use the arrow keys to highlight the block you want to delete. When the block is highlighted, you can delete it by pressing the Return key. Then you can move the cursor to another place in your program—or even to a program on another disk—and place the deleted block there by simply pressing Control-V or ⍁-V.

You can copy a block to another position or to another program by following the same procedure, but substituting Control-C or ⍁-C for the Control-X or ⍁-X that you use when you want a block deleted. Other control commands recognized by the APW editor are listed in table 2–1.

Table 2–1
APW Editor Commands

Function	Command
Beep the speaker	Control-G
Beginning of line	⌘-, ⌘-<
Bottom of screen/Page down	Control-⌘-J ⌘-Down arrow
Change	See *Search and replace*
Clear	See *Delete*
Copy	Control-C ⌘-C
Cursor down	Control-J Down arrow
Cursor left	Control-H Left arrow
Cursor right	Control-U Right arrow
Cursor up	Control-K Up arrow
Cut	Control-X ⌘-X
Define macros	⌘-Esc
Delete	⌘-Delete
Delete character	Control-F ⌘-F
Delete character left	Delete Control-D
Delete line	Control-T ⌘-T
Delete to end of line	Control-Y ⌘-Y
Delete word	Control-W ⌘-W
End of line	⌘-. ⌘->
End macro definition	Option-Esc
Enter escape mode	See *Turn on escape mode*
Execute macro	Option-letter key
Find	See *Search*
Insert line	Control-B ⌘-B
Insert space	⌘-spacebar
Paste	Control-V ⌘-V
Quit	Control-Q ⌘-Q
Quit macro definitions	Option

Table 2–1 (cont.)

Function	Command
Remove blanks	Control-R Ó-R
Repeat count	1 to 32,767
Return	Return Control-M
Screen moves	Ó-1 to Ó-9
Scroll down one line	Control-P Ó-P
Scroll up one line	Control-O Ó-O
Search down	Ó-L
Search up	Ó-K
Search and replace down	Ó-J
Search and replace up	Ó-H
Set and clear tabs	Ó-Tab Control-Ó-I
Start of line	Ó-, Ó-<
Tab	Tab Control-I
Tab left	Control-A Ó-A
Toggle auto indent mode	Ó-Return Ó-Enter Control-Ó-M
Toggle escape mode	Esc
Toggle insert mode	Control-E Ó-E
Toggle select mode	Control-Ó-X
Toggle wrap mode	Control-Ó-W
Top of screen/Page up	Control-Ó-K Ó-Up arrow
Turn on escape mode	Control-_
Undo delete	Control-Z Ó-Z
Word left	Ó-Left arrow Control-Ó-H
Word right	Ó-Right arrow Control-Ó-U

Examining the ZIP.SRC Program

After you're familiar with the operation of the APW editor, you're ready for the line-by-line examination of the ZIP.SRC program, beginning with the first line:

```
KEEP ZIP
```

Now what does that mean?

Assembler Directives

In source code written using the APW assembler-editor system, statements called *assembler directives* are often placed in the headings of programs, before the first lines of executable code. The line `KEEP ZIP` is such a directive. When the ZIP.SRC program is assembled, the `KEEP ZIP` directive tells the assembler to save the machine language version of the program as a file named ZIP. Because the source code version of the program is titled ZIP.SRC, there is no conflict between these two filenames.

The next line of the program:

```
LIST ON
```

is also an assembler directive. It is there because the APW assembler will not generate a listing when a program is assembled unless you tell it to. The `LIST ON` directive tells the assembler to produce a listing.

Program Segments

The next line of the program:

```
Zippy    START
```

is made up of two parts: a label and an assembler directive. The label is `Zippy` and the directive is `START`. We'll look at the `START` directive first.

The APW assembler, unlike most assemblers designed for small computers, generates programs divided into modules called *program segments*. The division of programs into segments greatly facilitates the writing of well-designed modular programs. Thanks to the use of program segments, a long complex program written with the APW system can consist of one small segment, or main loop, that calls other segments as needed. Furthermore, each segment can include a set of local variables used only in that segment—and the program can use a set of global variables recognized by every segment in the program.

Because local variables in an APW program have no effect outside the segments in which they appear, local variables in one segment can have the same names as local variables in another segment, without conflict. Even if a local variable is given the same name as a global variable, it will not cause a conflict; APW simply uses the local variable and ignores the global one.

Now turn your attention again to the line:

```
Zippy    START
```

As pointed out, this line consists of two parts: the label `Zippy` and the directive `START`. It marks the beginning of a program segment named `Zippy` and, in this case, also marks the beginning of the ZIP.SRC program. The

segment ends, as all APW program segments do, with the **END** directive. Because the ZIP.SRC program is only one segment long, the **END** directive also marks the end of the program.

In programs written using the APW assembler-editor system, every program segment begins with a line that includes **START** or a similar directive (**DATA** is used to begin data segments, for example), and every program ends with the **END** directive. When a **START** or **DATA** directive begins a segment, the directive must be preceded by a label that provides the segment's name.

Comments

The next two lines in the ZIP.SRC program are the first lines that contain executable code. They are

```
phk                             ; make program bank
plb                             ; and data bank the same
```

The abbreviations `phk` and `plb` are assembly language instructions, or *mnemonics*. The words that follow the semicolons in the right-hand column are *comments*, which are used like **REM** statements in BASIC programs. They are ignored by the APW assembler, but can provide valuable information to the next person who reads and tries to make sense of a program. (And that person could be you, because even people who write programs often find it difficult to figure out what they were trying to do after the ink on a program is dry.)

In programs written using the APW assembler, comments are usually preceded by semicolons, asterisks, or exclamation points. Asterisks and exclamation points are often used to identify remarks that take up a whole line. Semicolons must be used to set off comments that appear in the right-hand column of an APW source code program.

Stack Operations

Now back to the program in progress. The mnemonics `phk` and `plb` are often encountered in the initialization sections of IIGS assembly language programs. They set up two internal registers in the 65C816—the data bank register and the program bank register—so that both registers point to the same bank of memory. We won't cover the memory architecture of the IIGS until chapter 4, and the internal registers of the 65C816 aren't introduced until chapter 5. For now, it's sufficient to note that placing data used by a program and the program itself in the same memory bank simplifies matters greatly for the 65C816 processor when the program is assembled and run.

The `phk` and `plb` mnemonics belong to a category of instructions called *stack operations* because they manipulate a special area of memory called the *stack*. In assembly language jargon, a stack is an area of memory in which data is stored temporarily in the order last-in, first-out, abbreviated LIFO. A stack is sometimes compared with a spring-mounted stack of plates in a cafeteria. When a plate is placed on top of the stack, it covers up the plate that was previously on top, and it must be removed before the next plate can again be accessed.

In 65C816 assembly language, the `phk` instruction means *push the program bank register on the stack*, and the `plb` instruction means *pull the*

data bank register off the stack. When you use these two instructions together, they transfer the contents of the program bank register into the data bank register, using the stack as a temporary storage area for the data being transferred. This roundabout procedure is used because there is no 65C816 instruction for accomplishing the transfer more directly. More details about the stack—and about the `phk` and `plb` mnemonics—are presented in chapters 5 and 6.

Now let's move on to the next two lines of the ZIP.SRC program:

```
pea  testmsg|-16          ; push msg bank on stack
pea  testmsg              ; push msg address on stack
```

The `pea` mnemonic, like the `phk` and `plb` instructions, is a stack operation. It means *push effective address.* In the ZIP.SRC program, it pushes the address of a text message onto the stack so that the message can be displayed on the screen. The address being pushed on the stack is the starting address of a string called `testmsg`. That string appears, along with an identifying label, in the last line of the program:

```
testmsg # dc c'Are we programming yet?',h'00'
```

The rather cryptic formatting of this line is discussed in a few moments, when we get to the end of the program. First, though, look again at the two lines that push the address of `testmsg` onto the stack.

In chapter 5, you'll see why the `pea` instruction has to be used twice to push the address of the `testmsg` string onto the stack. Briefly, though, this is the reason. Because the 65C816 is a 16-bit chip, it can perform manipulations on pieces of data up to 16 bits long. But because it has a 24-bit data bus, it can access addresses that are up to 24 bits long. So it takes two operations to push an address onto the stack: one to push the 8-bit bank number of the address and another to push the 16-bit remainder of the address. When a 24-bit address is pushed on the stack in this way, it must be pulled off the stack in a similar fashion, but in reverse order. If you don't quite understand this, don't worry. Stack operations are covered in more detail in chapter 6.

Operands

Now you're ready to take a look at the operands used by the `pea` mnemonic in these same two lines:

```
pea  testmsg|-16          ; push msg bank on stack
pea  testmsg              ; push msg address on stack
```

As you have seen, the `testmsg` operand is a label that identifies a text string. In the ZIP.SRC program, `testmsg|-16` means *the first 16 bits of the address of the testmsg string.* For reasons that become clearer in chapters 4 and 5, the first 16 bits of the address of the `testmsg` string hold the bank number of the address. So, in the ZIP.SRC program, the statement `pea testmsg|-16` pushes the bank number of the address in ques-

tion onto the stack. Then the statement `pea testmsg` pushes the rest of the address.

The next two lines print the string labeled `testmsg` on the screen:

```
ldx #$200C                    ; put tool no. in x reg
jsl $E10000                   ; long jump to tool dispatcher
```

To understand what these two lines do, you need to know something about how the Apple IIGS Toolbox works. The Toolbox isn't examined until chapter 7, but it wouldn't hurt to point out now that each tool in the Toolbox has a 2-byte identification number, and a program can call any tool in memory by using its identification number.

In the ZIP.SRC program, a utility called the *tool dispatcher* calls a tool with the identification number $200C. Tool number $200C, as you can verify by looking at the list of IIGS tools presented in appendix B, is a tool called `WriteCString`. The `WriteCString` call is part of the Text Tool Set. It can be used to print a C-style string (a string ending in $00) on a text output device such as a printer or a monitor screen.

Using the Tool Dispatcher

The ZIP.SRC program uses the tool dispatcher to make the `WriteCString` call, which prints the string labeled `testmsg` on the screen. More information about tool calls is provided in chapter 7. For the moment, it's sufficient to note that the following steps must be taken to call a tool using the tool dispatcher:

1. Certain parameters (in this case the address of the string to be printed) must be pushed on the stack.
2. The identification number of the tool to be called must be placed in the 65C816's X register. (More information about the X register and the 65C816's other internal registers is presented in chapter 5.) In the ZIP.SRC program, the statement used to load `WriteCString`'s ID number into the X register is `ldx #$200C`.
3. The tool dispatcher must be called with the statement `jsl $E10000`, which means *jump to a subroutine located at memory address $E10000*. The `jsl` mnemonic, which stands for *jump to subroutine—long*, is often used in Apple IIGS programs to access subroutines that lie across bank boundaries.

   The last line of executable code in the ZIP.SRC program is

```
rtl                                   ; long return
```

The `rtl` mnemonic, which stands for *return from subroutine—long*, is used at the end of a subroutine (or the end of a program) that is called from across bank boundaries. This instruction is examined in greater detail in chapter 5.

Text in an Assembly Language Program

Now we have come to the line

```
testmsg # dc c'Are we programming yet?',h'00'
```

In this line, `testmsg` is a label that identifies the string that follows. The abbreviation `dc`, which comes next in the line, stands for *define constant* and means, obviously, a constant is being defined. The abbreviation `c`, which comes next, means a character string follows.

The text that follows `c` and is enclosed in single quotation marks is the string printed on the screen when you run the program. After the string is a comma, then the abbreviation `h`, which tells the assembler that the next value it encounters is a hexadecimal number.

The hex number that follows `h` is also enclosed in single quotation marks. The number is $00, the conventional terminator for C-style strings.

The last word in listing 2–1 is, appropriately enough

```
END
```

This ends the program segment labeled `Zippy` and also ends the ZIP.SRC program.

The APW Editor's Menu

When you finish typing the ZIP.SRC program, you can leave the APW editor by typing Control-Q. Your program disappears from the screen and is replaced by the APW editor's menu. By picking menu choice S, you can save the ZIP.SRC program under the filename you chose when you entered the editor (this filename appears at the top of the menu). Or, by selecting menu choice N, you can save it under a different name. After you save the program, you can choose menu selections to load another file, return to the editor (and to the program you just finished editing), or exit from the editor.

Assembling the ZIP.SRC Program

When you have typed the ZIP.SRC program and have made sure that it contains no mistakes, return to the APW shell by selecting menu choice E. You can then assemble and link your program by typing

```
ASML ZIP.SRC
```

You can then run it by typing:

```
ZIP
```

When the ZIP.SRC program prints its important question on the screen, you can answer it with a resounding yes!

CHAPTER 3

Programming the IIGS in C

And Learning More About the APW Development System

If you want to learn how to program the Apple IIGS in C, this is the chapter you've been waiting for. Even if you are interested only in assembly language, it is strongly suggested that you read this chapter because it contains valuable information about the APW system that you won't find elsewhere in this book.

It's important to note, however, that this chapter does not teach you C programming from the ground up. If that's what you need, you'll have to supplement this book with an introductory text on C programming. (A few are listed in the Bibliography.) But even if you've never written a line of C code, you are still invited to type, compile, and run the two sample programs in this chapter.

If you're an experienced C programmer, you'll be ready to write C programs for the IIGS when you finish this chapter. If you're new to C, you'll get some hands-on experience in writing simple C programs using the Apple Programmer's Workshop, plus a basic understanding of how things are done in C. If you know a little about C and are interested in learning more, this chapter and the information on C in the rest of this book provide a general understanding of how the language works and how it fits into the Apple IIGS programming environment.

The C Language

Before you start programming in C, we'll present some historical and technical information about the language. The C language was invented by Dennis Richie of Bell Laboratories and was originally designed for developing applications and utilities in the UNIX environment. Since then, it has become popular among professional and amateur programmers as a general-purpose language. C programs have been written for virtually every kind of microcomputer, minicomputer, and mainframe computer. Apple recognized C's usefulness and popularity by making it the first high-level language for the Apple IIGS.

C is successful because it offers a balance between the programmer-friendly features of a high-level language and the speed and versatility of assembly language. It is almost (though not quite) as easy to work in as a high-level language such as Pascal. Yet it offers the kind of unrestricted access to the IIGS's memory, operating system, and I/O functions that is otherwise available only in assembly language.

Structure of a C Program

A C program is a collection of functions, or sets of instructions for performing specific tasks. Information to be processed in a C program is passed to a function with a *parameter list*. A parameter list is a list of values, separated by commas and all contained between parentheses, that follows the function's name. The parameter list doesn't have to contain any parameters. But if there are no parameters, the name of the function must still be followed by a pair of parentheses, like this:

```
function()
```

Parentheses are not the only punctuation marks you'll find in a C program. C uses the semicolon as a separator between statements in a program and uses braces to group statements into blocks.

Any C expression that has a value can be used as a parameter in a parameter list. A C function usually returns a value as its result. So a function itself can be used as a parameter or as an argument to another function.

The value returned by a function does not have to be used by the program in which the function appears. A function can also perform other actions called side effects. Many C functions are used only for their side effects.

Important Features

C provides several ways to make decisions, perform looping operations, and assign and store data. In addition, a number of preprocessor (or compiler) directives facilitate the development of large programs and provide easy access to commonly used code and definitions. APW C also supports enumerated types, and assignments and comparisons between structured variables of the

same type. If you're an experienced C programmer, you'll understand this. If not, these and other features of APW C are explored in the programs in this chapter and the rest of this book.

C in the APW Environment

The Apple Programmer's Workshop (APW), an Apple product, is the development system used to write the C programs in this book. In addition to the standard integer arithmetic offered by most C development systems, the APW system also supports floating-point math. And, along with the standard C libraries—which provide some compatibility with C code developed using other systems—APW C also has a large set of interface libraries to support the Apple IIGS Toolbox. These libraries contain a complete set of function declarations, along with definitions of constants and data structures, that are designed to be used with the IIGS Toolbox. This means you can access the Toolbox directly from C as well as from programs written in assembly language.

Pascal Functions

One noteworthy feature of APW C is that you can define Pascal-style functions. Pascal functions make it possible to use the calling and parameter-passing conventions of Pascal in a C program. Many Toolbox routines were developed using Pascal-style conventions, and APW C's Pascal function type makes it possible to use them. Pascal functions also allow routines written in Pascal and linked with a C program to be called from C.

A Limitation of APW C

As any C buff will tell you, you can generally do anything in C that you can do in assembly language. In APW C, however, there is a major exception because the 65C816 chip has a "split personality."

As you saw in previous chapters, the 65C816 has a native (16-bit) mode that takes advantage of 16-bit registers and data paths and a 6502/65C02 emulation (8-bit) mode that emulates earlier members of the 6502 family. Emulation mode enables the IIGS to run most software designed for earlier Apple IIs. It also allows assembly language programmers to create and assemble programs that are compatible with earlier machines.

But APW C is strictly a native mode language; you can't use it to write programs in 8-bit emulation mode. Even when it's used to write native mode programs, sometimes its inability to deal with 8-bit machine code is a limitation. In most applications, though, this is not a problem. The APW C compiler also supplies an inline assembler that allows the programmer to insert assembly language code directly into C functions.

When it comes to creating native mode applications for the IIGS—complete with windows, menus, desk accessories, color graphics, and sound—APW C is a powerful and efficient tool.

Installing APW C

If you typed, assembled, and executed the assembly language program in chapter 2, you should have no trouble getting used to the APW C development system. When you work with the C programs, you'll use the same editor that you used in chapter 2. When you compile and link them, you'll use similar APW commands.

In a moment, you'll fire up your APW development system and start writing programs in C. First, though, it must be pointed out that the following instructions apply to a version of APW that may no longer be current by the time you read these words.

As explained in chapter 2, the APW system used to write the programs in this book is a text-based utility that does not make use of the IIGS's sophisticated graphics interface and event-driven programming capabilities. If APW has been completely overhauled by the time you read this, some of the details in the next few paragraphs may not apply to your APW system. But most of the information that follows should prove helpful, even if APW has been modified.

Using APW C with a Hard Disk

Adding C to the APW environment is simple if you have a hard disk. Simply start up the APW shell on your hard disk, insert the /APWC floppy in a 3.5-inch disk drive, and type this line following APW's # prompt:

```
copy /apwc/languages/= 5
```

Then type

```
copy /apwc/libraries/= 2
```

Using APW C Without a Hard Disk

If you don't have a hard disk, the previous method won't work because there is not enough room on one 3.5-inch disk for both a C and an assembly language APW package. One way to deal with this problem is to copy one or more of the large directories in the APW system onto another floppy disk or onto a RAM disk. Then set APW's shell prefixes so they look for the transferred files in their new locations.

You can also set up two stripped-down versions of APW—one for assembly language and one for C—so that you can put a fairly complete assembly language development system on one floppy and a fairly complete C development system on the other. They won't be on the same disk, however.

If you want to work in both C and assembly language using two floppy disks, here is a relatively painless way to get started:

1. Back up your original APW disks, store them in a safe place, and use your backup copies to conduct the following operations.
2. Start up the computer using a copy of the APW disk. Start APW from your finder disk.

3. Insert a copy of /APWC in your second drive and type the following commands (not the # prompts, just what follows them):

```
#copy  2/=  /apwc/libraries
#delete -c 2/ainclude/=
#copy /apwc/languages/=   5
#delete -c  /apwc/languages/=
#delete  /apwc/languages
#prefix 2  /apwc/libraries
```

These commands set up the APW assembler and compiler on one disk, and the C and assembler support libraries on another.

If you are planning to use this configuration regularly, you can tailor the APW LOGIN file (an exec file that calls APW when the APW disk is booted) so that everything is ready to go as soon as you boot up. To edit the LOGIN file, simply type this line following APW's # prompt:

```
edit 4/login
```

When the editor comes up, add this line to the end of the LOGIN file:

```
prefix 2 /apwc/libraries
```

To save your amended LOGIN file, press Control-Q to leave the editor, then make menu choices S and E. Each time you want to use APW, make sure the modified copy of /APWC is in one of your disk drives when you load APW.SYS16 from the IIGS finder or (on older system disks) the IIGS launcher.

After you've used APW C for a while, you may find many files on the /APWC disk you can do without. You may want to create a custom configuration that can save you even more disk space—and time.

The Language Barrier

As mentioned previously, C programs for the IIGS are created using the APW editor. They are compiled using commands—such as `compile` and `assemble`—that can also be used with the assembler.

To create a C program using the editor, however, you first must set APW's language to C. You can do this by simply typing the following command after APW's # prompt:

```
cc
```

After you use the `cc` command, any new files you create using the editor are recognized by the APW system as C language source files. APW compiles them using the C compiler when you issue a `compile` command. If you work mostly in C, you can use the editor to add the `cc` command to your LOGIN file. The editor then makes all new files C language source files.

Writing a C Program

Now, at last, you're ready to write a program in C. To begin, start up the editor with a new filename:

```
#edit myprog.c
```

C source files written under APW do not require the c suffix. But it is a good idea to use the c suffix because it distinguishes C source files from other kinds of files and makes them easy to spot when you catalog your directory.

When your editor comes up, you can type in a C program like you would type in an assembly language program. Some tips are provided in chapter 2. However, APW C programs, unlike APW assembly language programs, are standard-looking pieces of code. In fact, as long as they use the IIGS's standard text input and output mode, and don't require the use of graphics calls in the IIGS Toolbox, they look just like C programs written for any other machine.

For example, type in listing 3–1, the Hello World program found in so many texts on C.

Listing 3–1
Hello World program

```
main()
{
    printf("Hello World!\n")
}
```

When you've typed the program, you can leave the editor by pressing Control-Q. Then choose menu selection S to save your work and menu selection E to return to the APW shell's familiar # prompt.

Next, look at the directory of the current disk to make sure myprog.c was saved as a C language source file. To list the program, type, after APW's # prompt:

```
cat myprog.c
```

APW shows you a screen display like the one in figure 3–1.

Note that the last item on the second line in figure 3–1, under the heading

```
Name      Type   Blocks   Modified          Created           Access   Subtype
MYPROG    SRC         1   9 Jun 87 20:30    9 Jun 87 20:30    DNBWR    CC
```

Figure 3–1
Cataloging a single file

Subtype, is CC. That shows myprog.c has indeed been saved as a C language source file. If there's something else under Subtype in your disk directory, you probably didn't use the `cc` command before you made the new file. In this case, type the following line to change the subtype of myprog.c before compiling it:

```
change myprog.c cc
```

Compiling a C Program

After you save a C source file and exit the editor, you can compile the file by typing a line like this:

```
#compile myprog.c keep=myprog
```

The `compile` command in the previous line means exactly the same thing as APW's `assemble` command. You can use either one, in C or in assembly language, because the shell looks at the source file's language to decide whether to invoke the C compiler or the assembler. The `keep` directive in the command line tells the compiler to create an object file named myprog.root in the current directory. Any valid full or partial pathname can be used as the value of the `keep` command.

Linking a C Program

When you wrote an assembly language program in chapter 2, you assembled and linked it using the command `ASML`, which means *assemble and link.* And when APW received that command, it assembled and linked the program automatically. To create an executable C file, however, you must invoke the linker by specifically using a `link` command.

Before we link our Hello World program, it might be helpful to explain how the APW linker works. All APW assemblers and compliers, including the APW C compiler, generate object code files that have the same format. This format is called *object module format,* or OMF. To the linker, it doesn't matter whether a program was written in C, assembly language, or Pascal. In fact, because all assembled and compiled APW files have the same format, the APW linker can link object files written in any combination of development languages available under APW.

From an object module file created by the APW assembler or C compiler, the linker generates a load file, a file the system loader can load into memory. If necessary, the linker resolves any external references (references to segments of machine code outside the OMF file it is linking) and creates relocation dictionaries that the system loader uses later, at load time, to relocate the load file produced by the linker.

To instruct the linker to link an OMF file and produce a load file, type a command line like this:

```
link 2/start myprog keep=myprog
```

There are a few points about this command line that haven't been explained. But go ahead and type and enter the line, and after you link and run the program, we'll do the explaining.

Linking a C program can take a while, but when it's done you'll see the # prompt in its usual place, waiting for your next command. Then you can run the program you have linked by simply typing

```
myprog
```

followed by a carriage return. The greeting "Hello World!" is printed on your screen. The # prompt then appears on the next line, letting you know that myprog has finished executing and you can enter another command.

Now let's go back for another look at the line you typed to link the myprog program:

```
link 2/start myprog keep=myprog
```

To understand what the more cryptic parts of this line mean, it helps to know something about how C programs work.

Part of what makes programs like Hello World so much shorter and easier to write in C than in assembly language is that the compiler takes care of many details. For example, you don't need to worry about whether to use `jsl` or `jsr` when calling a subroutine, what to do with values placed on the stack, how many words to take off the stack, or what addressing mode to use. The compiler knows how to do all this. But it doesn't know anything about how to start or end a program, or how to read input from the keyboard or print to the screen.

The secret behind the brevity of the Hello World program (it is condensed into one line of code) is the existence of C libraries, which include a number of useful programs. Here's how a few of them work.

START.ROOT File

If you look in the LIBRARIES subdirectory of your /APWC disk, you'll see a file called START.ROOT and another file called CLIB. START.ROOT is the object code of an assembly language program on the /APWC disk. Typing `start` following the `link` command links the code in START.ROOT to your program.

When you link a C program, it is first linked to START.ROOT. When you execute a C program, the function named `main()` is called as a subroutine from a machine language program. And START.ROOT is that program. START.ROOT calls `main()` using the machine language equivalent of a `jsl` instruction. The program then returns to START.ROOT using the machine language equivalent of an `rtl` instruction, which is placed at the end of `main()` by the compiler. Details of how the `jsl` and `rtl` instructions work are in appendix A.

When the START.ROOT program is called, it does whatever is necessary to start up a C program. It also handles any arguments typed on the

command line so that they are accessible through the C input parameters `argc` and `argv`, if applicable. It then carries out the machine language equivalent of the assembly language statement `jsl main()`, which causes the machine code generated by the C program labeled `main()` to be executed.

At the end of the C function `main()` (which, as its name indicates, is always the main function in a C program), START.ROOT encounters the machine language equivalent of an `rtl` instruction—which, as noted, is placed there by the compiler. This instruction returns control to START.ROOT, which then takes care of returning to the shell prompt # or, if you launched your program from the finder or the program launcher, to one of those utilities.

Another Look at Link

Now let's review again the line that you typed to link the Hello World program:

```
link 2/start myprog keep=myprog
```

In this line, the names listed after `link` are pathnames—they can be full or partial pathnames—that tell the linker where to find the object files that make up your program. When C programs are linked, there are always at least two such pathnames in the `link` command line. The linker automatically looks for files with the suffix ROOT, so there's no need to include the ROOT suffix in your filenames. The 2/ prefix in `2/start` refers to the LIBRARIES subdirectory.

The `keep` directive, as noted, tells the linker where to send its output. Again, you can specify any legal pathname. Typically, an executable file is given the same name as its corresponding source code and object code files. Because executable files, by convention, do not have a suffix, the linker creates a load file called simply myprog.

CLIB File

Now you're ready to examine CLIB, another important file in the LIBRARIES directory on the /APWC disk. As you've seen, the START.ROOT program takes care of initializing and ending C programs, relieving that burden from the programmer. And, as you may notice when you look at the code for the Hello World program, C also relieves the programmer of such chores as reading inputs from the keyboard and printing characters on the screen. These details, as well as those needed for other kinds of input and output operations, are provided by the CLIB file.

The CLIB file is a special file created by the MAKELIB program. (MAKELIB is in the UTILITIES directory on the /APWC disk.) CLIB is made up of object files containing routines, most of which are written in C, that take care of many common programming actions in a standard manner.

To understand how the CLIB file works, look at how it was used when you compiled the myprog.c program. When the C compiler compiled the program, it didn't know anything about how to print on the screen. It also didn't know anything about CLIB. It created a storage area containing the ASCII codes for the message "Hello World," generated code to put the address of that storage area on the stack, then tried to generate a line of machine language code that would carry out the C statement

```
printf("Hello World!\n")
```

To create a machine language statement that would execute a `printf` function, the compiler generated an object code statement equivalent to the assembly language statement `jsl printf`. Then the `jsl` instruction was converted into a machine language opcode. But the `printf` instruction remained the same because the compiler didn't know what it meant. In other words, the compiler treated `printf` as a *symbolic reference.*

Symbolic References

In assembly language jargon, a symbolic reference is another name for a label that identifies a program segment—that is, a segment of code or data that begins with the `start` directive. In C, a compiler generates a symbolic reference to identify the location of a function or variable.

The APW linker treats symbolic references in the same way in C and assembly programs. In both, one of the jobs of the APW linker is to resolve symbolic references. When the linker encounters a symbolic reference in a program being linked, it first scans each program listed on the `link` command line to see if it contains the reference in question. If it doesn't find the segment there, it searches for it in any files that appear in the LIBRARIES subdirectory and have the file type LIB.

When the linker linked the Hello World program, there were no other filenames on the `link` command line. So, when it encountered the C function `printf`, it went directly to the LIBRARIES directory and searched for it there.

Finally, in the CLIB file, the linker found what it was looking for: a code segment labeled `printf`. It added that segment to the executable file it was creating. Then the linker replaced the symbolic reference operand of the `jsl printf` statement with a value marking the location of the start of the `printf` routine in relation to the beginning of the load file being created by the linker.

Standard C Libraries

The analysis of the `printf` function has served as an introduction to a useful set of prewritten C functions called standard C libraries. These libraries, stored in the CLIB file, include more than 40 routines. Most of the routines emulate the behavior of the standard C routines available in UNIX systems. Many of them deal with various aspects of input and output, such as file handling, reading the keyboard, and printing text. In addition to I/O routines, there are mathematic routines, such as sine and cosine functions, and memory allocation routines, such as `malloc`, `calloc`, and `free`. The routines are explained in chapter 5 of the *Apple IIGS Programmer's Workshop C Reference.*

CLIB also contains routines that are not called directly from C programs. These provide an interface with the SANE floating-point math routines in the IIGS Toolbox. When you include floating-point arithmetic expressions in your C code, the C compiler generates calls to these SANE interfaces to perform the calculations. Much of the functionality of the standard C libraries can also be achieved by making direct calls to the tools in the Toolbox and to ProDOS. In fact, standard C libraries make extensive use of routines in the Text Tool Set and ProDOS for text I/O and file handling. The standard C

libraries not only simplify your work, they also make it possible to port C source code written for other machines over to the IIGS.

Another Sample Program: The Name Game

Now that you've typed and run a very simple C program and understand how to create a C program, you're ready to write a slightly more complex program. The name of the program is the Name Game. It was written in 1980, in BASIC. Since then it has been translated into five programming languages and has appeared in various forms in more than a dozen books and magazine articles. It will also turn up, in an assembly language version, in chapter 7.

Now you're ready to type, compile, execute, and analyze the Name Game. Load APW and type this line following the # prompt:

```
edit namegame.c
```

When the editor comes up, you can type in the Name Game program, which appears in listing 3–2.

Listing 3–2
Name Game program (C version)

```
#include <stdio.h>

main()
{
char replay = 'Y';
char name[25];

  while ((replay == 'Y')||(replay == 'y')){
    putchar(0x8C);
    printf("**** The Name Game ****\n\n\n");
    printf("Hello, what's your name? ");
    scanf("%24s",name);
    fflush(stdin);

    while(strcmp(name,"George")&&strcmp(name,"george")&&
strcmp(name,"GEORGE")) {
       printf("\nGo away %s, bring me George!\n\n",name);
       printf("What is your name? ");
       scanf("%24s",name);
       fflush(stdin);
    }

   printf("\nHi George! Try again? (Y/N) ");
   replay = getchar();
   }
```

When you have finished typing and correcting the program, press Control-Q, select S and E to leave the editor, save your work, and return to the shell command line.

Compiling the Name Game

You can compile the Name Game by typing the command line

```
#compile namegame.c keep=namegame
```

If you typed in the program exactly as shown in listing 3–2, the compiler generates a screen display that looks like the one in figure 3–2.

Now type a `cat` command, like this:

```
#cat namegame=
```

Your disk directory includes a new file called NAMEGAME.ROOT.

If you made any mistakes in typing the program, the compiler presents a list of error messages. If there are any error messages on the screen, they contain the numbers of the lines in which errors occurred. If the compiler has found errors, enter the editor and compare the lines you typed with the lines in listing 3–2. Then leave the editor, save your changes, and compile the program again.

If you made so many errors that the first one scrolls off the screen (and that's easy to do, because one error in a C program can cause the compiler to generate many error messages), use the APW shell's redirection capability to save the compiler's error messages in a file. Or, if you have a printer hooked up, send them to the printer.

To redirect the compiler's error messages to a special error file, just type this command:

```
#compile namegame.c keep=namegame >errors
```

Then, to view your file of error messages, you can type

```
                    Apple IIGS APW C Compiler
                            V1.0B7
      Copyright Apple Computer Inc. and Megamax, Inc. 1986, 1987
                        All Rights Reserved

FNAME='/RAM1/NAMEGAME.C'PARMS=''LANG=''DFILE='/RAM1/NAMEGAME'
Compiling /RAM1/NAMEGAME.C
Exit
_exit(0)
```

Figure 3–2
APW C compiler's screen display

```
#type errors
```

While APW is printing your error file on the screen, you can stop the display from scrolling by pressing a key. You can resume scrolling by pressing another key.

To redirect your error file to the printer, type

```
#compile namegame.c keep=namegame >.printer
```

Then, if the printer is hooked up and online, you'll get a paper copy of the compiler's output.

Even if you didn't make any errors in typing the Name Game program, you might like to try these exercises in file redirection, just to see how they work. They will come in handy eventually.

Linking the Program

To link the Name Game, type the command line

```
#link 2/start namegame keep=namegame
```

This line works like the line that linked the Hello World program. It creates a load file called NAMEGAME in the current directory. If the linker displays an error message, you'll have to activate the editor, correct the errors, and compile and link the program again.

If the linker finds any errors in your program, it will probably present a display similar to the one shown in figure 3–3.

The error shown in figure 3–3 was caused by the misspelling of a subroutine's name. In the example, the *f* was not included in the function name `printf` somewhere in the program.

If all goes well and you don't get an error message, you can now run the Name Game by simply typing the command

```
#namegame
```

```
Link Editor V1.0 B5.1

Pass1:  ..........................................................
Pass2:  ...
Error at 0000013B past main PC = 00000275 : Unresolved reference
Label: print.....................................................

1 errors found during link.
8 was the highest error level.

    There are 3 segments, for a combined length of $00006F71 bytes.
```

Figure 3–3
An error message from the linker

Playing the Name Game

Because the Name Game is a game, please read no further until you play it! Then come back and look at the following play-by-play listing of what should happen when you play the game.

1. The screen clears, and the title **** *THE NAME GAME* **** appears at the top of the screen.
2. The greeting *Hello, what's your name?* appears three lines below the title.
3. As you type in your name, the letters you type appear after the ? prompt.
4. If you don't type George, george, or GEORGE, the computer responds:

```
Go away the name you typed in, bring me George!
What is your name?
```

5. Steps 3 and 4 repeat until you type George.
6. When you finally give up and type George, the computer responds:

```
Hi George! Try again? (Y/N)
```

7. If you type Y or y, the computer starts the Name Game over again, beginning with step 1. If you type anything else, you return to the shell's # prompt.

The Art of Debugging

If the program doesn't work in the manner described, you probably didn't type it exactly as shown in listing 3–2. Unfortunately, no compiler or linker can spot and report every type of error that can be made in a program. Here are a few types of errors that may not be noticed by the APW system:

- Misspellings.
- Discrepancies in the layout of a screen display.
- The program won't print *Hi George!* even if you type in George or keeps playing even after you type N. If one of these problems occurs, press Control-Ó-Reset (at the same time) to reboot the machine.
- After performing all, part, or none of the steps listed in the play-by-play description, the machine just freezes. You'll have to reboot for this one, too.

In programs that you write, errors like the last two are usually the hardest to find. In such cases, all you can do is carefully go over your code until you find your error. Then, each time you find an error and track down its cause, it's a good idea to think for a moment about why the error occurred.

When you start debugging your programs, you'll have to think in reverse. You'll need to figure out what kind of mistake was likely to cause a

certain problem before you even know where to look in your source code! This process is called *debugging*, and it's an important part of programming—in any language.

How the Name Game Works

If your Name Game program is debugged and running, you're ready for a line-by-line description of how it works. Let's start at the top:

```
#include <stdio.h>
```

The term `#include` is a compiler directive, and the `#include` directive is a standard feature of C. The `#include` directive replaces the line the directive is on with the contents of the named source file. The < and > around the filename tell the compiler to search for the filename in the 2/CINCLUDE directory.

Macros

The Name Game program needs the contents of the <stdio.h> file because they provide definitions for the `putchar` and `getchar` macros. Macros are often found in Apple IIGS programs written in both assembly language and C. When they are included in C programs, they are used like the functions in the CLIB file. In the Name Game program, for example, the `putchar` and `getchar` macros read each character input from the keyboard and print every character displayed on the screen.

Macros, though they may look obscure to the uninitiated, are time-saving and labor-saving aids for assembly language and C programmers. A macro makes it possible to write a complex sequence of code using a single word or a word followed by one or more symbolic variables. When the program is compiled, the macro is replaced by the code it represents.

Macros are often used when the actual code for a frequently performed action is obscure. So they not only save programming time, but also make code more readable. In C programs, macros are more efficient than function calls because the code replacement they require is handled at compile time, and `jsl` and `rtl` instructions are not required. Also, symbolic variables can be used more easily in macros than in subroutines.

Macros do have one disadvantage, however. When a macro is used repeatedly in a program, it uses much more memory than if it were written as a subroutine. A macro is replaced by the sequence of code it calls every time it is used, but a subroutine can be used over and over without using any additional memory.

More information about macros is presented in part 2. For now, all you need to know about macros is that if you didn't include <`stdio.h`> in the heading of the Name Game program, `putchar` and `getchar` wouldn't work. The fact that macros are implemented in a slightly different manner than true

function calls is not too important at the moment and is mostly transparent to the programmer.

The Main() Attraction

Now let's move to the next line in the Name Game program:

```
main()
```

As noted, every C program must have a function called `main()`. For example, in the description of the START.ROOT routine, `main()` is the label the routine jumped to.

To the C compiler, `main()` is just another function definition and is treated the same as any other. When the compiler compiles a `main()` function, it simply generates an OMF file segment whose start is labeled `main`. To create this segment, it uses all the code between the first and last braces that follow the `main()` declaration. Often, in longer C programs, the `main()` function consists almost entirely of calls to other functions. (A general rule for beginning C programmers is to avoid writing any C function that is too long to fit on the computer screen at one time. If you follow this rule, it reduces your chances of writing convoluted, hard-to-understand "spaghetti code.")

A Prompt and a Response

Now on to the next line in the Name Game program:

```
char replay = 'Y';
```

This line is included in the program because you need a place to store the response to the *Try again? (Y/N)* prompt. Because you will store a letter, you declare the `replay` variable to be type char. The program ends whenever `replay` is not equal to `'Y'`, so you start out making `replay` equal to `'Y'` to ensure that the game is played the first time through. `'Y'` is a character constant. The single quotes around `Y` tell the compiler that it is not the name of a variable. C stores the ASCII value of the letter *Y* in the byte of memory it associates with the name `replay`.

Setting Up a String

Now for the line

```
char name[25];
```

This line is included in the program because you also need a place to store the name the user types in. The statement sets aside 25 bytes to hold the name. The identifier `name` refers to the address of the first byte in the string. The memory area addressed by the identifier `name` is an array of type char.

The While Statement

After the `name[]` array is set up, a line is skipped in the program, and this line appears:

```
while ((replay == 'Y')||(replay == 'y')){
```

The skipped line and the indentation before `while` are conventions that make C programs easier to read. (Refer to the complete program, listing 3–2, to see the indentation.) The compiler ignores them.

The statement itself has two parts. The first part—inside the parentheses that follow the word `while`—is a condition. The second is a block of code enclosed by braces. Only the first brace appears on the same line as `while`. The closing brace appears farther down in the program, preceding the closing brace of `main()`. When the program is run, it repeatedly executes the block of code between the braces as long as the condition inside the parentheses that follow the `while` statement is true. In this case, the block that is executed is the rest of the program.

Logical OR Operator

The `||` symbol in the `while` statement is C's logical OR operator. As long as the variable `replay` is equal to either *Y* or *y*, the `while` statement's condition is true, and the block that follows it is executed.

Both an uppercase *Y* and a lowercase *y* are used in the `while` statement because the C language is case sensitive—that is, it distinguishes between uppercase and lowercase letters. So, in C programs with inputs that are not case sensitive, you often need to write code that forces C to accept either uppercase or lowercase letters as inputs from the keyboard.

The next line in the program:

```
putchar(0x8C);
```

calls the `putchar` macro defined in the header file <stdio.h>. This line illustrates a fast way to send a single ASCII code to the program's output stream—in this case, the screen. If you wanted to print a single letter on the screen, the argument to `putchar` (the value inside the parentheses that follow the name of the function) would be the desired letter, enclosed in single quotation marks.

Because the Apple-style ASCII code to clear the screen is not a printable character, but the hexadecimal value $8C, you can just send the code number itself by omitting the single quotation marks. The 0x preceding the value 8C means 8C is a hexadecimal number. In C, hex constants are indicated by the prefix 0x. So `0x8C` represents the same value as $8C in assembly language.

The Name of the Name Game

The next line:

```
printf("**** THE NAME GAME ****\n\n\n");
```

calls the CLIB routine `printf`. In this case, the C compiler reserves a space in memory for the characters inside the quotation marks, stores them there with a terminating 0 (null character), and passes the address to the `printf` routine.

We'll discuss what the `printf` routine does in a moment. But first,

we'll describe the 0 that CLIB adds to the characters inside the quotation marks before `printf` goes into action.

In C, the word *string* describes an array of characters whose last value is 0. A 0 is called a null character because it does not represent any letter or control character. So 0 is used to mark the end of a string. It tells various C routines that work with strings when they have found the end of a string.

Now you can move to the `printf` routine. The C compiler interprets another special character—the backslash character (\)—as an escape character. Instead of placing a backslash in the stored string, it treats the character that follows it in a special way. For example, *n* following a backslash stands for newline, which in C talk means a carriage return. So the three \n's before the closing quotation marks in the line

```
printf("**** THE NAME GAME ****\n\n\n");
```

insert three newlines (carriage returns) in the string passed to `printf`. This means two lines are skipped before the next item is displayed on the screen.

In the next line

```
printf("Hello, what is your name? ");
```

you do not include \n because you want the player's answer to appear on the same line as the question.

The Scanf Routine

The `scanf` routine in the statement

```
scanf("%24s",name);
```

is another powerhouse from CLIB. It works like `printf`, but in reverse. It takes values of text data from the keyboard, echoes them to the screen as they are typed, and stores them in a designated variable or string.

In the `scanf` routine there are two arguments inside the parentheses, separated by a comma. The first argument, `%24s`, instructs `scanf` to read up to 24 characters from the keyboard and place them, in the order they are input, in a string (character array). The second argument, `name`, is the address of 25 bytes of storage. This tells `scanf` where to store the character string.

When the user types a carriage return or has input 24 characters, `scanf` stops accepting characters. If input is ended by a white space character—a space, tab, or newline character—`scanf` does not add it to the stored string. When input has ended, a 0 is placed at the end of the string of characters that have been typed in, making the array called `name` a C string. Control then returns to the next statement in the calling routine.

Counting Characters

In a `scanf` string like the one in the Name Game program, the `%` symbol preceding 24 limits the length of the string to 24 characters, plus the terminating 0 that makes it a C string. This is a total of 25 characters, which is the size of the character array `name`. If you allowed an unlimited number of input characters, `scanf` would blindly store every character the user enters

in the area of memory that begins with the first character of the array `name`. If more than 24 characters were input, the program could eventually crash or overwrite other data stored in memory.

Other values can follow `%` in a `scanf` argument to cause the function to read and store data in different ways. You can find more information on this topic in the *Apple IIGS Programmer's Workshop C Reference*.

The next three lines in the program are

```
printf("What is your name? ");
scanf("%24s",name);
fflush(stdin);
```

`stdin` is defined in <stdio.h>. It represents the standard input stream, which is normally the keyboard. `fflush` is a standard library call that removes any data "queued," or waiting to be read from or written to. The `scanf` call, which precedes `fflush` in the program, takes in whatever is typed up to, but not including, the first white space character typed. Sometimes, you will be interested in this character. In this case, you are not, so `fflush` disposes it.

If you left the `fflush` call out of the program, the next input request—the `getchar()` call near the end of the program—would accept the pending white space character as its input instead of waiting for the user's response.

A Loop Within a Loop

Now for the next statement in the Name Game program:

```
while(strcmp(name,"George")&&strcmp(name,"george")&&
strcmp(name,"GEORGE")){
```

You may notice that the `while` loop in this statement is on two lines. This was done simply because the statement is too long to fit on one line. C doesn't care about extra spaces and carriage returns in source code, as long as they are not within a name or between quotation marks.

Now let's see what the statement does. Although the program is already inside a `while` loop that recycles the Name Game as many times as users want, you can create another `while` loop that keeps users typing in entries until they decide to go get George (or lie and tell the computer that their name is George).

This loop within a loop introduces another new CLIB routine, `strcmp`. The `strcmp` function compares the C string `name` with the C string `George` and generates a value of 0 if the strings are the same. In C, 0 stands for the logical value false, and any nonzero value stands for true. Our goal is to repeat the `while` loop that asks for George as long as the character array `name` is different from three variations of the name `George`.

Because the result of `strcmp` is nonzero (true) when the string stored in `name` is different from the string stored in `George`, you use the logical

AND operator && to make the comparison. This says: "While name is different from George, AND name is different from george, AND name is different from GEORGE, carry out the following block of code." Otherwise, the program moves to the statement following the closing brace of the block:

```
printf("\nGo away, %s, bring me George!\n\n",name);
```

What's new here is that %s, the same term used in the scanf statement, is now used in a printf statement. In this case, it causes printf to print on the screen the string stored in name. This operation is the reverse of the one carried out by scanf, which replaces the contents of name with the string of characters typed at the keyboard. So in this context, you can think of the screen and the keyboard as the input and output sides of the same device.

These are the next two lines in the inner while loop:

```
printf("What is your name? ");
scanf("%24s",name);
```

In these two statements, printf prints a line on the screen and scanf places a new string in the variable name. There is nothing new here, but the results are important. The scanf statement provides a new value to be tested by the strcmp routine at the start of the loop. If this operation did not take place, even typing George would not help the poor users. They would have to reboot the machine to get it to stop its dialog.

This brings us to an important point in programming. When you write a while loop, something must eventually happen within the loop to make the condition being tested false and bring the loop to an end.

Now we come to the last statement in the inner while loop:

```
fflush(stdin);
```

After the printf and scanf routines are carried out, the fflush routine "flushes" the queue.

The end of the program's inner while loop is marked by a closing brace placed beneath the *w* that began the loop. This convention makes C code easier to read and understand.

When Your Name Is George

The next line is one you can't get to unless you claim your name is George:

```
printf("\nHi, George! Try again? (Y/N)");
```

At this point, you can decide whether you want to play the game again, though I can't think of why anyone would want to.

This line stores your reply in the variable replay:

```
replay = getchar();
```

The `getchar()` macro, which looks and works like an ordinary C function, simply returns the ASCII code for the next character typed at the keyboard. The statement in which it appears also makes it possible to end the program. If you type any character other than *Y* or *y*, the condition for the `while` loop near the beginning of the program is not met. As a result, the program passes control to the next statement after this block. But the only thing after the } that ends this `while` loop is the } that ends `main()`. The compiler places the `rtl` instruction at the end of the generated code, so execution continues with the next statement after `jsl main()` in START.ROOT. The result is a return to the shell's # prompt.

Making a Standalone Application

I hope you have now succeeded in getting the Name Game running. If you have, you're ready to turn it into a standalone application. But before you can do that, you'll have to tell the IIGS that your name is George, so that the Name Game will end and return to the APW shell. Then you can type the command line

```
#filetype namegame s16
```

This changes the file type of the Name Game from exe, a file type which can be executed only under APW, to s16, a file type that can be loaded from the IIGS finder (or, on older system disks, the IIGS launcher).

Now you can astound your friends by letting them play the Name Game. The program may not be impressive enough to put on the market. But with a little imagination—and some fancy graphics tricks you'll learn in this book—you'll soon be able to turn it into something more complex and more or less annoying than the original.

CHAPTER 4

Memory Magic

Mapping the Apple IIGS

The engineers who created the Apple IIGS accomplished a remarkable feat: they stuffed more than 9 megabytes of memory capacity into a computer originally designed to work with 48K of RAM. The secret of how they did it can be summed up in two words: bank switching.

Bank switching is based on the principle that two blocks of memory can share the same address as long as they don't try to use it at the same time. When a computer uses bank switching, blocks of memory are assigned identical addresses. Special switching facilities are provided so that memory segments that use the same addresses can be switched into and out of the space they share.

In the Apple IIc and the expanded Apple IIe, blocks of memory that use bank switching are controlled by special electronic circuits called *soft switches*. A soft switch is a microcomputer circuit that can be turned on and off, just like a light switch. You'll take a closer look at some of the soft switches built into Apple II computers later in this chapter. First, though, let's pause for a brief look at the memory architecture of microcomputers in general and the Apple IIGS in particular.

Memory Pages

The term *page* is often used in memory mapping. A page is simply a block of 256 bytes of memory, or $100 bytes in hex notation. It is a convenient

unit of memory measurement because the 256 memory addresses in a page can be expressed using the hex values $00 through $FF. For example, page 0 on the Apple II memory map is made up of memory addresses $00 through $FF, and page 1 includes memory addresses $100 through $1FF. The address at which a page number changes—for example, memory address $1FF, which is the last address on page 1—is known in assembly language as a page boundary.

(Incidentally, in Apple II graphics programming, the word *page* is also used to describe one screenful of graphics memory. These different uses of the same word should not be confused. You'll enounter graphics pages again later in this chapter.)

Memory Banks

Another important unit of memory measurement is a *bank*. A bank is a group of 256 pages, or a total of 65,536 (64K) banks of memory. The earliest models of the Apple II—the original Apple II and the Apple II+—have just one bank of memory, or a total of 64K. The Apple IIc (and the expanded Apple IIe) have two banks of memory, or 128K. A basic Apple IIGS, without a memory expansion card, has four banks of memory, or 256K. The IIGS's central processor, the 65C816, can address up to 256 banks, or 16 megabytes, of memory (that is, 16,384,000 bytes, or $FA0000 bytes in hex notation).

Because the 65C816 can address 16 megabytes of memory, the address space of the IIGS also totals 16 megabytes—at least in theory. Actually, however, only 8.25 megabytes of memory are available for RAM expansion, and 1 megabyte is available for ROM expansion. The IIGS also comes with four banks, or 256K, of RAM. Figure 4–1 is a simplified memory map of an unexpanded Apple IIGS, just as it comes out of the box: with 256K of RAM. (A memory map of a fully expanded IIGS is presented in figure 1–2.)

The Memory Manager

Until the advent of the IIGS, people who wrote an assembly language program for an Apple II had to decide exactly where in memory their program would be loaded. Then they had to make sure the program would work properly when it was assembled and loaded into the chosen locations. In other words, it was the programmer's responsibility to allocate and manage memory.

With the introduction of the IIGS, this situation changed dramatically. The IIGS, as mentioned in chapter 1, is equipped with an ultrasophisticated programming tool that takes all responsibility for memory management from the programmer. This tool, called the Memory Manager, can allocate blocks of memory, discard blocks of memory when they are no longer needed, and even rearrange blocks of memory so that available RAM space can be used more efficiently. If you use the Memory Manager—and Apple strongly advises that you do—you will never again have to decide where in memory to start a program or a data segment, and you will never again have to juggle

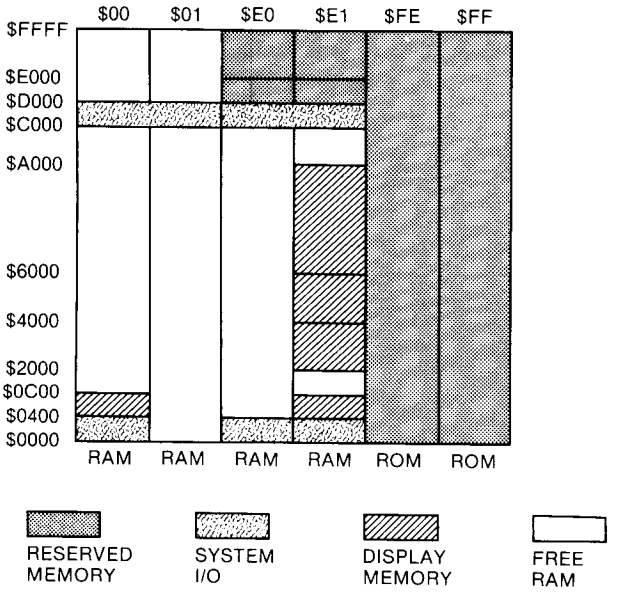


Figure 4–1
Memory map of an unexpanded Apple IIGS

blocks of memory so that they don't "bump" into each other. All those tasks—and virtually every kind of task that involves memory management—are now jobs for the IIGS Memory Manager.

But the IIGS programmer still needs to know something about the memory architecture of the computer. The IIGS has a lot of firmware (prewritten programs) installed in specific locations in ROM, and it is sometimes helpful to know where they are. It is also helpful to know where screen memory starts and ends, where color tables and other graphics-related data are stored, and where important I/O routines can be found.

Another good reason for understanding the memory architecture of the IIGS is that it is sometimes necessary to place user-written routines in bank 0, so that they can access firmware designed for pre-GS Apple IIs without moving across bank boundaries.

Now that you know why memory sometimes must be managed manually, let's take a closer look at the Memory Manager. The Memory Manager is built into ROM and goes to work automatically as soon as you turn on the computer. Every time you load an application program, a utility called the system loader (mentioned in chapter 1) calls the Memory Manager and requests memory space for the program. The loader then loads the program into memory at the address returned by the Memory Manager.

After an application program is running, it can summon the Memory Manager and request (or allocate) additional memory. It can also ask the Memory Manager to release (or deallocate) memory when it is no longer needed, and it can query the Memory Manager at any time to find out how much memory is available.

Managing Desk Accessories

The Memory Manager is so meticulous in its record keeping that it always knows which blocks of memory are in use, which programs are using them, and which blocks are free. So when the Memory Manager is active—and it always is—several programs can be present in memory at the same time (coresident), and you can switch back and forth among them at any time. This ability to handle several coresident programs is an important feature of the Memory Manager because it enables the IIGS to use desk accessories. Desk accessories are programs that can be loaded into memory once, then called up and used whenever desired, even while an application is running. Some accessories that can be handled in this way include clocks, calendars, calculators, and note pads.

The Memory Manager also makes it possible for a IIGS to be equipped with any amount of memory ranging from the standard 256K to 8.25 megabytes and for application programs to use the maximum amount of available memory in a way that is transparent to the user (and to the programmer as well).

APW and the Memory Manager

Because the Memory Manager is such an integral part of the IIGS, the APW assembler-editor and the APW C compiler are designed to work closely with the Memory Manager. When you use the APW assembler to write and assemble an assembly language program for the IIGS, you are advised not to assign the program a specific starting point in memory and not to use addressing modes that require literal addresses except when absolutely necessary.

When you follow Apple's guidelines for using the Memory Manager, the APW assembler automatically produces machine code that is relocatable and, therefore, can be handled easily by the Memory Manager. The Memory Manager can handle a relocatable program easily because it can load the program into any block of available RAM, and it can later move the program to another block if needed.

Pointers and Handles

To keep track of the IIGS's memory, the Memory Manager uses two important types of variables: pointers and handles. A pointer is a pair of memory addresses that contain, or point to, a second memory address. In C and assembly language programs, a pointer is a convenient tool for accessing a memory address because the block of memory can be changed by simply altering the addresses stored in the pointer. You examine how pointers work, and how they are used in Apple IIGS programs, in chapter 6. Figure 4–2 gives a rough idea of how a pointer is used in an assembly language program.

When the Memory Manager allocates a block of memory, it usually returns a handle rather than a pointer. A handle is a pair of memory addresses that point to a pointer, which in turn points to still another address. Because of the indirect way in which a handle is used, it is sometimes described as a pointer to a pointer. The use of handles is illustrated in figure 4–3.

The concept of a handle may sound obscure, but the Memory Manager has a good reason for using handles. The machine code produced by the APW assembler is relocatable and can therefore be shuffled around in memory at will by the Memory Manager. But even when a piece of machine code is

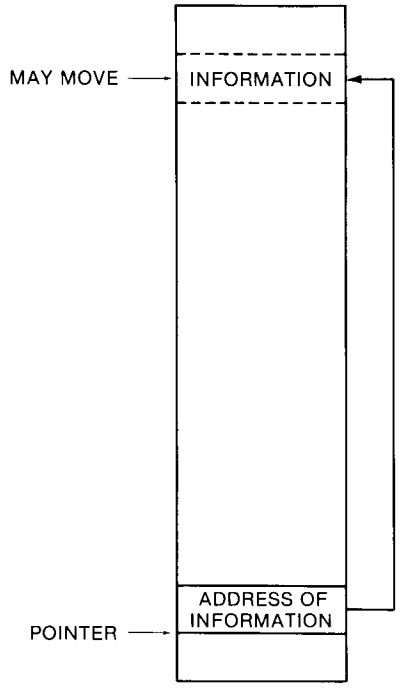


Figure 4–2
Using a pointer in an assembly language program

relocatable, moving it around in memory can still cause problems. For example, if a program contains a pointer and the code the pointer is supposed to access is moved, the pointer contains an invalid address and will almost certainly crash whatever program is running the next time it is used.

To keep this kind of disaster from occurring, the Memory Manager does not assign a pointer when it allocates a block of memory. Instead, it stores a pointer to the block in a non-relocatable table. The block's handle is the fixed address to this pointer. In other words, a handle is simply a 4-byte space in which the current address of a block is kept. As the block is moved, this pointer changes, but the correct pointer can always be found in the same place: the handle.

Using this procedure, the Memory Manager can always keep track of any block of code, and blocks of code can always access each other, no matter how many times their addresses change.

The IIGS Memory Map

Now that you've seen how the Memory Manager works, you are ready to examine the memory map of the IIGS in more detail. Refer back to figure 4–1, the simplified IIGS memory map at the beginning of this chapter.

As you learned in chapter 1, the IIGS's memory space can be divided into five major segments. Each of these segments can be subdivided into 64K

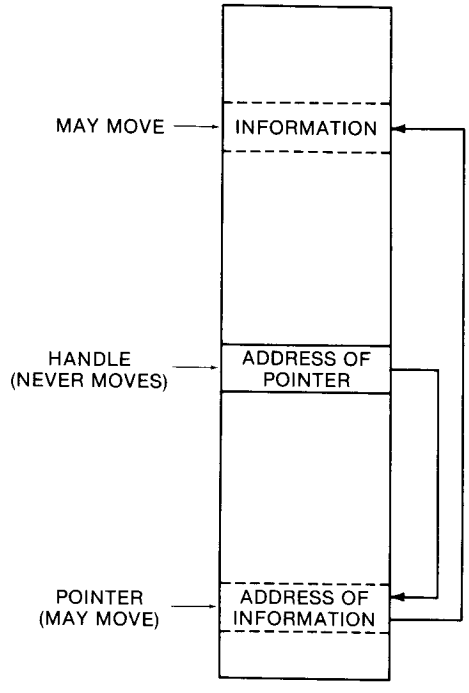


Figure 4–3
Using a handle in an assembly language program

memory banks. Here is an outline of what each block of memory in the IIGS contains:

- Banks $00 and $01 (memory addresses $000000 through $01FFFF) include both free RAM and system memory. When the IIGS is in Apple IIc/IIe emulation mode, the addresses in these two banks are the only addresses available.
- Banks $02 through $7F (memory addresses $020000 through $7FFFFF) are available for RAM expansion.
- Banks $E0 and $E1 (memory addresses $E00000 through $E1FFFF) include some free RAM, but are also used for system, input/output (I/O), and display memory.
- Banks $F0 through $FD (memory addresses $F00000 through $FDFFFF) are available for ROM expansion.
- Banks $FE and $FF (memory addresses $FE0000 through $FFFFFF) are used for system firmware.

A more detailed map of the Apple IIGS is presented later in this chapter.

Mapping the IIGS in Emulation Mode

As noted previously in this chapter and in chapter 1, the Apple IIGS can be used in two modes: Apple IIc/IIe emulation mode and native mode (that is, as a fully equipped Apple IIGS). In this section, you'll see how the memory of the IIGS is apportioned in emulation mode. Then you'll examine the computer's memory layout in native mode.

Figure 4–4 is a memory map of the Apple IIGS in Apple IIc/IIe emulation mode. In this mode, the IIGS operates as a 128K computer, and banks $00 and $01 are referred to as main memory and auxiliary memory—the same names they are known by in the Apple IIc and the expanded Apple IIe.

If you're familiar with Apple IIc or Apple IIe assembly language programming, the map in figure 4–4 will be familiar. If you're new to Apple II programming, though, a little map reading is in order. So let's pause for a closer look at what the various blocks of memory in figure 4–4 contain when the IIGS is in emulation mode.

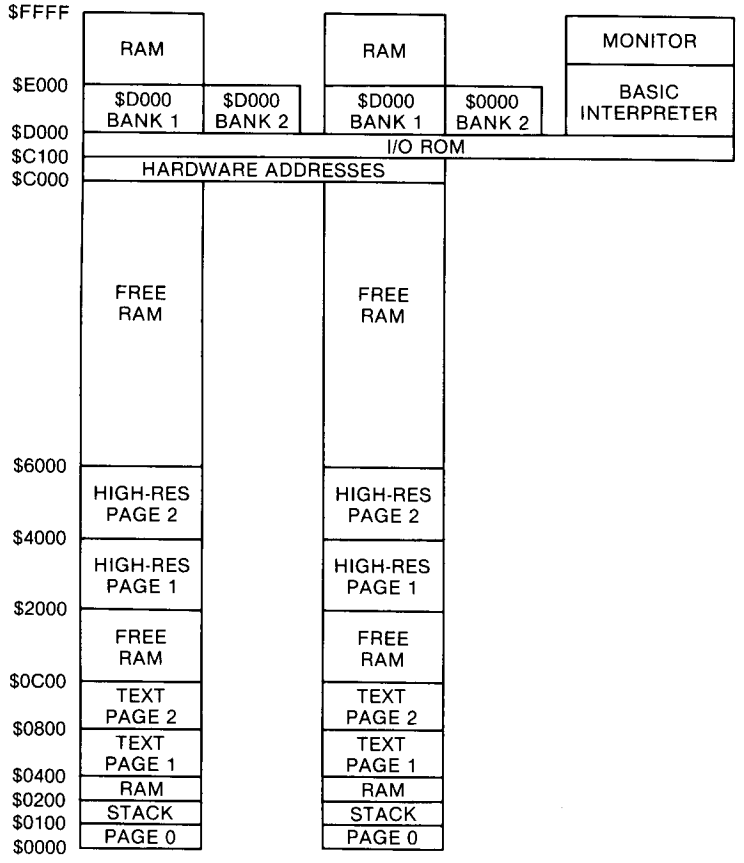


Figure 4–4
A map of the IIGS in emulation mode

- Addresses $00 to $FF (page 0). As you will see in chapter 5, memory addresses $00 to $FF, also known as page 0, are an important part of the memory map of any microcomputer. When the operand of an assembly language statement is a page 0 address, the instruction can be carried out faster because a page number does not have to be specified. And, as you shall see in chapter 6, some addressing modes require their operands to be on page 0.

  For now, it's sufficient to note that in an Apple IIc or an expanded Apple IIe, there are two bank-switchable page zeros: one in main memory and one in auxiliary memory. When the IIGS is operated in native mode, any page in bank $00 can be used as page 0—but we'll save further discussion of that point for chapters 5 and 6.

- Addresses $100 to $1FF (stack). The stack is a temporary storage area where values can be tucked away until needed. How the stack works and how it is used are examined in chapter 6.

  In the Apple IIc and the expanded Apple IIe, there are two bank-switchable stacks: one in main memory and one in auxiliary memory. When the IIGS is operated in native mode, the stack, like page 0, can be located anywhere in bank $00. This operation is also covered in chapters 5 and 6.

- Addresses $0200 to $03FF (input buffer, vectors, and link addresses). In bank $00, these addresses are used by the Applesoft input buffer and for certain operating system vectors and link addresses. In bank $01, they are available as free RAM.

- Addresses $0400 to $0BFF (text and low-resolution pages 1 and 2). As noted, the block of memory in which a screen display is stored is sometimes referred to as a page. In the earliest models of the Apple II, there were four such pages: two for text and low-resolution screen displays, and two for high-resolution displays. In the Apple IIc, the expanded Apple IIe, and the Apple IIGS, a second pair of high-resolution graphics pages and a second pair of text and low-resolution graphics pages are provided in auxiliary RAM.

  In all Apple II computers, animated displays can be created by using soft switches to flip between one high-resolution page and another, or between one text or low-resolution display and another. In the Apple IIGS, however, this capability exists only when the computer is in emulation mode, with IIc/IIe-style text or graphics displays. Soft switches are examined at the end of this chapter.

  As figure 4–4 illustrates, text and low-resolution page 1 extends from $0400 to $07FF, and text and low-resolution page 2 extends from $0800 to $0BFF. In application programs that do not use Apple IIc/IIe-style text or low-resolution graphics, both of these blocks of memory can be used as RAM.

- Addresses $0C00 to $1FFF (free RAM). In both bank $00 and bank $01, this block of memory is available for use as free RAM.

- Addresses $2000 to $5FFF (high-resolution pages 1 and 2). In all Apple II computers, addresses $2000 through $3FFF are used for data displayed on high-resolution page 1, and addresses $4000 to $5FFF are used for data displayed on high-resolution page 2. On the Apple IIc, the expanded Apple IIe, and the Apple IIGS, the same blocks of addresses can be used for the same purposes in auxiliary memory. In programs that do not use IIc/IIe-style high-resolution graphics, all of these memory blocks can be used as free RAM.
- Addresses $6000 to $BFFF (free RAM). In banks $00 and $01, this block of memory is available for use as free RAM.
- Addresses $C000 to $CFFF (hardware addresses and I/O ROM). In bank $00, this segment of memory is reserved for system hardware addresses and system I/O ROM. In bank $01, it is available for use as free RAM.
- Addresses $D000 to $DFFF (language card area). This block of memory consists of bank-switched RAM that is reserved mostly for use by ProDOS and for other system uses. When BASIC is used, addresses $D000 through $F7FF in bank $01 are claimed by the IIGS's BASIC interpreter. Why this segment of memory is called the language card area is explained later in this chapter.
- Addresses $E000 to $FFFF (bank-switched RAM and monitor firmware). When both the IIGS monitor and Applesoft BASIC are not in use, addresses $E000 through $FFFF in bank $00 and bank $01 can be used as free RAM. When BASIC is in use, however, it occupies addresses $D000 to $F7FF in bank $01. When the monitor is active, it claims memory addresses $F800 through $FFFF in bank $01.

How Pre-GS Programs Use Memory

When you load a program written for a pre-GS Apple II computer into the Apple IIGS, the IIGS firmware automatically sets up banks $00 and $01 as main and auxiliary memory and configures both banks for Apple IIc/IIe-style operations. The firmware also allocates pages $00 and $01 in bank $00 for use as page 0 and the stack, respectively. (There's more about page 0 and the stack later in this chapter and in chapter 6.)

When the IIGS configures itself for emulation mode, memory outside banks $00 and $01 is not available for use in programs. But it can be used as a big RAM disk, designated /RAM5.

As you can see by looking at figure 4–4, the largest block of memory in main memory, or bank $00, is labeled main RAM. The largest block in auxiliary memory, or bank $01, is labeled auxiliary RAM. When the IIGS is in emulation mode, main RAM extends from $6000 to $BFFF in bank $00, and auxiliary RAM uses the same block of memory in bank $01. Application programs can use both of these blocks as free RAM.

In the Apple IIGS, just as in earlier Apple IIs, an application can switch

between bank $00 and bank $01 using soft switches—bytes in memory that, like a light switch, can be turned on and off to change memory banks and control IIc-style and IIe-style text and graphics displays.

Language Card Area

In the memory addresses that extend from $D000 to $DFFF in both bank $00 and bank $01, there is another block of bank-switchable memory that has come to be known as the language card area of RAM. It got its name when the Pascal language was first introduced for the Apple II and required more memory than what was available. The card added to accommodate Pascal no longer exists—it is now built into the main circuit board of Apple II computers—but this area of memory retains its original name.

Because there are two language card areas—one in bank $00 and one in bank $01—there are actually four banks of useable RAM between memory addresses $D000 and $E000. In bank $00, most of the language card space in both main memory and bank-switched memory is reserved for use by ProDOS (which is covered in chapter 12) and for other needs of the IIGS operating system. In bank $01, the bank-switched portion of the language card area is also reserved for use by system memory, but the portion that does not have to be bank switched is available for use as free RAM.

Now that you've had a good look at the emulation mode memory map of the IIGS, it should be pointed out that the map is misleading in one respect. When the IIGS is running in emulation mode, it does not directly address banks $00 and $01. Instead, all data in banks $00 and $01 is copied into banks $E0 and $E1. It is the copied data that the IIGS reads from and writes to when it is running an emulation program. This process, known as *memory shadowing*, is carried out because banks $E0 and $E1 are synchronized for use with emulation mode programs, but banks $00 and $01 are not. A fuller description of memory shadowing is presented at the end of this chapter.

As noted, the Apple IIGS has two memory maps; it uses one in emulation mode and the other in native mode. You've just examined the emulation mode memory map, and in a few moments you'll see how the map changes when the IIGS is switched to native mode. Before that, though, it is helpful to explore how the Apple IIGS emulates an Apple IIc.

Mega II Chip

As you may remember from chapter 1, the designers of the IIGS faced a double-edged problem. They wanted to build a computer that would not only run programs designed for earlier Apples, but also take full advantage of the increased operating speed and expanded memory addressing capabilties of the 65C816 microprocessor. They came up with an ingenious solution. They created a new integrated chip, the Mega II, to interface the new features of the IIGS with the old features of earlier members of the Apple II family.

The first job for the designers of the Mega II chip was achieving some kind of compatibility between the 2.8 MHz operating speed of the Apple IIGS and the 1 MHz operating speed of earlier Apples. They attained this goal by incorporating the Mega II into the design of the IIGS, as illustrated in figure 4–5.

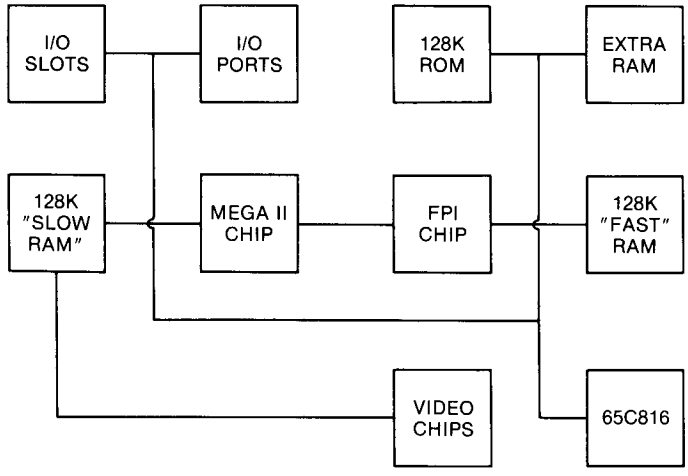


Figure 4–5
Incorporating the Mega II chip into the IIGS's design

As figure 4–5 shows, the Mega II chip is connected to

- The Apple IIGS's ports and slots, which are operated under the control of a 1 MHz chip and are therefore compatible with the ports and slots in earlier Apple IIs.
- A 128K block of RAM called *slow RAM*, which is built into the IIGS to make it compatible with earlier members of the Apple II family.
- The video chips that generate the IIGS's text and graphics displays when it is running in IIc/IIe emulation mode.
- The VGC (video graphics controller) chip, which generates the IIGS's super high-resolution graphics display. Although the VGC chip was designed specifically for the IIGS and is not found in earlier Apple IIs, it operates at a 1 MHz clock speed so that it is synchronized with other video circuitry that is IIc/IIe compatible.

To interface the Mega II module with the 65C816 and the components it controls, Apple engineers designed another special chip called the *fast processor interface*, or FPI. The FPI, as figure 4–5 shows, is connected not only to the Mega II chip and its 1 MHz components, but also to all the IIGS components that operate at 2.8 MHz. These components include

- A 128K block of *fast RAM* that is laid out exactly like the 128K of *slow RAM* controlled by the Mega II
- All the 128K of ROM built into the IIGS
- All expansion RAM that the IIGS owner may install
- The 65C816 processor (which must be switched from 2.8 MHz to 1 MHz before the IIGS can operate in IIc/IIe emulation mode)

Memory Shadowing

Now you're ready to study the concept of memory shadowing, which was briefly mentioned in this chapter. Memory shadowing is a technique the IIGS uses to copy data from banks $00 and $01 into banks $E0 and $E1 so that programs can be run from banks $E0 and $E1 when the computer is in emulation mode. Here, as promised, is an explanation of why memory shadowing is used in the IIGS and how it works.

Because programs written for the IIc and the IIe use memory addresses $0000 through $FFFF, the designers of the IIGS had to build the computer so that IIc and IIe programs could be run in banks $00 and $01. But banks $00 and $01 are also important to the operation of the IIGS in native mode, so they were designed to operate at the native mode speed of 2.8 MHz, not at the emulation mode speed of 1 MHz (the speed at which IIc/IIe programs must be run).

To make the IIGS compatible with programs written for earlier Apple IIs, the creators of the IIGS had to equip it with at least two banks of 1 MHz RAM. They didn't want to slow down banks $00 and $01 just to make them IIc/IIe compatible, so they decided to slow down banks $E0 and $E1—the only other two banks available on a bare-bones IIGS—and make them run at 1 MHz.

Banks $E0 and $E1 also have all the features needed to run Apple IIc/IIe programs. These features include language card mapping in memory addresses $D000 through $DFFF, space for hardware and I/O memory in addresses $C000 through $CFFF, and display buffers used for IIc/IIe-style video displays.

After all these features were incorporated into banks $E0 and $E1, only one problem remained: how to run emulation mode programs designed to be executed from banks $00 and $01 using the clock speed and IIc/IIe features built into banks $E0 and $E1. To solve this problem, the designers of the IIGS used the technique of memory shadowing. Here's how it works.

The Quagmire State and the Shadow Register

To find out the current status of the IIGS's shadowing operations, you can read the status of a memory location called the *shadow register*. The shadow register keeps track of the IIGS's shadowing state, which is also known as the computer's *quagmire state* because shadowing can make memory locations move around like shifting sand. The shadow register, or quagmire register, is at memory address $C035 in bank $E0.

In addition to controlling memory shadowing, the shadow register can also activate or deactivate the I/O and language card areas at addresses $C000 through $DFFF. See table 4–1.

When the shadow register selects shadowing for an area, the IIGS hardware executes any instruction that writes into the selected area in bank $00 or $01 by writing into both the selected area and the same address in bank $00 or bank $01. Then, because the RAM in banks $E0 and $E1 runs at 1 MHz, all code that is shadowed is executed at slow speed.

Shadowing of the I/O and language card spaces is controlled by bit 6 of the shadow register, sometimes referred to as the IOLC (I/O and language

Table 4–1
The Shadow Register

Bit	Value	Description
0	1	Text page 1 shadowing disabled
1	1	High-res page 1 shadowing disabled
2	1	High-res page 2 shadowing disabled
3	1	Super high-res buffer shadowing disabled
4	1	Shadowing of auxiliary high-res pages disabled
5		Reserved—do not find modify
6	1	I/O and language card operation disabled
7		Reserved—do not modify

card) bit. This bit is normally set to 0, which enables I/O in the $CXXX memory addresses and maps the 4K of RAM that ordinarily resides in that space into a second bank of RAM in the $DXXX address range. Figure 4–6 illustrates this operation.

Shadowing and Interrupts

Some of the interrupt routines used in emulation mode are in ROM in the I/O space of the $C07X address range. For this code to operate, I/O must remain enabled in the $CXXX range of memory in bank $00, and the high 16K of RAM must remain mapped as a language card. In other words, the IOLC bit of the shadow register must be clear. If a program changes the IOLC bit so that it can use RAM in the $CXXX range, the interrupt routines in that area won't work. So IOLC shadowing must be left on even by programs running in native mode, which otherwise do not use language card mapping.

Display Shadowing

Programs run on the IIGS can also use *display shadowing*, which works a little differently than I/O shadowing. When I/O shadowing is used, both reading and writing are slowed to 1 MHz. When only display shadowing is selected, however, the slowdown affects only instructions that write to the shadowed areas. The 65C816 still reads from the display areas of banks $00 and $01 at 2.8 MHz.

When the IIGS loads a program, it automatically sets display shadowing to whatever is appropriate for the program's operating system: on for DOS 3.3, UCSD Pascal, and ProDOS 8, and off for ProDOS 16 (the operating system used in native mode). An application can turn off shadowing of individual displays by setting individual bits in the shadow register.

More details about memory shadowing and how the shadow register works can be found in the *Apple IIGS Hardware Reference*.

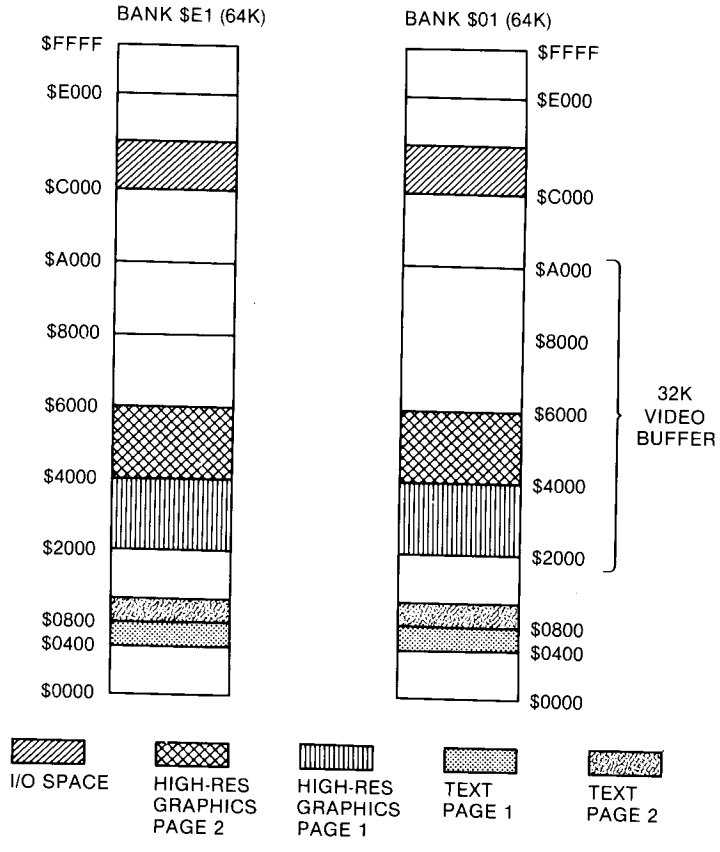


Figure 4–6
Memory shadowing in the Apple IIGS

Mapping the IIGS in Native Mode

The memory map used by the IIGS in native mode is considerably different from the one used in emulation mode. The most obvious difference is the native mode map is bigger. It can contain at least 256K of memory and as much as 8.25 megabytes of memory. There are other differences, too. For example, to give native mode programs as much free RAM as possible in banks $00 and $01, the computer's native mode ROM is in banks $FE and $FF, opening up almost all the memory space in banks $00 and $01 for use as free RAM. System ROM includes Applesoft BASIC, the IIGS monitor, port firmware, and the part of the IIGS Toolbox built into ROM.

Figure 4–7 shows how memory is allocated when the IIGS is in native mode. Programs can occupy most of the space in banks $00 and $01, and all the expansion RAM space in banks $02 through $7F (if expansion RAM is installed). Applications can call the Memory Manager to obtain the memory they need in those areas.

In banks $E0 and $E1, however, there are some blocks of memory that

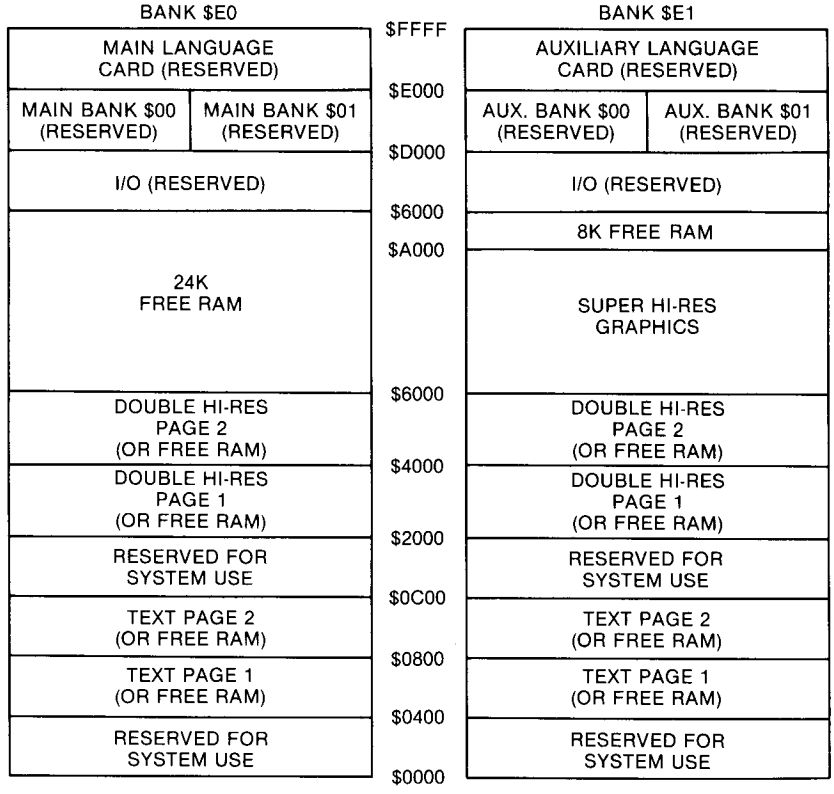


Figure 4–7
Detailed memory map of banks $E0 and $E1

are not available for use as free RAM, even when the IIGS is in native mode. For example, the I/O space in the $CXXX region and text page 1 are shadowed from memory banks $00 and $01 into banks $E0 and $E1. These areas have to be shadowed for the proper operation of interrupts and peripheral cards, and thus cannot be used as free RAM by application programs.

There are other areas in banks $E0 and $E1, however, that are available for use in application programs. If you decide to use these banks in a program, remember that they are timed to operate as slow RAM—operating at 1 MHz—when they are written to. But they can be read from at the fast speed of 2.8 MHz. If a program merely reads from them, without writing to them, they won't slow the program.

Here is an outline of how the various blocks of memory in banks $E0 and $E1 are used when the IIGS is running in native mode:

- Addresses $0000 to $03FF in bank $E0. Reserved for system use. This block of RAM—used for shadowing page 0, the stack, and other important addresses when the IIGS is in emulation mode—is reserved for future expansion. It is not managed by the Memory Manager, but you can use it by managing it yourself. If you do, though,

your application may not be compatible with future models of the IIGS.

- Addresses $0400 to $07FF in bank $E0 (text page 1). Text page 1 is shadowed into this area even when the IIGS is in native mode. It is not managed by the Memory Manager, but you can use it if you manage it yourself. That could get you into trouble, however, because you never know when something such as a desk accessory might decide to use text page 1 and try to use this segment of memory.
- Addresses $0800 to $0BFF in bank $E0 (text page 2). Text page 2 is not likely to be used by a desk accessory (though it could be), so this region is fairly safe for use by an application program. The Memory Manager doesn't manage it, though, so once again, beware.
- Addresses $0C00 to $1FFF in bank $E0. Reserved for use by the IIGS system.
- Addresses $2000 to $5FFF in bank $E0 (high-resolution pages 1 and 2). Available for use by application programs that don't use high-resolution graphics pages 1 and 2. Managed as *special memory* by the Memory Manager (more about that in chapter 7).
- Addresses $6000 to $BFFF in bank $E0 (free RAM). This 24K chunk of memory is allocated as free RAM and is managed by the Memory Manager.
- Addresses $C000 to $FFFF in bank $E0. Used by the IIGS system. This segment of memory includes I/O space, the language card area, and other addresses used by the IIGS system. It's off-limits to application programs.
- Addresses $0000 to $03FF in bank $E1. Reserved for system use. Not managed by the Memory Manager. Use at your own risk.
- Addresses $0400 to $0BFF in bank $E1 (alternate text pages 1 and 2). Rarely used and probably safe, but not managed by the Memory Manager.
- Addresses $0C00 to $1FFF in bank $E1. Reserved for use by the IIGS system.
- Addresses $2000 to $5FFF in bank $E1 (alternate high-resolution pages 1 and 2). Available for use by programs that don't use alternate high-resolution pages 1 and 2. Managed as special memory by the Memory Manager. The special memory designation is covered in chapter 7.
- Addresses $6000 to $BFFF in bank $E1 (super high-resolution display). This is the super high-resolution screen display area of the IIGS. It can be managed as special memory by the Memory Manager. But most programs written for the IIGS use super high-resolution graphics, so using this area of memory as free RAM—even by a program that doesn't require super high-res graphics—is strongly discouraged.

- Addresses $A000 to $BFFF in bank $E1 (free RAM). Free RAM managed by the Memory Manager.
- Addresses $C000 to $FFFF in bank $E1. Reserved for system use. Not managed by the Memory Manager and not recommended for use as free RAM by application programs.
- Banks $F0 through $FD. Reserved for use by a ROM expansion card used for additional firmware and by applications that are stored as ROM disk files.

Soft Switches

If you're an old hand at Apple II programming, you may be familiar with the concept of *soft switches:* bytes in memory that perform operations by simply being read from or written to.

If you like to manage Apple II operations using soft switches, you'll be happy to know that the IIGS has all the soft switches its predecessors have—and an extra register to help you access them conveniently.

The soft switches in the IIGS, like the ones in earlier Apple IIs, reside in the $CXXX block of memory in bank $00. And, like their counterparts, they can be used for bank switching, I/O and graphics operations, and protecting certain blocks of memory by making it possible to read from them but not write to them. Table 4–2 lists some of the most often used soft switches in the Apple IIGS and earlier Apple IIs.

Accessing Soft Switches

There are three ways to manipulate the soft switches in the IIGS:

1. Some soft switches can be toggled on or off with either a read operation, such as `lda`, or a write operation, such as `sta`. For example, you can change the setting of the Page2 soft switch at $C055 with a statement such as

   ```
   sta $C055
   ```

   or a statement like

   ```
   lda $C055
   ```

   More details of how the Page2 soft switch works are presented in a moment.

2. Some soft switches can be turned on or off with a write operation. For example, you can turn on the RAMWrt switch at $C005 by writing any value to it, using a statement such as

   ```
   sta $C005
   ```

Table 4–2
Soft Switches

Arranged by Name

Name	Address	Access	Function
80Store	$C000	Write	Off: RAMRd and RAMWrt determine RAM locations
80Store	$C001	Write	On: Page2 switches between main and auxiliary display pages
AltZP	$C008	Write	Off: Using main-memory page 0 and stack
AltZP	$C009	Write	On: Using auxiliary-memory page 0 and stack
Bank Select	$C080	Two Reads	Read RAM; no write; use $D000 bank 2
Bank Select	$C081	Two Reads	Read ROM; write RAM; use $D000 bank 2
Bank Select	$C082	Read	Read ROM; no write; use $D000 bank 2
Bank Select	$C083	Two Reads	Read and write RAM; use $D000 bank 2
Bank Select	$C088	Read	Read RAM; no write; use $D000 bank 1
Bank Select	$C088	Read	Read RAM; no write; use $D000 bank 1
Bank Select	$C089	Two Reads	Read ROM; write RAM; use $D000 bank 1
Bank Select	$C08A	Read	Read ROM; no write; use $D000 bank 1
Bank Select	$C08B	Two Reads	Read and write RAM; use $D000 bank 1
DHiRes	$C05E	Read/Write	On: If OIUDis is on, turn on double high resolution
DHiRes	$C05F	Read/Write	Off: If IOUDis is on, turn off double high resolution
HiRes	$C056	Read	Off: Display text page
HiRes	$C057	Read	On: Show high-res pages; make Page2 switch between high-res pages
IOUDis	$C07F	Write	On: Disable IOU access for $C058-$C05F; enable zDHiRes switch access
IOUDis	$C07F	Write	Off: Enable IOU access for $C058-$C05F; disable DHiRes switch access
Page2	$C054	Read	Off: Select text page 1 and high-resolution page 1
Page2	$C055	Read	On: If 80Store off, use main memory displays; if on, use auxiliary displays
RAMRd	$C002	Write	Off: Read main 48K RAM
RAMRd	$C013	Write	On: Read auxiliary 48K RAM
RAMWrt	$C004	Write	Off: Write to main 48K RAM
RAMWrt	$C005	Write	On: Write to auxiliary 48K RAM
Rd80Store	$C018	Read bit 7	Bit 7 tells whether 80Store is on (1) or off (0)
RdAltZP	$C016	Read bit 7	Bit 7 tells whether auxiliary memory (1) or main memory (0) accessed
RdBnk2	$C011	Read bit 7	Bit 7 tells whether $D000 is bank 2 (1) or bank 1 (0)
RdDHiRes	$C07F	Read bit 7	Read DHiRes switch (1 = on)

Table 4–2 (cont.)

Arranged by Name

Name	Address	Access	Function
RdHiRes	$C01D	Read bit 7	Bit 7 tells whether high resolution is on (1) or off (0)
RdIOUDis	$C07E	Read bit 7	Read IOUDis switch (1 = off)
RdLCRAM	$C012	Read bit 7	Reading RAM (1) or ROM (0)
RdPage2	$C01C	Read bit 7	Bit 7 tells whether Page2 is on (1) or off (0)
RdRAMRd	$C013	Read bit 7	Bit 7 tells whether main memory (0) or auxiliary memory (1) is being accessed
RDRAMWrt	$C014	Read bit 7	Read whether main memory (0) or auxiliary memory (1) is being accessed

Arranged by Address

Address	Name	Access	Function
$C000	80Store	Write	Off: RAMRd and RAMWrt determine RAM locations
$C001	80Store	Write	On: Page2 switches between main and auxiliary display pages
$C002	RAMRd	Write	Off: Read main 48K RAM
$C004	RAMWrt	Write	Off: Write to main 48K RAM
$C005	RAMWrt	Write	On: Write to auxiliary 48K RAM
$C008	AltZP	Write	Off: Using main-memory page 0 and stack
$C009	AltZP	Write	On: Using auxiliary-memory page 0 and stack
$C011	RdBnk2	Read bit 7	Bit 7 tells whether $D000 is bank 2 (1) or bank 1 (0)
$C012	RdLCRAM	Read bit 7	Reading RAM (1) or ROM (0)
$C013	RAMRd	Write	On: Read auxiliary 48K RAM
$C013	RdRAMRd	Read bit 7	Bit 7 tells whether main memory (0) or auxiliary memory (1) is being accessed
$C014	RdRAMWrt	Read bit 7	Read whether main memory (0) or auxiliary memory (1) is being accessed
$C016	RdAltZP	Read bit 7	Bit 7 tells whether auxiliary memory (1) or main memory (0) is being accessed
$C018	Rd80Store	Read bit 7	Bit 7 tells whether 80Store is on (1) or off (0)
$C01C	RdPage2	Read bit 7	Bit 7 tells whether Page2 is on (1) or off (0)
$C01D	RdHiRes	Read bit 7	Bit 7 tells whether high resolution is on (1) or off (0)
$C054	Page2	Read	Off: Select text page 1 and high-resolution page 1
$C055	Page2	Read	On: If 80Store off, use main memory displays; if on, use auxiliary displays
$C056	HiRes	Read	Off: Display text page
$C057	HiRes	Read	On: Show high-res pages; make Page2 switch between high-res pages

Table 4–2 (cont.)

Arranged by Address

Address	Name	Access	Function
$C05E	DHiRes	Read/Write	On: If OIUDis is on, turn on double high resolution
$C05F	DHiRes	Read/Write	Off: If IOUDis is on, turn off double high resolution
$C07E	RdIOUDis	Read bit 7	Read IOUDis switch (1 = off)
$C07F	IOUDis	Write	On: Disable IOU access for $C058-$C05F; enable DHiRes switch access
$C07F	IOUDis	Write	Off: Enable IOU access for $C058-$C05F; disable DHiRes switch access
$C07F	RdDHiRes	Read bit 7	Read DHiRes switch (1 = on)
$C080	Bank Select	Two Reads	Read RAM; no write; use $D000 bank 2
$C081	Bank Select	Two Reads	Read ROM; write RAM; use $D000 bank 2
$C082	Bank Select	Read	Read ROM; no write; use $D000 bank 2
$C083	Bank Select	Two Reads	Read and write RAM; use $D000 bank 2
$C088	Bank Select	Read	Read RAM; no write; use $D000 bank 1
$C088	Bank Select	Read	Read RAM; no write; use $D000 bank 1
$C089	Bank Select	Two Reads	Read ROM; write RAM; use $D000 bank 1
$C08A	Bank Select	Read	Read ROM; no write; use $D000 bank 1
$C08B	Bank Select	Two Reads	Read and write RAM; use $D000 bank 1

3. You can read some soft switches to see whether a given bit is on or off. For example, you can read bit 7 of the RAMWrt switch, at $C014, to find out whether main memory (bank $00) or auxiliary memory (bank $01) is being used for writing.

4. As a precaution against accidents, some soft switches have to be accessed twice in succession before they respond. For example, to turn on the soft switch at $C083, you must carry out a pair of operations, like this:

```
lda $C083
lda $C083
```

Please note that in this case, memory address $C083 is not being written to, but is merely being accessed with a read operation (`lda`). If you were writing to it—for example, with a `sta` instruction—it wouldn't matter what was in the accumulator when the operation was carried out. That's because it's the act of accessing the switch, not the value written to it, that causes the switch to do its work. When you access a switch with a write operation, you can store any value in it (even a 0) and the result is always the same.

Using Soft Switches

As you may notice in table 4–2, the same name is sometimes used for two or more soft switches. That's because some switches are activated with one switch and deactivated with another. And some switches are turned on with one address, turned off with another, and read from with still another. In table 4–2, nine switches that select memory banks are grouped under the same name: bank select. The following sections explain the operation of some important switches.

Selecting Main or Auxiliary RAM

Two switches, RAMRd and RAMWrt, select main or auxiliary RAM in the 48K memory space in banks $00 and $01 when the IIGS is in emulation mode. When RAMRd is on and the 80Store switch (which controls display memory) is off, RAMRd selects auxiliary memory for reading. When both 80Store and RAMRd are off, RAMRd selects main memory for reading. When RAMWrt is on and the 80Store switch is off, RAMWrt selects auxiliary memory for writing. When both RAMWrt and the 80Store switch are off, RAMWrt selects main memory for writing. That may sound quite complicated, but after you start using these three soft switches, you'll become accustomed to how they work.

Both the RAMRd and RAMWrt switches use three memory addresses. One address turns the switch on, one turns it off, and one reads its state. To read the state of RAMRd, RAMWrt, or any other three-address switch listed in table 4–2, just check bit 7 of the appropriate memory address. If the switch is off, bit 7 is cleared to 0. If the switch is on, bit 7 is set to 1.

Selecting Display Memory

When the IIGS is displaying IIc/IIe-style high-resolution graphics, three soft switches—80Store, HiRes, and Page2—can select the portion of RAM used for screen memory. Each of these switches has three memory addresses—one that turns it on, one that turns it off, and one that reads its state by checking bit 7.

If the HiRes switch is off, Page2 switches between text pages 1 and 2. If HiRes is on, Page2 switches between high-resolution graphics pages 1 and 2.

If 80Store is off, RAMRd and RAMWrt determine whether to use the display pages in main or auxiliary RAM, and Page2 selects pages for display only—not for reading or writing. If 80Store is on, however, it overrides RAMRd and RAMWrt with respect to the display pages selected by HiRes and Page2.

The Machine State Register

There is one drawback in using the soft switches in table 4–2. Because they are in slow RAM—memory that runs at the emulation speed of 1 MHz, instead of the native mode speed of 2.8 MHz—the system is slowed down every time a soft switch is accessed directly.

But there is a way to access eight of the most commonly used soft switches without paying the penalty of changing operating speeds. That method is to use a special memory address called the *machine register*. (It's also called the state register or machine state register.) This register is situated at memory address $C068. Table 4–3 shows how each bit in the machine register is used.

Table 4–3
The Machine State Register

Bit	Name	Description
Bit 0	INTCXROM	Determines whether internal or slot card ROM will be used in the $C100 to $C7FF block of memory
Bit 1	ROMBank	Selects the ROM bank in main memory (0) or auxiliary memory (1)
Bit 2	Bank2	Selects the main RAM bank (0) or auxiliary RAM bank (1)
Bit 3	RdROM	Activates the correct bank select switch to read ROM
Bit 4	RAMWrt	Turns the RAMWrt switch off and on
Bit 5	RAMRd	Turns the RAMRd switch off and on
Bit 6	Page2	Turns the Page2 switch off and on
Bit 7	AltZP	Turns the AltZP switch off and on

In this chapter, you saw how much memory is in the IIGS, its location, how it is accessed, and its uses. In chapter 5, you take an inside look at the 65C816 processor and see what makes it go.

CHAPTER 5

In the Chips

Inside the 65C816 Microprocessor

One major component that sets the Apple IIGS apart from earlier members of the Apple II family is the 65C816 central processing unit, or CPU. The 65C816, as noted in chapter 1, is a 16-bit chip that runs almost three times as fast as the 6502 and 65C02 processors in earlier Apple IIs.

The 65C816 has other advantages over its 8-bit predecessors. Because of its 16-bit data-handling capacity, programs written for the 65C816 are 25 to 50 percent shorter than programs written for earlier 6502–style processors. The 65C816 can also address far more memory than any of its 8-bit counterparts.

In this chapter and in chapter 6, you see how the 65C816 does all those things and what its advanced features mean to the Apple IIGS programmer. The instruction set of the 65C816 is described in appendix A.

All in the (6502) Family

The 65C816 is a member of the venerable 6502 family of microprocessors. The first Apple II, built in 1977, was designed around a 6502 chip. Since then, various models of the 6502 have been built into every computer in the Apple II line. The CPU in the Apple IIe was a slightly improved 6502 called the 6502B. The Apple IIc was built around a further expanded 6502 called a 65C02. The 65C02 is equipped with 27 more assembly language instructions

than the original 6502, plus an expanded set of addressing modes. A few months after the 65C02 appeared in the Apple IIc, it became standard equipment in the Apple IIe.

Apple is not the only manufacturer that has used 6502 chips in its products. The Commodore 64's CPU is a 6502-style chip called the 6510, and the Commodore 128 runs on a version of the 6502 called an 8502. Atari still uses 6502 chips in its line of 8-bit computers. Because of their versatility, availability, and low price, 6502-family chips have been widely used in standalone configurations in the fields of robotics and computer-aided manufacturing.

There are a number of important differences between the 65C816 and all its 6502 predecessors, including the original 6502 and the 65C02. For example:

- The 65C816 is the first 16-bit chip in the 6502 family. It can perform calculations on 16-bit values—numbers ranging from 0 to 65,535—without dividing them into smaller numbers as its predecessors had to do.
- All previous 6502-family chips had 16-bit address buses. Therefore, they could address memory locations ranging from $0000 to $FFFF, or from 0 to 65,535 in decimal notation. But the 65C816 has a 24-bit address bus, so it can address up to 16 megabytes of memory (although only 8.25 megabytes of its RAM addressing capability are utilized by the Apple IIGS).
- The 65C816 has nine internal registers, three more than its predecessors. In this chapter, you'll examine all nine of the 65C816's internal registers.
- The 65C816 operates at a clock speed of 2.8 MHz, compared with a clock speed of 1.024 MHz for all previous members of the 6502 family.
- The 65C816 recognizes 9 new addressing modes and 78 new machine language opcodes. Thus, it can do more with less code than its 8-bit predecessors.
- The 65C816 can be operated in two modes: in native mode as a full-featured 16-bit chip and in an emulation mode as a 65C02. The processor's emulation mode makes the Apple IIGS compatible with earlier Apple IIs.

Inside the 65C816

The most important components of the 65C816 are illustrated in figure 5–1. They include:

- A 16-bit data bus
- A 24-bit address bus
- Nine internal registers

- An arithmetic and logic unit, or ALU

In this chapter, you'll examine these components in detail, beginning with the 65C816's data and address buses.

Buses

The rectangles across the top and bottom of figure 5–1 represent buses, lines used for the transmission of addresses, instructions, and data. The bus at the top of the illustration is a data bus, and the one at the bottom is an address bus.

Data buses are quite appropriately named; they move data between the registers in the CPU and the memory registers in a computer's RAM and ROM. An address bus transmits the addresses that data is being moved from and to.

When the 65C816 is operated in 8-bit emulation mode, it has an 8-bit data bus and a 16-bit address bus. It can perform operations on numbers ranging from $00 to $FF (0 to 255 in decimal) and can access memory addresses ranging from $0000 to $FFFF (0 to 65,535 in decimal).

When the processor is running in native mode, it has a 16-bit data bus and a 24-bit address bus. It can perform operations on numbers ranging from $0000 to $FFFF (0 to 65,535 in decimal) and can access memory addresses ranging from $000000 to $FFFFFF (0 to 16,772,215 in decimal).

Internal Registers

As mentioned, the 65C816 has nine internal registers. They are the

- Accumulator
- X register
- Y register
- Program counter
- Stack pointer
- Processor status register

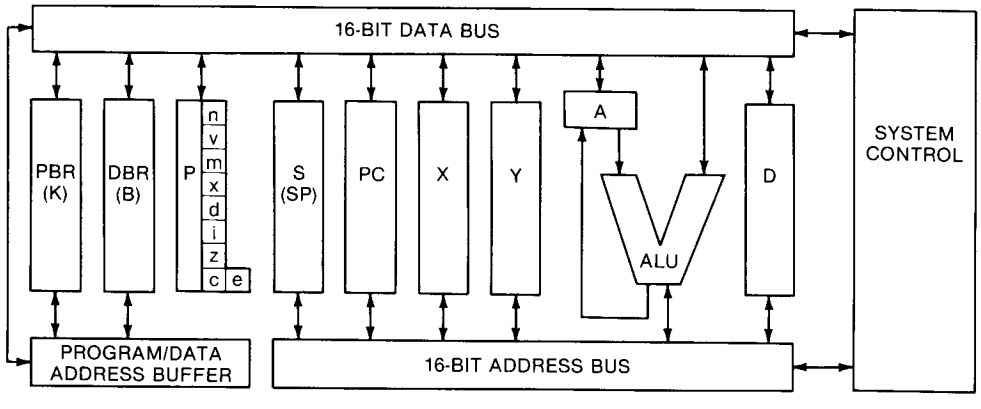


Figure 5–1
Simplified block diagram of the 65C816

- Data bank register
- Program bank register
- Direct page register

Three of the 65C816's registers—the data bank register, program bank register, and direct page register—handle the extended addressing functions of the 65C816 and are initialized to 0 when the chip is in emulation mode. But when the 65C816 is in native mode, all nine of its internal registers are active.

Figure 5–2 shows how the 65C816's registers are used when the chip is in native mode. Figure 5–3 shows the configuration of the registers when the 65C816 is in emulation mode. Now let's examine each register, in both native mode and emulation mode.

Accumulator

The accumulator (abbreviated A or C) is a 16-bit register divided into two 8-bit registers when the 65C816 is in emulation mode. When the 65C816 is in native mode, the accumulator is referred to as the A register. But when the register is split for emulation mode operations, its low-order byte is abbreviated A, its high-order byte is abbreviated B, and the register as a whole is abbreviated C. The accumulator is the 65C816's busiest register. You'll take a closer look at it later in this chapter.

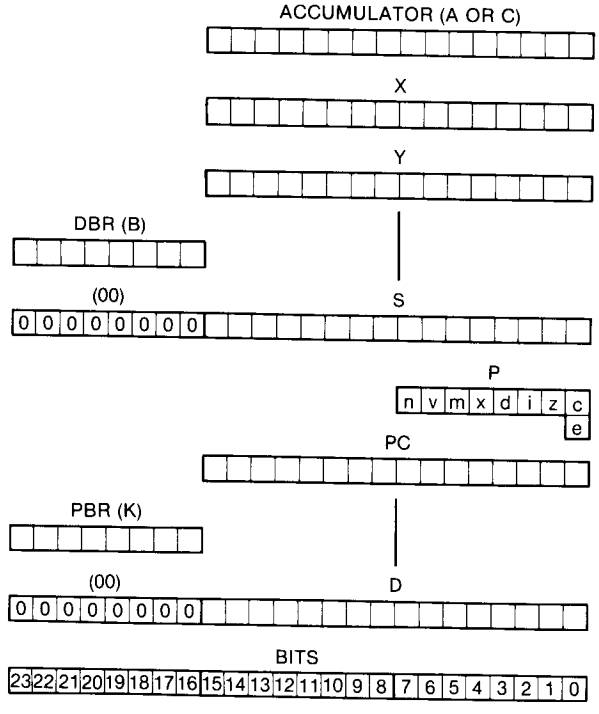


Figure 5–2
65C816 register configuration in native mode

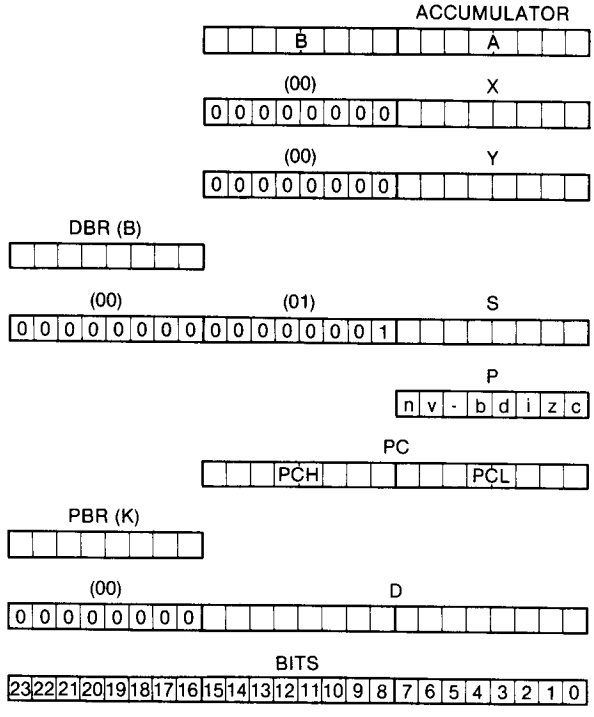


Figure 5–3
65C816 register configuration in emulation mode

X Register

The X register (abbreviated X) is an 8-bit register when the 65C816 is in 8-bit emulation mode, but expands into a 16-bit register when the processor is in 16-bit native mode. In the 65C816, as in other 6502-family processors, the X register is often used for the temporary storage of data. But it also has an important special feature. It can be incremented with a simple 1-byte assembly language instruction (`inx`) and decremented with another 1-byte instruction (`dex`). It is therefore used quite often as a counter and as an index register during loops in programs.

Y Register

The Y register (abbreviated Y) is also an 8-bit register when the 65C816 is in 8-bit emulation mode and expands to a 16-bit register when the processor is in 16-bit native mode. The Y register, like the X register, can be incremented and decremented with a pair of 1-byte instructions (`iny` and `dey`). The Y register is also used as an index register and for storing data.

Program Counter

The program counter (abbreviated PC) is a pair of 8-bit registers. In both emulation mode and native mode, these two registers are combined and used as one 16-bit register.

The two 8-bit registers that make up the program counter are sometimes referred to as the program counter low (PCL) register and the program counter high (PCH) register. During native mode operations, the contents of the PCL and PCH registers are appended to the value of another 8-bit register called the program bank register. The combined contents of all three registers are then treated as a single 24-bit address. You'll learn more about the program bank register later in this chapter.

It is important to remember that the program counter (and the program bank register, if the 65C816 is running in native mode) always contains the memory address of the next instruction to be executed. When that instruction is carried out, the address of the instruction that follows it is loaded into the program counter.

Stack Pointer

The stack pointer (abbreviated S or SP) is a register that always contains the address of the next available memory address in a block of RAM called the stack. It is an 8-bit register in emulation mode and a 16-bit register in native mode. As you may recall from chapter 2, the 65C816 stack is a special block of memory in which data is often stored temporarily during the execution of a program. When the 65C816 is in emulation mode, the stack is always on page 1 in bank $00 (unless a soft switch shifts it to bank $01), so the stack pointer has to be only 1 byte long. But in native mode the stack can start anywhere in bank $00, so the stack pointer has to be 2 bytes long.

When subroutines are used in assembly language programs, the 65C816 often uses the stack as a temporary storage location for return addresses. The stack is also available for use in application programs. The operation of the stack is discussed in more detail in chapter 6, which is devoted to 65C816 addressing.

Processor Status Register

The processor status register (often called simply the status register, but abbreviated P) is an 8-bit register that keeps track of the results of operations performed by the 65C816. The processor status register is such an important part of the 65C816 processor that you'll take a closer look at it later in this chapter.

Program Bank Register

The program bank register (abbreviated PBR or K) is an 8-bit register initialized to 0 when the 65C816 is in 8-bit emulation mode. When the processor is in native mode, however, the program bank register becomes very important. In native mode, every time the 65C816 has to get an instruction from memory, it gets it from the location pointed to by the concatenation of the

program bank register and the program counter. So, when the 65C816 is in native mode, it uses the program bank register to extend the addressing capability of the program counter to 24 bits.

Because of the hybrid nature of the 65C816, it is not quite accurate to view the program counter and the program bank register as a single register. Sometimes they do work as one register, but more often they don't. Most of the instructions the 65C816 inherited from the 6502 use the address stored in the program counter, but ignore the bank number stored in the program bank register. In other words, they recognize only short addresses. But there are a few new or redesigned instructions that do treat the PC and the PBR as one 24-bit register. In other words, they recognize long addresses.

Instructions that recognize only short addresses work fine in programs written for the native mode 65C816; they just can't cross bank boundaries. That usually doesn't cause any serious problems in IIGS programs because a IIGS program segment can't cross a bank boundary. If it tries, the program counter simply rolls over to memory address $0000 in whatever bank the segment started in. For example, if the program counter increments past $FFFF, it rolls over to $0000 without incrementing the program bank register.

Instructions that recognize long addresses are a little easier to work with. You can move them from any address to any other address, without worrying about bank boundaries. Unfortunately, there are only five such instructions: `jmp` (when it is used to jump to an absolute long or indirect long address), `jsl` (jump to subroutine—long), `rtl` (return from subroutine—long), `brl` (branch to long address), and `rti` (return from interrupt).

Because the program bank register always contains the bank number of the program currently being executed, there is no assembly language instruction for changing the value of the PBR. But there is an instruction— `phk`— that pushes the value of the PBR onto the stack so that it can be pulled off the stack and into another register. More information on that topic is provided in chapter 6.

Data Bank Register

The data bank register (abbreviated DBR or B) is an 8-bit register that is initialized to 0 when the 65C816 is in 8-bit emulation mode. When the 65C816 is in native mode, the DBR designates the bank currently being used as a data bank by instructions that read and write data.

Usually, the data bank register and the program bank register contain the same bank number, because assembly language programs are ordinarily stored in the same bank as the data they access. But sometimes it is more convenient to store a program in one bank and place a long data segment, such as a bit map, in another. The value of the data bank register can be changed temporarily to permit access to the bit map.

The data bank register works much like the program bank register. When the 65C816 is in native mode and an instruction for fetching or storing data is used with a 16-bit operand, the address specified by the operand is con-

catenated with the value of the data bank register to form a 24-bit address. For example, if a program is running in bank $06, and the 65C816 enounters the instruction

```
lda $FEF0
```

the accumulator is loaded with the contents of memory address $06FEF0. There are ways to force the 65C816 to access addresses in other banks with instructions such as `lda`, but you won't get into that subject until chapter 6.

The data bank register can be accessed with the instructions `phb` and `plb`. The `phb` instruction pushes the address of the DBR on the stack. The `plb` instruction can be used to pull a value off the stack and place it in the data bank register. These operations are explained in more detail in chapter 6.

Direct Page Register

An area of memory called page 0 is a very valuable piece of real estate in the memory map of pre-GS Apple IIs. In the Apple IIc and the Apple IIe, page 0 extends from memory address $00 to memory address $FF in bank $00 or bank $01 (depending on the soft switch settings), and can therefore be accessed with a 1-byte operand. So instructions that address memory locations on page 0 run faster than they would if they accessed locations elsewhere in memory.

That is not the only reason that space on page 0 is so valuable. Some 65C02 addressing modes, called *indirect addressing modes*, require their operands to be page 0 addresses. As a result, space on page 0 is at a real premium in 8-bit Apple IIs.

In programs written for the Apple IIGS, however, page 0 is no longer the high-rent district. With the help of a new 16-bit register called the direct page register (abbreviated D), a IIGS programmer can move what was once called page 0 to any 256-byte area of memory in bank $00 that begins on a byte boundary. Because it has become a moveable page in the Apple IIGS, it is no longer called page 0, but is referred to as the direct page.

When you want to instruct the IIGS to use a given page as a direct page, all you have to do is place the starting address of the direct page of your choice in the direct page register. You can even give different segments in a program different direct pages, so that a direct page used by one part of a program doesn't conflict with the direct page used by another.

There are two instructions for accessing the direct page register: `phd`, which pushes the value in the direct page register on the stack, and `pld`, which pulls a value off the stack and places it in the direct page register. More details about these instructions and direct page addressing are provided in chapter 6.

The Arithmetic and Logical Unit

The arithmetic and logical unit, or ALU, is a component that can perform arithmetic and logical operations on data stored in a computer. It does its work with the help of the 65C816's busiest internal register, the accumulator.

As you shall soon see, the 65C816 wouldn't be much of a microprocessor if someone took away its accumulator. Every time the 65C816 is called upon to perform an operation on a value, the value first has to be placed in the accumulator.

The accumulator does its work with the help of another very busy component, the ALU. Every time the IIGS performs a calculation or a logical operation, the ALU is where the work is actually done.

The ALU performs only two kinds of calculations: addition and subtraction. The ALU solves division and multiplication problems by sequences of addition and subtraction operations.

Another job of the ALU is to compare values. But as far as the 65C816 chip is concerned, the comparison of two numbers is also an arithmetic operation. When the 65C816 chip compares two values, it subtracts one value from the other. Then, by merely checking the results of this subtraction, it can determine whether the subtracted value is more than, less than, or the same as the value it was subtracted from.

As figure 5–1 illustrates, the ALU is often depicted in diagrams as a V-shaped hopper. The ALU has two inputs (traditionally illustrated as the two arms of the hopper) and one output (represented as the bottom of the V). When two numbers are added, subtracted, or compared, one number is placed in the ALU through one of its inputs and the other number is put in through the other input. The ALU then carries out the requested calculation and puts the answer on a data bus so it can be transported to another register.

Here's a more detailed look at what happens inside the accumulator and the ALU when two numbers are added, subtracted, or compared. First, a number is stored in the 65C816's accumulator. Next, the accumulator deposits that number in the ALU through one of the ALU's inputs. The other number is placed in the ALU through its other input. Then the ALU carries out the requested calculation, and the result of the calculation finally appears at the output of the ALU. As soon as the answer appears, it is placed in the accumulator, where it replaces the value originally stored there.

Listing 5–1, a tiny assembly language program titled ADDNRS.S, shows how this process works.

Listing 5–1
ADDNRS.S program, version 1

```
lda #2
adc #2
sta $8000
```

The first statement in the ADDNRS.S program, `lda #2`, means *load*

the accumulator with the literal number 2. As you may recall from chapter 1, the # in front of the numeral 2 means the 2 is interpreted as a literal number. If there were no #, the 2 would be interpreted as the address of a memory register.

The second instruction in the listing, `adc`, means *add with carry*. In 65C816 arithmetic, the addition of two numbers often results in a carry from a low-order word to a high-order word (or from a low byte to a high byte if the processor is in emulation mode)—in much the same way that you carry numbers from one column to another in ordinary pencil-and-paper addition. If there was a carry in the ADDNRS.S program, the `adc` instruction would be able to handle it. Later in this chapter you'll find out how. But in this addition problem, there is no number to be carried, so the `adc` instruction only adds 2 and 2.

When the statement `adc #2` is executed, the 2 that has been loaded into the accumulator is deposited into one of the ALU's inputs. The instruction `adc #2` is placed in the ALU's other input. The ALU then carries out this instruction; it adds 2 and 2, and places the sum back in the accumulator.

Now you're ready for the third and last instruction in the ADDNRS.S program. The numbers 2 and 2 have been added, and their sum is now in the accumulator. The instruction in line 3, `sta`, means *store the contents of the accumulator* (in the memory address that follows). Because the accumulator now holds the value 4 (the sum of 2 and 2), the number 4 will be stored somewhere.

The memory address that follows the instruction `sta` is $8000—the hexadecimal equivalent of the decimal address 32768. So it appears that the number 4 will be stored in memory register $8000.

Now take a close look at the operand in line 3: the hexadecimal number $8000. There is no # in front of the value $8000, so the APW assembler will not interpret it as a literal number. Instead, $8000 is interpreted as a memory address—which is what a number has to be in assembly language if it is not designated as a literal number and carries no other identifying labels.

(Incidentally, if you want the assembler to interpret $8000 as a literal number, you have to write #$8000. When # and $ both appear before a number, the number is interpreted as a literal hexadecimal number. If the third line of the program was `sta #$8000`, however, there would be a syntax error. That's because `sta` is an instruction that must be followed by a value that can be interpreted as a memory address—not by a literal number.)

The Processor Status Register

The processor status register (P) is built differently from the other registers in the 65C816 and is used differently, too. Unlike the 65C816's other registers, the processor status register isn't designed for storing or processing numbers. Instead, its 8 bits are flags that keep track of several kinds of important information. Figure 5–4 shows the layout of the processor status register.

As illustrated in figure 5–4, the processor status register can be visu-

alized as a rectangular box containing eight square compartments, with a ninth and tenth compartment sitting on top. (More about those later.) Each of the lower compartments in figure 5–4 represents one of the register's 8 bits. If a bit has the binary value 1, it is set. If it has the binary value 0, it is reset, or clear.

The bits in the 65C816 status register—like the bits in all 8-bit registers—are customarily numbered from 0 to 7. By convention, the rightmost bit in an 8-bit register is referred to as bit 0, and the leftmost bit is referred to as bit 7.

The P Register Flags at a Glance

Now let's look briefly at each of the P register's ten flags. Then the operation of each flag is described in greater detail.

Status Flags

Four of the processor status register's eight bits are called status flags. They

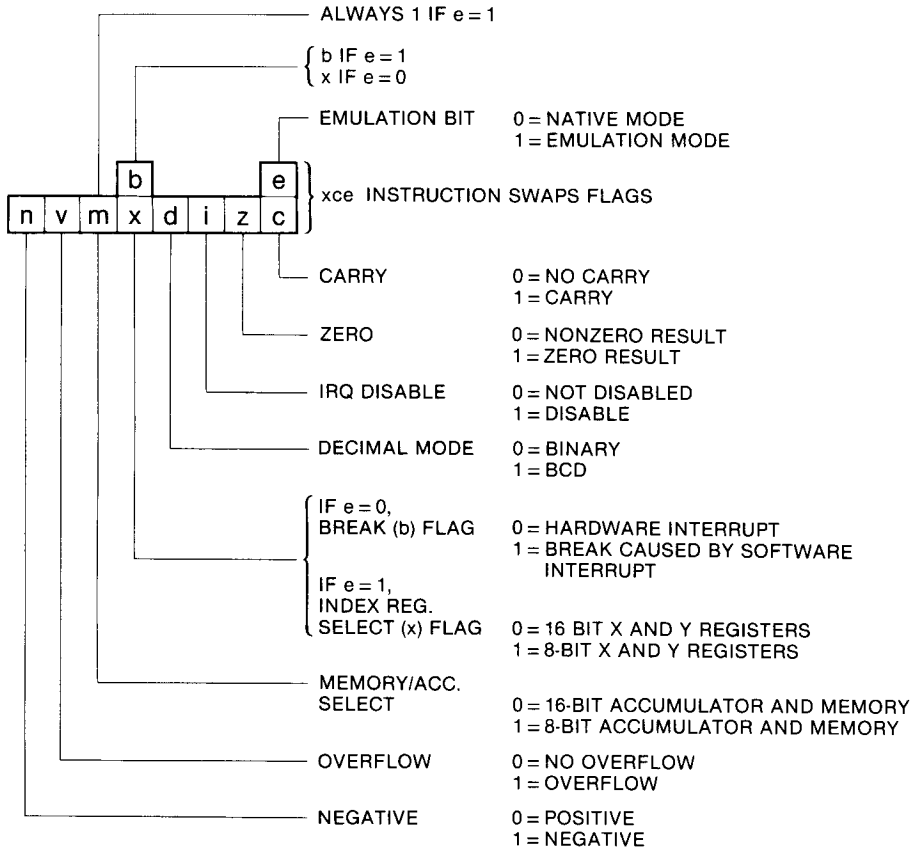


Figure 5–4
Processor status register

keep track of the results of operations carried out by the other registers inside the 65C816 processor.

- Bit 0: carry (c) flag. In arithmetic operations, the carry flag determines whether a number will be carried from one 16-bit integer to another (if the 65C816 is in native mode) or from one 8-bit byte to another (if the 65C816 is in emulation mode).
- Bit 1: zero (z) flag. Novice programmers often get confused about the way this flag works; it does the opposite of what you might expect. When the result of a calculation is 0, the zero flag is set. When the result of a calculation is not 0, the zero flag is cleared.
- Bit 6: overflow (v) flag. This bit determines if there has been a carry, or overflow, to the leftmost bit in a byte or word as the result of a calculation involving signed numbers.
- Bit 7: negative (n) flag. If the result of a calculation is negative, this flag is set. If the result of a calculation is not negative, the flag is cleared.

Condition Flags

The other four bits in the processor status register are called condition flags. They determine if certain conditions exist with respect to the configuration of the IIGS or the operation of a program.

- Bit 2: IRQ disable (i) flag. If the IRQ (interrupt) disable flag is set, interrupts are disabled. If it is clear, they are enabled.
- Bit 3: decimal mode (d) flag. If the decimal flag is set, the 65C816 performs addition and subtraction operations in binary coded decimal (BCD) mode. If it is clear, the processor will add and subtract in its normal binary mode.
- Bit 4: index register select (x) flag. This flag, together with the e flag (described in a moment), determines whether the 65C02 treats its X and Y registers as 8-bit or 16-bit registers.
- Bit 5: memory/accumulator select (m) flag or break (b) flag. When the 65C816 is in emulation mode, bit 5 is a break flag and can be read following an interrupt to determine whether the interrupt was hardware generated or software generated. When the 65C816 is in native mode, however, it doesn't need a break flag because a set of interrupt vectors make a break flag unnecessary.

  Because a break flag is not needed in native mode operations, bit 5 of the P register is free to be used for another purpose when the 65C816 is in 16-bit mode. During native mode operations, bit 5 is called the memory/accumulator select flag and is used to determine whether the accumulator and the IIGS's memory registers are treated as 8-bit or 16-bit registers.

Toggling Between Native and Emulation Mode

The processor status register also has a tenth flag. The emulation (e) flag determines whether the 65C816 will operate in native mode or emulation mode. Because the P register contains only eight bits, the e flag is a "hanging bit" that shares bit 0 with the carry (c) flag. Normally, bit 0 is a carry flag, but a special assembly language instruction—`xce`—exchanges the positions of the two flags, placing the e flag in bit 0 and making the c flag the hanging bit. The e flag can then be set or cleared using the mnemonics `sec` (set carry) and `clc` (clear carry). After the e flag is set or cleared, the `xce` mnemonic can switch the e flag and the c flag back to their original positions. As you may have guessed by now, there are some significant differences between the way the 65C816 works in native mode and in emulation mode. Switching the 65C816 back and forth between native mode and emulation mode can be a tricky business. It involves three P register flags—the e, m, and x flags—and setting them so they work together is an important part of 65C816 programming. Here are some handy facts and tips about the e, m, and x flags.

Emulation Flag

The e (emulation) flag of the processor status register determines whether the 65C816 will operate as a full-featured 16-bit chip or as an 8-bit 65C02 chip. When the e flag is set to 1, the 65C816 processor is in emulation mode and works exactly like the 65C02 chip in the Apple IIc and later models of the Apple IIe. For example, when the 65C816 is in emulation mode

- It uses an 8-bit accumulator, 8-bit X register, 8-bit Y register, and 8-bit stack pointer.
- It can address only one 64K bank of memory—either bank $00 or bank $01, depending upon soft switch settings.
- It uses page $00 as page 0, and it uses page $01 as the stack.
- To perform arithmetic and logical operations on numbers greater than 8 bits (numbers greater than 255), it must break them into smaller increments.
- When it receives an instruction to fetch data (for example, `lda`), it fetches 1 byte of data at a time, from just one memory location. When it receives an instruction to store data (for example, `sta`), it stores 1 byte of data at a time, in just one memory location.

When the e flag is cleared to 0, the 65C816 goes into native mode. Then it becomes a 16-bit chip, with these characteristics:

- Its accumulator, X register, and Y register are expanded into 16-bit registers.
- Its program bank register and data bank register are activated, giving the capability of addressing up to 16 megabytes of memory (although only 8.25 megabytes of memory are available in the Apple IIGS).
- Its stack pointer is expanded into a 16-byte register, providing it

with the capability of using a stack situated anywhere within bank $00, not limited to a memory capacity of 256 bytes.

- Its direct page register is activated, providing it with the capability of placing its direct page (the equivalent of a page 0) anywhere in bank $00.
- It becomes capable of carrying out arithmetic and logical operations on 16-bit numbers (numbers ranging from 0 to 65,535) without breaking them into smaller increments.
- When it receives an instruction to fetch data (for example, `lda`), it fetches 2 bytes of data at a time, from two consecutive memory locations. When it receives an instruction to store data (for example, `sta`), it stores 2 bytes of data at a time, in two consecutive memory locations.

As explained, the e flag can be set and cleared using the instructions `xce`, `sec`, and `clc`. There are also APW commands and macros that perform the same actions. You'll learn more about those in chapter 7 and later chapters.

Memory/Accumulator Flag

When the 65C816 is running in emulation mode—that is, when the P register's e flag is set—the 65C816 accumulator is always 8 bits wide. But when the processor is running in native mode—that is, when the P register's e flag is clear—the width of the accumulator can be set to either 8 bits or 16 bits, depending upon the setting of the P register's m (memory/accumulator) flag.

When the 65C816 is in 8-bit mode and the accumulator is 16 bits wide, its low-order bit is the A register, its high-order bit is the B register, and both bytes combined are sometimes referred to as the C register. When the accumulator is configured in this fashion, the accumulator's B register becomes an extra 65C816 register in which 8-bit values can be stored.

Here's how the B register works. When the 65C816 is switched from 16-bit mode to 8-bit mode, the accumulator's high-order bit becomes the B register, and any value that was there remains there. Any time thereafter, a new 65C816 instruction, `xba`, can exchange the values of the A and B registers. No other 65C816 instruction affects the B register. As long as the 65C816 remains in 8-bit mode, the "hidden" B register can be used as a safe storage space for any 8-bit value.

Here, in summary, is the formula for setting the width of the accumulator. If e = 1, the 65C816 is in emulation mode and the accumulator is 8 bits wide. If e = 0 and m = 0, the 65C816 is in native mode, the accumulator is 16 bits wide, and the accumulator always addresses memory 2 bytes at a time. But if e = 0 and m = 1, the 65C816 is in native mode, the accumulator is 8 bits wide, and the accumulator always addresses memory 1 byte at a time.

When the 65C816 is in native mode and the m flag is used to shorten the accumulator to 8 bits, the data stored in the B register (the accumulator's high byte) simply stays there. Because the 65C816 does not use the B register during 8-bit operations, the data remains there, untouched, until it is moved

into the lower 8 bits of the accumulator using the `xba` instruction or until the accumulator is switched back into 16-bit mode.

If you're wondering why anyone would want to use an 8-bit accumulator in 16-bit mode, there's a simple answer. For example, when you need to read a string of 1-byte ASCII characters stored in a block of memory, it's desirable to fetch them and process them 1 byte at a time. Similarly, it's sometimes desirable to write a series of 1-byte values into memory. An 8-bit accumulator can often perform jobs like that more easily and conveniently than a 16-bit accumulator.

The m flag is set using the assembly language mnemonic `sep`, which stands for *set status bits*. To use the instruction, just follow it with a 1-byte value that has a set bit in the position corresponding to the bit in the P register you want to set. You don't have to do any bit masking because zeros in the operand have no effect on their corresponding bits. Because the P register's m flag is bit 5 when the 65C816 is in native mode, you set it with the statement

```
sep %00100000
```

or

```
sep #$20
```

which means the same thing.

The m flag is cleared with the instruction `rep`, which stands for *reset status bits*. `rep` works like `sep`, but in reverse. Give it an operand with a bit set, and it clears the corresponding bit in the P register, without affecting any bits that correspond to zeros in the operation. You could therefore clear the P register's m flag with the statement

```
rep %00100000
```

or

```
rep #$20
```

It is easier to set and clear the m flag with APW directives and macros. You'll see how those methods work starting in chapter 7.

Index Register Select Flag

When the 65C816 is running in emulation mode—that is, when the P register's e flag is set—the 65C816's X and Y registers (like its accumulator) are always 8-bit registers. But when the processor is running in native mode—that is, when the P register's e flag is clear—the widths of the X and Y registers (like the width of the accumulator) can be set to either 8 bits or 16 bits, depending upon the setting of the P register's index register select (x) flag.

The x flag sets the width of both the X register and the Y register. The formula for using it is much like the formula for setting the width of the

accumulator. If e = 1, the 65C816 is in emulation mode and its X and Y registers, like its accumulator, are 8-bit registers. If e = 0 and x = 0, the 65C816 is in native mode and the X and Y registers are 16-bit registers that always access memory 2 bytes at a time. But if e = 0 and x = 1, the 65C816 is in native mode and the X and Y registers are 8-bit registers that always address memory 1 byte at a time.

The X and Y registers can be placed in 8-bit mode for the same reason that the accumulator can be turned into an 8-bit register. For example, when you need to read a string of 1-byte ASCII characters stored in a block of memory, it's desirable to access them using the X register or the Y register. And when the accumulator is in 8-bit mode, it's usually a good idea to shorten the X and Y registers, too, because it's easier to keep track of registers that are the same length.

One note of caution should be mentioned regarding the use of the x flag. When it is used to reduce the size of the X and Y registers to 8 bits, the contents of their high-order bytes are lost. So before you slice the X and Y registers in half, be sure to save the values of their high bytes if you want to use them later.

The x flag, like the m flag, can be set using the assembly language mnemonic `sep`. Because the P register's x flag is bit 4, it can be set with the statement

```
sep %00010000
```

or

```
sep #$10
```

which means the same thing.

The x flag, like the m flag, can be cleared with the `rep` instruction:

```
rep %0001100000
```

or

```
rep #$120
```

APW directives and macros make it easier to set and clear the x flag. They are covered starting in chapter 7.

A Closer Look at the P Register's Flags

Now, as promised, let's take a closer look at each bit, or flag, in the processor status register.

Carry Flag

As pointed out in chapter 2, the 65C816 cannot perform arithmetic operations on numbers longer than 16 bits (greater than 65,535) without dividing them into smaller numbers. When the 65C816 chip is in 8-bit emulation mode, its

arithmetic capabilities are reduced even further. In emulation mode, when you need to perform an operation involving a number greater than 255—or even a calculation with a result greater than 255—each number greater than 255 must be broken down into smaller numbers. When the calculation is completed, all numbers that have been split must be patched together before they can be output in a form that makes sense to the user. When the 65C816 is in native mode, it can handle larger numbers. But when an arithmetic operation involves the use of numbers greater than 65,535, they must be broken down into smaller units even when the processor is running in 16-bit mode.

This kind of mathematic "cutting and pasting," as you can imagine, involves a lot of carrying (in addition problems) and borrowing (in subtraction problems). The carry flag of the P register (bit 0) keeps up with all of this carrying and borrowing.

It is therefore considered good programming practice to clear the carry flag prior to an addition operation and to set the carry flag prior to a subtraction operation. If you don't do this, your calculations may be thrown off by the leftover results of previous calculations. The assembly language instruction to clear the P register's carry bit is `clc`, which stands for *clear carry*. The instruction to set the carry bit is `sec`, which stands for *set carry*.

Here's how the carry bit works in 6502/65C816 addition and subtraction operations. Before a multiprecision addition problem (one that requires the use of more than one word) is performed in 65C816 assembly language, the carry flag of the P register is customarily cleared using `clc`. Then the low-order words of the two numbers (or the low-order bytes, if the 65C816 is in emulation mode) are added. If this operation results in a carry to a high-order word (or byte), the 65C816 automatically sets the carry flag. Then, when the high-order words (or bytes) of the two numbers are added, the chip automatically adds the value of the carry flag. If the carry flag holds a 0, there is no carry. If it holds a 1, there is a carry, and the result of the operation is correct.

Because it is recommended that the carry flag be cleared before any addition operation, the ADDNRS.S program in listing 5–1 can be improved as shown in listing 5–2. Preceding the addition operation with the `clc` instruction clears the carry bit, ensuring that no unwanted carry is included in the operation. You'll see more examples of how the carry bit works in addition problems later in this book.

Listing 5–2
ADDNRS.S program, version 2

```
clc
lda #2
adc #2
sta $8000
```

The carry flag is also used in subtraction problems, but in the opposite way from its use in addition problems. Before a subtraction operation, the carry bit is usually set using `sec`. Then, if the subtraction operation requires

that a low-order word or byte borrow a number from a high-order word or byte, the number needed is provided by the carry bit. The carry flag has other uses, most of which are described in later chapters.

Zero Flag

When the result of an arithmetic or logical operation is 0, the status register's zero flag (bit 1) is automatically set. Addition, subtraction, and logical operations can all result in changes to the status of the zero flag. The zero flag is often tested in programming loops that count down to 0 and to see if two numbers are equal.

When you write routines that use the zero flag, it's important to remember one 6502/65C816 convention that may seem odd at first. When the result of an operation is 0, the zero flag is set to 1. When the result of an operation is not zero, the zero flag is cleared to 0. This convention is easy to forget—and can trip you up if you aren't careful.

There are no assembly language instructions to clear or set the zero flag. It's strictly a read bit, so instructions to write to it are not provided.

Interrupt Disable Flag

The Apple IIGS, unlike many earlier members of the Apple II family, supports a wide variety of interrupts, instructions that halt all 6502/65C816 operations temporarily so that more time critical operations can take place. Some interrupts are called *maskable interrupts* because you can prevent them from taking place by setting the interrupt disable flag (bit 2) of the processor status register. Other interrupts are called *nonmaskable interrupts* because they are essential to the operation of a computer and you can't stop them from taking place.

The most common reason for using the P register's interrupt disable flag is to write a sequence of code that would not work properly if an interrupt took place while the code is executed. For example, if a program is setting up an interrupt and gets cut off in midstream by another interrupt, the whole program might crash. The best way to keep this kind of disaster from happening is to set the interrupt disable flag, execute the sensitive segment of code, and then clear the interrupt disable flag. That way, an unexpected interrupt cannot come along and crash the program.

The assembly language instruction to clear the interrupt flag is `cli`. The instruction to set the interrupt flag is `sei`. Examples showing how this flag works are presented in later chapters.

Decimal Mode Flag

The 65C816 processor normally operates in binary mode, using standard binary numbers. But the chip can also operate in binary coded decimal, or BCD, mode. To put the computer in BCD mode, you have to set the decimal flag of the 65C816 status register.

When the 65C816 is in BCD mode, it uses the same ten digits used in the standard decimal system: the numbers 0 through 9. Because the hexa-

decimal digits A through F are not used in the BCD system, they are not recognized by the 65C816 when the IIGS is in BCD mode.

Table 5–1 shows how the IIGS converts the numbers 0 through 9 into BCD numbers when the 65C816 is in BCD mode. It also shows the hexadecimal and binary equivalents of the decimal numbers 0 through 15.

As table 5–1 shows, the binary numbers 1010 through 1111, which equate to the digits A through F in the hexadecimal system and 10 through 15 in the decimal system, are nt used when the 65C816 chip is in BCD mode. Instead, the numbers 10 through 15 are written in the BCD system as the separate digits 1 and 0 through 1 and 5, just as they are in the standard decimal system. For example, the number 13 is written in BCD as the binary equivalent of 1 (0001) and 3 (0011). So, when the 65C816 is in BCD mode, it converts the decimal values 11 through 15 into the binary numbers 0001 0000 through 0001 0101.

Because the binary numbers 1010 through 1111 are not used in the BCD system, it takes more memory to store numbers using BCD notation than it does to store non-BCD binary numbers. In many applications (for example, in floating-point arithmetic operations), a full byte of memory is used for each decimal digit in a BCD number. When BCD notation is used in this way, BCD numbers require even more memory.

Figure 5–5 shows how the decimal number 255 is stored in memory as a BCD number if each digit in the number is expressed as an individual byte. In comparison, figure 5–6 shows how the 65C816 chip stores the decimal number 255 in memory if the BCD flag is not set.

As figures 5–5 and 5–6 illustrate, at the rate of one byte per digit, it takes three times as many bytes to store the number 255 in BCD notation as

Table 5–1
BCD-to-Binary Conversion

Decimal	Hexadecimal	BCD Notation	Binary Notation
0	0	0000	0000
1	1	0001	0001
2	2	0010	0010
3	3	0011	0011
4	4	0100	0100
5	5	0101	0101
6	6	0110	0110
7	7	0111	0111
8	8	1000	1000
9	9	1001	1001
10	A	0001 0000	1010
11	B	0001 0001	1011
12	C	0001 0010	1100
13	D	0001 0011	1101
14	E	0001 0100	1110
15	F	0001 0101	1111

```
BCD NUMBER:              2        5        5
BINARY EQUIVALENT: 00000010 00000101 00000101
```

Figure 5–5
Expressing a number in BCD mode

```
DECIMAL NUMBER:           255
HEXADECIMAL EQUIVALENT    FF
BINARY EQUIVALENT         11111111
```

Figure 5–6
Expressing a number in binary mode

it does in binary notation. There are many applications in which BCD numbers use even more memory. For example, when the 65C816 performs floating-point arithmetic, extra bytes are usually required to indicate how many digits are in the number, whether the number is positive or negative, and how many decimal places are in the number.

In floating-point arithmetic—which is often used in "number-crunching" operations because of its high degree of accuracy—it could take six or more binary numbers to express a three-digit decimal number. Figure 5–7 shows how the number 2.55 is expressed as a 6-byte BCD number. This illustration shows only one of the many methods for converting decimal numbers into BCD numbers for use in floating-point operations.

In addition to using extra memory, BCD arithmetic is slower than binary arithmetic. But because BCD numbers, like decimal numbers, are based on 10, they are also more accurate in arithmetic operations that use fractions and decimal values. So BCD arithmetic is often used in programs in which accuracy of calculations is more important than speed or memory efficiency.

Converting BCD numbers into decimal numbers is also easier than converting standard binary numbers. So BCD numbers are sometimes used in programs that require the instant display of numbers on a video monitor.

When the status register's decimal mode flag is set, the 65C816 chip performs all its arithmetic using BCD numbers. You probably won't be using much BCD arithmetic in your assembly language programs—at least not for

```
DECIMAL NUMBER:       2.55
FLOATING-POINT BCD:   0011 0010 0000 0010 0101 0101

                MEANING OF EACH BCD DIGIT

0011 (3):  THE NUMBER HAS THREE DIGITS
0010 (2):  DECIMAL POINT IS TO THE LEFT OF THE DIGIT 2
0000 (0):  THE NUMBER IS POSITIVE (0001 WOULD MEAN NEGATIVE)
0010 (2):  FIRST DIGIT (2)
0101 (5):  SECOND DIGIT (5)
0101 (5):  THIRD DIGIT (5)
```

Figure 5–7
A floating-point binary number

a while—so you'll usually want to make sure that the decimal flag is clear before the computer starts performing arithmetic operations.

The assembly language instruction that clears the decimal flag is `cld`. The `sed` instruction sets it. The `cld` instruction is often used before arithmetic operations take place to ensure that the 6502/65C816 chip has not been placed and left in decimal mode. So a further improved version of the ADDNRS.S program presented in listing 5–1 is shown in listing 5–3.

Listing 5–3
ADDNRS.S program, version 3

```
cld
clc
lda #2
adc #2
sta $8000
```

Index Register Select Flag or Break Flag

Bit 4 of the processor status register is an index register select (x) flag when the 65C816 is in native mode and a break (b) flag when the processor is in emulation mode.

You have seen how bit 4 works in its role as an index register select flag. Now you will take a brief look at how it is used in emulation mode, in its capacity as a break flag.

When the assembly language instruction `brk` halts a program and the 65C816 is in emulation mode, an interrupt is generated, the program halts, and the b flag is set automatically. If an interrupt is hardware generated, however, the b flag is not set.

The `brk` instructions that result in the setting of the break flag are often used by program designers during debugging. After a program is debugged, any `brk` instructions placed in the program for use during debugging are usually removed. Other than the `brk` menmonic, there are no assembly language instructions that set or clear the break flag.

Memory/Accumulator Select Flag

When the 65C816 is in native mode, bit 5 is the memory/accumulator select flag (m), which we have discussed. In emulation mode and in pre-65C816 processors, bit 5 is not used.

Overflow Flag

The overflow flag, bit 6, detects an overflow from the next-to-leftmost bit to the leftmost bit in a binary number. The overflow flag is used primarily in addition and subtraction problems involving signed numbers. When the 65C816 microprocessor performs calculations on signed numbers, each number is expressed as a 15-bit value (or as a 7-bit value in emulation mode), with the leftmost bit designating the number's sign. When the leftmost bit is

used in this way, an overflow from the next-to-leftmost bit to the leftmost bit can make the result of a calculation incorrect. So after a calculation involving signed numbers is performed, the v flag is often tested to see whether such an overflow has occurred. Then, if an unwanted overflow has occurred, you can take corrective action.

The assembly language instruction that clears the overflow flag is `clv`. The v flag is a read-only bit, so there is no specific instruction to set it.

Negative Flag

The negative flag, bit 7, is set when the result of an operation is negative and cleared when the result of an operation is 0. The negative flag is often used in operations involving signed numbers. The negative flag also can be tested to see whether one number is less than another number and used to detect whether a counter in a loop has decremented past 0. Other uses are discussed in later chapters. There are no instructions to set or clear the negative flag; it's strictly a read-only bit.

CHAPTER 6

The Right Address

The Addressing Modes of the 65C816

In chapter 2, you saw the one-to-one correlation between assembly language and machine language. For every mnemonic in an assembly language program, there's a numeric machine language instruction that means the same thing.

In chapter 5, you saw that while that's the truth, it isn't quite the whole truth. Most instructions in 6502/65C816 assembly language have more than one equivalent instruction in machine language. For example, when the `adc` mnemonic is used in a IIGS program, it can be converted into 15 different numeric instructions when it is assembled into machine language. To understand why this is true, you need to know how to use addressing modes in 6502/65C816 assembly language.

In the world of assembly language programming, an addressing mode is a tool for locating and using information stored in a computer's memory. The 65C816 can access the memory locations in the IIGS in 24 ways; in other words, it has 24 addressing modes.

In this chapter, you examine all 24 of the 65C816's addressing modes, and you see how to use them in IIGS assembly language. First, though, let's look at the 15 ways that one mnemonic—`adc`—can be converted into machine language. See table 6–1.

Later in this chapter, you'll examine all these addressing modes and see how they work in assembly language programs. First, though, let's compare the assembly language statements and the machine language statements listed in table 6–1.

Table 6–1
15 Ways to Address the adc Mnemonic

Addressing Mode	Assembly Language Statement	Machine Language Equivalent	Bytes
Immediate	`adc #$03`	69 03	2
Direct	`adc $03`	65 03	2
Direct indexed with X	`adc $03,x`	75 03	2
Absolute	`adc $0300`	6D 00 03	3
Absolute indexed with X	`adc $0300,x`	7D 00 03	3
Absolute indexed with Y	`adc $0300,y`	79 00 03	3
Direct indexed indirect	`adc ($03,x)`	61 03	2
Direct indirect indexed	`adc ($03),y`	71 03	2
Direct indirect	`adc ($0300)`	72 03	2
Stack relative indexed indirect	`adc (3,S),y`	73 03	2
Stack relative	`adc 3,s`	63 03	2
Direct indirect long	`adc [$03]`	67 03	2
Direct indirect long indexed	`adc [$03],y`	77 03	2
Absolute long	`adc $030300`	6F 00 03 03	4
Absolute long indexed with X	`adc $030300,x`	7F 00 03 03	4

In the assembly language column, all 15 statements have the same mnemonic, but each has a different operand. In the machine language column, the statements have quite a different structure. There are 15 different opcodes, but only three kinds of operands: the 1-byte operand 03, the 2-byte operand 00 03, and the 3-byte operand 00 03 03.

This arrangement illustrates an important difference between assembly language and machine language, a difference that you first observed in chapter 2. In 6502/65C816 machine language, addressing modes are distinguished by differences in their opcodes. But in 6502/65C816 assembly language, the 24 available addressing modes can be identified by differences in their operands.

The Addressing Modes of the 65C816

Table 6–2 shows the 24 addressing modes recognized by the 65C816. As you can see, they can be divided into five categories:

- Simple addressing
- Indexed addressing
- Indirect addressing

- Stack addressing
- Block move addressing

In this chapter, you'll examine these five addressing modes and all 24 of the 65C816's addressing modes.

Table 6–2
The 65C816's 24 Addressing Modes

Simple Addressing		
Addressing Mode	Example	Identifier
Implied	`rts`	i
Immediate	`lda #2`	#
Absolute	`lda $0C00`	a
Absolute long	`lda $030300`	al
Direct	`sta $FA`	d
Accumulator	`inc a` (or `ina`)	Acc
Program counter relative	`bcc label`	r
Program counter relative long	`brl label`	rl
Indexed Addressing		
Addressing Mode	Example	Identifier
Absolute indexed with X	`lda $0C00,x`	a,x
Absolute indexed with Y	`lda $0C00,y`	a,y
Direct indexed with X	`lda $FA,x`	d,x
Direct indexed with Y	`stx $FA,y`	d,y
Absolute long indexed with X	`lda $030300,x`	al,x
Indirect Addressing		
Addressing Mode	Example	Identifier
Direct indirect	`lda ($FA)`	(d)
Direct indirect long	`lda [$FA]`	[d]
Absolute indirect	`jml ($0300)`	(a)
Absolute indexed indirect	`jsr ($0300,x)`	(a,x)
Direct indexed indirect	`lda ($FA,x)`	(d,x)
Direct indirect indexed	`lda ($FA),y`	(d),y
Direct indirect long indexed	`lda [$03],y`	[d],y
Stack Addressing		
Addressing Mode	Example	Identifier
Stack	`pha`	s
Stack relative	`lda $30,s`	r,s
Stack relative indirect indexed	`lda ($30,s),y`	(r,s),y
Block Move Addressing		
Addressing Mode	Example	Identifier
Block source bank, destination bank	`mvn 6,0`	xya

Simple Addressing Modes

The 65C816 has the following simple adddressing modes:

- Implied addressing
- Immediate addressing
- Absolute addressing
- Absolute long addressing
- Direct addressing
- Accumulator addressing
- Program counter relative addressing
- Program counter relative long addressing

Listing 6–1, titled AddrDemo1, uses four addressing modes. They are all simple addressing modes, but one of them—simple stack addressing—can also be classified as a stack addressing mode (as it is in table 6–2). First you'll examine each addressing mode in the AddrDemo1 program. Then you'll see how each instruction in the program works and what the program does.

Listing 6–1
AddrDemo1 program

```
*
*    ADDRESSING DEMO #1: Four kinds of addressing
*

             KEEP AddrDemo1

Demo         START

result       equ $2000

             phk                              ; stack addressing
             plb                              ; stack addressing

             lda #$2200                       ; immediate address
             clc                              ; implied address
             adc #$0022                       ; immediate address
             sta result                       ; absolute address

             brk                              ; implied address

             END
```

The four addressing modes used in listing 6–1 are:

- Stack addressing
- Implied addressing

- Immediate addressing
- Absolute addressing

Let's take a close look at each of these four addressing modes. Then, with the help of some other short programs, you'll examine the rest of the 65C816's 24 addressing modes.

Stack Addressing

To understand how stack addressing works, it helps to know what a stack is. A stack, sometimes known as a hardware stack, is an area of RAM that is often compared with a stack of plates in a diner. When you place a value on the stack, it "covers up" the value previously in the top position on the stack and becomes the new top value on the stack. To get to the value that was previously on top, you have to remove the value that was just added. Then the value that was on the top of the stack before becomes the top value again.

This stacked plate analogy, as you shall see later in this chapter, is not completely accurate. But we can use it to explain how stack addressing works in the AddrDemo1 program.

In the AddrDemo1 program, stack addressing is used in the lines

```
phk                                     ; stack addressing
plb                                     ; stack addressing
```

In these two lines, the value of the 65C816 program bank register is placed on the stack. Then it is pulled off the stack and deposited in the 65C816 data bank register.

The mnemonic in the first line, `phk`, means *push the program bank register on the stack.* It does exactly what its name suggests. The mnemonic in the second line, `plb`, means *pull the top value off the stack and place it in the data bank register.* It does what its name implies, too.

When the `phk` and `plb` instructions are used together at the beginning of a program, as they are in AddrDemo1, they ensure that the program and its data use the same 64K bank of memory. It is sometimes desirable—even necessary—for a program to access data stored in another bank. On those occasions, the value of the data bank register can be changed temporarily. But most of the time, the program bank and the data bank should be the same. If they aren't, instructions that fetch and store data—such as `lda` and `sta`—might try to access data in the wrong banks, causing crashes and other programming catastrophes.

Using stack addressing to change the value of the data bank register is indirect and inconvenient, but there's one good reason for it. It's the only method the 65C816 instruction set provides.

Types of Stack Addressing

As table 6–2 shows, there are three major types of stack addressing: simple stack addressing, stack relative addressing, and one complex form of stack addressing called stack relative indirect indexed addressing. In the AddrDemo 1 program, the `phk` and `plb` instructions use simple stack addressing. The other two kinds of stack addressing are covered later in this chapter.

Mnemonics that use stack addressing are all 1-byte instructions (which means they don't have operands), and all but three—`rts`, `rtl`, and `rti`—start with *p*. Some stack instructions push values onto the stack, some pull values off the stack, and three—the three that begin with *r*—pull addresses off the stack and use them as addresses to jump to.

Emulation Mode and Native Mode

There are some differences between the way stack addressing works when the 65C816 is in 16-bit native mode and 8-bit emulation mode. For example, in emulation mode, the stack pointer is always on page 1 and has only 256 addresses. But when the processor is in native mode, the stack can start at any address in bank 0, and the length of the stack is limited only by the amount of available RAM in that bank.

Another difference is that some instructions push only 1 byte onto the stack in emulation mode, but all instructions push at least 2 bytes onto the stack when the processor is in native mode. The differences between native mode and emulation mode operations are described in table 6–3.

Table 6–3
Simple Stack Addressing Operations

Instructions	Operations
`brk, cop` (software interrupts)	Push PBR, P, and PC onto the stack
`irq, nmi, abort, res` (hardware interrupts)	Push PBR, P, and PC onto the stack
`rti`	Pull P, PC, and PBR off the stack
`rts`	Pull PC off the stack
`rtl`	Pull PC and PBR off the stack
`pei`	Push a direct page word onto the stack
`pea`	Push bytes 3 and 2 of the instruction onto the stack; this is really a push immediate instruction
`per`	Push onto the stack a value obtained by adding the PC to the contents of bytes 3 and 2 of the instruction
`pha, phb, phd, phk, php, phx, phy`	Push register contents onto the stack. (Number of bytes pushed varies, depending on the register pushed and the processor mode.)
`pla, plb, pld, plp, plx, ply`	Pull the top element off the stack and into the register. (Number of bytes pulled varies, depending on the register pushed and the processor mode.)

Implied Addressing

Another kind of 1-byte addressing—implied addressing—appears in these two lines of the AddrDemo1 program:

```
clc                                     ; implied address
```

and

```
brk                                    ; implied address
```

In the implied addressing mode, the operand is not spelled out, but merely understood, like the understood object of an intransitive verb in English grammar. When you use implied addressing, all you have to type is the three-letter assembly language instruction. Its syntax does not require (in fact does not allow) the use of an expressed operand.

Immediate Addressing

Two lines in the AddrDemo1 program use immediate addressing:

```
lda #$2200
```

and

```
adc #$0022
```

When immediate addressing is used in a 65C816 instruction, the operand that follows the opcode mnemonic is a literal number—not the address of a memory location. So in a statement that uses immediate addressing, # (the symbol for a literal number) always appears before the operand.

When an immediate address is used in an assembly language statement, the assembler does not have to `peek` into a memory location to find a value. Instead, the value itself is placed directly into the accumulator. Then the operation that the statement calls for can be immediately performed; in other words, an immediate address forms the effective address of an operand.

When the 65C816 is in native mode and its accumulator and index registers are in their 16-bit modes, every instruction that uses immediate addressing has a 2-byte operand. But when the 65C816 is in emulation mode, or when its accumulator and index registers are in their 8-bit modes, instructions that use immediate addressing have 1-byte operands.

The immediate addressing mode is often used to create pointers, or addresses that point to other address. For example, the following code segment converts the address of a block of data called `Picture` into a pointer stored in a variable called `PicPtr`:

```
lda #<Picture
sta PicPtr
lda #^Picture
sta PicPtr+2
```

This fragment of code uses two forms of addressing: immediate addressing and absolute addressing, which are covered in the next sections. Absolute addressing uses an operand that specifies a memory location as its effective address.

In this code, the statements that use immediate addressing are `lda #<Picture` and `lda #^Picture`. The statements that use absolute addressing are `sta PicPtr` and `sta PicPtr+2`.

This code loads the 24-bit address of the data segment `Picture` into a pointer situated in a pair of memory addresses labeled `PicPtr` and `PicPtr+2`. If the fragment were encountered in an assembly language program, it would load the 24-bit address of the data segment `Picture` into a 2-word pointer labeled `PicPtr`, depositing the low-order word of the address in `PicPtr` and placing the high-order word in `PicPtr+2`.

In this code, < and ^ are special symbols recognized as directives by the APW assembler. They are used as prefixes of the label `Picture` so that the APW assembler will split the address of the data segment specified by the label `Picture` into two 16-bit words. One word can then be loaded into the pointer `PicPtr`, and the other can be loaded into `PicPtr+2`.

When the APW assembler encounters the statement `lda #<Picture`, it loads the 2 low bytes of the address of `Picture` into the pointer `PicPtr`. When it reaches the statement `lda #^Picture`, it loads the 2 high bytes of the address of `Picture` into `PicPtr+2`. The full address of the data segment `Picture` is stored, in the 65C816's typical low-byte-first format, in the two memory addresses labeled `PicPtr` and `PicPtr+2`. For example, if the address of the data block `Picture` is $E12000, the value $2000 (the low word of the address) is stored in `PicPtr`, and the value $00E1 (the high word of the address) is stored in `PicPtr+2`.

The symbol < in the statement `lda #<Picture` is optional. It can be eliminated, as it is in these lines of code:

```
lda #Picture
sta PicPtr
lda #^Picture
sta PicPtr+2
```

Absolute Addressing

One line in the AddrDemo1 program uses absolute addressing:

```
sta result                              ; absolute address
```

In this line, the word `result` is a symbolic label defined previously in the program:

```
result   equ $2000
```

So the symbolic label `result` in the statement

```
sta result
```

stands for the hexadecimal value $2000.

If this line was written as

```
sta #result
```

the APW assembler would assemble the value $2000 into a literal number, and the addressing mode used in the statement would be immediate addressing.

In this case, however, the operand of the `sta` mnemonic is not preceded by #, so the APW assembler does not interpret it as a literal number. Instead, as you have seen in programs in chapter 2, the operand in the statement `sta result` is interpreted as a memory address. Another way of saying this is that in the AddrDemo1 program, the statement `sta result` uses absolute addressing.

Now you see that in a statement using absolute addressing, the operand is a memory location, not a literal number. In reading and writing operations that use absolute addressing, the operation called for is always performed on the value stored in the specified memory location, not on the operand itself. When a jump instruction (`jmp` or `jsr`) uses absolute addressing, however, the address jumped to is the absolute address that is expressed as the operand.

In both native mode and emulation mode, every instruction that uses absolute addressing has a 16-bit operand. When the 65C816 is in native mode, however, the assembler extends the effective address of the operand to 24 bytes by concatenating it with a bank register. If the instruction that uses absolute addressing is a read or write instruction, such as `lda` or `sta`, the assembler extends the operand to 3 bytes by combining it with the 65C816's data bank register. If the instruction is a jump instruction (`jmp` or `jsr`), the assembler extends the operand to 3 bytes by combining it with the program bank register.

How the AddrDemo1 Program Works

You have completed an analysis of the addressing modes in the AddrDemo1 program and are ready to see how it works.

As noted, the lines

```
phk                                    ; stack addressing
plb                                    ; stack addressing
```

copy the contents of the program bank register into the data bank register, so the program accesses data from the same bank in which the program is running. Now let's look at the lines

```
lda #$2200                             ; immediate address
clc                                    ; implied address
adc #$0022                             ; immediate address
sta result                             ; absolute address
```

In the statement `lda #$2200`, the 65C816's accumulator is loaded with

the literal value $2200. Then the mnemonic `clc` clears the P register's carry flag in preparation for an addition operation.

Next, in the statement `adc #$0022`, the literal value $0022 is added to the value of $2200 that is already in the accumulator. Finally, the statement `sta result` stores the result of the addition—the number $2222—in an absolute memory address.

What is this memory address? Because the symbolic label `result` was assigned the value $2000 and the mnemonic `sta` is a write instruction and not a jump instruction, the APW assembler calculates the effective address of the operand result by concatenating the value of the result with the contents of the 65C816's data bank register. In other words, the effective address of the operand is the address $2000 in whatever data bank the program is loaded into.

And what data bank is that? Well, frankly, there's no way of knowing. As you learned in chapter 4, it is up to the IIGS system loader, not the IIGS programmer, to decide where to place a program when it is loaded into memory. And when a program has been loaded into memory, the IIGS Memory Manager can move it. So, when you write a program for the IIGS, you can never be sure where the program will start in memory or even what bank it will be loaded into.

When you type, run, assemble, and load the AddrDemo1 program, you can only be sure that the result of the addition of the numbers $2200 and $0022 are stored in memory addresses $2000 and $2001 in some bank of memory.

You won't have to stay in the dark for very long, however. The last line in AddrDemo1 is

```
brk                                    ; implied address
```

As soon as you run the program, you will hear a beep from your computer and will discover that the `brk` instruction, which ends the program, has "bounced" the program into the IIGS monitor. You will see the contents of all the 65C816's registers, including its data bank register (D), listed on the screen. You can use your monitor's display memory functions (described in chapter 2) to list the contents of memory addresses $2000 and $2001 in the 64K bank pointed to by the data bank register. If the 2-byte value stored in those two addresses is $2222—the sum of $2200 and $0022—you'll know that the AddrDemo1 program worked properly.

Direct Addressing

If you're an experienced 6502/65C02 programmer, you're familiar with the concept of page 0 addressing, a technique that can save time and allow memory locations to be addressed in some tricky (and quite useful) ways.

In pre-GS Apple IIs, page 0 is a 256-byte block of RAM that extends from memory address $00 through memory address $FF. Every memory location on page 0 has a 1-byte address and thus can be addressed using a 1-byte operand. Another noteworthy fact about page 0 is that some addressing—as you shall see later in this chapter—actually requires direct page operands.

Because the 256 memory addresses on page 0 are so valuable, page 0 is the high-rent district in pre-GS Apple IIs. It is such a desirable piece of real estate, in fact, that the designers of the Apple II operating system, the Apple II monitor, and Applesoft BASIC claimed most of it for themselves. They left only a few bytes free for use in application programs.

Because space on page 0 is so useful and so scarce, designers of 6502-based computers tried for years to increase the amount of page 0 storage space. In designing the Apple IIGS, they finally succeeded. In the IIGS, as you may recall from chapter 4, the concept of page 0 addressing is expanded into something called direct page addressing. This form of addressing allows any page in bank $00 to be used as a page 0 and allows different programs, or even different segments of the same program, to use different pages in bank $00 as their own private page 0.

Because a IIGS program can use any page in bank $00 as a page 0, the form of addressing that was called page 0 addressing is now more properly referred to as direct page addressing. The page of bank $00 memory that is accessed through direct page addressing is no longer known as page 0, but is more properly referred to as the direct page.

In a statement that uses direct page addressing, the operand always consists of just 1 byte—a number from $00 to $FF. When the 65C816 assembles a statement that uses direct addressing, it interprets the operand as an offset that, when added to the contents of the data bank register, specifies the operand's effective address.

That's quite a mouthful, but listing 6–2 is a short program that shows how direct addressing works.

Listing 6–2
AddrDemo2 program

```
*
*    ADDRESSING DEMO #2: Direct addressing
*
             KEEP AddrDemo2

Demo         START

             phk                          ; make program bank and
             plb                          ; data bank the same
             lda #$2000                   ; make the direct page
             tcd                          ; start at $2000
             lda #$5500                   ; immediate address
             clc
             adc #$0055                   ; immediate address
             sta $60                      ; direct page address

             brk                          ; quit to the monitor

             END
```

How the AddrDemo2 Program Works

AddrDemo2, like AddrDemo1, starts with the instructions

```
phk                             ; make program bank and
plb                             ; data bank the same
```

These statements, as their comments now reveal, make the program bank and the data bank the same.

The next lines are

```
lda #$2000                      ; make the direct page
tcd                             ; start at $2000
```

These two lines are very important. They set aside page #$20 in bank $00, memory addresses $2000 through $20FF, for use as a direct page.

The next three lines work much like their corresponding lines in the previous program:

```
lda #$5500                      ; immediate address
clc
adc #$0055                      ; immediate address
```

They add the literal numbers $5500 and $0055, taking care to clear the carry flag before the addition is carried out so that the result of the operation is correct.

The next line is the part of the AddrDemo2 program that you have been waiting for:

```
sta $60                         ; direct page offset
```

Using the value $60 as an offset, this line stores the result of the addition of $5500 and $0055 in the direct page address $2060.

The AddrDemo2 program, like the AddrDemo1 program, ends with a `brk` instruction so that you can use the IIGS monitor to check its results. Type, assemble, and run the program. Then use your monitor to `peek` into memory addresses $00/2060 and $00/2061. If everything has worked correctly, those two memory locations now hold the 2-byte value $5555—the sum of the addition of $5500 and $0055.

Forcing Absolute Addressing

Now that you know how the AddrDemo2 program works, let's go back and take another look at the line

```
sta $60
```

If you've written assembly language programs for pre-GS Apple IIs, you may notice that this statement works much differently in the AddrDemo2 program than it would in a 6502 or 65C02 program. In the AddrDemo2 program, the operand $60 in the statement `sta $60` is not a complete address, but merely an offset that is used to calculate a direct page address. But if the AddrDemo2 program were written for an 8-bit chip—or for a 65C816 chip running in emulation mode—the operand $60 would be interpreted as a literal address: the page 0 address $60.

This brings us to a problem faced by Apple IIGS assembly language programmers. Because the 65C816 interprets the 1-byte operand in a statement like `sta $60` as an offset for calculating a direct page address, there is no straightforward way to access 1-byte addresses in the program bank or data bank currently in use. In other words, there is no direct way to access the addresses $00 through $FF in the current program or data bank.

Suppose you are writing a 65C02 program. You want the operand in the statement `sta $60` to be assembled not as a direct page offset, but as absolute memory address $0060 in the current data bank. What would you do?

Fortunately, there is a way out of this dilemma. If you are writing a program with the APW assembler, and you want the statement `sta $60` to mean *store the value of the accumulator in the absolute address $XX0060* (with *XX* representing the current data bank), you could force APW to assemble it that way by merely writing

```
sta |$60
```

or

```
sta !$60
```

You can use a vertical bar or an exclamation point as a prefix to force absolute addressing.

The prefix | or the prefix ! can also force absolute addressing in statements that use symbolic labels as operands. For example, if the symbolic label `memloc` is defined as the value $333 in an assembly language program, the statement

```
lda |memloc
```

or the statement

```
lda !memloc
```

cause the operand `memloc` to be interpreted as the absolute address $XX0333. So the accumulator is loaded with the value stored at that physical address—not at the address calculated by adding $333 to the contents of the direct page register.

Forcing Absolute Long Addressing

Now that you have dealt with the problem of forcing absolute addressing, you're ready to look at another problem that arises often in IIGS assembly language programming. Suppose you are writing a 65C02 program, and you want the operand in the statement `sta $60` to be assembled as the absolute address $000060—in other words, as an absolute long address in bank $00. What would you do?

The APW assembler also provides a solution to this problem. If you are writing a program in which you want the statement `sta $60` to mean *store the value of the accumulator in address $000060,* you can force the assembler to assemble it as an absolute long address by writing

```
sta >$60
```

The > prefix forces absolute long addressing. You'll see more examples of absolute long addressing later in this chapter.

The > prefix can also force absolute long addressing in statements that use symbolic labels as operands. For example, if the symbolic label `memloc` is defined as the value $333 in an assembly language program, the statement

```
lda >memloc
```

causes the operand `memloc` to be interpreted as an absolute long address. So the accumulator is loaded with the value stored in memory address $000333. But the statement

```
lda memloc
```

is interpreted as a direct address. In this case, the accumulator is loaded with the value stored in a direct page address calculated by adding the literal value $333 to the contents of the direct page register.

A direct page operand can be written using the < prefix, as in the following examples:

```
lda <$60
lda <memloc
```

When < is used in this way, it is ignored by the APW assembler. It merely shows people reading the program that the addressing mode is direct addressing.

Absolute Long Addressing

Another example of absolute long addressing appears in listing 6–3, AddrDemo3.

Listing 6–3
AddrDemo3 program

```
*
*    ADDRESSING DEMO #3: Absolute long addressing
*

             KEEP AddrDemo3

Demo         START

             phk                          ; make the program bank
             plb                          ; and data bank the same

             lda #$BB00                   ; immediate address
             clc
             adc #$00BB                   ; immediate address
             sta $012030                  ; absolute long address

             brk                          ; quit to the monitor

             END
```

In the AddrDemo3 program, the lines

```
lda #$BB00                   ; immediate address
clc
adc #$00BB                   ; immediate address
sta $012030                  ; absolute long address
```

add the literal numbers $BB00 and $00BB, and store their sum in the absolute long address $012030. After you type, assemble, and run the program, you can confirm that it works by using the IIGS monitor to view the contents of memory addresses $01/2030 and $01/2031.

In the AddrDemo3 program, the absolute long address $012030 is expressed in the easiest possible way: as a literal number. Operands are usually expressed as literal numbers in programs that use absolute long addressing.

Accumulator Addressing

The accumulator addressing mode performs an operation on a value stored in the 6502/65C816 processor's accumulator. When you use accumulator addressing mode, some assemblers require that you use an `a` as an operand. The APW assembler requires the use of the `a` operand in all but three cases. The aliases `cpa`, `dea`, and `ina` can be substituted for the assembly language statements `cmp a`, `dec a`, and `inc a`.

Another example of a statement that uses the accumulator addressing mode (no alias allowed) is `asl a`. This statement rotates each bit in the accumulator one position to the left, with the leftmost bit (bit 15 in native mode or bit 7 in emulation mode) dropping into the carry bit of the processor status (P) register.

Program Counter Relative Addressing

Program counter relative addressing is used for branching—a method for instructing a program to jump to a given routine under certain conditions. There are nine branching instructions in 65C816 assembly language. All begin with *b*, which stands for *branch to*, and eight use program counter relative addressing.

Some examples of branching instructions are

- `bcc`: Branch to a specified address if the carry flag is clear.
- `bcs`: Branch to a specified address if the carry flag is set.
- `beq`: Branch to a specified address if the result of an operation is equal to 0.
- `bne`: Branch to a specified address if the result of an operation is not equal to 0.
- `bra`: Branch always.

The `bra` mnemonic is one of two unconditional branching instructions used in 65C816 assembly language. The other unconditional branching mnemonic, `brl` (branch always—long), uses another form of addressing, called program counter relative long addressing, which is covered in the next section. All nine branching instructions are described in chapter 5, in the section devoted to the 65C816 instruction set.

The nine branching mnemonics are often used with three other instructions called comparison instructions. Typically, a comparison instruction compares two values, and the conditional branch instruction then determines what should be done according to the result of the comparison.

The three comparison instructions are

- `cmp`: Compare the number in the accumulator with . . .
- `cpx`: Compare the value in the X register with . . .
- `cpy`: Compare the value in the Y register with . . .

Conditional branching instructions can also follow arithmetic operations, logical operations, and various kinds of bit testing operations.

Usually, a branch instruction causes a program to branch to a specified address if certain conditions are met or not met. A branch might be made,

for example, if one number is larger than another, if two numbers are equal, or if an operation results in a positive, negative, or zero value.

(The AddrDemo4 program shows one way to use program counter relative addressing. We present this program and examine it line by line in a few moments.)

Program Counter Relative Long Addressing

As you saw in chapter 5, one disadvantage of the eight branching instructions that use program counter relative addressing is their very short range: a displacement of −128 bytes to +127 bytes counting from the end of the branching instruction.

But the 65C816 has one branching instruction—`brl`—that can cause a program to branch to any address within the current program bank. So `brl`, instead of accepting a 1-byte operand like all other branching instructions, takes a 2-byte operand. The `brl` instruction's 2-byte operand is interpreted as an offset. This offset is added to the value of the program bank register to calculate the destination address of the branch.

Because `brl` is an unconditional branching instruction, you cannot use it to test the outcome of an arithmetic or comparison operation and then branch if some condition is or is not met. You can use it, however, with conditional branching instructions to extend their range. For example, in this code sequence

```
        lda value
        bne next
        brl longbranch
next    lda something
```

the value of the variable labeled `value` is tested to see if it equals 0. If it equals 0, the `brl` instruction causes a long-range branch to a segment of code labeled `longbranch`. If `value` is not equal to 0, the program continues. Except for a few extra cycles of machine time, the effect is the same as if the segment were coded

```
        lda value
        beq shortbranch
```

but the branch is a long one.

Indexed Addressing

In indexed addressing, the 65C816's X and Y registers provide an index that is used to calculate an effective address. The 65C816 has five kinds of indexed addressing:

- Absolute indexed addressing with X
- Absolute indexed addressing with Y
- Direct indexed addressing with X

- Direct indexed addressing with Y
- Absolute long indexed addressing with Y

Let's examine each of these five types of indexed addressing.

Absolute Indexed Addressing with X

An indexed address, like a relative address, is calculated using an offset. But in an indexed address, the offset is determined by the current contents of the X or Y register.

A statement that uses absolute indexed addressing with X can be written this way:

```
lda $0C00,x
```

The second and third bytes of the statement are added to the X register to form the low-order 16 bits of the operand's effective address. The high-order 8 bits of the effective address are taken from the data bank register. In other words, the value of the X register is used as an offset to calculate the lower 16 bits of the effective address, and the upper 8 bits come from the direct page register.

Listing 6–4, AddrDemo4, is a short program that uses indexed addressing. The routine is designed to move byte-by-byte through a string of ASCII characters, storing the string in a text buffer. When the string is stored in the buffer, the routine ends.

Listing 6–4
AddrDemo4 program

```
*
*  ADDRESSING DEMO #4: Program counter relative addressing
*                      and absolute indexed addressing
*

           KEEP AddrDemo4

demo       START

txtbuf     equ $2000
eol        equ $0d

           phk                              ; make the program bank
           plb                              ; and data bank the same

           ldx #0
loop       lda text,x
           sta txtbuf,x
           cmp #eol
           beq fini
           inx
```

```
            bra loop

fini        brk

text        dc c'This sentence is really moving!',h'0d'

            END
```

The text to be moved is labeled `text`, and the buffer to be filled with text is labeled `txtbuf`. As you can see by looking at the line labeled `text`, the text to be read ends with an end-of-line (EOL) character, the ASCII character `$0d`. The EOL character equates to the Return key on the IIGS keyboard.

As the program proceeds through the string, it tests each character to see if it is a carriage return. If the character is not a carriage return, the program moves to the next character. If the character is a carriage return, there are no more characters in the string, and the routine ends.

In addition to showing how absolute indexed X addressing works, the program also demonstrates the use of program counter relative addressing. In the sequence

```
            ldx #0
loop        lda text,x
            sta txtbuf,x
            cmp #eol
            beq fini
            inx
            bra loop
```

the branching instructions `beq` and `bra` control the loop that prints text on the screen.

Absolute Indexed Addressing with Y

Absolute indexed addressing with Y works like absolute indexed addressing with X except it uses a different index register. A statement that uses absolute indexed addressing with Y can be written as

```
lda $0C00,y
```

The second and third bytes of the statement are added to the Y register to form the low-order 16 bits of the operand's effective address. The high-order 8 bits of the effective address are taken from the data bank register. In other words, the value of the Y register is used as an offset to calculate the lower 16 bits of the effective address, and the upper 8 bits come from the direct page register.

Direct Indexed Addressing with X

A statement that uses direct indexed addressing with X looks like one that uses absolute indexed addressing with X, except it has a 1-byte operand. For example:

```
lda $30,x
```

In this statement, the second byte is added to the sum of the direct page register and the X register to form a 16-bit effective address. In other words, the X register is used as an offset to calculate the lower 16 bits of the effective address, and the upper 8 bits come from the direct page register.

The APW assembler always interprets a 2-byte instruction written in the form `lda $30,x` as a direct indexed address. You must use special prefixes when you want the operand to be interpreted as a data bank offset or as a long address in bank $00, rather than as a direct page offset. These prefixes are the same ones that distinguish between direct addressing and absolute addressing.

In indexed addressing modes, as in unindexed addressing modes, the prefix | (or !) forces the APW assembler to interpret a 1-byte indexed operand as an absolute indexed address. And the prefix > forces the assembler to interpret a 1-byte indexed operand as an absolute long indexed address. Thus, in the statement

```
lda |$40,x
```

the assembler concatenates the address $40 with the contents of the data bank register. Then it adds the value of the X register to calculate the effective address.

In the statement

```
lda >$40,x
```

the value of the X register is added to the address $000040. The result of that calculation is the effective address.

Direct Indexed Addressing with Y

Direct indexed addressing with Y works like direct indexed addressing with X, except it uses a different register. The following statement uses direct indexed addressing with Y:

```
lda $30,y
```

In this statement, the second byte of the instruction is added to the sum of the direct page register and the Y register to form a 16-bit effective address. In other words, the Y register is used as an offset to calculate the lower 16 bits of the effective address, and the upper 8 bits are taken from the direct page register.

It should come as no surprise by now to learn that the APW assembler always interprets a 2-byte instruction written in the form `lda $30,y` as a

direct indexed address. So, in this case also, you must use a special prefix when you want the operand to be interpreted as a data bank offset or as a long address in bank $00. This prefix is the same one you have been using for the same purpose in other addressing modes: the symbol |. Thus, in the statement

```
lda |$40,x
```

the assembler concatenates the address $40 with the contents of the data bank register. It then adds the value of the X register to calculate the effective address.

There is nothing new in any of this, but you may be surprised to know that the syntax

```
lda >$40,y
```

is never invoked to force the assembler to use absolute long indexed addressing with Y. That's because there is no such addressing mode. In 65C816 assembly language, the X register is the only index register that can be used for absolute indexed addressing.

Absolute Long Indexed Addressing with X

In absolute long indexed addressing, the effective address is calculated by adding a long (24-bit) address to the value of the X register. There is no comparable addressing mode that uses the Y register.

A statement that uses absolute long indexed addressing with X can be written this way:

```
lda $E16000,x
```

The value of the X register is added to the long address $E16000 to form the operand's effective address.

Indirect Addressing

In 65C816 assembly language, indirect addressing modes are modes in which data in memory is accessed indirectly, that is, through pointers contained in other memory locations.

The 65C816 has seven indirect addressing modes:

- Direct indirect addressing
- Direct indirect long addressing
- Absolute indirect addressing
- Absolute indexed indirect addressing
- Direct indexed indirect addressing

- Direct indirect indexed addressing
- Direct indirect long indexed addressing

We'll sort this out in the following sections.

Absolute Indirect Addressing

Absolute indirect addressing is really made up of two addressing modes: one is used with the `jmp` (jump) instruction and the other is used with the `jml` (jump—long) instruction.

When absolute indirect addressing is used with `jmp`, the syntax is

```
jmp ($4000)
```

A `jml` instruction that uses absolute indirect addressing looks like this:

```
jml ($E1A000)
```

In both formats, a symbolic label can be substituted for the address inside the parentheses.

When absolute indirect addressing is used with the `jmp` instruction, the address inside the parentheses is a pointer to a memory address. This address and the following memory address contain the lower 16 bits of the effective address of the operand. The program bank register contains the upper 8 bits of the effective address. These two values are concatenated, and the result is the complete effective address of the operand.

When the absolute indirect addressing mode is used with the `jml` instruction, the parentheses that follow the instruction contain a long (24-byte) address. This address and the next two memory addresses contain all 3 bytes of the destination address.

Direct Indirect Addressing

Direct indirect addressing uses the syntax

```
lda ($FB)
```

or

```
lda (<$FB)
```

Notice that in each case, the value inside the parentheses is only 1 byte long.

When you use direct indirect addressing, the operand is an offset that is added to the contents of the direct page register to calculate the lower 16 bits of the operand's effective address. The upper 8 bits of the effective address are taken from the direct page register.

Direct Indirect Long Addressing

Direct indirect long addressing uses the syntax

```
lda [$FB]
```

or

```
lda [<$FB]
```

Notice that in each case, the value inside the parentheses is only 1 byte long.

When you use direct indirect long addressing, the operand is an offset that is added to the contents of the direct page register to calculate the operand's long (24-byte) effective address.

Direct Indexed Indirect Addressing

Two of the 65C816's indirect addressing modes—direct indexed indirect addressing and direct indirect indexed addressing—are so closely related that it makes sense to examine them in combination.

If you think their names are confusing, you're not the first one with that complaint. Here's a memory trick to help eliminate the confusion. Direct indexed indirect addressing—which has an *x* in the second word of its name—is an addressing mode that uses the X register. Direct indirect indexed addressing—which doesn't have an *x* in the second word of its name—uses the Y register. With that introduction, let's examine both of these indirect addressing modes—beginning with direct indexed indirect addressing.

The syntax for a statement that uses direct indexed indirect addressing is

```
lda ($FB,x)
```

or

```
lda (<$FB,x)
```

Notice that the value inside the parentheses is only 1 byte long.

The most common use for direct indexed indirect addressing is to calculate addresses using tables of pointers, or jump tables, located on the direct page. Each address in a direct page jump table is 16 bits long, and must be added to the contents of the current data bank register to yield an effective address. Hence, each item in a direct page jump table is a 2-byte pointer to a 3-byte address situated in the data bank of the program currently being executed.

In a statement that uses direct indexed indirect addressing, both the value of the X register and the value that appears in front of it are offsets used to calculate the operand's final address.

When the 65C816 encounters a statement that uses direct indexed indirect addressing, it first adds the value of the X register to the contents of the direct page register. Then it adds this sum to the value inside the parentheses (that is, the second byte of the instruction). The result is a pointer to the low-order 16 bits of the operand's effective address. The high-order 8 bits of the effective address are taken from the data bank register.

An example might help clarify this process. Suppose memory address $B0 on the direct page holds the number $00, memory address $B1 on the direct page holds the number $80, and the X register holds the number 0, as follows:

```
Direct page + $B0 = #$00
Direct page + $B1 = #$80
X register        = #$00
```

Now suppose you are running a program that contains the direct indexed indirect instruction `lda ($B0,x)`. If all these conditions exist when the IIGS encounters the instruction `lda ($B0,x)`, the 65C816 chip adds the contents of the X register (0) to the hexadecimal number $B0. The sum of $B0 and 0 is $B0.

Next, the 65C816 checks the contents of the direct page memory addresses $B0 and $B1. It finds the number $00 in the direct page memory address $B0 and the number $80 in the direct page address $B1.

Because the 65C816 convention is to store 16-bit numbers in memory with the low byte first, the processor interprets the number in $B0 and $B1 as $8000. So it loads the accumulator with the number $8000, the 16-bit value stored in $B0 and $B1. It then concatenates that value with the contents of the data bank register. The result is the operand's effective address.

Now let's suppose when the IIGS encounters the statement `lda ($B0,x)`, its 65C816's X register holds the number $04, instead of the number $00.

Here is a table illustrating those values, plus a few more equates you'll be using soon:

```
Direct page + $B0 = #$00
Direct page + $B1 = #$80
Direct page + $B2 = #$0D
Direct page + $B3 = #$FF
Direct page + $B4 = #$FC
Direct page + $B5 = #$1C
X register        = #$04
```

If these conditions exist when the IIGS encounters the instruction `lda ($B0,x)`, the 65C816 adds the number $04 (the value in the X register) to the number $B0. It then checks memory addresses $B4 and $B5. In those two addresses, it finds the address $1CFC (low byte first). It then concatenates that value with the contents of the data bank register. The result is the operand's effective address.

Until the advent of the 65C816 and direct page addressing, direct indexed indirect addressing was called simply indexed indirect addressing and required the use of jump tables on page 0. Free space on page 0 was so difficult to find that indexed indirect addressing was not used very often in application programs.

With the 65C816, there is no longer any reason to avoid using direct indexed indirect addressing. In programs written for the IIGS, direct page addresses are so readily available that any application program can use as many as the programmer desires. So, if you ever need to include jump tables in a IIGS program, you might consider using direct indexed indirect addressing.

Direct Indirect Indexed Addressing

Direct indirect indexed addressing uses the syntax

```
lda ($FB),y
```

or

```
lda (<$FB),y
```

Direct indirect indexed addressing uses the Y register (never the X register) as an offset to calculate the base address of the beginning of a table. The starting address of the table has to be stored on the direct page, but the table itself is stored in the bank currently being used as a data bank.

When the APW assembler encounters a direct indirect indexed address in a program, it first adds the number in parentheses—the second byte of the instruction—to the contents of the data bank register. The sum of that operation is combined with the contents of the data bank register to form a 24-bit base address. Finally, that address is added to the value of the Y register to form the effective address of the operand.

Here's an example of how direct indirect indexed addressing is used. Suppose the 65C816 chip is running a program and comes to the instruction `lda ($B0),y`. First it looks into direct page memory addresses $B0 and $B1. Suppose it finds the number $B0 in direct page address $00 and the number $50 in direct page address $B1. And suppose the Y register contains a 0. The following illustrates these conditions:

Direct page + $B0 = #$00
Direct page + $B1 = #$50
Y register = #$04

If these conditions exist when the 65C816 encounters the instruction `adc ($B0),y`, the processor concatenates the numbers $00 and $50, and it comes up with the address $5000 (in the 65C816 chip's peculiar low-byte-first fashion). It then adds the contents of the Y register ($04) to the number $5000—for a total of $5004.

The processor then combines the 16-bit number $5004 with the 8-bit value of the data bank register. The result is the 24-bit effective address of the operand.

Direct indirect indexed addressing is a valuable tool in assembly language programming. Only one address—the starting address of a table—has to be stored on the direct page. Yet that address, added to the contents of the Y register, can be used as a pointer to locate any other address in memory.

Direct Indirect Long Indexed Addressing

Direct indirect long indexed addressing uses the syntax

```
lda [$FB],y
```

or

```
lda [<$FB],y
```

In direct indirect long indexed addressing, the Y register is used as an offset to calculate the base address of the beginning of a table. The starting address of the table has to be stored on the direct page, but the table itself can be stored anywhere in memory.

In direct indirect long indexed addressing, the value in parentheses (the second instruction of the address) is added to the contents of the direct register. The sum of these two numbers is an address on the direct page. In this address and the two addresses that follow, a 24-bit base address is stored. This base address is added to the value of the Y register to form the 24-bit effective address of the operand.

Absolute Indexed Indirect Addressing

Absolute indexed indirect addressing is used with only two instructions: `jmp` (jump) and `jsr` (jump to subroutine). It provides a means for jumping to any address in memory with a jump table placed in the current program bank. The syntax is

```
jmp ($0300,x)
```

Or, when a 1-byte operand is used and the assembler must be forced to generate a 2-byte instruction, the syntax is:

```
jmp (|$30,x)
```

A symbolic label can be substituted for the literal address in each of these examples.

In a statement that uses absolute indexed indirect addressing, the value inside the parentheses is added to the value of the X register to form a 16-bit address. This address is combined with the contents of the program bank register to form a 24-bit base address. Finally, this base address is added to the value of the X register, forming the operand's 24-bit effective address.

Stack Addressing

The 65C816 has three stack addressing modes:

- Stack relative addressing
- Stack relative indirect indexed addressing
- Simple stack addressing

To understand how stack addressing works, it is essential to have an understanding of what a stack is, and what it does.

The Stack

A stack, as pointed out in the beginning of this chapter, is an area of memory often used for the temporary storage of data that is waiting to be processed. In pre-GS Apple IIs, the stack is exactly 256 bytes long and occupies page 1—memory addresses $100 through $1FF—in either main or auxiliary memory. In the IIGS, the stack can be placed anywhere in bank $00. The only restriction on its length is the availability of free RAM in bank $00.

In both the IIGS and earlier Apples, the stack is called a LIFO (last-in first-out) block of memory. It is often compared to a spring-loaded stack of plates in a diner. When you put a number in the memory location on top of the stack, it covers up the number that was previously on top. So the number on top of the stack must be removed before the number under it—which was previously on top—can be accessed.

Although the stacked plate analogy is a useful technique for describing how the stack works, it is not completely accurate. Actually, the stack is nothing but a block of RAM—and blocks of RAM don't really move up and down like a stack of plates inside the IIGS. When you place a number on the 65C816 stack, here's what really happens.

Suppose, for simplicity, that you have placed the stack on page 1 in memory bank $00. (The stack was in this location in earlier Apple IIs, so we'll keep it there for this description.)

Now the block of memory in which the stack is situated—in this case, page 1 in bank $00—is used in stack operations from high memory downward. The first number stored in a page 1 stack is in register $01FF, the next number is placed in register $01FE, and so on. A program can keep placing values on the stack, in this from-the-top-down fashion, until it runs out of free RAM. When the stack is on page 1, it will run out of free memory when it reaches memory address $100 because all RAM below that address is claimed by page 0. By starting the stack higher in memory, you can make the stack bigger. But because we're using page 1 for the stack in this example, the last stack address that we can currently use is memory address $100.

As you saw in chapter 5, the 65C816 chip keeps track of stack manipulations with the help of a special register called the stack pointer. In the 65C816, the stack pointer is a 16-bit address, and the upper 8 bits always hold the number of the page where the stack starts. When the stack starts on page 1, for instance, the high byte of the stack pointer holds a 1.

When there is nothing stored on the stack, the value of the stack pointer's low byte is $FF. If there are 256 bytes on the stack, the value of the stack pointer's low byte is $00.

As soon as a value is stored on the stack, the 65C816 chip automatically decrements the stack pointer by 1. And each time another value is stored on the stack, the stack pointer is decremented again. Therefore, the stack pointer always points to the address of the next value that will be stored on the stack.

Stack Operations

Suppose several numbers are stored on the stack. And let's also suppose you want to retrieve one of those values from the stack. What will happen?

When a number stored on the stack is retrieved, the value of the stack pointer is incremented by 1. That effectively removes one value from the stack, because the next value stored on the stack has the same position on

the stack as the one that was removed. That's a little tricky to comprehend, given the upside-down nature of the stack. Figure 6–1 will help you understand how this works. This figure shows an empty stack, with the stack pointer pointing to the first available address on the stack: $01FF.

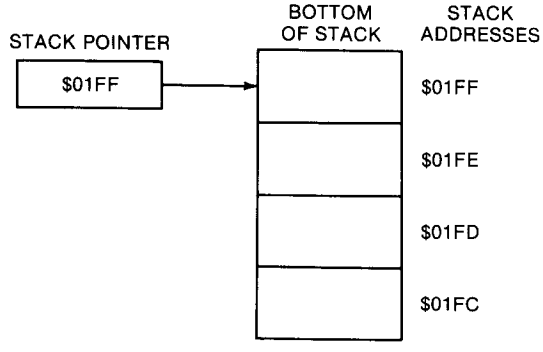


Figure 6–1
How the stack pointer works

Now let's place a number (whose value is arbitrary) on the stack. This kind of operation is illustrated in figure 6–2. In this figure, the value of the stack pointer is decremented, and the number placed on the stack is now stored at the highest address in the stack, memory register $01FF.

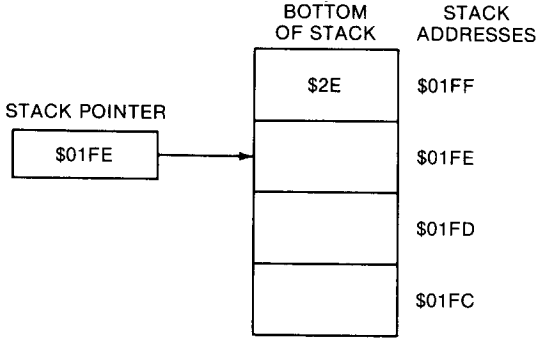


Figure 6–2
Placing a number on the stack

Figure 6–3 shows what happens if you place another number (also with an arbitrary value) on the stack. The stack pointer is decremented again, and a second number is now on the stack.

Figure 6–4 shows what happens if you "remove" one number from the stack. Stack address $01FE still holds the value $B0, but the value of the stack pointer is decremented and now points to memory address $01FE. So the next number placed on the stack will be stored at memory address $01FE. When that happens, the number previously stored in that stack position—$B0—will be erased.

To see how that works, we'll store one more number on the stack. This time, for no special reason, the value of the number placed on the stack is $17. This process is illustrated in figure 6–5. Register $01FE now holds the

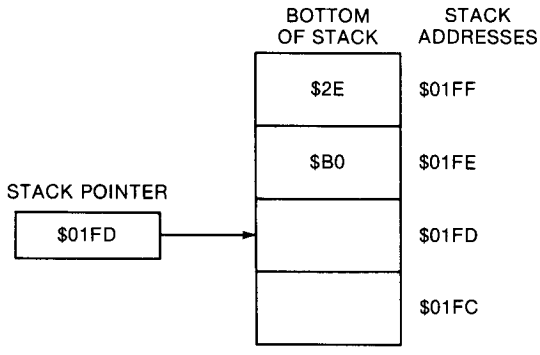


Figure 6–3
Placing another number on the stack

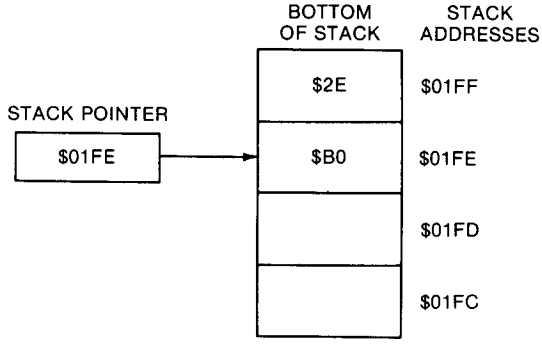


Figure 6–4
Pulling a number off the stack

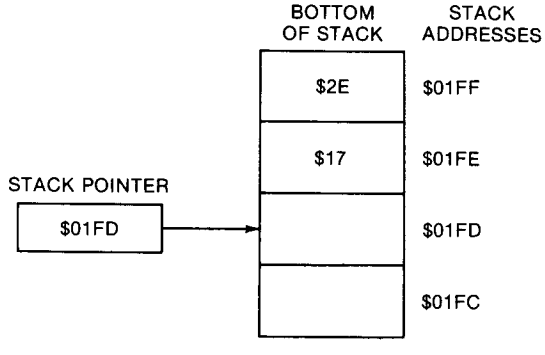


Figure 6–5
One last stack manipulation

value $17. The value of the stack pointer is decremented, the value $B0 is erased by the value $17, and the next number placed on the stack will be stored in memory register $01FD.

How the IIGS Uses the Stack

As mentioned, the 65C816 often uses the stack for temporary data storage during the operation of a program. When a program jumps to a subroutine, for example, the processor pushes onto the top of the stack the memory address

that the program will have to return to. Then, when the subroutine ends with an `rts` instruction, the return address is pulled from the top of the stack and loaded into the 65C816's program counter. Then the program can return to the proper address, and normal processing can resume.

Instructions that Use Stack Addressing

As you saw at the beginning of this chapter, `phk` and `plb` are two instructions that use stack addressing. Other mnemonics that use this addressing mode include

- `pha`: Push the contents of the accumulator onto the stack.
- `phx`: Push the contents of the X register onto the stack.
- `phy`: Push the contents of the Y register onto the stack.
- `php`: Push the contents of the P register onto the stack.
- `pla`: Pull the top value off the stack and deposit it in the accumulator.
- `plp`: Pull the top value off the stack and deposit it into the P register.

The `php` and `plp` operations are often included in assembly language subroutines so that the contents of the P register won't be erased during subroutines. When you jump to a subroutine that may change the status of the P register, it's a good idea to begin the subroutine by pushing the contents of the P register onto the stack. Then, just before the subroutine ends, you can restore the P register's previous state with a `php` instruction. This ensures that the P register's contents aren't destroyed during the subroutine.

Programs written for the IIGS often use stack addressing because of the way the IIGS Toolbox is designed. As you shall see in part 2, most routines in the Toolbox are called by placing values on the stack, accessing a macro, and then pulling values off the stack when the macro returns. We go into more detail about how to do that in chapter 7.

The 65C816, as pointed out at the beginning of this section, has three addressing modes that use the stack. One of these modes, simple stack addressing, was covered at the start of this chapter. The other two, stack relative addressing and stack relative indirect indexed addressing, are examined next.

Stack Relative Addressing

Stack relative addressing is the first addressing mode in the 6502 chip family that has made it possible to access a byte in the stack without removing the last byte pushed onto the stack. A statement that uses stack relative addressing is written in this format

```
lda 3,s
```

The value that follows the mnemonic is an offset that is added to the contents of the stack pointer to form the effective address. When the statement is assembled into machine code, the operand is assembled as a single byte. Because the stack is always in bank $00, the effective address calculated by adding the operand to the stack pointer is always 16 bytes long.

In determining what offset to use to access a value on the stack, it is important to remember that offsets used in stack relative addressing start at 1, not at 0. The stack pointer always points to the next available stack location, which is 1 byte below the last byte pulled off the stack. So, to load the accumulator with the last value pushed onto the stack using stack relative addressing, you would use this statement:

```
lda 1,s
```

Stack Relative Indirect Indexed Addressing

You can use stack relative indirect indexed addressing to access a value indirectly, with a pointer pushed onto the stack. The format of a statement that uses stack relative indirect indexed addressing is

```
lda ($30,s),y
```

Stack relative indirect indexed addressing works much like direct indirect indexed addressing. The value between the parentheses is a 1-byte offset. The 65C816 adds this offset to the contents of the stack pointer to form the lower 16 bits of a base address in bank $00. The upper 8 bits of the base address are taken from the data bank register. Finally, the value of the Y register is added to this base address, and the result is the effective 24-byte address of the operand.

A Warning

Now that you know how stack addressing works, it's time to add a note of warning: The 65C816 stack can be a dangerous section of memory for novice programmers to play with. When you use the stack in an assembly language routine, it's extremely important when the routine ends to leave the stack exactly as you found it. If you've placed a value on the stack during a routine, it must be removed from the stack before the routine ends and normal processing resumes. Otherwise, there might be "garbage" on the stack when the next routine is called, and that can result in program crashes, memory wipeouts, and various other programming disasters. Remember: Mismanagement of the hardware stack is extremely hazardous to the health of assembly language programs.

Block Move Addressing

The 65C816 has one addressing mode for block moves. It is called block source bank, destination bank addressing. This addressing mode is used by two instructions: `mvn` (block move next, or block move negative) and `mvp` (block move previous, or block move positive). The syntax is

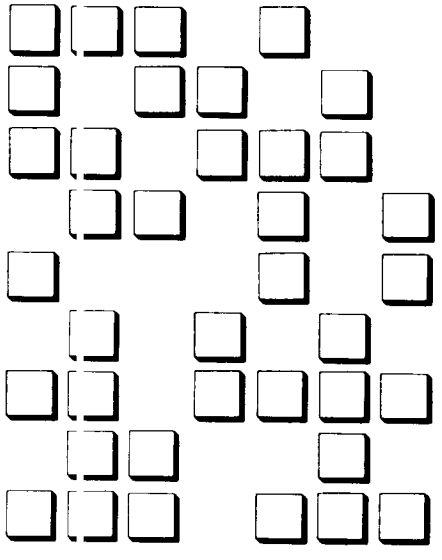
```
mvn 0,0
```

A statement that uses block move addressing takes a 2-byte operand. In source code written using the APW assembler, these 2 bytes are separated by a comma. The first byte of the operand specifies the 64K bank of memory that

a block of data is being moved to, and the second byte specifies the bank in which the source data lies. The Y register contains the lower 16 bits of the destination address. The X register contains the lower 16 bits of the source address. The number of bytes to be moved, minus 1, is contained in the C register, the 65C816's 16-bit accumulator. More details about how block move addressing mode works can be found in chapter 5 and appendix A, which deals with the 65C816 instruction set.

PART 2

The Apple IIGS Toolbox



CHAPTER 7

Introducing the IIGS Toolbox

Using the Event Manager and the Memory Manager

The biggest difference between the Apple IIGS and earlier members of the Apple II family is the IIGS has a gigantic Toolbox: a collection of more than 800 prewritten routines that greatly simplify the writing of sophisticated programs.

We have encountered a number of the tools in the IIGS Toolbox in previous chapters, but we haven't gone into detail about how they work. In this chapter, you are formally introduced to the various tool kits in the Toolbox, and you take a close look at what they are and what they do.

Tool Sets

The 800-plus routines in the IIGS Toolbox are divided into *tool sets,* or collections of related routines. Each routine in a tool set performs one function, or fundamental operation. For example, the QuickDraw II tool set contains one routine that draws a rectangle, another that draws an oval, and so on.

Some tool sets in the Toolbox manage important features of the Apple IIGS and are therefore called *managers.* For example, the IIGS Memory Manager allocates, deallocates, and keeps track of all memory used by the computer. The Event Manager keeps track of mouse and keyboard operations and

calls other manager tools, such as the Menu Manager and the Window Manager. The Menu Manager and the Window Manager, as their names imply, manage IIGS operations that involve menus and windows.

What the Toolbox Can Do

The most important reason for learning how to use the Toolbox is that it can take care of much of the drudgery that otherwise is the responsibility of the programmer. It can free your application so it can concentrate on its most important tasks rather than deal with routine background work and trivial details.

Another reason for using the Toolbox is that its routines are always available to help you perform many important tasks. Most of the remarkable capabilities of the IIGS are accessed easily through the IIGS Toolbox, the various tool sets in the Toolbox, and each set's individual tools.

What the Toolbox Contains

The tools in the IIGS Toolbox are listed in chapter 1. We'll list them again, in more detail.

The Big Five

Five vital IIGS tool sets are dubbed the "Big Five." All these tools must be used in every event-driven IIGS application because they are the basic framework upon which other tools build. The "Big Five" tools are

- The Tool Locator, which provides the mechanism for dispatching tool calls. You don't need to know a tool's memory location; the Tool Locator knows, and it retrieves the tool when you make a tool call.
- The Memory Manager, which allocates, deallocates, and keeps track of all memory used in every program. When your application needs memory, it must request it from the Memory Manager. When a well-written application ends, it calls the Memory Manager again to deallocate the memory it no longer needs.
- The Miscellaneous Tool Set, which includes mostly system-level routines that must be available for other tool sets. The Miscellaneous Tool Set is vital to the operation of the IIGS. It keeps track of mouse movements, takes care of battery-powered memory functions, and handles interrupts. All tools except the Tool Locator and the Memory Manager depend on the tools in the Miscellaneous Tool Set in some way.
- QuickDraw II, which controls the graphics environment of the IIGS and draws objects and text when the computer is in super high-resolution graphics mode. QuickDraw II draws the menus, windows, controls, and other object used by many of the tools in the Toolbox.

- The Event Manager, which manages all the IIGS's event-driven programming. The Event Manager keeps track of keyboard and mouse events, maintains a queue of the events that take place, and passes information about events to the application.

Desktop Interface Tool Sets

Another important group of tools control the IIGS desktop interface. The desktop interface tool group is the interface between the IIGS user and the computer's programs. Most of the IIGS programs you write will use desktop interface utilities such as the Window Manager, Menu Manager, Dialog Manager, and LineEdit Tool Set.

The tool sets in the desktop interface group are

- The Window Manager, which draws, updates, maintains, and deallocates windows. Because the IIGS is designed to be used in a window environment, the Window Manager is one of the most important tools in the Toolbox.
- The Control Manager, which draws pushbuttons, scroll bars, and other objects on the super high-resolution screen. When the Control Manager draws controls, you can activate them by clicking the mouse. In this way, you can scroll windows, turn functions off and on, and cause various other things to happen. The Control Manager is primarily a low-level tool set whose functions are used by other tools such as the Window Manager. But the Control Manager can also create and manipulate user-designed controls.
- The Menu Manager, which controls and maintains pull-down menus and items in menus. Because the IIGS is designed to be used in a menu environment, the Menu Manager is one of the most important tool sets in the Toolbox.
- The LineEdit Tool Set, which performs much the same function in the super high-resolution environment that the Text Tool Set performs when the computer is in text mode. The LineEdit Tool Set places text on the screen and allows the user to edit it. In addition, LineEdit offers "cut-and-paste" operations that provide convenient methods for editing, deleting, and moving text.
- The Dialog Manager, which offers a convenient and easy-to-use interface between the IIGS and the user. The Dialog Manager creates windows to display messages and can accept user input. Windows created by the Dialog Manager can warn the user of dangers or special situations and provide the user with an easy method for making decisions and passing them to a IIGS program.
- The Scrap Manager, which offers the user a method of storing information temporarily so it can be moved to another location, document, or application. When information is no longer needed, the Scrap Manager can delete it.
- The Desk Manager, which manages desk accessories, mini-applications executed while other applications are running. The Desk

Manager controls clocks, calculators, note pads, and other useful desktop utilities.

- The Standard File Operations Tool Set, which provides an easy-to-use interface with ProDOS 16 in a super high-resolution environment. When the Standard File Operations Tool Set is activated, it presents a special dialog window that can load, save, open, and close disk files without requiring the user to master the technical details of ProDOS 16.
- The List Manager, a low-level tool set used primarily by other tool sets, such as the Standard File Operations Tool Set and the Font Manager. The List Manager creates lists of items, such as files and fonts, and is also available for use in application programs.
- The Font Manager, which keeps track of the character fonts available to the IIGS and provides applications with information about them. The Font Manager can tell you how many fonts are available and the characteristics of those fonts. It can also underline text, print in boldface or italics, and print text of various sizes on a printer or the screen.
- QuickDraw II Auxiliary, which adds special capabilities to QuickDraw II. The tools in the QuickDraw II Auxiliary tool set can combine various drawing calls into a single picture, shrink and reduce drawn objects and the bit maps used to create screen windows, and draw icons and other objects on the super high-resolution screen.

Math Tool Sets

The Apple IIGS has two tool sets that perform arithmetic and mathematic operations:

- The Integer Math Tool Set, which includes routines that perform operations using integers. The Integer Math Tool Set handles integers, long integers, and signed fractional numbers. It can also convert integers, hexadecimal numbers, and decimal numbers from one form to another and from one base to another.
- The SANE Tool Set, which supports the Standard Apple Numerics (SANE) mathematics package. With the SANE Tool Set, the IIGS can carry out extended-precision calculations in accordance with the widely accepted IEEE standard.

The Print Manager

The Print Manager allows applications to use standard QuickDraw II routines to print text or graphics on a printer. It can interface an application with a standard dot-matrix printer such as an Apple ImageWriter, or a laser printer such as the Apple LaserWriter, or a network of laser printers.

Sound-Related Tool Sets

The IIGS has three sound-related tool sets:

- The Sound Tool Set, which provides a method for using the IIGS's sound hardware without using specific hardware input-output addresses.
- The Note Synthesizer, which creates notes, sound patterns, and waveforms with sound-synthesizing techniques similar to those used by synthesizers in professional sound studios.
- The Note Sequencer, which provides a convenient method for incorporating sequences of musical notes into a program.

Specialized Tools

The Apple IIGS has one group of tools that are categorized as specialized tools. They include

- The Apple Desktop Bus (ADB) Tool Set, which interfaces the IIGS with its keyboard, mouse, and other I/O devices such as graphics tablets and game controllers.
- The Scheduler, which prevents a tool call from crashing the system by asking for a temporarily unavailable system resource.
- The Text Tool Set, which provides an interface between Apple II character device drivers and applications running in native mode.

How To Use the Toolbox

In early models of the IIGS, many of the tools in the Toolbox were provided on the system disk and had to be loaded into RAM. In newer models, increasing numbers of tools have been taken off the system disk and included in ROM. More tools are instantly available to application programs without using disk space, loading time, or what would otherwise be free memory.

You seldom need to keep track of a tool's location or even whether the toolkit that contains the tool is in ROM or RAM. A tool set called the Tool Locator keeps track of all tools for you and takes care of most of the "housekeeping" involved in loading and unloading tools.

The Tool Locator is automatically initialized when ProDOS 16 is booted, and from then on you can use it any time you like. In an assembly language program written using APW, the easiest way to use the Tool Locator is to decide what tools you will use in a program and then make the APW macro call

```
_LoadTools
```

All the tools you'll need in your program are then loaded into memory.

Making the LoadTools Call

Before you can make a `LoadTools` call, you have to design a tool table containing the identification number and lowest suitable version number of each tool set you plan to use in your program. The available tools are listed in table 7–1.

Table 7–1
Tools in the IIGS Toolbox

Tool Number	Tool	Version on System Disk 3.0
1	Tool Locator	0201
2	Memory Manager	0200
3	Miscellaneous Tool Set	0200
4	QuickDraw II	0202
5	Desk Manager	0202
6	Event Manager	0201
7	Scheduler	0200
8	Sound Manager	0200
9	Apple Desktop Bus	0201
10	SANE	0202
11	Integer Math Tool Set	0200
12	Text Tool Set	0200
13	Not used	
14	Window Manager	0201
15	Menu Manager	0200
16	Control Manager	0201
17	System Loader	0103
18	QuickDraw Auxiliary	0202
19	Print Manager	0102
20	LineEdit Tool Set	0200
21	Dialog Manager	0200
22	Scrap Manager	0102
23	Standard File Operations Tool Set	0200
24	Disk Utilities	0100
25	Note Synthesizer	0100
26	Note Sequencer	0100
27	Font Manager	0201
28	List Manager	0201

The `LoadTools` call must be used with a carefully designed tool table to work properly. The first word in the tool table must contain the number of tool sets that will be loaded. Next, you must list the ID number of each tool set, along with the minimum acceptable version number of each tool set to be loaded. Listing 7–1 shows how the `LoadTools` call is included in a IIGS assembly language program.

Listing 7–1
Tool loading routine

```
*
*ROUTINE FOR LOADING TOOLS
*

LoadEmUp        START

                PushLong #ToolTable
                _LoadTools

                rts

ToolTable       dc i'13'                ; no. of tools to load
                dc i'$04,$0100'         ; quickdraw
                dc i'$05,$0100'         ; desk manager
                dc i'$06,$0100'         ; event manager
                dc i'$0E,$0000'         ; window manager
                dc i'$0F,$0100'         ; menu manager
                dc i'$10,$0100'         ; control manager
                dc i'$12,$0000'         ; qd auxiliary
                dc i'$13,$0000'         ; print manager
                dc i'$14,$0000'         ; line edit
                dc i'$15,$0000'         ; dialog manager
                dc i'$17,$0100'         ; std file manager
                dc i'$1B,$0100'         ; font manager
                dc i'$1C,$0000'         ; list manager

                END
```

Initializing Tools

Some tool sets—such as the Tool Locator, the Text Tool Set, and the Integer Math Tool Set—are present in ROM at all times and thus do not have to be loaded before they are used. But most tool sets do have to be loaded and then have to be started up, or initialized. When you're finished using a tool set, you should shut it down.

It is very easy to initialize a tool set, and it is just as easy to shut one down. To initialize or shut down a tool set, you make a specific call. The following call, for example, initializes the LineEdit Tool Set:

```
_LEStartup
```

and this call shuts it down:

```
_LEShutdown
```

The sample programs in the rest of this book give you plenty of practice in starting up and shutting down tool sets.

There are two important points to think about when you plan to call a IIGS tool from your application. One is *tool dependency*, making sure certain tools are loaded and initialized before other tools that rely on them are called. The second point is making sure the IIGS is in 16-bit native mode before any tools are loaded, initialized, and called.

It is very important to start up tools in the correct order. A tool set dependency chart, which shows what tools have to be started before other tools can be used, appears in table 7–2. You can practice starting up tool sets in the proper order by typing, assembling (or compiling), and running the sample programs in chapters 8 through 13.

Table 7–2
Tool Set Dependency

			Dependencies (with minimum version number needed)											
Hex	Dec	Tool Set	Tool Locator	Memory Manager	Misc. Tool Set	Quick-Draw II	Event Manager	Window Manager	Control Manager	Menu Manager	LineEdit Tool Set	Dialog Manager	Scrap Manager	List Manager
$01	1	Tool Locator												
$02	2	Memory Manager	$0102											
$03	3	Misc. Tool Set	$0102	$0102										
$04	4	Quick-Draw II	$0102	$0102	$0102									
$12	18	Quick-Draw II Auxiliary	$0102	$0102	$0102	$0102								
$06	6	Event Manager	$0102	$0102	$0102	$0102								
$0E	14	Window Manager	$0102	$0102	$0102	$0102	$0100							
$10	16	Control Manager	$0102	$0102	$0102	$0102	$0100	$0103						
$0F	15	Menu Manager	$0102	$0102	$0102	$0102	$0100	$0103	$0103					
$14	20	LineEdit Tool Set	$0102	$0102	$0102	$0102	$0100							
$15	21	Dialog Manager	$0102	$0102	$0102	$0102	$0100	$0103	$0103	$0103	$0100			
$16	22	Scrap Manager	$0102	$0102										
$05	5	Desk Manager	$0102	$0102	$0102	$0102	$0100	$0103	$0103	$0103	$0100	$0101	$0101	
$17	23	Standard File Tool Set	$0102	$0102	$0102	$0102	$0100	$0103	$0103	$0103	$0100	$0101		
$1C	28	List Manager	$0102	$0102	$0102	$0102	$0100	$0103	$0103					
$13	19	Print Manager	$0102	$0102	$0102	$0102	$0100	$0103	$0103	$0103	$0100	$0101		$0100
$1B	27	Font Manager	$0102	$0102	$0102	$0102	$0100	$0103	$0103	$0103	$0100	$0101		$0100

It is also important to make sure the IIGS is in native mode, rather than emulation mode, when you use the Toolbox in a program. When the 65C816 is in native mode, its e, m, and x flags are all set to 0, providing it with a 16-bit accumulator and 16-bit index registers. Almost all the tools in the Toolbox require the 65C816 to be in native mode and simply will not work if the processor is in its 8-bit state. Exceptions to this rule are rare and are documented in the *Apple IIGS Toolbox Reference*.

The Memory Manager

The Memory Manager, like the Tool Locator, resides in ROM and thus does not have to be loaded or initialized. It goes into action as soon as you turn on the IIGS. From then on, it controls the allocation, deallocation, and positioning of every byte in the computer's memory. The Memory Manager constantly keeps track of how much memory is free and which blocks of memory are allocated to which programs.

The Memory Manager works closely with ProDOS 16 and the System Loader to provide needed memory spaces for loading programs and data and to provide buffers for input and output. All Apple IIGS software, including the System Loader and ProDOS 16, must obtain memory space by making requests (calls) to the Memory Manager.

When a block of memory is allocated by the Memory Manager, it is assigned a number of important attributes that the Memory Manager stores in a special location. These attributes determine how the Memory Manager may modify each block (such as moving it or deleting it), if each block can be purged from memory, if it must be placed in memory in a special way (for example, starting on a page boundary), and what program owns it.

When a program asks for a block of memory, it must pass to the Memory Manager a list of attributes that it wants to assign to the block. These attributes are passed in the form of a word. This is shown in figure 7–1 and is examined more closely later in this chapter. When a group of attributes is assigned to a block of memory, the Memory Manager takes them into account whenever it has to purge, allocate, deallocate, or compact memory.

How an Application Obtains Memory

When an application makes a ProDOS 16 call that requires allocation of memory (such as opening a file or writing from a file to a memory location), ProDOS 16 first obtains any needed memory blocks from the Memory Manager and then performs its tasks. Likewise, the System Loader requests any needed memory either directly or indirectly (through ProDOS 16 calls) from the Memory Manager. Conversely, when an application informs the operating system that it no longer needs memory, that information is passed to the Memory Manager, which in turn frees the application's allocated memory. In all these cases, the memory allocation and deallocation is automatic as far as the application is concerned.

When an application needs memory for its own use, it must request the memory directly from the Memory Manager. In a few moments, you'll see how a program can request memory from the Memory Manager.

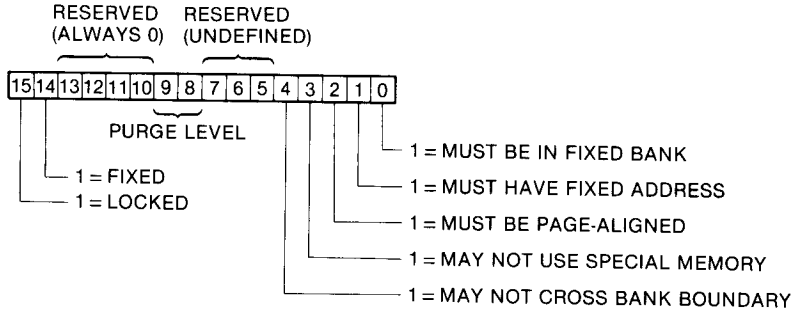


Figure 7–1
Attributes word used by the Memory Manager

How the Memory Manager Uses Memory

From the Memory Manager's point of view, the memory in the IIGS is divided into three categories:

- Allocatable memory (managed by the Memory Manager). There are no special restrictions on the use of this memory. It includes banks $02 through $DF and parts of banks $E0 and $E1.
- Special memory (managed by the Memory Manager and allocatable except under special conditions). There are certain restrictions on the use of this memory because it is used like Apple IIe and IIc memory when the IIGS is in emulation mode. Special memory may not be used by desk accessories, tool sets, and other routines that might be called while IIc/IIe-style applications are running. Banks $00 and $01 and parts of banks $E0 and $E1 are special memory.
- Unmanaged memory (reserved and not managed by the Memory Manager). This category of memory includes the language card area from $D000 to $DFFF in banks $00, $01, $E0, and $E1, addresses $0000–$0800 in banks 0 and 1, and addresses $0000–$2000 in banks $E0 and $E1. The Memory Manager marks this memory as "busy" at startup time and does not interfere with it thereafter.

Figure 7–2 shows how the Memory Manager handles allocatable, special, and unmanaged memory.

Pointers and Handles

Because the Memory Manager can move a memory block and thus change its starting address, IIGS applications cannot use simple pointers to access entry points in memory. Instead, each time the Memory Manager allocates a memory block, it returns to the requesting application a special kind of pointer called a *handle*. Then the application that owns the memory block can always access it safely by using its handle, rather than an ordinary pointer.

A handle is sometimes described as a "pointer to a pointer." It is a fixed, or unmovable, memory location that contains the address of a simple pointer. The handles used by the IIGS are kept in an unmovable, unpurgeable

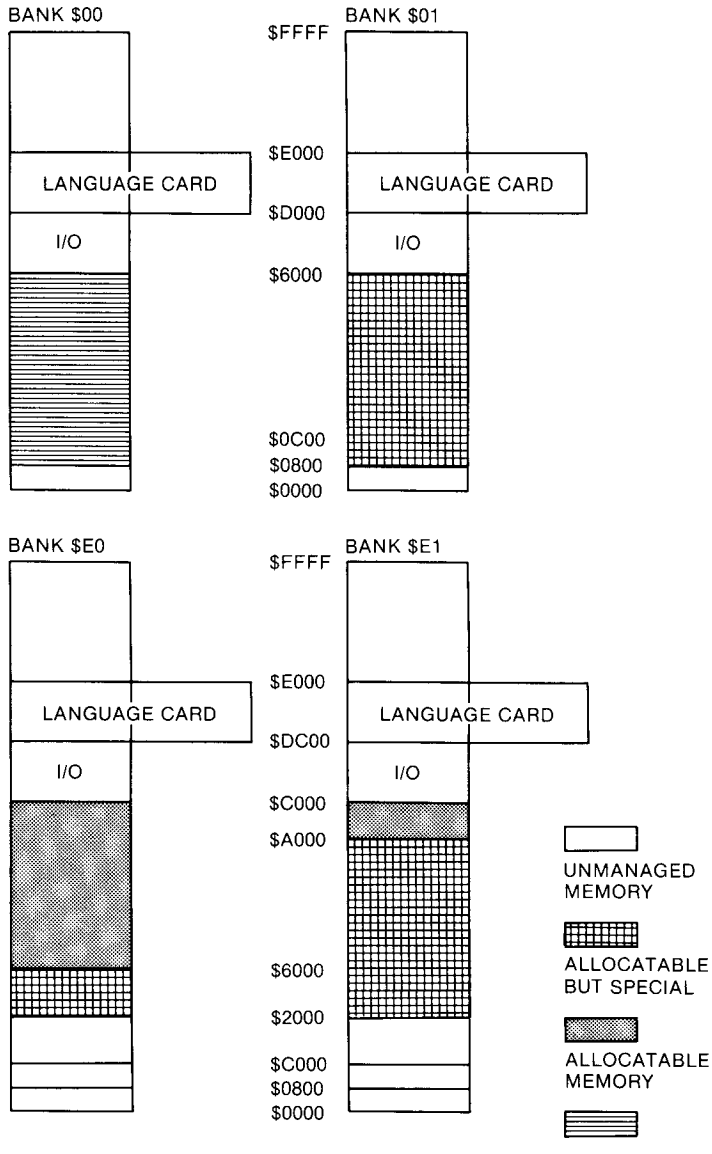


Figure 7–2
Allocatable, special, and unmanaged memory

block of memory that starts at memory address $E11700. Each time a block of memory is assigned, the Memory Manager stores its starting address, along with its attributes, into one of the handles that start at $E11700.

How To Assign a Handle

Before a program can request a block of memory (and a handle) from the Memory Manager, it must obtain a user identification code, or user ID, from the Memory Manager. To get a user ID, a program can make the `MMStartup` call, in this fashion:

```
PushWord #0                    ;space for return
_MMStartup
PullWord MyID
```

In this example, a word is pushed onto the stack so that **MMStartup** has a place on the stack to push its data. Then the APW macro **_MMStartup** makes the **MMStartup** call. When you make the call, it assigns a user ID number and places it on the stack. When the call returns, the user ID assigned by the Memory Manager is pulled off the stack and placed in a program variable called **MyID**.

If you're wondering why the **MMStartup** call has to be made to get a user ID, the answer is simply that that's the way it's done. Because the Memory Manager is a ROM-based tool and is always active, it doesn't have to be started with a startup call. But the conventional way to get a user ID is to request it with an **MMStartup** call. And more than one **MMStartup** call can be made in a program. This would all be less confusing if the **MMStartup** call had a different name. You just have to remember that the **MMStartup** call does not really start up the Memory Manager. It is used primarily for obtaining user IDs.

After you have a user ID from the Memory Manager, you can request as many memory blocks as you like. As long as the Memory Manager has enough free RAM available, it will assign memory blocks and return handles. You have to keep track of the handles the Memory Manager assigns and what you're using them for. One good reason to keep track of handles is that you must dispose of any handles before you end the application. Otherwise, they remain in memory after the application ends, wasting memory space and possibly interfering with other programs.

Before you can dispose of a handle, though, you have to get one. Listing 7–2 is a fragment of assembly language code that shows how to get a handle from the Memory Manager.

Listing 7–2
Getting a handle from the Memory Manager

```
PushLong #0                    ; space for result
PushLong #$8000                ; size of block
PushWord MyID                  ; user ID
PushWord #0                    ; attributes
PushLong #0                    ; Location (0=don't care)
_NewHandle
PullLong MyHandle
```

The call to get a block of memory (along with a handle) is **NewHandle**. But before you make a **NewHandle** call, you must push these parameters on the stack:

- A space for a result (1 word).
- The size of the block of memory being requested (2 words). In

listing 7–2, the length of the block being requested is $8000, or 32K.

- The user ID of the application requesting the block (1 word).
- The block's attributes (1 word). A diagram of this word appears in figure 7–1. (An explanation of each bit is provided later in this chapter.)
- The starting address of the block (2 words). Unless there is an overwhelming need to store a block in a specific location, this parameter should be #0 so that the Memory Manager determines where to store the block being requested.

How the Memory Manager Uses Handles

After a handle is assigned to a block of memory and the program that owns the handle is told what the handle is, the Memory Manager can move the block as often as needed, and the block's handle will not change. If the Memory Manager changes the location of the block, it updates the address stored in the handle, but does not change the address of the handle. Thus, the application that owns the memory block can always use the block's handle to access it, no matter how often the block itself is moved in memory.

Dereferencing a Handle

If an application is sure that a block of memory will always remain in the same spot—that is, if it has requested a locked and unpurgeable handle—the application can access the block by its pointer as well as by its handle. To obtain a pointer to a particular block or location, a special kind of operation called *dereferencing* is used. Listing 7–3 is a routine that dereferences a handle—that is, it tells you what its handle is. The segment of code that appears in listing 7–3 is used in several programs in part 2.

Listing 7–3
Dereferencing a handle

```
Deref           START
                sta DPTemp
                stx DPTemp+2
                ldy #4
                lda [DPTemp],y
                ora #$8000
                sta [DPTemp],y
                dey
                dey
                lda [DPTemp],y
                tax
                lda [DPTemp]
                rts

                END
```

In a dereferencing operation, the application reads the address stored

in the location pointed to by the handle. That address is the pointer to the block. If the Memory Manager moves the block, the pointer is no longer valid.

Memory Fragmentation and Compaction

Because the Memory Manager does not allocate and deallocate memory in any order, memory can become fragmented into a jumble of free and allocated memory blocks. When this happens, the Memory Manager may not be able to allocate a requested block, even if enough free memory is available. So the Memory Manager has the capability of compacting memory, or moving all relocatable blocks so that bigger blocks of memory become available. Figure 7–3 shows how the Memory Manager compacts memory.

As you can guess by looking at figure 7–3, when fixed and locked blocks are present in memory, the Memory Manager can't do a very good job of compacting memory. For this reason, applications should avoid requesting fixed and locked blocks, and settle for movable blocks when possible.

Purging Memory

If the Memory Manager compacts as much memory as possible and still can't allocate a block, it tries to purge any blocks marked unlocked and purgeable. When a block is purged, its contents are discarded, and the memory it occupied is free for other uses.

When a block is purged, its handle remains allocated, but the value of the master pointer that its handle points to is set to 0, or nil. A handle that points to a nil master pointer is called an *empty handle*. When the block of memory assigned to a handle is purged, an application asks the Memory

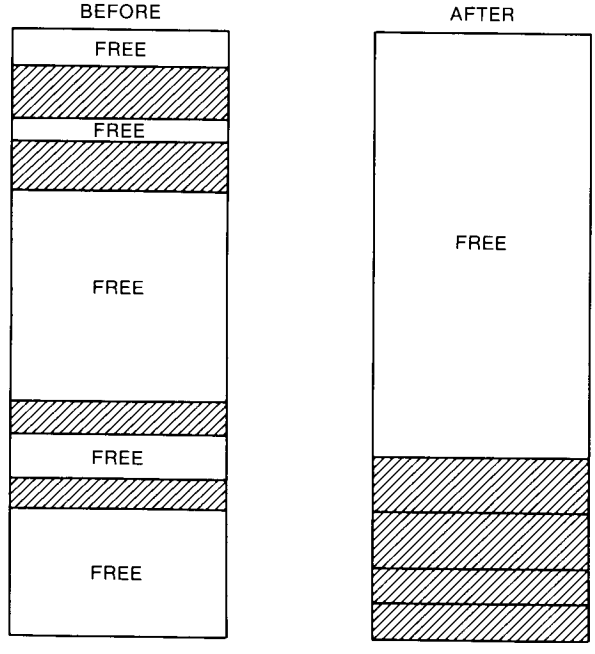


Figure 7–3
How the Memory Manager compacts memory

Manager to reallocate the purged block. After a block of memory is purged, however, the data in it is irretrievably lost, so only the memory—not the data—can be retrieved by a program.

Properties of Memory Blocks

As mentioned, an application program can control the properties of a memory block by setting up a memory attributes word and passing it to the Memory Manager in a `_NewHandle` call. Most attributes in an attributes word are defined when the block is allocated and can't be changed. Some attributes, however, can be modified after allocation.

The layout of a memory attributes word is shown in figure 7–1. In each bit position, a value of 1 means the attribute defined by the bit applies to the block. You might think of setting the bit to 1 as applying a restriction to the block.

Allocation Attributes

When a block is allocated, several bits in the attributes word set restrictions on how the block is allocated. These attributes can only be set when the block is allocated. The allocation attributes are

- Fixed. If a block is fixed, it cannot be moved when memory is compacted. Code blocks are usually fixed, but data blocks usually should not be fixed.
- Bank boundary limited. Specifies that a block must not cross banks. Code blocks may never cross banks, and making a data bank cross bank boundaries is very risky.
- Special memory usable. Specifies that the block may be allocated in special memory, or memory used by the IIc and IIe. Special memory includes banks $00 and $01 and screen memory.
- Page aligned. For timing or other special reasons, code or data may need to be page aligned.
- Fixed address. The block must be at a specified address when allocated. A fixed address attribute should be used only in special situations, for example, in allocating a graphics screen.
- Fixed bank. The block must start in a specified bank, for example, on the direct page.

Modifiable Attributes

As noted, the Memory Manager can move or purge a block while making room for a new block. The attributes that determine whether a block can be moved or purged can be changed by an application after a block is created. These attributes are

- Locked. When a block is locked, it is unmovable and unpurgeable regardless of the setting of the fixed or purge level attributes. A block can thus be locked temporarily while it is being executed or referenced.

- Purge level. Purge level is a 2-bit number defining the purge priority of a block.

When the Memory Manager starts purging blocks of memory, the order of the purging is based on the purge level of the block. The purge level is a 2-bit number specifying the purging priority of the block. The values are

3 Most purgeable (used by System Loader)
2 Next most purgeable
1 Least purgeable
0 Not purgeable

Application programs should use only purge levels 0, 1, and 2; level 3 is reserved for the System Loader. When some applications exit, the memory is not freed but its blocks are set to level 3. The old application can thus be restarted without accessing the disk if the new application did not need the space. If the Memory Manager purges any blocks of an application in this state, it purges all of that application's blocks.

The Event Manager

Because the IIGS is designed to use event-driven programming, the Event Manager is a vital tool set. It allows applications to monitor the actions initiated by the user—such as movements using the mouse, keyboard, and keypad—and to respond accordingly.

In an event-driven program, the actions tracked and handled by the Event Manager are known, logically enough, as *events*. For example, when the user presses or releases the button on top of the mouse, that is a mouse down or mouse up event. When the user presses a key on the keyboard, that is a key down event. If the user presses a key and holds it down, that is an auto-key event.

When an event recognized by the Event Manager takes place, the Event Manager may report it immediately or place it in a queue, according to its priority. When the Event Manager has a series of events waiting in its queue, it removes and reports them, one at a time. But they are not necessarily reported in the order in which they were detected because some events have higher priorities than others. You examine the priorities of events later in this section.

When the Event Manager detects a user-generated event—such as a mouse button being pressed or a key being held down—it places information about the event in a record in memory called an *event record*. The application can then access the contents of the event record to find out what kind of event has taken place so that it can determine what to do. You see what an event record looks like and how it is used later in this section.

When a user-generated event is detected, and information about it is placed in an event record, the application using the Event Manager decides what to do about the reported event. But not all events detected by the Event

Manager are generated by the user. The Event Manager is also used by other tools in the IIGS Toolbox. For example, the Window Manager uses events to coordinate the order and display of windows on the screen. When toolkits such as the Window Manager use the Event Manager, they often decide what to do about the event notifications they receive.

Later in this section, you see how application programs and other tools in the IIGS Toolbox use the Event Manager. Before that, though, let's see what kinds of events are handled by the Event Manager.

Types of Events

Events handled by the Event Manager can be categorized by types. Some types of events report actions by the user. Others are generated by the Window Manager, the Control Manager, device drivers, or even the application being executed. The IIGS system handles some events before the application ever sees them, but it leaves others for applications to handle. We'll now pause to examine the types of events the Event Manager can handle.

Mouse Events

When you press the button on the top of the IIGS mouse, the system generates a mouse down event. When you release the button, the system generates a mouse up event. Movements of the mouse update the cursor position but are not reported as events.

Keyboard Events

When you press any character key on the IIGS keyboard or keypad, the system generates a key down event. The character keys include all keys except Shift, Caps, Lock, Control, Option, and Apple, which are called modifier keys. Modifier keys are treated differently and generate no keyboard events of their own. When an event is posted, the state of the modifier keys is reported in a special modifier field in the event record. The program using the Event Manager then decides what the pressing of a modifier key should do.

The character keys on the keyboard and keypad also generate auto-key events when you hold them down. Two different time intervals are associated with auto-key events. The first auto-key event is generated after an initial delay has elapsed since the key was originally pressed. This is called the *repeat delay*. Subsequent auto-key events occur each time a certain repeat interval has elapsed since the last such event. This is called the *repeat speed*. You can change these values by using the IIGS Control Panel.

Window Events

The Window Manager generates events to coordinate the display of windows on the screen. (You examine the Window Manager in greater detail in chapter 10.) Events generated by the Window Manager are divided into two categories: activate events and update events.

An activate event is generated each time an inactive window becomes active or an active window becomes inactive. Activate and deactivate events generally take place in pairs; that is, one window is deactivated and then another is activated.

An update event takes place when all or part of a window's contents need to be drawn or redrawn, usually as a result of the user opening, closing, activating, or moving a window.

Other Events

There are other events the Event Manager can handle. For example:

- Device driver events, which (as you might guess) are generated by device drivers. A device driver event can signify the receipt or interruption of I/O data.
- A desk accessory event, which takes place when you activate a classic desk accessory such as the IIGS Control Panel.
- Application events, which are defined by application programs. A program can define as many as four application events of its own and can use them for any purpose. A call titled `PostEvent` places application-defined events in the event queue.

Priorities of Events

When the Event Manager is active, it collects events from a variety of sources and reports them to the application on demand, one at a time. But, as noted, the events are not necessarily reported in the order in which they took place because some have a higher priority than others. The Event Manager places events in a queue and handles them according to a strict priority system.

In general, the Event Manager retrieves events from the event queue in the order in which they were posted. But the way in which types of events are generated and detected causes some events to have a higher priority than others. Also, not all events are kept in the event queue. Furthermore, when an application asks for an event, it can specify the types of events it is interested in, and this can cause the Event Manager to pass over some events in favor of others.

If the queue becomes full, the Event Manager begins discarding old events to make room for new ones as they're posted. Discarded events are always the oldest ones in the queue.

Events are carried out by the Event Manager in the following order:

1. Activate events (a window becoming inactive before another window becomes active). Activate events have priority over all other types of events. They are detected in a special way and are never actually placed in the event queue. The Event Manager's `GetNextEvent` and `EventAvail` routines (which you look at in more detail later) check for pending activate events before looking in the event queue, so they always return such an event if one is available. Because of the special way the routines detect activate events, there can't be more than two such events pending at the same time. At most, there is one event for a window becoming inactive, followed by another event for a window becoming active.

2. Switch events (reserved for future use). Switch events also remain outside the event queue. If no activate events are pending, the `GetNextEvent` and `EventAvail` routines check for a switch event before looking in the event queue. If a switch event is available, the routines check to see if any update events are pending. If so, they return the update event to the application. `GetNextEvent` and `EventAvail` return switch events to the application only if update events are pending. This ensures that all windows are updated before the application is switched.
3. Mouse down, mouse up, key down, auto-key, device driver, application-defined, and desk accessory events (handled in order of posting). This category includes all event types placed in the event queue. With the exceptions noted previously, the Event Manager retrieves them from the queue in the order of their posting. The `GetNextEvent` and `EventAvail` calls only return events from this category.
4. Update events (in front-to-back window order). Update events, like activate and switch events, are not placed in the event queue, but are detected in another way. If no higher priority event is available, `GetNextEvent` and `EventAvail` check for windows whose contents need to be drawn. If they find one, they return an update event for that window. `GetNextEvent` and `EventAvail` also check the order (from front to back) in which windows are displayed on the screen. If two or more windows require updating, `GetNextEvent` and `EventAvail` return an update event for the frontmost window.

Event Records

When the Event Manager detects an event, it returns information about the event in an event record. The event record includes the following information:

- Type of event detected
- Time the event was posted, in ticks since system startup
- Location of the mouse when the event was posted, expressed in global (screen) coordinates
- State of mouse buttons and modifier keys when the event was posted
- Additional information that might be required for a particular type of event, such as which key the user pressed or which window is being activated

Every event, including a null event, results in data being entered into an event record by the Event Manager. Listing 7–4 shows how an event record is included in a data section of a program.

Listing 7–4
An event record

```
EventRecord     anop
What            ds 2        ; event code (word)
Message         ds 4        ; event message (long)
When            ds 4        ; ticks since startup (long)
Where           ds 4        ; mouse location (point)
Modifiers       ds 2        ; modifier flags (word)
```

In listing 7–4, the `When` field contains the number of ticks since the system was last started. The `Where` field contains the location of the mouse, in global coordinates, when the event was posted. Now you'll examine the contents of the other fields in an event record.

The What Field

The `What` field of an event record contains an event code that tells what kind of event was detected by the Event Manager. The Event Manager's event codes, and their meanings, are listed in table 7–3.

The Message Field

The `Message` field contains an event message that returns additional information about the detected event. The nature of this message depends on the event type, as shown in table 7–4.

The Modifiers Field

The `Modifiers` field of an event record shows the state that various keys and control buttons were in when an event was posted. In addition, the `ActiveFlag` and `ChangeFlag` bits in the `Modifiers` field provide further information about activate events. See table 7–5.

Table 7–3
Event Manager's Event Codes

Code	Name	Meaning
0	`NullEvt`	Null event
1	`MouseDownEvt`	Mouse down event
2	`MouseUpEvt`	Mouse up event
3	`KeyDownEvt`	Key down event
4		Undefined
5	`AutoKeyEvt`	Auto-key event
6	`UpdateEvt`	Update event
7		Undefined
8	`ActivateEvt`	Activate event
9	`SwitchEvt`	Switch event
10	`DeskAccEvt`	Desk accessory event
11	`DriveEvt`	Device driver event
12	`App1Evt`	Application-defined event
13	`App2Evt`	Application-defined event
14	`App3Evt`	Application-defined event
15	`App4Evt`	Application-defined event

Table 7–4
Event Messages

Event Type	Event Message
Mouse down	Button number (0 or 1) in low word; high word undefined
Mouse up	Button number (0 or 1) in low word; high word undefined
Key down	ASCII code in low word, low byte; low word, high byte clear; upper 3 bytes undefined
Auto-key	ASCII code in low word, low byte; low word, high byte clear; upper 3 bytes undefined
Activate	Pointer to window
Update	Pointer to window
Device driver	Defined by device driver
Application	Defined by application
Switch	Undefined
Desk accessory	Undefined
Null	Undefined

Table 7–5
Modifiers Field of an Event Record

Bit	Name	Value
0	`ActiveFlag`	0 = Window being deactivated 1 = Window being activated
1	`ChangeFlag`	0 = No change 1 = Active window being changed to system or application window
2	Reserved	
3	Reserved	
4	Reserved	
5	Reserved	
6	`Btn0State`	0 = Mouse button down 1 = Mouse button up
7	`Btn1State`	0 = Mouse button 2 down 1 = Mouse button 2 up
8	Apple key	0 = Apple key up 1 = Apple key down
9	Shift key	0 = Shift key up 1 = Shift key down
10	Caps lock key	0 = Caps lock up 1 = Caps lock down
11	Option key	0 = Option key up 1 = Option key down
12	Control key	0 = Control key up 1 = Control key down
13	Keypad	0 = Key pressed on keyboard 1 = Key pressed on keypad
14	Reserved	
15	Reserved	

Bits 6 through 13 of the `Modifiers` field show the state of the mouse button and modifier keys at the time an event was posted. The `Btn0State` and `Btn1State` bits (bits 6 and 7) are set to 1 if the corresponding mouse button is up. The bits for the five modifier keys are set to 1 if their corresponding keys are down.

The `ActiveFlag` is set to 1 if a window pointed to by the event message is being activated or set to 0 if it is being deactivated. The `ChangeFlag` bit is set to 1 if the active window is being changed from an application window to a system window, or vice versa. Otherwise, it is set to 0.

Loading and Initializing the Event Manager

Now that you know how to interpret event records, you're ready to load and initialize the Event Manager. Before the Event Manager tool set is started up, it must be loaded. In most cases, the best way to load the Event Manager is with the Tool Locator's `_LoadTools` call, described previously in this chapter.

When the Event Manager is loaded, several other operations must be carried out before it can be started. For example, before a call to start the Event Manager can be issued, these tool sets must be in memory and initialized:

- Tool Locator. (No action needed; initialization is automatic.)
- Memory Manager. (Does not have to be loaded; must be initialized if a user ID is needed.)
- Miscellaneous Tool Set. (Must be loaded and initialized.)
- QuickDraw II. (Must be loaded and initialized.)

Before a program can start up the Event Manager, it must also obtain four direct pages that are reserved for use by QuickDraw II and the Event Manager. The QuickDraw tool set requires three reserved direct pages and the Event Manager requires three. Listing 7–5 is a fragment of code that shows how to set up three private direct pages for QuickDraw and one for the Event Manager.

Listing 7–5
Reserving direct pages for QuickDraw and the Event Manager

```
PushLong #0                ; space for handle
PushLong #$300             ; eight pages
PushWord MyID
PushWord #$C001            ; locked, fixed, fixed bank

PushLong #0
_NewHandle
```

```
pla
sta DPHandle
pla
sta DPHandle+2

lda [DPHandle]
sta DPPointer
```

In listing 7–5, the Memory Manager call **NewHandle** obtains the direct page workspace that QuickDraw and the Event Manager need. The parameters passed to **NewHandle** specify a block size of $400 (three pages for QuickDraw and one for the Event Manager) and an attribute word of $C001, or %1100 0000 0000 0001. This parameter tells the Memory Manager that the block it assigns should be locked and fixed and should be situated in bank $00.

When QuickDraw and the Event Manager have the reserved page zeros they need, they can be started up with the calls **QDStartup** and **EMStartup**. Listing 7–6 shows how QuickDraw and the Event Manager are initialized in a program.

Listing 7–6
Starting the Event Manager

```
*** INITIALIZE QUICKDRAW II ***

            lda DPPointer              ; pointer to direct page
            pha
            PushWord #ScreenMode       ; either 320 or 640 mode
            PushWord #160              ; max size of scan line
            PushWord MyID
            _QDStartup
            ErrorCheck 'Could not start QuickDraw.'

*** INITIALIZE EVENT MANAGER ***

            lda DPPointer              ; pointer to direct page
            clc
            adc #$300                  ; QD direct page + #$300
            pha                        ; (QD needs 3 pages)
            PushWord #20               ; queue size
            PushWord #0                ; Xclamp low
            PushWord #MaxX             ; Xclamp high
            PushWord #0                ; Y clamp low
            PushWord #200              ; Y clamp high
            PushWord MyID
            _EMStartup
            ErrorCheck 'Could not start Event Manager.'
```

Writing an Event Loop

When you load the Event Manager, start the tools it uses, and supply QuickDraw and the Event Manager with the direct page space they need, you are ready to write a program that uses an event loop handled by the Event Manager.

Some ruffles and flourishes would be appropriate at this point because learning to write event loops is one of the most important skills you'll master in your quest to become an Apple IIGS programmer. If you follow Apple's IIGS interface guidelines—and you should, if you want your programs to be user-friendly and compatible with future models of the IIGS—every program you write has to be based on an event loop. After you start writing event loop programs, you'll probably be glad you did. Event-driven programs are easier to write, understand, and use than old-fashioned sequential-style programs. In an event-driven program, a very short main loop controls an extremely complex program, and a quick glance usually tells you a lot about how the program works.

Listing 7–7 is the main loop of a simple event-driven program, called EVENT.S1, which is listed in its entirety later in this section. Let's pause for a look at its main loop and then move on to the complete program.

Listing 7–7
Main loop of an event-driven program

```
Again       PushWord #0                 ; space for result
            PushWord #$000A             ; key down & mouse down events
            PushLong #EventRecord
            _GetNextEvent

            pla
            beq Again

            lda EventWhat               ; get event code
            asl a                       ; code * 2 = table ocation
            tax                         ; X is index register

            jsr (EventTable,x)          ; look up event's routine
            lda QuitFlag
            beq again

            rts
```

How an Event Loop Works

As listing 7–7 illustrates, the heart of a typical event loop is the Event Manager call `GetNextEvent`. When you call `GetNextEvent`, you have to pass it three parameters:

- A 1-word space on the stack, which `GetNextEvent` fills with a value before it returns.
- A 1-word mask, which tells `GetNextEvent` what kinds of events to look for and what kinds of events to ignore. An event mask is a word in which each bit stands for one type of event. By setting certain bits and leaving other bits clear, you instruct the Event Manager to be on the lookout for certain types of events, and to pay no attention to others. Table 7–6 lists the Event Manager's event mask. When the Event Manager is in an event loop, it reports each type of event that has a bit set in the event mask and ignores each event whose corresponding bit is clear. If you pass the Event Manager an event mask of $FFFF, it reports on all events detected.
- A pointer to an event record. When an application uses the Event Manager, it must place an event record somewhere in memory. Then, when the Event Manager posts an event, it can place important information about the event in the event record.

When the Event Manager processes a `GetNextEvent` call, it returns a 1-word Boolean value: a nonzero value (true) if an event was detected and a zero value (false) if there was no event.

The `GetNextEvent` call is usually used in a loop. In listing 7–7, `GetNextEvent` is used in the loop labeled `Again`. Each time the loop makes a cycle, `GetNextEvent` is called. Then the 1-word Boolean value returned by `GetNextEvent` is pulled off the stack. If `GetNextEvent` does not detect an event, the program branches back to the line labeled `Again` and makes another `GetNextEvent` call.

Interpreting the Event Record

If `GetNextEvent` detects an event, it places information about the event in an event record, which must be set up by the program using the Event Manager. Listing 7–8 is an event record you'll be using in the EVENT.S1 program later in this chapter.

Listing 7–8
Event record in the EVENT.S1 program

```
EventData       DATA

EventRecord     anop
EventWhat       ds 2
EventMessage    ds 4
EventWhen       ds 4
EventWhere      ds 4
EventModifiers  ds 2

                END
```

Table 7–6
Event Manager's Event Mask

Bit	Name	Value
0	Not used	
1	Mouse down mask	0 = No mouse down event 1 = Mouse down event
2	Mouse up mask	0 = No mouse up event 1 = Mouse up event
3	Key down mask	0 = No key down event 1 = Key down event
4	Not used	
5	Auto-key mask	0 = No auto-key event 1 = Auto-key event
6	Update mask	0 = No update event 1 = Update event
7	Not used	
8	Activate mask	0 = No activate event 1 = Activate event
9	Switch mask	0 = No switch event 1 = Switch event
10	Desk accessory mask	0 = No desk accessory event 1 = Desk accessory event
11	Device driver mask	0 = No device driver event 1 = Device driver event
12	Not used	
13	Application-defined events	
14	Application-defined events	
15	Application-defined events	

As listing 7–8 shows, the event record in the EVENT.S1 program has five elements, or fields:

- `What` field, called `EventWhat`. In this field, the Event Manager returns a code telling what kind of event was detected. The event codes that can be returned in this field are listed in table 7–3.
- `Message` field, called `EventMessage`. The nature of this message depends on the type of event detected, as shown in table 7–4.
- `When` field, called `EventWhen`. In this field, the Event Manager returns the number of clock ticks since the system was last started.
- `Where` field, called `EventWhere`. In this field, the Event Manager places the location of the mouse, in global coordinates, when the event was posted.
- `Modifiers` field, called `EventModifiers`. When a `GetNextEvent` call returns, this field contains information about

activate events and the states of certain keyboard keys and hand-controller buttons when an event was posted. A bit-by-bit explanation of this field is in table 7–5.

Using an Event Table

When the Event Manager detects an event and places information about it in an event record, the EVENT.S1 program uses a block of data called an *event table* to decide what to do about the event. An event table is simply a table of pointers to subroutines that an application program uses to respond to events of various types. In the EVENT.S1 program, when the `GetNextEvent` call detects an event and places its event code in the `What` field of an event record, an addressing mode called absolute indexed indirect addressing interprets the event code returned by the Event Manager and jumps to the appropriate subroutine. Listing 7–9 shows the event table used in the EVENT.S1 program.

Listing 7–9
Event table in the EVENT.S1 program

```
EventTable      DATA

                dc i'ignore'            ; 0 null
                dc i'doQuit'            ; 1 mouse down
                dc i'ignore'            ; 2 mouse up
                dc i'doQuit'            ; 3 key down
                dc i'ignore'            ; 4 undefined
                dc i'ignore'            ; 5 auto-key down
                dc i'ignore'            ; 6 update event
                dc i'ignore'            ; 7 undefined
                dc i'ignore'            ; 8 activate
                dc i'ignore'            ; 9 switch
                dc i'ignore'            ; 10 desk acc
                dc i'ignore'            ; 11 device driver
                dc i'ignore'            ; 12 application
                dc i'ignore'            ; 13 ap
                dc i'ignore'            ; 14 ap
                dc i'ignore'            ; 15 ap
                dc i'ignore'            ; 0 in desk

                END
```

Listing 7–10, a fragment of code, uses indexed indirect addressing to loop through an event table to look for an event code returned by the `GetNextEvent` call.

In the first line of listing 7–10, the 65C816 accumulator is loaded with the event code that the Event Manager placed in the `EventWhat` field of the event record. In the next line, an `asl a` instruction multiplies the event code

Listing 7–10
Looping through an event table

```
lda EventWhat                ; get event code
asl a                        ; code * 2 = table location
tax                          ; X is index register
jsr (EventTable,x)           ; look up event's routine
```

now in the accumulator by 2. Because each address in the event table is 2 words, this operation converts the code in the accumulator to the proper offset for the address in the table the program is looking for.

When this offset is calculated, the `tax` instruction copies it into the X register. Finally, in the last line of the example, the absolute indexed indirect addressing mode is used to jump to the desired subroutine.

The EVENT.S1 Program

Now that you know how event loops work, you're ready to type, assemble, and execute the EVENT.S1 program. This program prints a message on the screen and then goes into an event loop. During the event loop, an event mask allows the `GetNextEvent` call to respond only to key down and mouse down events, so nothing more will happen until a key or the button on the IIGS mouse is pressed. When the mouse button or a key is pressed, another message is printed on the screen and the program ends. The complete listing of the EVENT.S1 program (listing 7–12) is at the end of this chapter.

Using the IIGS Toolbox from C

If all you wanted to do in C was write standard, vanilla-flavored, UNIX-style programs, you probably wouldn't be using an Apple IIGS. The real fun (and possible profit) in using the IIGS is in creating programs with razzle-dazzle features like windows, pull-down menus, and glorious color and sound. Thanks to the IIGS C Interface Library, which allows you to make IIGS Toolbox calls from C programs, you can put the IIGS through all its spectacular paces from programs written in C.

The APW C compiler, which was used to write all the C programs in this book, fully supports the use of the IIGS Toolbox from C. In addition to the definitions needed to use the standard C library routines, the APW directory LIBRARIES/CINCLUDE contains all you need (probably more than you need) to use all the Toolbox calls and data structures in C programs. In addition, APW has made thousands of predefined tool-related constants available to C programmers. These include bit-flag attribute values and the error codes returned by tools. The IIGS C Interface Library also contains many other miscellaneous values to convey special information to and from various tool calls.

Pascal-Type Functions

APW C implements an extension to standard C that allows you to use a special set of Pascal calling conventions as well as standard C conventions. In Pascal, the arguments passed to a function are pushed onto the stack from left to right, so the rightmost argument ends up at the top of the stack. In normal C functions, the leftmost argument winds up on top. Pascal-type functions—and this includes all IIGS Toolbox routines and any functions you compile from Pascal source code—expect space for any values they return to be pushed onto the stack before they are called. Instead of returning values in the A and X registers as you might expect a well-behaved C call to do, they place the values they return in the space reserved for them on the stack. Naturally, if the space is not reserved, whatever is there is "clobbered" by the returned values, and your program gets the wrong values back when the call returns.

You'll rarely have to worry about any of this, however, as long as you use the IIGS C Interface Library. Unless you are writing modules in Pascal that are called from C or writing your own Toolbox routines, you won't need to declare anything as Pascal to make Toolbox calls. In APW C, all the conversion details needed to make Toolbox calls are included in a special collection of header files in APW/LIBRARIES/CINCLUDE.

C Toolbox Header File

You don't need to look at the contents of APW's header files to use them in making Toolbox calls. All you have to do is use an `#include` definition to include the names of the tool sets you need in the heading of your program, then make sure you follow the calling conventions listed under *C* at the bottom of each page in the *Apple IIGS Toolbox Reference*. It may be instructive, however, to look at one or two of APW's header files. You can print one to the printer by making this shell call:

```
#type 2/cinclude/control.h >.printer
```

If you use the APW editor instead of your printer to look at a header file, make sure you don't inadvertently change the file's contents. If you do, be sure you don't save the changes when you quit. Better yet, lock your disk or make a copy of the file and open the copy with the editor.

When you print the contents of a header file on the screen or the printer, the first thing you'll see is a heading, which is a comment. Under that, you'll see something like this:

```
#ifndef _quickdraw_
#include <quickdraw.h>
#endif

#ifndef _event_
#include <event.h>
#endif

#ifndef _control_
#define _control_
```

Next are the real contents of the file. Because the definitions that follow depend in part on definitions provided in other headers, they have to be included first. That's why two `#include` statements head up the file. Because C "complains" if you try to compile the same group of definitions more than once, conditional compilation protects against this occurrence:

```
#ifndef _control_
```

The last line:

```
define _control_
```

ensures that the definitions that follow are never recompiled during this compilation.

Next you'll see a long list of constant definitions, each preceded by the expression

```
#define
```

These definitions allow you to use certain named constants described in Apple's Toolbox and C manuals without looking up their values. They make your code easier to write and read. The comments tell you a little about the use of each constant. The ones that say `error` are values placed in the global variable `_toolErr` if an error is detected by one of the tool calls.

After the constant definitions, you'll see a listing of type definitions. These allow you to declare variables in your source that match the structures expected by various tool calls. For instance, you can write:

```
CtlRecHndl myCtl;
```

You can then store a control's handle, returned by `NewControl` or another function, in the variable called `myCtl`. For example:

```
myCtl = NewControl(........);
```

Then there is a listing of function declarations. For example:

```
extern Pascal CtlRecHndl NewControl()
inline(0x0000, dispatcher);
```

This declares a Pascal function returning 4 bytes (long) to be interpreted as a pointer-to-a-pointer to `CtlRecHndl`. It also tells the compiler to insert the inline trap instructions in the object code instead of the usual `jsl` function name generated for normal C functions.

At the end of the function declarations is the line

```
#endif
```

That's the ending required by the conditional compilation directive at the beginning of the file.

The Inline Trap Call

In IIGS C, almost all Toolbox routines are called with the aid of an inline trap. This mechanism is provided so that the linker won't go looking in C libraries for Toolbox routines when it runs across their names in C programs. The inline trap mechanism distinguishes Toolbox calls from C library calls so that this won't happen.

Because tool calls are not located where the linker can find them and because they may be moved as tools are revised, a routine called the Tool Dispatcher, which is always located at address $E1000, uses a jump table to access each tool. This table is updated with each revision of the tools. To call a tool in assembly language, you push the tool number onto the stack and then do a long jump (`jsl`) to $E1000. The engineers who designed APW C could have placed assembled routines for making each call into CLIB, and then you could have called them just as you would any other library routine. But this method would increase the size of CLIB and be inefficient, because it would turn each tool call into two nested subroutine calls.

Instead, they designed the inline trap, which inserts dispatcher calls directly into the object code generated by the C compiler. That's why it is called inline. You will never need to use this call directly; it is used automatically by the function definitions in the headers. But knowing how it works and why it is there gives you a better understanding of what happens when you make a tool call.

Making Calls with Glue

A few tool routines are not accessed using inline dispatcher calls placed in your object code. These routines return too much data on the stack, have arguments smaller than a word (less than 2 bytes), or are otherwise not directly compatible with the APW C compiler. For these, routines called glue have been written in assembly language, assembled, and added to CLIB. The glue routines accept input supplied by compiled C code, adapt it (if necessary) to the format required by the call, execute an inline trap, and pass any results back to the calling routine in a way that can be handled easily in C. If you look in an appropriate C header file, you'll see that such calls look like ordinary C function declarations. For example, in the file misctool.h, you can find this line:

```
extern TimeRec ReadTimeHex();
```

Because of this function, the call `ReadTimeHex` is accessed by a long jump (`jsl`) instead of an inline trap call. This, in turn, causes the APW linker to find a glue routine called `ReadTimeHex` in CLIB and link it with

your program. Again, all the details are handled for you. All you have to do is make the call and pass it any required arguments (in this case there are none).

Pointers, Handles, and the Memory Manager

Two very important definitions in the types.h file are

```
char *Pointer;
```

and

```
Pointer *Handle;
```

Many of the tools in the IIGS Toolbox deal with handles, or pointers to pointers. A handle, as you may recall from chapter 4, is a variable in which the address of another variable, called a master pointer, is stored. All handles must be assigned by the Memory Manager. Much of the data used by the tools in the Toolbox has to be referenced with handles, rather than directly with pointers. The use of handles allows the Memory Manager to compact memory by shuffling data around and purging programs and data that are no longer being used. During this procedure, the address of the master pointer, which the handle points to, remains constant. But the value contained in the master pointer is updated by the Memory Manager whenever the data to which it points is moved.

The definitions of pointer and handle in the types.h file are generic definitions. Because the data type char is a byte, the smallest addressable unit of memory, the definitions `char *Pointer;` and `Pointer *Handle;` are handy for referencing general-purpose data. Most Toolbox routines don't require you to specify the data structure. You just indicate the location of the data structure or, specifically, its master pointer. Variables of type handle are perfect for storing this information. If you want to access the first byte of information pointed to by a handle's master pointer you can write

```
**myhandle
```

In some cases, the data pointed to, or at least the part of the data closest to the beginning of the block, has a specified structure. In such cases, an appropriate data structure is defined in an appropriate toolbox header file. These definitions use the C `typedef` statement. A `typedef` statement declares certain names to stand for a particular data structure or some other complex data type. For each of these definitions, a pointer type and a handle type are also provided. For example, at the end of the definition of a `CtlRec` in ctl.h, you'd see

```
} CtlRec, *CtlRecPtr, **CtlRecHndl;
```

There is an advantage to defining a type that is a handle to a specific structure. When you make a call that gives you a handle to some data that is structured as follows:

```
CtlRecHndl myHandle;
myHandle = GetWindowControls();
```

myHandle is set to the address of the master pointer for the active window's first control. If you want to know the size, shape, and location of this control, you can write

```
Rect myRect;
myRect = (*myHandle)->ctlRec;
```

The EVENT.C Program

The EVENT.C program needs no introduction. It's a C language version of the EVENT.S1 program. The EVENT.C program appears in listing 7–13 at the end of this chapter.

The EVENT.C program uses the standard C library routine `printf` to display a message on the IIgs text screen. Because this program is interested only in key down and mouse down events, a `#define` statement creates a mask for the Event Manager `GetNextEvent` call. Thus, the result of `GetNextEvent` can be treated as a Boolean-type value. It returns a nonzero value (true) when a key or the mouse button is pressed, and it returns a zero value (false) if a key down or mouse down event is not detected. By setting a `done` flag to a nonzero value and using it for the condition of the `while` loop in the EVENT.C program, you guarantee that the loop will end.

Actually, you can compress the `while` loop even more, eliminating the need for a `done` flag:

```
while(!GetNextEvent(SIMPLE_MASK,&myEvent));
```

Although this line accomplishes the same thing as the loop in the program, the syntax we chose is more commonly encountered in event loops that actually do something. That is why it is used in the EVENT.C program.

Listing 7–11, titled INITQUIT.C, is not a complete C program. You can tell that right away because it doesn't have a `main()` function. Instead, it's an `include` file designed to be used with the EVENT.C program. If you want to type and run EVENT.C, you have to type INITQUIT.C, save it on disk, and then include it in EVENT.C with the line

```
#include "initquit.c"
```

which is the first line of the EVENT.C program.

INITQUIT.C does two important things. First, using `#include` statements, it provides EVENT.C with the Toolbox interface files it needs. It then provides the C functions needed to start up and shut down the tools that are loaded and initialized.

The INITQUIT.C program is designed to be used not only with the EVENT.C program, but also with two other programs—PAINTBOX.C and SKETCHER.C—that you encounter in chapter 8. So it's easy to see why it is separated from the rest of the code in EVENT.C. By typing it separately and treating it as an `include` file, you can create it once and then use it in three different programs. It can be modified and used in even more programs—and you will see it again, in expanded versions, in later chapters.

Listing 7–11
INITQUIT.C program

```
#include <TYPES.H>
#include <LOCATOR.H>
#include <MEMORY.H>
#include <MISCTOOL.H>
#include <QUICKDRAW.H>
#include <EVENT.H>

#define MODE 0          /* 320 graphics mode */
#define MaxX 320        /* max X for cursor (for Event Mgr) */
#define dpAttr attrLocked+attrFixed+attrBank /* for allocating direct page
space */

int  MyID;              /* for Memory Manager */

int ToolTable[] = {2,
                   4, 0x0100,  /* QD version 1.1 */
                   6, 0x0100,  /* Event version 1.1 */
                   };

StartTools()            /* start up these tools: */
{
    TLStartUp();              /* Tool Locator */
    MyID = MMStartUp();       /* Mem Manager  */
    MTStartUp();              /* Misc Tools   */
    LoadTools(ToolTable);     /* load tools from disk */
    ToolInit();               /* start up the rest */
}

ToolInit()              /* init the rest of needed tools */
{
char **y;

    y = NewHandle(0x400L,MyID,dpAttr,0L);     /* reserve 4 pages */
```

```
    QDStartUp((int) *y, MODE, 160, MyID);     /* uses 3 pages */
    EMStartUp((int) (*y + 0x300), 20, 0, MaxX, 0, 200, MyID);
}

ShutDown()              /* shut down all of the tools we started */
{
    GrafOff();
    EMShutDown();
    QDShutDown();
    MTShutDown();
    MMShutDown(MyID);
    TLShutDown();
}
```

EVENT.S1 and EVENT.C Listings

Listing 7–12
EVENT.S1 program

```
*
* EVENT.S1
*

; This program prints a message on the screen and then goes into
; an event loop. During the loop, the _GetNextEvent mask allows
; the Event Manager to look only for key down and mouse down
; events. When one of these is detected, the loop ends, another
; message is printed on the screen, and the program ends.

*** A FEW ASSEMBLER DIRECTIVES ***

                    Title 'Event'

                    ABSADDR on
                    LIST off
                    SYMBOL off
                    65816 on
                    mcopy event.macros

                    KEEP Event

*
*   BEGINNING OF PROGRAM
*
```

```
Begin           START
                Using QuitData

                jmp MainProgram         ; skip over data

                END

*
*  SOME DIRECT PAGE ADDRESSES AND A FEW EQUATES
*

DPData          START

DPPointer       gequ    $10             ; direct page pointer
DPHandle        gequ    DPPointer+4

ScreenMode      gequ $00                ; 320 mode
MaxX            gequ 320                ; X clamp high

                END

*
*  MAIN PROGRAM LOOP
*

MainProgram     START

                phk
                plb

                tdc                     ; get current direct page
                sta MyDP                ; and save it for the moment

                jsr ToolInit            ; start up all tools we'll need

*** SET UP INPUT AND OUTPUT SLOTS ***

                PushWord #0             ; set input to slot 3
                PushLong #3
                _SetInputDevice
                PushWord #0
                _InitTextDev

                PushWord #0
                PushLong #3             ; set output to slot 3
                _SetOutputDevice
                PushWord #1
                _InitTextDev
```

```
                jsr PrintMsg1           ; print message on screen
                jsr EventLoop           ; check for key & mouse events

*** WHEN EVENT LOOP ENDS, WE'LL SHUT DOWN ***

                jsr Shutdown
                jmp Endit

MyDP            ds  2

                END

*
*  EVENT LOOP
*

EventLoop       START
                Using QuitData
                Using EventTable
                Using EventData

Again           PushWord #0             ; space for result
                PushWord #$000A         ; key down & mouse down events
                PushLong #EventRecord
                _GetNextEvent
                pla
                beq Again
                lda EventWhat           ; get event code
                asl a                   ; code * 2 = table location
                tax                     ; X is index register
                jsr (EventTable,x)      ; look up event's routine
                lda QuitFlag
                beq again

                rts

                END

*
*  ROUTINE THAT PRINTS OPENING STRING
*

PrintMsg1       START

                _GrafOff

                PushWord #$8C           ; clear screen
                _WriteChar
```

```
                PushLong #StartMsg
                _WriteCString

                rts

StartMsg        dc c'Press any key to continue: ',h'0d00'

                END

*
*  THIS IS WHERE WE INITIALIZE OUR TOOLS
*

ToolInit        START
                using MMData

*** START UP TOOL LOCATOR ***

                _TLStartup                  ; Tool Locator

*** INITIALIZE MEMORY MANAGER ***

                PushWord #0
                _MMStartup

                pla
                sta MyID

*** INITIALIZE MISC. TOOLS SET ***

                _MTStartup

*** GET SOME DIRECT PAGE MEMORY FOR TOOLS THAT NEED IT ***

                PushLong #0                 ; space for handle
                PushLong #$800              ; eight pages
                PushWord MyID
                PushWord #$C001             ; locked, fixed, fixed bank
                PushLong #0
                _NewHandle

                pla
                sta DPHandle
                pla                 sta DPHandle+2

                lda [DPHandle]
                sta DPPointer
```

```
*** INITIALIZE QUICKDRAW II ***

                lda DPPointer           ; pointer to direct page
                pha
                PushWord #ScreenMode    ; either 320 or 640 mode
                PushWord #160           ; max size of scan line
                PushWord MyID
                _QDStartup

*** INITIALIZE EVENT MANAGER ***

                lda DPPointer           ; pointer to direct page
                clc
                adc #$300               ; QD direct page + #$300
                pha                     ; (QD needs 3 pages)
                PushWord #20            ; queue size
                PushWord #0             ; X clamp low
                PushWord #MaxX          ; X clamp high
                PushWord #0             ; Y clamp low
                PushWord #200           ; Y clamp high
                PushWord MyID
                _EMStartup

                rts

                END

*
*    THE ROUTINE THAT ENDS THE PROGRAM
*

EndIt           START
                Using QuitData
                Using MMData

                PushWord #$8C           ; clear screen
                _WriteChar

                PushLong #EndMsg
                _WriteCString

                PushWord MyID
                _MMShutdown

                jsr Shutdown

                _Quit QuitParams
```

```
EndMsg          dc c'Thank You.',h'0d00'

                END

*
*  SHUT DOWN ALL THE TOOLS WE STARTED UP
*

ShutDown        START
                Using MMData

                _EMShutDown
                _QDShutDown
                _MTShutDown

                PushLong DPHandle
                _DisposeHandle

                PushWord MyID
                _MMShutDown
                _TLShutDown

                rts

                END

*
*  ROUTINE THAT SETS THE QUIT FLAG
*

doQuit          START
                Using QuitData

                lda #$8000
                sta QuitFlag
                rts

                END

*
*  A USEFUL AND CONVENIENT WAY NOT TO DO ANYTHING
*

Ignore          START

                rts

                END
```

```
*
*  DATA SEGMENTS
*

EventTable      DATA

                dc i'ignore'            ; 0 null
                dc i'doQuit'            ; 1 mouse down
                dc i'ignore'            ; 2 mouse up
                dc i'doQuit'            ; 3 key down
                dc i'ignore'            ; 4 undefined
                dc i'ignore'            ; 5 auto-key down
                dc i'ignore'            ; 6 update event
                dc i'ignore'            ; 7 undefined
                dc i'ignore'            ; 8 activate
                dc i'ignore'            ; 9 switch
                dc i'ignore'            ; 10 desk acc
                dc i'ignore'            ; 11 device driver
                dc i'ignore'            ; 12 application
                dc i'ignore'            ; 13 application
                dc i'ignore'            ; 14 application
                dc i'ignore'            ; 15 application
                dc i'ignore'            ; 0 in desk

                END

EventData       DATA

EventRecord     anop                    ; table for Event Manager
EventWhat       ds 2
EventMessage    ds 4
EventWhen       ds 4
EventWhere      ds 4
EventModifiers  ds 2

                END

QuitData        DATA

QuitFlag        ds 2

QuitParams      dc  i4'0'
                dc  i4'0'
                dc  i4'0'

                END

MMData          DATA
```

```
MyID            dc   i'0'                    program ID word

                END
```

Listing 7–13
EVENT.C program

```
#include "initquit.c"
#include <stdio.h>      /* needed for putchar */

#define SIMPLE_MASK (mDownMask + keyDownMask)

EventRecord myEvent;
Boolean done = false;

main()
{
StartTools();
PrintMsg();
EventLoop();
ShutDown();
}

PrintMsg()  /*  send message to stdout, then switch display     */
{
    putchar(0x8C);    /* clear screen  */
    printf("Press any key to continue\n");
    GrafOff();        /* display standard text screen */
}

EventLoop()
{
    while(!done)
        done = GetNextEvent(SIMPLE_MASK,&myEvent);
}
```

CHAPTER 8

IIGS Graphics

Using QuickDraw II

There are more than 800 tools in the Apple IIGS Toolbox, and more than a fourth of them are in one tool set: QuickDraw II. QuickDraw II is the tool set that draws everything on the screen when the IIGS is in super high-resolution screen mode. It is used not only by application programs, but also by other tools. When the Window Manager places a window on the screen, all the window's components—scroll bars, title bar, and so on—are drawn by QuickDraw II. When a pushbutton appears in a dialog box, the button and its contents are drawn by QuickDraw II. Even text displayed on a super high-resolution screen is drawn by QuickDraw II.

You can also use the QuickDraw II tool set in your own application programs. This chapter contains two type-and-run programs that demonstrate some of QuickDraw's capabilities. One of the programs, PAINTBOX, draws a rectangle on the screen. The other, SKETCHER, displays a white screen on which you can draw sketches using the IIGS mouse.

Before those programs are presented, though, a description of how QuickDraw II works is helpful. So the first section of this chapter is devoted to a description of QuickDraw II.

What QuickDraw II Can Do

When the Apple Macintosh was designed, its high-resolution screen display was controlled by a tool set called QuickDraw. Now, with the advent

of the IIGS, a IIGS version of the original QuickDraw tool set has been designed—QuickDraw II. When IIGS programmers talk about QuickDraw II, they often leave off the II and refer to it simply as QuickDraw. So when you see the term *QuickDraw* in this book, please remember that, unless otherwise specified, we are discussing QuickDraw II.

The QuickDraw II tool set can draw various kinds of objects on a screen:

- Lines (straight or irregular)
- Rectangles (including squares)
- Ovals (including circles)
- Arcs (actually segments of circles)
- Polygons (multisided figures)
- Regions (collections of other kinds of objects)

QuickDraw can perform the following graphic operations on rectangles, rounded-corner rectangles, ovals, arcs, regions, and polygons:

- Framing, which outlines the shape
- Painting, which fills the shape with a specified color or pattern
- Erasing, which paints the shape using the current background color or background pattern
- Inverting, which inverts the pixels in the shape

Point Data Structure

Every object drawn in QuickDraw is made up of points. In QuickDraw, a *point data structure* contains two integers. The first integer in the structure defines the point's vertical, or Y, coordinate. The second integer defines the point's horizontal, or X, coordinate. Thus, a point can be defined in an assembly language program as

```
APoint    anop
YCoord    ds 2
XCoord    ds 2
```

Rectangle Data Structure

When you define a rectangle, QuickDraw stores it in memory as a data structure. In QuickDraw, a *rectangle data structure* is made up of two point structures. One of the points defines the upper left corner of the rectangle, and the other defines the lower right corner of the rectangle. Thus, it takes only four integers to define the size and location of a rectangle. So a rectangle can be defined this way in an assembly language program:

```
ARect     anop
UYCoord   ds 2
UXCoord   ds 2
LYCoord   ds 2
LXCoord   ds 2
```

Drawing a Rectangle

To draw a rectangle in QuickDraw, you pass its coordinates to a rectangle drawing call such as `FrameRect` or `DrawRect`. The `FrameRect` call outlines a rectangle using the current color, size, pattern, and mask of the current QuickDraw pen. The `PaintRect` call paints a rectangle on the screen using the current pen color, pen pattern, and pen mask. The QuickDraw pen and its attributes are described later in the chapter.

Drawing Ovals, Arcs, and Round Rectangles

The rectangle data structure is also used for drawing three other kinds of objects: ovals, arcs, and round rectangles. To draw an oval using QuickDraw, you define a rectangle and pass its coordinates to an oval drawing call, such as `FrameOval` or `PaintOval`. The `FrameOval` call works much like `FrameRect`. It outlines an oval using the current color, size, pattern, and mask of the current QuickDraw pen. The `PaintOval` call paints an oval on the screen using the current pen color, pattern, and mask.

In QuickDraw jargon, arcs are actually segments of circles. To draw an arc in QuickDraw, you first define the rectangle in which it will lie. Then you pass the rectangle's coordinates, along with the angle described by the arc, to the `FrameArc` or `PaintArc` call. From then on, the `FrameArc` and `PaintArc` calls work like `FrameOval` and `PaintOval`.

"Round rectangles," in QuickDraw lingo, are actually rounded-cornered rectangles. To draw a round rectangle in QuickDraw, you pass the rectangle's coordinates and the height and width of its rounded corners to a round rectangle drawing call such as `FrameRRect` or `PaintRRect`. QuickDraw takes care of the rest of the details.

Point and rectangle data structures are not the only kinds of data structures. QuickDraw uses many other data structures, and some of them are described later in this chapter.

Region and Polygon Data Structures

Regions and polygons make up a unique category in QuickDraw's library of data structures. A *region data structure* is a QuickDraw object made up of other QuickDraw objects. A *polygon data structure* is a figure that can have any number of straight sides.

To set up a region or a polygon, you can't just "fill in the blanks" as you do with other kinds of structures. The next section describes regions and polygons and how they are created in IIGS programs.

Regions

A region is a data structure that can contain other structures, such as rectangles, ovals, arcs, and rectangles. To initialize a region, you must use the QuickDraw call `NewRgn`. This call sets up a region and gives you a handle to it. After you create a region using the `NewRgn` call, you can open it for drawing using `OpenRgn`.

When you create and open a region, you can draw objects in it by using

the object framing calls FrameRect, FrameOval, and FrameRRect. Each call adds an object to the region you are creating.

When you finish drawing a region, you close it with the CloseRgn call. From then on, you can draw the region on the screen by passing its handle to a region drawing call such as FrameRgn or PaintRgn.

Polygons

Polygons are created in a similar way: with a sequence of calls to QuickDraw routines. Before you can start drawing a polygon, you issue the QuickDraw call OpenPoly. The OpenPoly call sets up a polygon and provides you with a handle to it. You can then define the polygon using LineTo calls.

You begin to define a polygon by moving the QuickDraw pen to the polygon's starting point and drawing a line from there to the next point. You can then draw another line from that point to the next point, and so on.

When you finish defining a polygon, you close it with the ClosePoly call. From then on, you can draw or paint it on the screen by passing its handle to polygon drawing calls such as FramePolyand PaintPoly.

The data structure for a polygon consists of two fixed length fields followed by a variable length array. The following shows the data structure for a polygon. (It is presented only for your information, because you will probably never have to set up a polygon data structure in a program. QuickDraw's polygon calls do that for you when they are used as described in this section.)

PolySize	An integer
PolyBBox	A rectangle
PolyPoints	An array [0 . . . ?] of points

The PolySize field of a polygon data structure contains the size, in bytes, of the polygon variable. The maximum size of a polygon is 32K bytes. The PolyBBox field is a rectangle that encloses the polygon. PolyPoints is a dynamic array that expands as necessary to contain the points of the polygon. It specifies the starting point of a polygon and each successive point to which a line is drawn.

When QuickDraw II draws a polygon, it moves its pen to the starting point of the polygon and then draws a series of lines to the remaining points, in the same way points are set up when the polygon is defined. In other words, QuickDraw "plays back" the same series of operations it uses to define the polygon. As a result, polygons are not treated exactly the same as other QuickDraw II shapes. For example, the procedure that frames a polygon draws outside the actual boundary of the polygon, because QuickDraw II line drawing routines draw below and to the right of the pen location.

Routines that fill a polygon with a pattern, however, stay inside the boundary of the polygon. If the polygon's ending point isn't the same as its starting point, these routines add a line between them to complete the shape.

A polygon is also scaled differently from a similarly shaped region if it is being drawn as part of a picture. When a slanted line is stretched, it is

drawn more smoothly if it's part of a polygon rather than part of a region. You may find it helpful to keep in mind the conceptual difference between polygons and regions. A polygon is treated more as a continuous shape; a region is treated more as a set of bits.

Pixel Maps and Conceptual Drawing Planes

When you create an object, QuickDraw places the object in a two-dimensional plane called a *conceptual drawing space*. When an object is placed in this drawing space, its position, like a position on a map, can be pinpointed with coordinates.

There is one fact about a conceptual drawing space that may be a little difficult to grasp. The plane that it describes does not exist anywhere in the IIGS's memory. When an object is defined in QuickDraw's conceptual drawing space, the object exists only as a mathematic image described by coordinates. The object thus takes up much less space in memory than it would if it were stored as a bit-mapped image.

But, before the object can be drawn—for example, on the IIGS screen or on a printer—enough space to hold the drawing must be reserved in memory. The memory area in which objects can be drawn is known as a *pixel map*. A pixel map is made up of tiny dots called picture elements, or pixels. After you create a pixel map, the objects drawn on it can be printed or displayed.

The Big Picture

The conceptual drawing space in which QuickDraw can store objects, measured in pixels, extends from − 16K to + 16K horizontally and from − 16K to + 16K vertically—a space large enough to hold 1,024,000,000 pixels. Figure 8–1 is a simplified diagram of the IIGS's conceptual drawing plane.

This plane is divided into four segments. The coordinate numbered 0,0 is in the middle of the plane. Thus, if you wanted to draw a point in the exact center of the plane, its coordinate would be 0,0.

The segments above and to the left of coordinate 0,0 use negative coordinates. Only the segments below and to the right of 0,0 use positive horizontal coordinates and positive vertical coordinates. For this reason, most of the drawing takes place in the lower right segment of QuickDraw's conceptual drawing plane.

If the entire conceptual drawing space of an Apple IIGS were transferred to a giant pixel map, the map would measure four screens wide by eight screens high (or eight screens wide by four screens high). You could create such a map and display it on your screen, using Window Manager scroll bars to move it, if the IIGS had enough memory capacity.

You don't need that much memory, however, to make full use of the conceptual drawing plane. Even with an unexpanded IIGS system, you can draw objects anywhere in QuickDraw's conceptual drawing space. But before you can transfer an object or a picture from QuickDraw's conceptual drawing

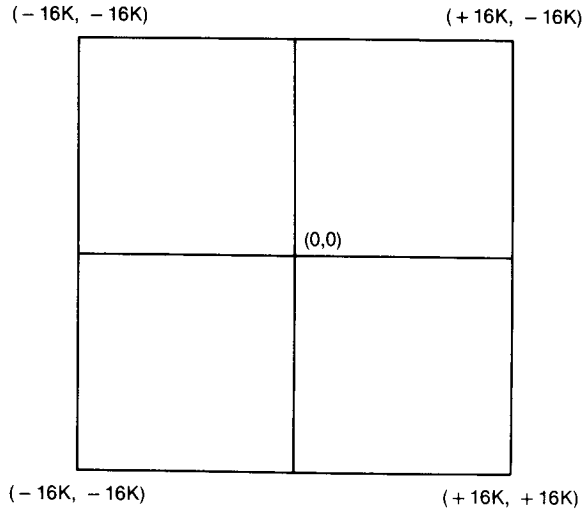


Figure 8–1
QuickDraw II conceptual drawing plane

space to an actual pixel map, you have to make sure there is enough room in the computer's memory to store the pixel map on which your object or picture will be drawn.

Using Pixel Maps

As mentioned, a pixel map is an area of memory that can contain an actual drawing of a graphic image. This image, like an image stored in a conceptual drawing space, is made up of a rectangular grid of pixels. Each pixel on a pixel map has a value that displays a color on the IIGS screen or prints it on a printer. Thus, the value assigned to each pixel in a pixel map is a color code.

Pixels on a pixel map, like coordinates in QuickDraw's conceptual drawing space, can be thought of as points in a Cartesian coordinate system; that is, each pixel on a pixel map has a horizontal coordinate and a vertical coordinate. In QuickDraw II, as in the original QuickDraw system for the Macintosh, the coordinates on a pixel map fall on lines that separate the pixels on the map, rather than on the pixels themselves. This method of assigning coordinates is illustrated in figure 8–2.

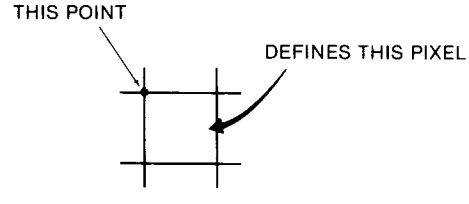


Figure 8–2
Coordinates of a pixel

This system of assigning coordinates makes it very easy to determine when a pixel falls within a given rectangle and when it does not. Knowing whether a pixel is inside a rectangle is quite important in QuickDraw II because many calls deal only with pixels that fall in specific rectangles.

Pixel Maps and Screen Memory

When QuickDraw is initialized, the pixel map it draws on is set by default to the same area of memory that displays the super high-resolution screen, memory address $E12000 to memory address $E19CFF. Thus, when you start QuickDraw, its default drawing area is the screen. However, QuickDraw can draw in any block of free RAM as easily as it can draw on the screen, and applications can instruct QuickDraw to draw anywhere in memory.

Graphics Modes

The IIGS has two super high-resolution graphics modes: a 320-pixel mode and 640-pixel mode. When the IIGS is in 320 mode, the pixel map it uses for its screen display measures 320 pixels wide by 200 pixels high. In 640 mode, its screen display measures 640 pixels wide by 200 pixels high.

Each horizontal line on the IIGS screen is called a *scan line*. So, in both 320 mode and 640 mode, the super high-resolution screen is 200 scan lines high.

Both super high-resolution screen modes use a "chunky"-style pixel organization; the bits used to draw a given pixel on the screen are contained in adjacent bits within 1 byte. In both 320 mode and 640 mode, each scan line on the screen uses 160 bytes of memory. But the degree of "chunkiness" used by each mode is different. In 320 mode, 4 bits represent each pixel display on the screen. In 640 mode, only 2 bits create each screen pixel. Consequently, using 640 mode doubles the number of pixels that can be displayed in each scan line, although the number of bytes used for each scan line is the same in 320 mode and 640 mode.

The use of 640 mode does involve one important trade-off, however. Because only 2 bits define each screen pixel in 640 mode and 4 bits define each pixel in 320 mode, the number of colors that can be displayed in 640 mode is reduced. In 320 mode, sixteen discrete colors can be displayed on the screen simultaneously. In 640 mode, only four discrete colors can be displayed.

This limitation of 640 mode is not as bad as it sounds. With the help of a technique called *dithering*, you can create repeating color patterns that make it appear that more than four colors are displayed. A full description of dithering is beyond the scope of this chapter, but complete instructions for using dithering techniques are in chapter 16 (the chapter on QuickDraw II) of the *Apple IIGS Toolbox Reference*.

The number of colors displayed in both 320 mode and 640 mode can be increased with special interrupts called scan-line interrupts. Instructions for using scan-line interrupts are in chapter 4 (the video and graphics chapter) of the *Apple IIGS Hardware Reference*.

Selecting a Graphics Mode

When QuickDraw is initialized, it determines which graphics mode to use by looking at a parameter passed to it in the `QDStartup` call. As you will see in the programs later in this chapter, the `QDStartup` call has four parameters, one of which is called `MasterSCB`. If you pass the value $00 to the `QDStartup` call in this parameter, QuickDraw starts in 320 mode. If you pass the parameter $80, QuickDraw starts up in 640 mode.

There are also calls that change the graphics mode used inside a program. Descriptions of these calls, and instructions for using them in programs, are in the *Apple IIGS Toolbox Reference*.

Selecting Colors

In both 320 mode and 640 mode, the IIGS selects colors to be displayed on the screen from a block of RAM data called a *color palette*. The IIGS has sixteen color palettes, and each scan line can take its colors from any color palette. Each pixel on a scan line can be drawn in any of the sixteen colors that make up the palette being used by that line. And the 16 colors in each palette can be chosen from 4,096 colors.

When you write programs for the IIGS, you will rarely, if ever, have to deal with color palettes by directly accessing their memory addresses. QuickDraw II has a full complement of calls to select and manipulate color palettes and the colors they contain. For example, the `SetColorTable` call sets a color table to specific values, and the `GetColorTable` call fills a color table with the contents of another color table. There are also calls for getting and setting single colors in color tables.

You can do just about anything with color palettes by using the color table and color entry calls QuickDraw provides. To use colors and color tables effectively, however, it is helpful to know a little about how the IIGS creates and displays color on its screen.

The color palettes used by the IIGS extend from memory address $E19E00 through memory address $E19FFF—an area that begins just 256 bytes higher than the RAM block used for screen memory. There are sixteen color palettes in this space, with 32 bytes used by each palette. Each color palette contains codes for sixteen colors, with 2 bytes used for each color.

A color table, then, is a table of sixteen 2-byte entries, or words. The low nibble of the low byte of each word represents the intensity of the color blue. The high nibble of the low byte represents the intensity of the color green. The low nibble of the high byte represents the intensity of the color red. The high nibble of the high byte is not used. The following illustrates the structure of each color represented in a color palette:

High Byte		Low Byte	
High Nibble	Low Nibble	High Nibble	Low Nibble
Reserved	Red	Green	Blue

As mentioned, each pixel is displayed differently in each of the super high-resolution modes: 4 bits represent each pixel color in 320 mode, and 2 bits represent each pixel color in 640 mode. The higher resolution in 640

mode carries a penalty. A pixel may be displayed in any of sixteen colors in 320 mode, but a pixel may be one of only four colors in 640 mode.

In both modes, the color information to display each pixel is placed in the RAM area reserved for screen memory in a linear and contiguous manner. The first byte of screen memory, in memory address $E12000, corresponds to the upper left corner of the screen display. The last byte in screen RAM, memory address $E19CFF, corresponds to the lower right corner of the screen. Each scan line uses 160 bytes of screen memory.

In 320 mode, it takes 4 bits to determine each pixel color, so two pixels are stored in every byte in the super high-resolution screen buffer. Because 4 bits of data determine the color of each pixel, each pixel on a scan line can represent one of the sixteen colors in the palette that controls the scan line on which the pixel appears.

In 640 mode, color selection is more complicated. In this mode, the 640 pixels in each horizontal line occupy 160 adjacent bytes of memory, and each byte holds 4 pixels that appear side by side on the screen. And the sixteen colors in the palette that controls the scan line are divided into four groups of four colors each. In other words, each palette used for a scan line in 640 mode contains four mini-palettes, each one made up of four colors.

By making careful use of the four mini-palettes used for each scan line, a program can increase the apparent number of colors used in each scan line in 640 mode. Unfortunately, the way in which colors are taken from the four mini-palettes used by each scan line is not intuitive.

The first pixel in each scan line can use any one of the four colors in the third mini-palette in the scan line's full palette. The second pixel can use any of the four colors in the full palette's fourth mini-palette. The third pixel can use any of the four colors in the main palette's first mini-palette. And the fourth pixel can use any of the four colors in the second mini-palette. The way this system works is shown in figure 8–3.

This process repeats itself for each successive group of four pixels in each scan line. Thus, even though a given pixel can be one of only four

PIXEL	VALUE	PALETTE
PIXEL 3	0	COLOR 1
	1	COLOR 2
	2	COLOR 3
	3	COLOR 4
PIXEL 4	0	COLOR 5
	1	COLOR 6
	2	COLOR 7
	3	COLOR 8
PIXEL 1	0	COLOR 9
	1	COLOR 10
	2	COLOR 11
	3	COLOR 12
PIXEL 2	0	COLOR 13
	1	COLOR 14
	2	COLOR 15
	3	COLOR 16

Figure 8–3
Mini-palettes in 640 mode

colors, different pixels in a line can take on any of the colors in a palette. With the help of dithering, software written in 640 mode can display 16-color graphics and 80-column text on the same screen.

Dithering techniques increase the apparent number of colors on a screen by placing certain colors next to each other. (Your eye blends them.) By alternating colors in even and odd mini-palettes, a skilled programmer can control this blending and can thus obtain full-color capabilities in 640 mode. Instructions for using dithering techniques are in chapter 16 of the *Apple IIGS Toolbox Reference*.

Scan-Line Control Bytes

In both 320 mode and 640 mode, the colors used for each scan line on the screen are controlled with a group of RAM bytes called *scan-line control bytes,* or SCBs.

Each scan-line control byte represents one scan line on the IIGS screen. For each horizontal screen line, you can use the appropriate scan-line control byte to select

- The 16-color palette from which the scan line will take its colors.
- If the scan line will use color fill mode. Color fill mode streamlines the process of drawing consecutive pixels in the same color on a scan line. Color fill is available only in 320 mode and is described more fully in the *Apple IIGS Hardware Reference*.
- If a scan-line interrupt should be generated for the scan line. (Instructions for using scan-line interrupts are in the *Apple IIGS Hardware Reference*.)
- Whether the scan line will use 320-pixel or 640-pixel resolution.

Each of these scan-line attributes is controlled by 1 bit, or group of bits, in the SCB for the line. The bits in a scan-line control byte, and what they do, are described in table 8–1.

How To Use SCBs

When you write programs for the IIGS, you will rarely, if ever, need to manipulate QuickDraw's scan-line control bytes by accessing them directly. The QuickDraw tool set has several calls to get and set SCBs. It is easier (and safer) to work with SCBs using these calls than it is to access them directly by their memory locations. Calls that can be used to control SCB settings include `GetSCB`, which returns the SCB setting for a given scan line, `SetSCB`, which sets an SCB that controls a given line, and `SetAllSCBs`, which sets all the SCBs on the screen to a specified value.

Descriptions of all SCB calls, and instructions for using them, are outlined in chapter 16 (the QuickDraw II chapter) of the *Apple IIGS Toolbox Reference*.

Where To Find SCBs

The block of memory that contains QuickDraw's scan-line control bytes extends from memory address $E19D00 through memory address $E19DFF.

Table 8–1
Structure of a Scan-Line Control Byte

Bit	Name	Value
7	320/640 mode flag	1 = Horizontal resolution equals 640 pixels. 0 = Horizontal resolution equals 320 pixels.
6	SCB interrupt flag	1 = Interrupt generated for this scan line. (When this bit is 1, the scan line interrupt status bit is set at the beginning of the scan line.) 0 = Scan line interrupts disabled for this scan line.
5	Color fill mode flag	1 = Color fill mode enabled. (This mode is available in super hi-res 320-pixel mode only. In 640-pixel mode, color fill mode is disabled.) 0 = Color fill mode disabled.
4		Reserved; do not modify.
0–3	Palette select code	Palette (0–15) chosen for this scan line.

This section of memory, as shown in figure 8–4, falls between the area of memory for the super high-resolution screen map and the area of memory for the color palettes that control the colors of the pixels on the screen.

The address of the scan-line control byte for each scan line is $E19DXX, where XX is the hexadecimal value of the line. For example, the control byte for the first scan line (line 0) is located in memory location $9D00, the control byte for the second scan line (line 1) is in location $9D01, and so on.

(Actually, only the first 200 bytes of the 255 bytes in the memory page beginning at $E19D00 are scan-line control bytes. The remaining 55 bytes are reserved for future expansion. To make sure your programs are compatible with future Apple II products, you should not modify these 55 bytes.)

GrafPorts

Now that you know a few facts about QuickDraw II, you're ready for more detail. To understand how QuickDraw II works, you need to be familiar with a data structure called a *GrafPort*. Without GrafPorts, there would be no such thing as a QuickDraw tool set.

Here is a summary of what GrafPorts are and what they do. First, a GrafPort is not a block of data designed to be displayed on the IIGS screen. Rather, it is a data structure that contains important information that QuickDraw uses to create a screen display.

A GrafPort, like most other kinds of QuickDraw data structures, is made up of records. Some of the records in a GrafPort data structure are also data structure. A GrafPort data structure also includes integers, pointers,

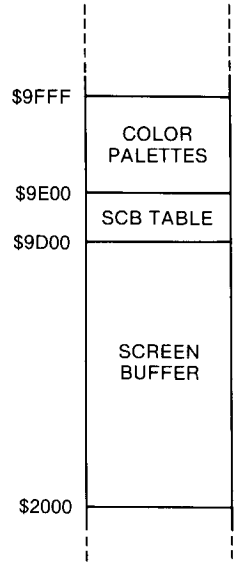


Figure 8–4
Memory map of screen buffer, SCB table, and color palettes

handles, rectangles, and other kinds of data. Understanding how these kinds of data are used by a GrafPort—and how they relate—is an important part of understanding QuickDraw II.

Drawing Environments

The data stored in a GrafPort is sometimes referred to as a *drawing environment.* A drawing environment is simply a collection of data that QuickDraw can refer to easily when it needs to draw a screen display.

The advantage of the GrafPort system is that it allows a complex drawing environment to be maintained in a single, easily accessible record. By switching between GrafPorts, QuickDraw can change drawing environments very rapidly and can thus create many different kinds of screen displays quite efficiently. More than one GrafPort can be stored in memory, and it is not unusual to have several GrafPorts in memory at one time. When a program uses several screen windows, for example, each window has a GrafPort of its own.

Using GrafPorts

In QuickDraw, all graphic operations are performed in GrafPorts. Before a GrafPort can be used, it must be allocated and initialized with the QuickDraw call `OpenPort`. But most applications do not call `OpenPort` directly. They use the IIGS Window Manager, which makes the call for them.

The QuickDraw call `ClosePort` closes a GrafPort when it is no longer needed. The GrafPort itself can be disposed of with the Memory Manager call `DisposeHandle`. The Window Manager will also make these calls for you when it is used to control the windows in a program.

In an application that uses multiple windows, each window is a separate GrafPort. If an application draws into more than one GrafPort, the `SetPort`

call sets up the GrafPort that is used for the drawing. Again, the Window Manager makes this call when it manages the windows in a program.

At times, an application needs to preserve the current GrafPort. In this case, the `GetPort` call saves the current port, and the `SetPort` call sets the port to be drawn in. Then, when drawing in the second port is completed, `SetPort` is used again to restore the previous port. The Window Manager also takes care of making these calls when it manages the windows and GrafPorts in a program.

Structure of a GrafPort Record

The fields in a GrafPort include information on such topics as

- The area of memory (the pixel map) in which images are drawn. This area of memory is pointed to by a pointer in the GrafPort record.
- Whether images are drawn in 320 mode or 640 mode.
- How drawings are trimmed, or clipped, to fit in the areas in which they lie.
- The size, shape, and pattern of the pen used for drawing.
- The font used for displaying text and how text is styled.
- Where objects that are drawn are stored in memory.

The structure of a GrafPort is no secret. It has been published by Apple and is listed in table 8–2. Apple strongly recommends, however, that programmers avoid the temptation of directly modifying the fields in GrafPorts. Instead, programmers are advised to access fields in GrafPorts only through QuickDraw calls.

If you count all the bytes in the GrafPort in table 8–2, you will see that a GrafPort data structure is 170 ($AA) bytes long. So, in an Apple IIGS assembly language program, the memory space required for one GrafPort could be set aside as follows:

```
GrafPort        ds $AA
```

PortInfo Data Structure

As mentioned, a GrafPort data structure includes many kinds of values: handles, integers, pointers, and even smaller data structures. In a GrafPort structure, each of these values is known as a field. Thus, the first field in a GrafPort structure, as table 8–2 illustrates, is a data structure within a data structure: in this case, a 16-byte structure called `PortInfo`. When a `PortInfo` structure lies outside a GrafPort structure, it is often called a `LocInfo` structure. And when a `LocInfo` structure is used in a call that transfers pixel map data from one area of memory to another (such as `PPToPort` or `PaintPixels`), it is often referred to as a `SrcLocInfo` structure. So, in QuickDraw jargon, a `PortInfo` structure, a `LocInfo` structure, and a `SrcLocInfo` structure are all the same.

Now let's see what a `PortInfo` (or `LocInfo`, or `SrcLocInfo`) structure looks like, and how it's used in a GrafPort data structure. The layout of a `LocInfo` structure is illustrated in listing 8–1.

Table 8–2
The Structure of a GrafPort

Field	Length	Description
	Port Information	
PortInfo	16 bytes	LocInfo data structure
PortRect	8 bytes	Rectangle data structure
ClipRgn	4 bytes	Handle to a region
VisRgn	4 bytes	Handle to a region
BkPat	32 bytes	Pattern data structure
	Pen State Data Structure	
PnLoc	4 bytes	Point structure
PnSize	4 bytes	Point structure
PnMode	2 bytes	Integer
PnPat	32 bytes	Pattern data structure
PnMask	8 bytes	Mask data structure
PnVis	2 bytes	Integer
	Font and Text Data	
FontHandle	4 bytes	Handle to a font
FontID	4 bytes	Long integer
FontFlags	2 bytes	Integer
TxSize	2 bytes	Integer
TxFace	2 bytes	Word
TxMode	2 bytes	Integer
SpExtra	4 bytes	Fixed point data structure
ChExtra	4 bytes	Fixed point data structure
	ForeGround and Background Color Data	
FGColor	2 bytes	Integer
BGColor	2 bytes	Integer
PicSave	4 bytes	Handle
RgnSave	4 bytes	Handle
PolySave	4 bytes	Handle
GrafProcs	4 bytes	Pointer (Usually a null pointer, set to 0)
ArcRot	2 bytes	Integer
UserField	4 bytes	Long integer
SysField	4 bytes	Long integer

Add up the bytes in a LocInfo structure, and you'll see that the structure is 16 bytes long. The first integer in a LocInfo structure is called a LocInfoSCB.

Listing 8–1
LocInfo Data Structure

```
LocInfo         anop
LocInfoSCB      ds 2            ;$00 for 320, $80 for 640
LocInfoPicPtr   ds 4            ;pointer to pixel image
LocInfoWidth    ds 2            ;scan line width (#160 is standard)
LIBoundsRect    ds 8            ;format: 0,0,200,320
```

LocInfoSCB Field

When a `LocInfo` structure appears inside a GrafPort data structure, the `LocInfoSCB` field defines the screen resolution of the pixel image that the GrafPort points to. If the value of `LocInfoSCB` is $00, the pixel image is displayed in 320 mode. If the value of `LocInfoSCB` is $80, the pixel image is displayed in 640 mode. An SCB can have other values, as explained previously in this chapter.

LocInfoPicPtr Field

The next field in a `PortLocInfo` structure—the `LocInfoPicPtr` field—is a pointer to the pixel map that the GrafPort describes. When a GrafPort is initialized, the pixel map that `PortLocInfo` points to is the super high-resolution screen. An application can change the `LocInfoPicPtr` field, however, to point to any area of memory in which a pixel map can be stored.

LocInfoWidth Field

The `LocInfoWidth` field of a `LocInfo` structure defines the maximum width, in bytes, of a scan line on the screen. In both 320 mode and 640 mode, the most common value for this field is the width, in bytes, of one screen-sized scan line: 160, or $A0 in hexadecimal notation.

LIBoundsRect Field

The `LIBoundsRect` field is a data structure that describes a rectangle. The rectangle described by the `LIBoundsRect` structure describes a bounds rectangle: a rectangle that encloses the pixel map (or, sometimes, a portion of the pixel map) that the current GrafPort is using. This pixel map is the same one pointed to by the `LocInfoPicPtr` field of the `LocInfo` data structure. More information about bounds rectangles is presented later in this chapter.

An `LIBoundsRect` structure is made up of four integers, or words. Each of these words defines one coordinate of the current GrafPort's bounds rectangle. The order of these coordinates is: top left Y coordinate, top left X coordinate, lower right Y coordinate, and lower right X coordinate. Because a IIGS screen measures 200 scan lines down by 320 pixels across (in 320 mode), the coordinates used in the `LIBoundsRect` structure exactly covering a 320-mode screen are 0,0,200,320.

Drawing with a Pen in QuickDraw II

QuickDraw does most of its drawing using a structure called a *pen*. Each GrafPort in a program has one (and only one) graphics pen, which the GrafPort uses for drawing lines, shapes, and text. A QuickDraw pen has five characteristics: location, size (height and width), drawing mode, drawing pattern, and drawing mask.

When a pen draws an image in a GrafPort, the pen location can always be expressed as a point in the GrafPort's coordinate system or, if a pixel map is used, as a pair of coordinates on the pixel map. The point that defines the location of a pen—like any other point used in QuickDraw—can be located using two integers, or words: an integer defining the point's vertical (Y) coordinate and an integer defining the point's horizontal (X) coordinate.

In QuickDraw, the position of a pen is defined as the point where the next line, shape, or character will begin. This point can be anywhere on a GrafPort's coordinate plane. The top left corner of the pen is at the pen location; the pen hangs below and to the right of this point. When a pen is in a given location, the QuickDraw call `LineTo` makes it draw a line, and the call `MoveTo` moves it to another point without drawing a line. The `MoveTo` and `LineTo` calls are used in a type-and-run program, SKETCHER, which is presented at the end of this chapter.

The pen used in QuickDraw II is rectangular. Its width and height are controlled by several different QuickDraw calls, including `SetPenSize`, `SetPenState`, `GetPenSize`, and `GetPenState`. The default size of a QuickDraw pen is a 1-by-1 pixel square. A pen can be set to this size with the QuickDraw call `PenNormal`. The width and height of a pen can range from coordinate $0000,$0000 to coordinate $3FFE,$3FFE (or 16382,16382 in decimal notation). If either the pen width or the pen height is less than 1, the pen will not draw a visible line.

Pen Patterns

In addition to having a specific size, a QuickDraw pen also has a specific pattern. A *pen pattern* is a 64-pixel image laid out as an 8-by-8 pixel square. When QuickDraw is initialized, it uses a pen pattern made up of all zeros. This type of pen pattern draws a solid line on the screen.

You can set the pen to draw in a pattern on the screen by setting up the pattern in memory and then making the QuickDraw call `SetPenPat`. When you want a pen to draw on the screen in a solid color other than black, you can use the QuickDraw call `SetSolidPenPat`. Instructions for using both of these calls are in chapter 16 of the *Apple IIGS Toolbox Reference*.

Actually, there are two kinds of QuickDraw patterns: pen patterns and background patterns. But both use the same kind of data structure: a 32-byte structure that is a small pixel image. After you set the contents of a pattern, you can use it as either a background pattern or a pen pattern. QuickDraw doesn't care.

In a data segment of a program, either kind of pattern is defined like this:

```
Pattern0        ds 32
```

QuickDraw programs often use pen patterns that define repeating designs. For example, when a pen pattern resembling a brick wall is created, the pen that uses the pattern draws a brick wall, instead of a solid line, on the screen. Figure 8–5 is a pen pattern resembling a brick wall. On the left is what the pattern looks like in memory; on the right is what the pattern looks like when a pen draws it on a screen.

Pen Masks

Another attribute of a QuickDraw pen is a mask. A *pen mask* is an 8-by-8 bit square that, like a pen pattern, defines a repeating design. See figure 8–6. As a line or an object is drawn, this design masks the pattern—only the pixels that "show through" the pen mask appear on the screen. In other words, only those pixels in the pattern aligned with a set bit in the pen mask are drawn.

A pen mask, then, is a special kind of pattern that a pen can draw through to create special effects on a screen. A pen mask is smaller than a pen pattern or a background pattern; a pen mask data structure is only 8 bytes long. In a data segment of a program, memory space for a pen mask is reserved in this manner:

```
Mask0           ds 8
```

The QuickDraw calls `GetPenMask` and `SetPenMask` transfer pen masks to and from GrafPorts. The effect of using a pen mask is illustrated in figure 8–7.

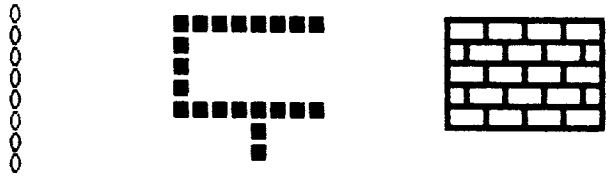


Figure 8–5
Pen pattern in memory and on the screen

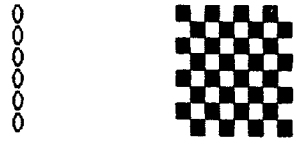


Figure 8–6
Pen mask

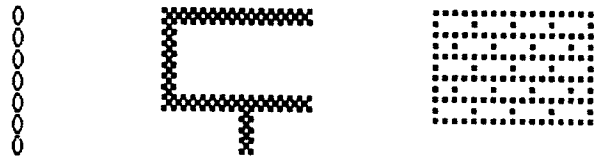


Figure 8–7
Effect of using a pen mask

Pen Modes

Still another attribute of a QuickDraw pen is its mode. The *pen mode* determines how the pen pattern will affect what is already in the pixel image when lines or shapes are drawn. When the pen draws, QuickDraw II first determines which pixels in the pixel image will be affected and finds their corresponding pixels in the pattern. QuickDraw II then does a pixel-by-pixel comparison based on the pen mode, which specifies one of eight Boolean operations to perform. The resulting pixel is stored in its proper place in the pixel image.

The QuickDraw calls `GetPenMode` and `SetPenMode` control the pen mode used in a GrafPort. The pen modes used in QuickDraw are listed in table 8–3.

A pen can be used for two kinds of drawing: normal drawing and erasing. In normal drawing, the pen mode determines what is drawn on the screen. Erasing just fills the affected pixels with the background pattern.

Pen State Structure

As mentioned, each QuickDraw GrafPort has its own drawing pen, and all the attributes of each pen are defined in a structure called a *pen state structure*. Listing 8–2 shows what a pen state structure looks like. For further details, refer to the *Apple IIGS Toolbox Reference*.

Listing 8–2
Pen State Structure

```
PenState    anop
PnLoc       ds 4        ;pen coordinates (Y and X)
PnSize      ds 4        ;pen size (width and height)
PnMode      ds 2        ;pen draws opaque or transparent pattern
PnPat       ds 32       ;pen pattern: 32-byte pixel image
PnMask      ds 8        ;pen mask: 8-byte pixel image
```

Bounds Rectangles

Two kinds of rectangles are very important in QuickDraw. One is a bounds rectangle, and the other is a port rectangle.

The *bounds rectangle* of a GrafPort, often abbreviated `BoundsRect`, is the rectangle defined by the `LIBoundsRect` field of a GrafPort's `LocInfo` data structure. When a GrafPort draws on the IIGS screen, the upper left corner of its bounds rectangle corresponds to the upper left corner of the screen, and the coordinates of its bounds rectangle and its pixel map are the same. If a

Table 8–3
QuickDraw II Pen Modes

Number	Name	Description
$0000	COPY	The default drawing mode. The source is copied into the destination, with source pixels replacing destination pixels.
$8000	notCOPY	The inverse of the source is copied into the destination, with the pixels being drawn replacing the destination pixels.
$0001	OR	Source pixels are overlayed nondestructively on top of destination pixels.
$8001	notOR	The inverse of the source pixels are overlayed nondestructively on top of the destination pixels.
$0002	XOR	Source pixels are exclusive-ORed (XOR) with destination pixels. If an image is drawn in XOR mode, the original appearance of the destination can be restored by drawing the image again in XOR mode.
$8002	notXOR	Source pixels are reversed, then exclusive-ORed with destination pixels.
$0003	BIC	Bit clear (BIC) pen with destination. This mode explicitly clears the pixels in the destination image before another image is copied in.
$8003	notBIC	Clears the pixels in a destination image, then copies the inverse of the source image pixels into the destination image.

GrafPort's bounds rectangle is smaller than the pixel map that the GrafPort is using, however, the coordinates of the GrafPort's bounds rectangle and the coordinates of its pixel map are not the same.

Port Rectangles

A *port rectangle,* or `PortRect`, outlines the section of a `BoundsRect` that is displayed on the super high-resolution screen. A port rectangle can be visualized as a window through which part of a bounds rectangle is viewed. A port rectangle can be the size of the screen or smaller. A good example of a `PortRect` is a window created and displayed by the Window Manager.

Regardless of the size of a port rectangle, the only part of a drawing that is displayed on the screen is the part that falls inside both the bounds rectangle and the port rectangle of the current GrafPort.

A newly created GrafPort has its pixel map initialized to include the entire screen. Its `BoundRect` and `PortRect` fields are set to rectangles enclosing the screen. Thus, coordinate 0,0 of the GrafPort's bounds rectangle and port rectangle corresponds to the top left corner of the screen. But this situation can be changed—and often is changed—by application programs.

Clip Regions

Two other attributes of a GrafPort are its clip region and its visible region. A *clip region,* or `ClipRgn`, is a structure that clips, or trims, pictures or drawings to a specified size. For a drawn object to be visible on the screen, it must be situated inside its GrafPort's clip region, as well as inside its GrafPort's bounds rectangle and port rectangle.

A clip region can be rectangular, or it can be drawn in any shape—even an irregular shape. Because of this feature, a clip region can create screens that are quite fancy. For example, if a GrafPort has a circle-shaped clip region, pictures displayed on the screen can be trimmed, or clipped, into round pictures.

A GrafPort's clipping region is defined with the `SetClip` and `ClipRect` calls. The `GetClip` and `SetClip` calls save a GrafPort's `ClipRgn` while other clipping functions are performed, for example, when you want to reset a `ClipRgn` so you can redraw a newly displayed portion of a document that's been scrolled.

Visible Regions

A *visible region,* or `VisRgn`, is the part of a port rectangle visible on the screen at a given time. A `VisRgn`, like a `ClipRgn`, can be rectangular but it doesn't have to be. When one window on a screen overlaps another, the Window Manager uses a `VisRgn` structure to determine which part of the partially hidden window should be displayed on the screen. Application programs can use visible regions for similar purposes. QuickDraw II contains a number of calls for manipulating visible regions.

QuickDraw Coordinates

When you define an object within QuickDraw's conceptual drawing plane, or draw an object on a pixel map, you must use coordinates to tell QuickDraw where to place the object. That can be a problem because QuickDraw uses two kinds of coordinate systems: a global coordinate system and a local coordinate system.

When a pixel map is stored in the IIGS's memory, its position within the conceptual drawing space is defined by a set of global coordinates. In the global coordinate system, coordinate 0,0 pinpoints where the upper left corner of a pixel map lies within the conceptual drawing plane.

In addition to QuickDraw's global coordinate system, each GrafPort created under QuickDraw has its own local coordinate system. In a GrafPort's local coordinate system, coordinate 0,0 defines the upper left coordinate of the GrafPort's bounds rectangle.

Coordinate Conversion

As mentioned, a newly created GrafPort has its pixel map set to point to the entire screen, and its bounds rectangle and port rectangle are both set to rectangles enclosing the screen. So, when a GrafPort is initialized, coordinate

0,0 corresponds to the screen's top left corner and also to the top left corners of its bounds rectangle and port rectangle.

But, as noted, a GrafPort does not have to use the screen as its pixel map, and its pixel map does not have to be the same size as its bounds rectangle. If a GrafPort's pixel image is larger or smaller than its bounds rectangle, its local and global coordinate systems are not the same.

Sometimes a IIGS program needs to convert coordinates from one system to another—from global to local and vice versa. One reason this is necessary is that some tools in the Toolbox use global coordinates for their operations, and others use local coordinates. For example, when the Event Manager reports an event, it gives the mouse location in global coordinates. But when you call the Control Manager to find out if the user clicked in a control in one of your windows, you must pass the mouse location in local coordinates.

Another reason coordinate conversion is necessary is that sometimes—for example, when windows are used—one coordinate system calculates coordinates on the screen, while another system calculates coordinates in individual windows. You'll see how and why this is done in chapter 10, which deals with the Window Manager.

Fortunately, there is an easy way to convert global coordinates to local coordinates and vice versa. The QuickDraw call `GlobalToLocal` converts any point expressed in global coordinates to a corresponding location expressed in local coordinates. Another QuickDraw call, `LocalToGlobal`, does the same job in reverse.

One call often used with onscreen rectangles is `SetOrigin`. The `SetOrigin` call allows a program to change the coordinates of a GrafPort's port rectangle so that its coordinates correspond to those of the GrafPort's bounds rectangle. When you use the `SetOrigin` call, the bounds and port rectangles remain the same size and in the same location relative to each other, but the upper left corner, or origin of the `PortRect`, is set to the point passed by `SetOrigin`. Details on the `SetOrigin` call are in the *Apple IIGS Toolbox Reference*.

If an application performs scrolling operations, it can use the `ScrollRect` call to shift the pixels of the image and then use `SetOrigin` to readjust the coordinate system after the shift. Details about the `ScrollRect` call are also in the *Apple IIGS Toolbox Reference*.

Strings and Text

QuickDraw recognizes three kinds of string and text structures:

- C-type strings. A C-type string ends with a null word (h'00') and is not preceded by a length byte.
- Pascal-type strings. A Pascal-type string is preceded by a length byte and does not have to end with a null word.
- Text structures. You can define a QuickDraw text structure with the `DrawText` call. When you make a `DrawText` call, you must pass

QuickDraw an integer that defines the number of bytes you want to write. A QuickDraw text structure can therefore be up to 65,535 bytes long.

FontInfoRecord and FontGlobalsRecord Structures

Two other kinds of text-related structures used by QuickDraw are the `FontInfoRecord` structure and the `FontGlobalsRecord` structure. These structures are used primarily by the Font Manager, but they are also available for use in application programs. Listing 8–3 shows how the `FontInfoRecord` and `FontGlobalsRecord` structures are defined in an assembly language program. If you're intrested in further details about these and other font-related and text-related structures, look in the *Apple IIGS Toolbox Reference*.

Listing 8–3
FontInfoRecord and FontGlobalsRecord Structures

```
FontInfoRecord  anop
Ascent          ds 2
Descent         ds 2
WidMax          ds 2
Leading         ds 2

FontGlobalsRec  anop
FontID          ds 2
FStyle          dc i'TextStyle'
FSize           ds 2
FVersion        ds 2
FWidMax         ds 2
fbrExtent       ds 2
```

BufSizeRecord

QuickDraw recognizes other kinds of structures that have special uses and are not described in detail here. QuickDraw uses `BufSizeRecord` to define the sizes and characteristics of buffers in which text is stored. Listing 8–4 shows how the structure of a `BufSizeRecord` is included in an assembly language program. `BufSizeRecord` is described in more detail in chapter 16 of the *Apple IIGS Toolbox Reference*.

Listing 8–4
BufSizeRecord Structure

```
BufSizeRecord  anop
MaxWidth       ds 2
TextBufHeight  ds 2
TextBufRowWrds ds 2
FontWidth      ds 2
```

Cursor Records

The cursor on the super high-resolution screen is user-definable. The data structure to define a cursor is called, logically enough, a cursor record. Listing 8–5 shows a cursor record included in an assembly language program.

Listing 8–5
Cursor Record

```
Cursor          anop
CursorHeight    ds 2
CursorWidth     ds 2
CursorImage     ds 32
CursorMask      ds 32
HotSpotY        ds 2      ;where cursor points, y coord
HotSpotX        ds 2      ;where cursor points, x coord
```

PaintParams Structure

QuickDraw has one special-purpose structure, called the `PaintParams` structure, which is used in just one call: `PaintPixels`. (This call is described in chapter 16 of the *Apple IIGS Toolbox Reference*.) Listing 8–6 shows the structure in an assembly language program.

Listing 8–6
PaintParams Structure

```
PaintParams     anop
LocInfo1Ptr     ds 4
LocInfo2Ptr     ds 4
SrcRectPtr      ds 4
DestPtPtr       ds 4
ScreenMode      ds 2
MaskHandle      ds 4
```

Loading and Initializing QuickDraw

Before QuickDraw is started up, the following tool sets must be loaded and started up:

- Tool Locator (always loaded and active)
- Memory Manager
- Miscellaneous Tool Set

After these tool sets are loaded and initialized, you can initialize QuickDraw.

The PAINTBOX Program

Now that you know a little about how QuickDraw works, you're ready to type, assemble, and run a few programs that use QuickDraw.

The first program is called PAINTBOX. This program draws a rectangle on the IIGS super high-resolution screen. The assembly language version of the program is PAINTBOX.S1 (listing 8–7). The C version is PAINTBOX.C (listing 8–8). Both program listings are at the end of this chapter.

PAINTBOX.S1 Program

When the PAINTBOX.S1 program is executed, it first loads and initializes QuickDraw II and the other tool sets it depends upon. Before QuickDraw is initialized, the Memory Manager call `NewHandle` reserves the three direct pages QuickDraw needs, plus one direct page required by the Event Manager. When `NewHandle` reserves the requested space, it returns with a handle to the space pushed onto the stack. The PAINTBOX.S1 program then pulls the handle off the stack, stores it in a variable called `DPHandle` (for direct page handle), and uses it to provide the necessary direct page space to QuickDraw and the Event Manager.

Next, in a program segment called `DrawRect`, the screen is cleared to white (color code $F) with the QuickDraw call `ClearScreen`. The call `PenNormal` is then used to set the pen color to black and the pen size to one pixel by one pixel.

When the pen state is set, the `SetRect` call defines a rectangle in QuickDraw's conceptual drawing space. The `PaintRect` call paints the rectangle on the screen.

After the rectangle is drawn, an event loop begins. This loop, like the one used in the EVENT.S1 program in chapter 7, keeps checking for a key down event or a mouse down event. As soon as it receives a notification of either kind of event, the program ends.

PAINTBOX.C Program

PAINTBOX.C is a C version of PAINTBOX.S1. It is designed to be used with the `#include` file INITQUIT.C, which appears in chapter 7.

From a program designer's point of view, PAINTBOX.C is almost identical to EVENT.C—although you'd never know it by just running the two programs! The only real difference is that PAINTBOX.C, instead of displaying a message on a text screen, goes into super high-resolution graphics and draws a black rectangle on a white screen.

PAINTBOX.C illustrates the advantage of writing programs split into short procedures and functions. To transform EVENT.C into PAINTBOX.C, you just replace the `PrintMessage` function with one that draws a rectangle on a super high-resolution screen.

The SKETCHER Program

The next program we'll look at, SKETCHER, is a little more complicated. With this program, you can use the IIGS mouse to draw sketches on a super high-resolution screen.

The assembly language version of the program is called SKETCHER.S1 (listing 8–9). The C version is SKETCHER.C (listing 8–10). Both listings appear at the end of this chapter.

SKETCHER.S1 Program

SKETCHER.S1, like PAINTBOX.S1, starts off by loading and initializing QuickDraw and clearing the screen to white. But then it gets considerably fancier. It uses the `ShowCursor` call to display the arrow-shaped cursor on the screen. Then it goes into an event loop that allows the user to draw sketches on the screen with the IIGS mouse. When the mouse moves, the cursor follows it. When the mouse button is pressed, the cursor starts drawing a line.

As long as the mouse button remains pressed, SKETCHER.S1 draws on the screen. When the mouse button is released, the program stops drawing, but the cursor still follows the movements of the mouse. The event loop in SKETCHER.S1 also looks for key down events. When it detects one, the program ends.

SKETCHER.C Program

SKETCHER.C is a C language version of the SKETCHER.S1 program. It is designed to be used with the `#include` file INITQUIT.C, which is listed in chapter 7.

SKETCHER.C is the first C language program you have encountered so far that has really justified the use of an event loop. It is the first one in which two or more different types of events require different responses. SKETCHER.C does more than just set a `done` flag to a value returned by a `GetNextEvent` call. It requires `done` to be true only when a key down event is detected. Mouse down events send the program to `Sketch`, a routine that sketches on the screen.

SKETCHER is the most ambitious program you have typed and run so far. You should be able to have some fun with it—particularly if you experiment with different pen colors, pen sizes, pen patterns, pen masks, background colors, and background patterns. You might want to add more event loop functions, such as a screen clearing function that doesn't end the program and a function that erases lines. You'll modify the SKETCHER program in some of these ways—and in other ways we haven't discussed yet—in later chapters.

PAINTBOX.S1 and PAINTBOX.C Listings

Listing 8–7
PAINTBOX.S1 program

```
*
* PAINTBOX.S1
*

*** A FEW ASSEMBLER DIRECTIVES ***

                Title 'PaintBox'
```

```
                ABSADDR on
                LIST off
                SYMBOL off
                65816 on
                mcopy paintbox.macros

                KEEP PaintBox

*
*  EXECUTABLE CODE STARTS HERE
*

Begin           START
                Using QuitData

                jmp MainProgram          ; skip over data

                END

*
*  SOME DIRECT PAGE ADDRESSES AND A FEW EQUATES
*

DPData          START

DPPointer       gequ    $10
DPHandle        gequ    DPPointer+4

ScreenMode      gequ $00                 ; 320 mode
MaxX            gequ 320                 ; X clamp high

                END

*
*  MAIN PROGRAM LOOP
*

MainProgram     START

                phk

                plb
                tdc                      ; get current direct page
                sta MyDP                 ; and save it for the moment

                jsr ToolInit             ; start up all tools we'll need
                jsr DrawRect             ; paint rectangle on screen
                jsr EventLoop            ; check for key & mouse events
```

```
*** WHEN EVENT LOOP ENDS, WE'LL SHUT DOWN ***

                jsr Shutdown
                jmp Endit

MyDP            ds  2

                END

*
*  THE ROUTINE THAT ENDS THE PROGRAM
*

EndIt           START

                Using QuitData

                _Quit QuitParams

*** THIS ERROR SHOULD NEVER OCCURR ***

                ErrorDeath 'We have returned from a quit call!!!'

                END

*
*  THIS IS WHERE WE INITIALIZE OUR TOOLS
*

ToolInit        START
                using MMData

*** START UP TOOL LOCATOR ***

                _TLStartup                  ; Tool Locator

*** INITIALIZE MEMORY MANAGER ***

                PushWord #0
                _MMStartup
                ErrorDeath 'Could not init Memory Manager.'
                pla
                sta MyID

*** INITIALIZE MISC. TOOLS SET ***

                _MTStartup
                ErrorDeath 'Could not init Misc Tools.'
```

```
*** GET SOME DIRECT PAGE MEMORY FOR TOOLS THAT NEED IT ***

                PushLong #0                     ; space for handle
                PushLong #$400                  ; four pages
                PushWord MyID
                PushWord #$C001                 ; locked, fixed, fixed bank
                PushLong #0
                _NewHandle

                ErrorDeath 'Could not get direct page.'

                pla
                sta DPHandle
                pla
                sta DPHandle+2

                lda [DPHandle]
                sta DPPointer

*** INITIALIZE QUICKDRAW II ***

                lda DPPointer                   ; pointer to direct page
                pha
                PushWord #ScreenMode            ; $00 for 320, $80 for 640 mode
                PushWord #160                   ; max size of scan line
                PushWord MyID
                _QDStartup
                ErrorDeath 'Could not start QuickDraw.'

*** INITIALIZE EVENT MANAGER ***

                lda DPPointer                   ; pointer to direct page
                clc
                adc #$300                       ; QD direct page + #$300
                pha                             ; (QD needs 3 pages)
                PushWord #20                    ; queue size
                PushWord #0                     ; Xclamp low
                PushWord #MaxX                  ; clamp high
                PushWord #0                     ; Y clamp low
                PushWord #200                   ; Y clamp high
                PushWord MyID
                _EMStartup
                ErrorDeath 'Could not start Event Manager.'

                rts

                END
```

```
*
*  SHUT DOWN ALL THE TOOLS WE STARTED UP
*

ShutDown        START
                Using MMData

                _EMShutDown
                _QDShutDown
                _MTShutDown

                PushLong DPHandle
                _DisposeHandle

                PushWord MyID
                _MMShutDown
                _TLShutDown

                rts

                END

*
*  EVENT LOOP
*

EventLoop       START
                Using QuitData
                Using EventTable
                Using EventData

Again           PushWord #0                 ; space for result
                PushWord #$000A             ; key down & mouse down events
                PushLong #EventRecord
                _GetNextEvent
                pla
                beq Again
                lda EventWhat               ; get event code
                asl a                       ; code * 2 = table location
                tax                         ; X is index register
                jsr (EventTable,x)          ; look up event's routine
                lda QuitFlag
                beq again

                rts

                END
```

```
*
*  ROUTINE THAT DRAWS A RECTANGLE
*

DrawRect        START

*** CLEAR SCREEN AND SET PEN STATE ***

                lda #$FFFF              ; color code for white,
*                                       ; typed four times (once
*                                       ; for each byte)

                pha                     ; push color code on the stack
                _ClearScreen            ; does what it says

                _PenNormal              ; make pen black & normal size

*** SET UP A RECTANGLE ***

                PushLong #RectPtr       ; pointer to a rectangle
                PushWord #$30           ; upper x coordinate
                PushWord #$30           ; upper y coordinate
                PushWord #$110          ; lower x coordinate
                PushWord #$98           ; lower y coordinate
                _SetRect                ; create a rectangle

*** PAINT RECTANGLE ON SCREEN ***

                PushLong #RectPtr       ; pointer to our rectangle
                _PaintRect              ; paint it on the screen

                rts

RectPtr         ds 8                    ; our rectangle

                END

*
*  ROUTINE THAT SETS THE QUIT FLAG
*

doQuit          START
                Using QuitData

                lda #$8000
```

```
                sta QuitFlag
                rts

                END

*
*  A USEFUL AND CONVENIENT WAY NOT TO DO ANYTHING
*

Ignore          START

                rts

                END

*
*  DATA SEGMENTS
*

EventTable      DATA

                dc i'ignore'            ; 0 null
                dc i'doQuit'            ; 1 mouse down
                dc i'ignore'            ; 2 mouse up
                dc i'doQuit'            ; 3 key down
                dc i'ignore'            ; 4 undefined
                dc i'ignore'            ; 5 auto-key down
                dc i'ignore'            ; 6 update event
                dc i'ignore'            ; 7 undefined
                dc i'ignore'            ; 8 activate
                dc i'ignore'            ; 9 switch
                dc i'ignore'            ; 10 desk acc
                dc i'ignore'            ; 11 device driver
                dc i'ignore'            ; 12 application
                dc i'ignore'            ; 13 application
                dc i'ignore'            ; 14 application
                dc i'ignore'            ; 15 application
                dc i'ignore'            ; 0 in desk

                END

***

EventData       DATA

EventRecord     anop                    ; table for Event Manager
EventWhat       ds 2
EventMessage    ds 4
```

```
EventWhen       ds 4
EventWhere      ds 4
EventModifiers ds 2

                END

***

QuitData        DATA

QuitFlag        ds 2

QuitParams      dc  i4'0'
                dc  i4'0'
                dc  i4'0'

                END

***

MMData          DATA

MyID            dc  i'0'                    ; program ID word

                END
```

Listing 8–8
PAINTBOX.C program

```
#include "initquit.c"

#define SIMPLE_MASK (mDownMask + keyDownMask)

EventRecord myEvent;
Boolean done = false;

main()
{
StartTools();
DrawRect();
EventLoop();
ShutDown();
}

DrawRect()  /* send message to stdout, then  switch display */
{
```

```
Rect myRect;

   ClearScreen(0xFFFF);
   PenNormal();
   SetRect(&myRect,0x30,0x30,0x110,0x98);
   PaintRect(&myRect) ;
}

EventLoop()
{
   while(!done)
      done = GetNextEvent(SIMPLE_MASK,&myEvent);
}
```

SKETCHER.S1 and SKETCHER.C Listings

Listing 8–9
SKETCHER.S1 program

```
*
*  SKETCHER.S1
*

*** A FEW ASSEMBLER DIRECTIVES ***

                Title 'Sketcher'

                ABSADDR on
                LIST off
                SYMBOL off
                65816 on
                mcopy sketcher.macros

                KEEP Sketcher

*
*  EXECUTABLE CODE STARTS HERE
*

Begin           START
                Using QuitData

                jmp MainProgram          ; skip over data

                END
```

```
*
*  SOME DIRECT PAGE ADDRESSES AND A FEW EQUATES
*

DPData          START

DPPointer       gequ    $10
DPHandle        gequ    DPPointer+4

ScreenMode      gequ $00                ; 320 mode
MaxX            gequ 320                ; X clamp high

                END

*
*  MAIN PROGRAM LOOP
*

MainProgram     START

                phk
                plb
                tdc                     ; get current direct page
                sta MyDP                ; and save it for the moment

                jsr ToolInit            ; start up all tools we'll need

*** CLEAR SCREEN AND SET PEN STATE ***

                lda #$FFFF              ; color code for white
                pha                     ; push it on the stack
                _ClearScreen            ; does what it says

                _PenNormal              ; make pen black & normal size
                _ShowCursor

                jsr EventLoop           ; check for key & mouse events

*** WHEN EVENT LOOP ENDS, WE'LL SHUT DOWN ***

                jsr Shutdown
                jmp Endit

MyDP            ds  2

                END
```

```
*
*  EVENT LOOP
*

EventLoop       START
                Using QuitData
                Using EventTable
                Using EventData

Again           PushWord #0                 ; space for result
                PushWord #$000F             ; key & mouse events
                PushLong #EventRecord
                _GetNextEvent
                pla
                beq Again
                lda EventWhat               ; get event code
                asl a                       ; code * 2 = table location
                tax                         ; X is index register
                jsr (EventTable,x)          ; look up event's routine
                lda QuitFlag
                beq again

                rts

                END

*
* ROUTINE TO DRAW SKETCHES ON THE SCREEN
*

MoveIt          START
                Using EventData

                _ShowPen

                lda EventWhere
                sta MouseHouse
                lda EventWhere+2
                sta MouseHouse+2

                PushLong MouseHouse
                _MoveTo

loop            pea 0                       ; space for return
                pea 0                       ; check button zero
                _StillDown
                pla
                beq out
```

```
                PushLong #MouseHouse
                _GetMouse
                PushLong MouseHouse
                _LineTo

                bra loop

out             _HidePen
                rts

MouseHouse      ds 4

                END

*
*   THE ROUTINE THAT ENDS THE PROGRAM
*

EndIt           START

                Using QuitData

                _Quit QuitParams

*** IF THIS COMES BACK, WE'RE DEAD ***

                ErrorDeath 'We just came back from a quit call!!!'

                END

*
*  THIS IS WHERE WE INITIALIZE OUR TOOLS
*

ToolInit        START
                using MMData

*** START UP TOOL LOCATOR ***

                _TLStartup                ; Tool Locator

*** INITIALIZE MEMORY MANAGER ***

                PushWord #0
                _MMStartup
                ErrorDeath 'Could not init Memory Manager.'
                pla
                sta MyID
```

```
*** INITIALIZE MISC. TOOLS SET ***

                _MTStartup
                ErrorDeath 'Could not init Misc Tools.'

*** GET SOME DIRECT PAGE MEMORY FOR TOOLS THAT NEED IT ***

                PushLong #0                ; space for handle
                PushLong #$800             ; eight pages
                PushWord MyID
                PushWord #$C001            ; locked, fixed, fixed bank
                PushLong #0
                _NewHandle

                ErrorDeath 'Could not get direct page.'

                pla
                sta DPHandle
                pla
                sta DPHandle+2

                lda [DPHandle]
                sta DPPointer

*** INITIALIZE QUICKDRAW II ***

                lda DPPointer              ; pointer to direct page
                pha
                PushWord #ScreenMode       ; either 320 or 640 mode
                PushWord #160              ; max size of scan line
                PushWord MyID
                _QDStartup
                ErrorDeath 'Could not start QuickDraw.'

*** INITIALIZE EVENT MANAGER ***

                lda DPPointer              ; pointer to direct page
                clc
                adc #$300                  ; QD direct page + #$300
                pha                        ; (QD needs 3 pages)
                PushWord #20               ; queue size
                PushWord #0                ; X clamp low
                PushWord #MaxX             ; X clamp high
                PushWord #0                ; Y clamp low
                PushWord #200              ; Y clamp high
                PushWord MyID
                _EMStartup
                ErrorDeath 'Could not start Event Manager.'
```

```
                rts

                END

*
*  SHUT DOWN ALL THE TOOLS WE STARTED UP
*

ShutDown        START
                Using MMData

                _EMShutDown
                _QDShutDown
                _MTShutDown

                PushLong DPHandle
                _DisposeHandle

                PushWord MyID
                _MMShutDown
                _TLShutDown

                rts

                END

*
*  ROUTINE THAT SETS THE QUIT FLAG
*

doQuit          START
                Using QuitData

                lda #$8000
                sta QuitFlag
                rts

                END

*
*  A USEFUL AND CONVENIENT WAY NOT TO DO ANYTHING
*

Ignore          START

                rts

                END
```

```
*
*  DATA SEGMENTS
*

EventTable    DATA

              dc i'ignore'          ; 0 null
              dc i'MoveIt'          ; 1 mouse down
              dc i'ignore'          ; 2 mouse up
              dc i'doQuit'          ; 3 key down
              dc i'ignore'          ; 4 undefined
              dc i'ignore'          ; 5 auto-key down
              dc i'ignore'          ; 6 update event
              dc i'ignore'          ; 7 undefined
              dc i'ignore'          ; 8 activate
              dc i'ignore'          ; 9 switch
              dc i'ignore'          ; 10 desk acc
              dc i'ignore'          ; 11 device driver
              dc i'ignore'          ; 12 application
              dc i'ignore'          ; 13 application
              dc i'ignore'          ; 14 application
              dc i'ignore'          ; 15 application
              dc i'ignore'          ; 0 in desk

              END

***

EventData     DATA

EventRecord   anop                  ; table for Event  Manager
EventWhat     ds 2
EventMessage  ds 4
EventWhen     ds 4
EventWhere    ds 4
EventModifiers ds 2

              END

***

QuitData      DATA

QuitFlag      ds 2
```

```
QuitParams      dc  i4'0'
                dc  i4'0'
                dc  i4'0'

                END

***

MMData          DATA

MyID            dc  i'0'                ; program ID word

                END
```

Listing 8–10
SKETCHER.C program

```
#include "initquit.c"

#define MY_MASK (mDownMask + mUpMask + keyDownMask)

EventRecord myEvent;
Boolean done = false;

main()
{
StartTools();
GrafPrep();
EventLoop();
ShutDown();
}

GrafPrep()
{
   ClearScreen(0xFFFF);
   PenNormal();
   ShowCursor();
}

EventLoop()
{
   while(!done)
      if ( GetNextEvent(MY_MASK,&myEvent) )
         switch (myEvent.what) {
```

```
                case mouseDownEvt:
                  MoveIt();
                  break;
                case keyDownEvt:
                  done = true;
            }
}

MoveIt()
{
Point MouseHouse;

   ShowPen();
   MoveTo(myEvent.where);

   while (StillDown(0)) {
      GetMouse(&MouseHouse);
      LineTo(MouseHouse);
   }

   HidePen();
}
```

CHAPTER 9

The Menu Manager

Creating Menus

One of the most important features of the IIGS is its ability to display pull-down menus—menus that allow the user to select almost any function or application at almost any time, without going through confusing levels of menus and without remembering command words or special keys. Pull-down menus were introduced with the unveiling of the Apple Macintosh—and the IIGS has windows almost identical to those that created such a sensation when they first appeared on the Mac.

Menus and the IIGS User

One reason why pull-down menus are so popular is that they are easy to use. To use a pull-down menu, you just place a cursor inside an onscreen bar called a menu bar, then click the button of the IIGS mouse over a menu title that also appears inside the menu bar. An application can then call the Menu Manager, which highlights the selected title by redrawing it in inverted colors.

When a menu title is selected, you can drag the cursor into a series of menu items that appear below the menu title. As long as the mouse button is held down, the selected menu title is highlighted, and the menu items below it are displayed. Dragging the mouse cursor up and down through the list of

menu items highlights each item or command while the cursor is positioned over it.

If the mouse button is released while an item is highlighted, the function or application that the item identifies is selected. The item blinks briefly to confirm the user's choice, and the menu disappears.

When you choose a menu item, the Menu Manager tells the application which item was chosen, and the application can then perform the appropriate action. When the application completes the action, it can remove the highlighting from the menu title, indicating that the operation is complete.

If you hold down the mouse button and move the cursor out of the menu, the menu remains visible, though none of its items are highlighted. If you release the mouse button outside the menu, no choice is made. The menu simply disappears, and the application does not take any action. Thus, you can always look at a menu without changing the document or the screen.

The IIGS can display menus in both 640-pixel mode and 320-pixel mode. Figure 9–1 is a 640-mode menu, and figure 9–2 is a 320-mode menu.

Menu Bars

Before we go into more detail about how the IIGS Menu Manager works, it is helpful to review some of the terminology used so far in this chapter.

A *menu bar* is a rectangle that usually appears across the top of the IIGS screen. Several *menu titles* are usually visible inside the bar. Some of these titles may be dimmed, indicating they are disabled. A disabled menu can still

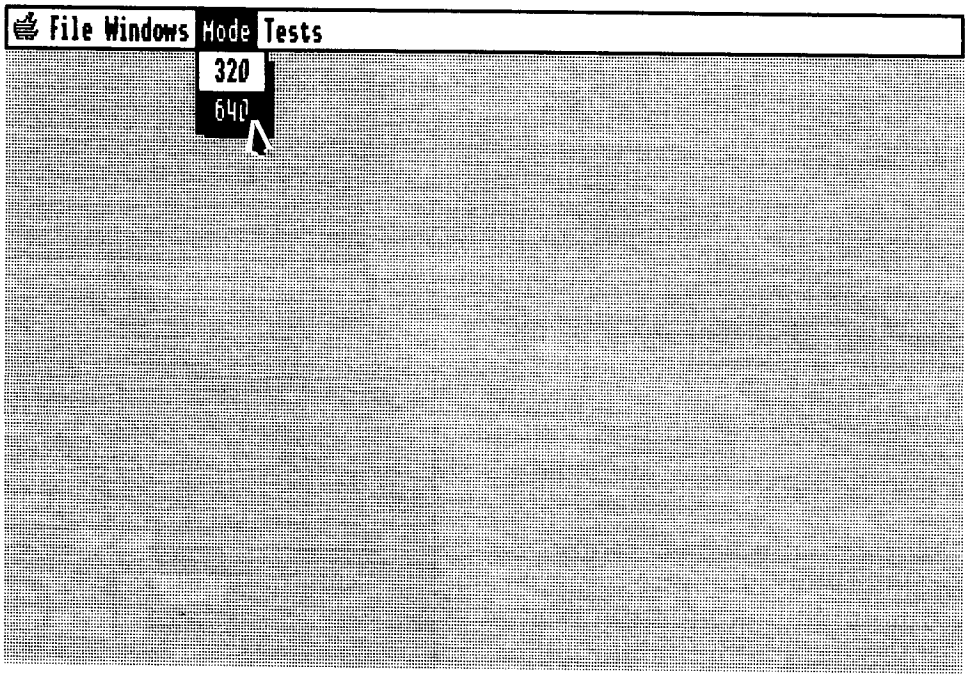


Figure 9–1
Menu in 640 mode

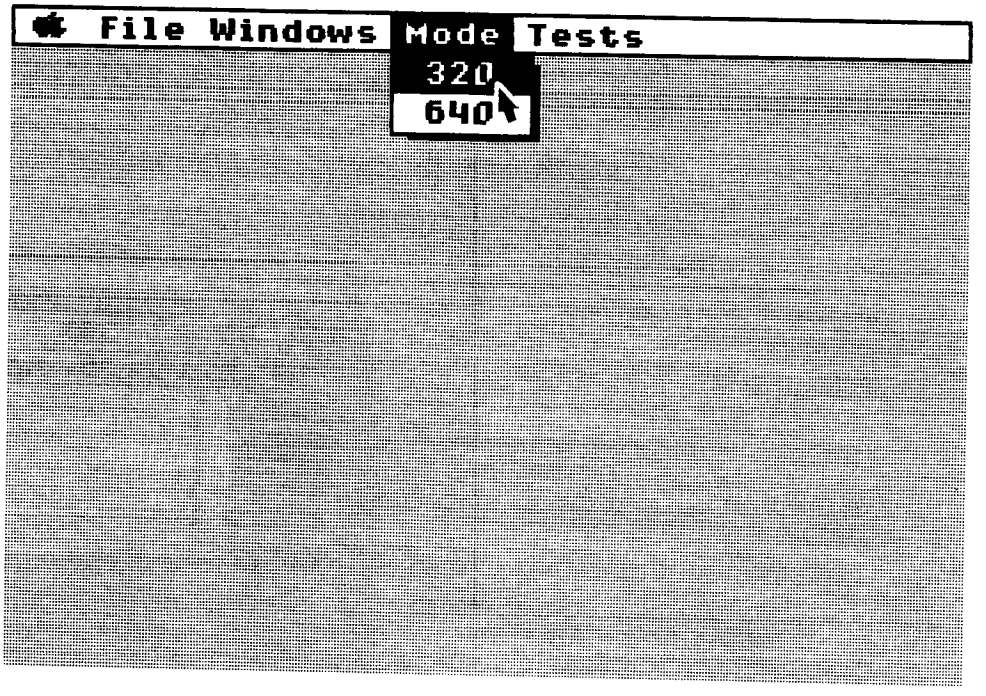


Figure 9–2
Menu in 320 mode

be pulled down, but all menu items under it will also be dimmed, and you usually cannot select them.

Underneath each menu title, an application can place the names of as many menu items as space allows. The items beneath a menu title, however, are not ordinarily visible unless you place the cursor over the menu title and pull the menu down.

A menu title and the items that appear beneath it make up a menu. Thus, several menus (as many as space allows) can appear inside a menu bar.

System Menu Bar

The Menu Manager has one special kind of menu bar called a system menu bar. Only one system menu bar can be on the screen at one time. The system menu bar is always positioned at the top of the screen, and only the cursor appears in front of it.

In applications that support desk accessories, the first menu on the menu bar—that is, the leftmost menu—should be a desk accessories menu. In programs written according to Apple's *Human Interface Guidelines,* the title of a desk accessories menu should always be a specially designed colored apple. In programs written for the Apple IIGS, a special Toolbox call, `FixAppleMenu`, sets up a desk accessories menu that has a colored apple as its title.

Desk accessories are special mini-applications that can be coresident in memory with other applications and thus can be executed at any time. A tutorial in writing desk accessory programs is beyond the scope of this book, but instructions for writing desk accessories are in the *Apple IIGS Toolbox Reference.*

Window Menu Bars

In addition to the system menu bar, an application can also use window menu bars. Because window menu bars can appear in individual windows, they can increase the number of menu titles visible on the screen. But they can also be confusing to the IIGS user, so they should be used in moderation, if at all.

More About Menus

A number of menu items make up a typical Apple IIGS menu. The items are listed vertically inside a shadowed rectangle, and each item may consist of the text of a command, an object or icon defined by an application, or just a line dividing groups of choices. Everything else on the screen, except the cursor, always appears behind menus.

Keyboard Equivalents for Menu Commands

An application program can set up a keyboard equivalent for any menu item so that you can issue a menu command from the keyboard, rather than the mouse. The character specified as a menu command equivalent is usually the first letter of a menu command. Typing the letter in either uppercase or lowercase is usually allowed. For example, typing either Q or q while holding down the Apple key can be used as an equivalent for a mouse selectable menu item titled Quit.

Initializing the Menu Manager

Before the Menu Manager is started, these tool sets must already be loaded and initialized:

- Tool Locator (always active)
- Memory Manager
- QuickDraw II
- Event Manager
- Window Manager
- Control Manager

The Menu Manager also requires one direct page. When one direct page is reserved, and the previous tool sets are started, the `MenuStartup` call initializes the Menu Manager. As soon as the Memory Manager is started, an empty menu bar appears at the top of the screen. The application that uses the menu bar must then finish drawing it by initializing a set of menus and printing their names in the bar.

Using the Menu Manager

An assembly language program titled MENU.S1 demonstrates how the Menu Manager is used in an assembly language program. There is also a C language version of the same program. (Both programs—listing 9–9 and listing 9–10—are at the end of this chapter.)

The MENU.S1 program prints a menu bar and a set of menus on the screen. Then it allows the user to place check marks in front of menu items by clicking the mouse. It also allows the user to quit the program by selecting a menu item titled Quit or by typing Q or q on the keyboard.

In the next few sections of this chapter, we divide the MENU.S1 program into parts and see how each part works. Then, at the end of the chapter, we put all the parts together and type and run the program.

Defining Menus and Items

The first step in creating a menu bar is to draw up a list of menus and menu items, and place the list in a data segment of a program. In the MENU.S1 program, menus and menu items are defined in the data segment titled `MenuData`.

Interpreting Menu Data

As the `MenuData` table shows, the MENU.S1 program has six menus, and there are several items under each menu title. In the data segment `MenuData`, the menus and menu items used in the program are listed in a special format required by the Menu Manager. For example, the menu titles in the listing are numbered consecutively beginning with 1, and the menu items in the listing are numbered consecutively beginning with 257. This numbering system is important because the Menu Manager uses it to distinguish between menu titles and menu items in a table of menu data. The number assigned to a menu title or a menu item is known as an ID number and is always preceded by the letter *N* in a table of menu data. Table 9–1 shows the ID numbers you can assign to menus and menu items and the uses for various ranges of ID numbers.

Special Characters in Menu Data Tables

In a menu data table, the title of each menu is preceded by the > symbol. The last item in each menu is followed by a line containing only a period. A number of other special characters also appear in the listing.

For example, the *L* that precedes the title of each menu and each menu item is merely a space filler required by the Menu Manager. If the > symbol appears in front of the L, the text string that follows the L is the title of a menu. If a space precedes the L, the string that follows the L is the title of a menu item.

Actually, L, >, the space character, and the period do not have to be used in the MENU.S1 program. You can substitute other characters as long as they are used consistently.

Table 9–1
Menu and Menu Item ID Numbers

Hex Number	Decimal Number	Meaning
	Menu ID Numbers	
$0000	0	For internal use. Usually used for the front (first) menu in a menu bar.
$0001–$FFFE	1–65534	Reserved for application use.
$FFFF	65535	For internal use. Usually used for the last item in a menu bar.
	Menu Item ID Numbers	
$0000	0	For internal use. Usually used for the front (first) item in a menu.
$0001–$00F9	1–249	Reserved for desk accessory items.
$00FA	250	Reserved for Undo edit item.
$00FB	251	Reserved for Cut edit item.
$00FC	252	Reserved for Copy edit item.
$00FD	253	Reserved for Paste edit item.
$00FE	254	Reserved for Clear edit item.
$00FF	255	Reserved for Close command item.
$0100–$FFFE	256–65534	Reserved for application use.
$FFFF	65535	For internal use. Usually used for the last item in a menu.

A number of reserved characters, however, always have the same meaning in tables of menu data. For example:

- The @ character, preceded by the symbols used for a symbol title and followed immediately by a backslash (\) always represents the colored Apple logo that usually appears as the leftmost element on a menu bar. This symbol appears in the line labeled `Menu1` in the `MenuData` table.
- The backslash character (\) always marks the end of a string of text and the beginning of a series of special characters.
- The letter *N*, as noted, is a prefix for each ID number in a table of menu data.
- The * symbol is a prefix for letters that can be used as keyboard equivalents for menu selections. Usually this symbol is followed by two letters: an uppercase letter and its corresponding lowercase letter. When the prefix is used in this way, it means the keyboard equivalent for the menu choice is not case sensitive. This prefix is used in the second line following the label `Menu2` in the `MenuData` table.
- The ASCII character 13, a carriage return, is an end-of-line symbol in tables of menu data. A null character (00) has the same meaning.

All of the characters that have special meanings in menu data tables are

listed in table 9–2. These characters can appear in any order following the backslash character that separates the text on each line from the special characters that follow it.

All of the characters in table 9–2 except the backslash character can be used in names of menu items, but the characters *, B, C, I, U, and V cannot be used in menu titles. There is no way to include a backslash character (\) in a text string because the Menu Manager always treats it as the beginning of a series of special characters.

Building a Menu

After a table of menu data is created and entered in a source code program, the Menu Manager calls `NewMenu` and `InsertMenu` can be used to build a menu. This is the syntax for issuing these two calls:

```
PushLong #0                ; space for return
PushLong #Menu6            ; ID number of menu
_NewMenu
PushWord #0                ; make this menu
_InsertMenu                ; the front menu
```

The `NewMenu` call takes two long parameters: a 0 to leave 2 words on the stack and a menu ID number. It returns one long parameter—a menu

Table 9–2
Special Characters in Table of Menu Data

Character	Meaning
\	Marks the end of a text string and the beginning of a series of special characters.
*	Prefix for a character (or characters) that can be used as a keyboard equivalent for a menu choice. This prefix is usually followed by an uppercase letter and a corresponding lowercase letter, indicating that the keyboard equivalent is not case sensitive.
B	Print the text of the preceding line in boldface.
C	Prefix for a character that can be printed in front of a menu item to mark it. The character is identified by its ASCII code. For example, C18 means use a check mark (ASCII code 18) to mark the preceding item.
D	Dim (disable) the preceding item.
H	A hexadecimal, non-ASCII ID number follows, in low-byte/high-byte order.
I	Italicize the text of the preceding item.
N	Prefix for the ID number of a menu title or a menu item.
U	Underscore the text of the preceding item.
V	Place an underline under the preceding item without requiring a separate item.
X	Color replacement, rather than an XOR operation, will be used for highlighting. This symbol is usually used with the colored Apple logo on a menu bar.

handle—which is left on the stack in the previous example. For the reason why, read on.

The `InsertMenu` call takes two parameters: a handle to a menu and the 1-word ID number after which the menu in question will be inserted. In the previous example, only the second parameter is passed because the first parameter—the menu handle just pushed onto the stack—is still there. If a 0 is passed as the second parameter, as it is in this example, the menu being inserted is placed in front of any other menus in the menu bar.

It's easy to use a 0 parameter to place an inserted menu on top of all the rest. So menus are usually built backwards, in back-to-front order, as you will see in the menu building segment of the MENU.S1 program.

After you build a menu, you can draw it with the `FixAppleMenu`, `FixMenuBar`, and `DrawMenuBar` calls.

Activating a Menu

After a menu is built, the next step in making it useful in a program is to write a routine that accepts input from the user. You can use an Event Manager loop, but it is much easier to use a tool called TaskMaster, which considerably expands the capabilities of the Event Manager call `GetNextEvent`.

Using TaskMaster

TaskMaster is a tool in the Window Manager tool set, but it also has capabilities designed to be used with the Menu Manager. When a program includes menus, windows, or both, it can call TaskMaster instead of making the Event Manager call `GetNextEvent`. When TaskMaster is called in a program, the first thing it does is call `GetNextEvent`. Then it checks for twelve events that `GetNextEvent` cannot handle, and it handles those events. Then it places some information on the stack and in a record called a *task record*. Finally, it returns to the calling program.

The following is a call to TaskMaster in an assembly language program:

```
PushWord #0          ; space for result
PushWord EventMask   ; standard GetNextEvent mask
PushLong TaskRecPtr ; pointer to a task record
_TaskMaster
PullWord TaskCode    ; a code returned by TaskMaster
```

As the example illustrates, a call to TaskMaster takes three parameters:

- A null word (a 0) to save space on the stack for the result of the call.
- An event mask. This 1-word parameter is the same as the `EventMask` parameter, which must be passed to the Event Manager call `GetNextEvent`.
- A pointer to a record called a task record. A task record, as you shall see, is just like an event record used by the Event Manager call `GetNextEvent`, except it has two extra fields.

Before a TaskMaster call returns, it places a word called a *task code* on the stack. If TaskMaster detects an event, the task code tells where on the desktop (that is, in what part of the screen) the event took place. The values returned as a task code can vary, depending upon what kind of item is detected by TaskMaster. For example, if TaskMaster detects any event that is not a key down or button down event, the task code that it returns is the same as the event code returned by the Event Manager. If TaskMaster detects a key down or button down event, however, the values that can be returned as a task code are the same as those returned by the Window Manager call FindWindow. These values, and their meanings, are listed in table 9–3.

How TaskMaster Works

One of the best ways to use TaskMaster is to set up a table including all tasks it can handle. One such table, labeled TaskTable, appears in the MENU.S1 program. The first seventeen items in the table are identical to the items in the event table used to make the GetNextEvent call in chapter 7. But at the end of the table there are twelve extra items: the events that TaskMaster looks for after it has called GetNextEvent.

When you call TaskMaster in a program, TaskMaster first makes the Event Manager call GetNextEvent. GetNextEvent handles all the events it can, then passes control back to TaskMaster.

Now TaskMaster goes to its expanded list of events and looks for events that GetNextEvent cannot handle. Specifically, TaskMaster looks to see if the mouse button has been clicked in

- the menu bar
- the system window (not an application window)
- the content region of any window
- the drag (title bar) region of any window

Table 9–3
Task Codes Returned by TaskMaster

Word	Code Name	Where Event Took Place
$0000	wNoHit	Not in a window or a menu
$0010	wInDesk	On the desktop
$0011	wInMenuBar	In the system menu bar
$0013	wInContent	In a window's content region
$0014	wInDrag	In a window's drag region
$0015	wInGrow	In a window's grow box
$0016	wInGoAway	In a window's close box
$0017	wInZoom	In a window's zoom box
$0018	wInInfo	In a window's information bar
$0019	wInSpecial	In a special menu item bar
$001A	wInDeskItem	Desk accessory selected from Apple menu
$001B	wInFrame	In a window frame area
$8XXX	wInSysWindow	In a system window

- the grow box of a window
- a window's go-away box
- a window's zoom box
- a window's information bar
- a window's vertical scroll bar
- a window's horizontal scroll bar
- a window's frame
- a menu's drop region

As you can see, most of the events TaskMaster looks for involve windows. We won't go into detail about window events now; they are covered in chapter 10.

In addition to looking for window-related events, TaskMaster can detect when the mouse button is clicked over a menu title or over a menu item—that is, in a menu's "drop region." These two capabilities make TaskMaster a valuable tool in programs that use the Menu Manager.

Event Records

When TaskMaster calls `GetNextEvent`, the `GetNextEvent` routine returns information in the usual way: by placing it in an event record. But the event record TaskMaster uses, like the event table, is slightly expanded. An event record in a program that uses TaskMaster has to be two fields longer than an ordinary event record. Listing 9–1 shows an event record used by TaskMaster in an assembly language program.

Listing 9–1
An event record used by TaskMaster

```
EventData       DATA

EventRecord     anop
EventWhat       ds 2
EventMessage    ds 4
EventWhen       ds 4
EventWhere      ds 4
EventModifiers  ds 2
TaskData        ds 4
TaskMask        dc i4'$0FFF'
```

The two extra fields used by TaskMaster are at the end of the event record in listing 9–1. In one of the extra fields, `TaskData`, TaskMaster returns information, in the same way that `GetNextEvent` returns data in the event record fields for which it is responsible.

The other extra field, `TaskMask`, can be used to tell TaskMaster what kinds of events to look for and what kinds of events to ignore. The `TaskMask` field is used much like the event mask passed to the `GetNextEvent` call as a parameter.

It is important to understand, however, that the event mask passed to TaskMaster as a parameter is different from the `TaskMask` passed to TaskMaster as part of a task record. The event mask passed to TaskMaster is the same kind of mask passed to the Event Manager in the `GetNextEvent` call. Table 9–4 shows the layout of an event mask.

The value TaskMaster returns in the `TaskData` field can vary, depending upon the kind of event TaskMaster has detected. For example, if TaskMaster detects a key down event, it makes the Menu Manager call `MenuKey` to determine if the key pressed is the keyboard equivalent of a mouse-controlled menu selection. If the key is a menu-related key, TaskMaster returns the ID number of the menu selected in the high word of the `TaskData` field and the ID number of the menu item selected in the low word. If the ID number ranges between 1 and 249 ($0000–$00F9), indicating a desk accessory item, TaskMaster makes the `OpenNDA` call to open a desk accessory. Then TaskMaster unhighlights the menu using the `HiliteMenu` call and returns a task code of 0.

If TaskMaster detects any other kind of key event, it returns a key down event: an ASCII character code (with the high bit clear) in the low-order byte of the `EventMessage` field and the upper 3 bytes of the field undefined.

If a button down event in a menu item is detected, TaskMaster returns with the menu's ID number in the high word of the `TaskData` field, the item's ID number in the low word of the `TaskData` field, and a task code of $0011 (`wInMenuBar`).

If TaskMaster detects a button down event in the menu bar but no menu item is selected, it returns a task code of 0. TaskMaster can also detect and handle a number of window-related events. These are covered in chapter 10.

Table 9–4
Bits in an Event Mask

Bit	Function
0	Not used
1	Mouse down mask
2	Mouse up mask
3	Key down mask
4	Auto-key mask
5	Update mask
6	Active mask
7	Switch mask
8	Desk accessory mask
9	Driver mask
10	Application 1
11	Application 2
12	Application 3
13	Not used
14	Not used
15	Not used

As mentioned, TaskMaster also returns a 1-word event code, which it pushes onto the stack. The task codes used by TaskMaster are listed in table 9–3.

Task Masks

A task mask is a 1-word parameter that must be passed to TaskMaster each time TaskMaster is called. An application uses a task mask to tell TaskMaster what events to look for or ignore.

In a task mask, bits 0 through 12 correspond to events TaskMaster can handle. Each bit corresponds to one type of event. If a bit is set, TaskMaster reports on the corresponding event. If a bit is clear, TaskMaster ignores the corresponding event. For TaskMaster to look for every type of event it can handle, the task mask should be $0000FFFF.

Bits 16 to 31 (the high word) in the task mask must always be clear. The bits in the task mask field and their functions are listed in table 9–5.

Table 9–5
Bits in the Task Mask Field

Bit	Function
0	Menu key
1	Update handling
2	Find window
3	Menu select
4	Open NDA
5	System click
6	Drag window
7	Select window if event is `wInContent`
8	Track go-away
9	Track zoom
10	Grow window
11	Scroll window
12	Handle special menu items
13	Not used
14	Not used
15	Not used
16–31	Must be clear

Accepting Input from the User

When you create a task table and an event record for TaskMaster, you can write a routine to accept input from the IIGS user. The main event loop of MENU.S1, `EventLoop` in listing 9–2, is one such routine.

The event loop in listing 9–2 is straightforward. It calls TaskMaster, pulls TaskMaster's event code off the stack, and then uses the code to jump to a subroutine listed in a jump table called `TaskTable`. This table is a standard event table of the type used by the Event Manager, with twelve additional events TaskMaster is designed to handle. The TaskMaster section of the event table used in MENU.S1 is in listing 9–3.

Listing 9–2
Event loop in MENU.S1

```
EventLoop       START
                Using QuitData
                Using TaskTable
                Using EventData

Again           PushWord #0                 ; space for result
                PushWord #$FFFF             ; recognize all events
                PushLong #EventRecord
                _TaskMaster
                pla
                asl a                       ; code * 2 = table location
                tax                         ; X is index register
                jsr (TaskTable,x)           ; look up event's routine
                lda QuitFlag
                beq again

                rts
                END
```

Listing 9–3
TaskMaster section of MENU.S1 event table

```
*
* TaskMaster Events
*

                dc i'DoMenu'            ; 1 in menu bar
                dc i'ignore'            ; 2 in system window
                dc i'ignore'            ; 3 in content of window (MoveIt)
                dc i'ignore'            ; 4 in drag
                dc i'ignore'            ; 5 in grow
                dc i'ignore'            ; 6 in go-away
                dc i'ignore'            ; 7 in zoom
                dc i'ignore'            ; 8 in info bar
                dc i'ignore'            ; 9 in ver scroll
                dc i'ignore'            ; 10 in hor scroll
                dc i'ignore'            ; 11 in frame
                dc i'ignore'            ; in drop

                END
```

As listing 9–3 shows, only the first item in the table—"in menu bar"—is activated. So each time TaskMaster loops through the table, it looks for only one kind of event: a button down event in the menu bar. If that event is detected, TaskMaster jumps to a subroutine labeled **DoMenu**, which appears in listing 9–4.

Listing 9–4
A routine that uses TaskMaster

```
*
* DoMenu
* Called when TaskMaster tells us a new menu item is selected.
*

DoMenu          START
                Using TaskTable
                Using EventData
                Using MenuTable

                Ida TaskData            ; get TaskData value
                cmp #256
                bcc GiveUp              ; this should never happen

                and #$00FF              ; mask off high byte
                asl a                   ; double the value
                tax                     ; for 2-byte addresses

                jsr (MenuTable,x)

GiveUp          anop
                PushWord #False         ; false=unhighlight
                PushWord TaskData+2     ; which menu?
                _HiliteMenu             ; unhighlight it

                rts

                END
```

The **DoMenu** routine is also straightforward. Each time it is called, it checks the **TaskData** field of the event record to see which item of which

menu (if any) the user selected. It then jumps to another table, labeled **MenuTable**, to determine what kind of action to perform. This table appears in listing 9–5.

Listing 9–5
MenuTable segment from MENU.S1

```
MenuTable       DATA

*               Menu 1 (apple)
                dc i'ignore'            ; one for the NDAs

                dc i'ignore'

*               Menu 2 (file)
                dc i'doQuit'            ; quit item selected

*               Menu 3 (appetizers)
                dc i'CheckIt'           ; 'salad'
                dc i'CheckIt'           ; 'jello'
                dc i'CheckIt'           ; 'slices'
                dc i'CheckIt'           ; 'juice'

*               Menu 4 (entrees)
                dc i'CheckIt'           ; 'duckling'
                dc i'CheckIt'           ; 'dumplings'

*               Menu 5 (beverages)
                dc i'CheckIt'           ; 'shake'
                dc i'CheckIt'           ; 'cola'
                dc i'CheckIt'           ; 'wine'

*               Menu 6 (desserts)
                dc i'CheckIt'           ; 'an apple'
                dc i'CheckIt'           ; 'pie'
                dc i'CheckIt'           ; 'turnover'

                END
```

The data segment labeled **MenuTable** is a jump table version of the table of menu data in listing 9–6. Both tables are in the MENU.S1 program at the end of this chapter. The table in listing 9–5 sends the MENU.S1 program to the subroutine the user selects. The table in listing 9–6 provides the Menu Manager with the information it needs to create a menu that works with the jump table in listing 9–5.

Listing 9–6
Data used to create a menu

```
MenuData        DATA

Return          equ 13

Menu1           dc c'>L@\XN1',i1'RETURN'
                dc c' LAn Apple Menu\N257',i1'RETURN'
                dc c'.'

Menu2           dc c'>L File  \N2',i1'RETURN'
                dc c' LQuit \N258*Qq',i1'RETURN'
                dc c'.'

Menu3           dc c'>L Appetizers   \N3',i1'RETURN'
                dc c' LApple Salad \N259',i1'RETURN'
                dc c' LApple Jello \N260',i1'RETURN'
                dc c' LApple Slices \N261',i1'RETURN'
                dc c' LApple Juice \N262',i1'RETURN'
                dc c'.'

Menu4           dc c'>L Entrees   \N4',i1'RETURN'
                dc c' LApple Duckling \N263',i1'RETURN'
                dc c' LApple Dumplings \N264',i1'RETURN'
                dc c'.'

Menu5           dc c'>L Beverages   \N5',i1'RETURN'
                dc c' LApple Shake \N265',i1'RETURN'
                dc c' LApple Cola \N266',i1'RETURN'
                dc c' LApple Wine \N267',i1'RETURN'
                dc c'.'

Menu6           dc c'>L Desserts   \N6',i1'RETURN'
                dc c' LApples \N268',i1'RETURN'
                dc c' LApple Pie \N269',i1'RETURN'
                dc c' LApple Turnover \N270',i1'RETURN'
                dc c'.'

                END
```

The MENU Program

Two programs that illustrate the use of the IIGS Menu Manager are at the end of this chapter. One, an assembly language program titled MENU.S1, is in listing 9–9. The other, a C program titled MENU.C, appears in listing 9–10.

MENU.S1 Program

MENU.S1 is a simple program; its menu table contains the names of only two subroutines. One, `Quit`, ends the program. The other, `CheckIt`, uses the Menu Manager call `GetMItemMark` to see if there is a check mark in front of the menu item selected. If there is no check mark, the `CheckIt` routine puts one there. If there is a check mark, `CheckIt` removes it.

Listing 9–7 is a source code listing of the `CheckIt` routine—and that concludes our analysis of the MENU.S1 program. When you have typed and run the program, be sure to save it. You'll use a similar menu, and add a windowing capability, in chapter 10.

Listing 9–7
CheckIt routine

```
CheckIt         START
                Using EventData

                PushWord #0                 ; space for result
                PushWord TaskData           ; menu item number
                _GetMItemMark
                pla
                beq putmark                 ; no check mark, so make one

erasemark       PushWord #0                 ; erase check mark
                PushWord TaskData           ; menu item number
                _SetMItemMark
                bra return

putmark         PushWord #18                ; ASCII for check mark
                PushWord TaskData           ; menu item number
                _SetMItemMark

return          rts

                END
```

MENU.C Program

MENU.C is the first program you have encountered so far that requires an expanded version of INITQUIT.C. In addition to the tool initialization in the original version of INITQUIT.C, the Menu Manager requires the use of the Window Manager and the Control Manager, so INITQUIT.C has grown. The revised version of INITQUIT.C appears in listing 9–8.

Listing 9–8
New version of INITQUIT.C

```
#include <TYPES.H>
#include <LOCATOR.H>
#include <MEMORY.H>
#include <MISCTOOL.H>
#include <QUICKDRAW.H>
#include <EVENT.H>
#include <CONTROL.H>
#include <WINDOW.H>
#include <MENU.H>

#define MODE mode640 /* 640 graphics mode def. from quickdraw.h */
#define MaxX 640     /* max X for cursor (for Event Mgr) */
#define dpAttr attrLocked+attrFixed+attrBank /* for allocating direct
 page space */

int  MyID;           /* for Memory Manager */
Handle zp;           /* handle for page 0 space for tools */

int ToolTable[]= {5,
                  4, 0x0100,  /* QD      */
                  6, 0x0100,  /* Event   */
                 14, 0x0100,  /* Window  */
                 16, 0x0100,  /* Control */
                 15, 0x0100,  /* Menu    */
                 };

StartTools()        /* start up these tools: */
{
   TLStartUp();             /* Tool Locator */
   MyID = MMStartUp();      /* Mem Manager  */
   MTStartUp();             /* Misc Tools   */
   LoadTools(ToolTable);    /* load tools from disk */
   ToolInit();              /* start up the rest    */
}
ToolInit()      /* init the rest of needed tools */
{
   zp = NewHandle(0x600L,MyID,dpAttr,0L);   /*reserve 6 pages */
   QDStartUp((int) *zp, MODE, 160, MyID);    /* uses 3 pages */
   EMStartUp((int) (*zp + 0x300), 20, 0, MaxX, 0, 200, MyID);
   WindStartUp(MyID);
   RefreshDesktop(NULL);
   CtlStartUp(MyID, (int) (*zp + 0x400));
   MenuStartUp(MyID, (int) (*zp + 0x500));
   ShowCursor();
}
```

```
ShutDown()              /* shut down all of the tools we started */
{
    GrafOff();
    MenuShutDown();
    CtlShutDown();
    WindShutDown();
    EMShutDown();
    QDShutDown();
    MTShutDown();
    DisposeHandle(zp); /* release our page 0 space */
    MMShutDown(MyID);
    TLShutDown();
}
```

Another significant difference between MENU.C and the event loop programs in previous chapters is that MENU.C uses the Window Manager call `TaskMaster` rather than the Event Manager call `GetNextEvent`. Because TaskMaster takes care of most of the event loop details in MENU.C, the rest of the event loop routine is interested in the answer to just one question: Was a menu item selected? If one was, you want to know whether it was the Quit item in the Files menu or simply an item that should be checked or unchecked.

The way in which the MENU.C program handles the checking of items is a little tricky. Because the Menu Manager call `CheckMItem` returns the ASCII value of a check mark when an item has been checked or a 0 if there is no check mark, you can treat the call's result as a Boolean value; true if an item is marked and false if it is not. Similarly, the `CheckMItem` call takes a Boolean value as an input and uses the value to determine whether to check or uncheck a menu item.

In the MENU.C program, you want to send a value of true to `CheckMItem` if you want an item marked, and you want to send a value of false if you want an item unmarked. By prefixing the logical inverse operator ! (pronounced "not" or, by UNIX fans, "bang") to `GetMItemMark`, you can pass the result returned by `GetMItemMark` directly to the `CheckMItem` routine.

Another trick used in the MENU.C program is the use of a pointer to refer to the contents of the `wmTaskData` field in TaskMaster's task record. By typecasting the address of this long word field to a pointer to a word called `data`, you can reference the low word of the field (the item number) as `*data` and the high word of the field (the menu number) as `*(data+1)`. Even though the contents of the `wmTaskData` field may change with each cycle through the event loop, the address of the information it contains always remains the same. Thus, you merely have to set the value of `data` to this address once before you begin the loop, and the value of `*data` and `*(data+1)` will always be equal to the latest results.

MENU.S1 and MENU.C Listings

Listing 9–9
MENU.S1 program

```
*
*  MENU.S1
*

*** A FEW ASSEMBLER DIRECTIVES ***

                Title 'Menu'

                ABSADDR on
                LIST off
                SYMBOL off
                65816 on
                mcopy menu.macros

                KEEP Menu

*
*  EXECUTABLE CODE STARTS HERE
*

Begin           START
                Using QuitData

                jmp MainProgram         ; skip over data

                END

*
*  SOME DIRECT PAGE ADDRESSES AND A FEW EQUATES
*

DPData          START

DPPointer       gequ    $00
DPHandle        gequ    DPPointer+4

TabPtr          gequ     $00

ScreenMode      gequ $80                ; 640 mode
MaxX            gequ 640                ; X clamp high

False           gequ $00
```

```
                END

*
*   MAIN PROGRAM LOOP
*

MainProgram     START
                Using GlobalData

                phk
                plb
                tdc                     ; get current direct page
                sta MyDP                ; and save it for the moment

                jsr ToolInit            ; start up all tools we'll need
                jsr BuildMenu           ; create and draw menu bar
                jsr EventLoop           ; check for key & mouse events

*** WHEN EVENT LOOP ENDS, WE'LL SHUT DOWN ***

                jsr Shutdown
                jmp Endit

                END

*
*  EVENT LOOP
*

EventLoop       START
                Using QuitData
                Using TaskTable
                Using EventData

Again           PushWord #0             ; space for result
                PushWord #$FFFF         ; recognize all events
                PushLong #EventRecord
                _TaskMaster
                pla
                asl a                   ; code * 2 = table location
                tax                     ; X is index register
                jsr (TaskTable,x)       ; look up event's routine
                lda QuitFlag
                beq again

                rts

                END
```

```
*
*   CREATE AND DRAW MENU
*
BuildMenu       START                   ; proceeding from back to front
                using MenuData

                PushLong #0             ; space for return
                PushLong #Menu6
                _NewMenu
                PushWord #0
                _InsertMenu

                PushLong #0             ; space for return
                PushLong #Menu5
                _NewMenu
                PushWord #0
                _InsertMenu

                PushLong #0             ; space for return
                PushLong #Menu4
                _NewMenu
                PushWord #0
                _InsertMenu

                PushLong #0             ; space for return
                PushLong #Menu3
                _NewMenu
                PushWord #0
                _InsertMenu

                PushLong #0             ; space for return
                PushLong #Menu2         ; 'wait' screen menu bar
                _NewMenu
                PushWord #0
                _InsertMenu

                PushLong #0             ; space for return
                PushLong #Menu1
                _NewMenu
                PushWord #0
                _InsertMenu

                PushWord #1             ; get NDAs for Apple Menu
                _FixAppleMenu

                PushWord #0             ; init & draw the menu bar
                _FixMenuBar
```

```
                pla                     ; discard menu bar height

                _DrawMenuBar

                rts

                END

*
*  DoMenu
*  Called when TaskMaster tells us a new menu item is selected.
*

DoMenu          START
                Using TaskTable
                Using EventData
                Using MenuTable

                lda TaskData            ; get TaskData value
                cmp #256
                bcc GiveUp              ; this should never happen

                and #$00FF              ; mask off high byte
                asl a                   ; double the value
                tax                     ; for 2-byte addresses

                jsr (MenuTable,x)

GiveUp          anop
                PushWord #False         ; draw normal
                PushWord TaskData+2     ; which menu?
                _HiliteMenu             ; unhighlight it

                rts

                END

*
*  ROUTINE TO PRINT A CHECK MARK IN FRONT OF A MENU ITEM
*

CheckIt         START
                Using EventData

                PushWord #0             ; space for result
                PushWord TaskData       ; menu item number
                _GetMItemMark
                pla
```

```
                beq putmark                ; no check mark, so make one

erasemark       PushWord #0                ; erase check mark
                PushWord TaskData          ; menu item number
                _SetMItemMark
                bra return

putmark         PushWord #18               ; ascii for check mark
                PushWord TaskData          ; menu item number
                _SetMItemMark

return          rts

                END

*
*  THIS IS WHERE WE INITIALIZE OUR TOOLS
*

ToolInit        START
                Using GlobalData
                Using ToolTable

*** START UP TOOL LOCATOR ***

                _TLStartup                 ; Tool Locator

*** INITIALIZE MEMORY MANAGER ***

                PushWord #0
                _MMStartup

                pla
                sta MyID

*** INITIALIZE MISC. TOOLS SET ***

                _MTStartup

*** GET SOME DIRECT PAGE MEMORY FOR TOOLS THAT NEED IT ***

                PushLong #0                ; space for handle
                PushLong #$800             ; eight pages
                PushWord MyID
                PushWord #$C001            ; locked, fixed, fixed bank
                PushLong #0
                _NewHandle
```

```
                pla
                sta DPHandle
                pla
                sta DPHandle+2

                lda [DPHandle]
                sta DPPointer

*** INITIALIZE QUICKDRAW II ***

                lda DPPointer           ; pointer to direct page
                pha
                PushWord #ScreenMode    ; either 320 or 640 mode
                PushWord #160           ; max size of scan line
                PushWord MyID
                _QDStartup

*** INITIALIZE EVENT MANAGER ***

                lda DPPointer           ; pointer to direct page
                clc
                adc #$300               ; QD direct page + #$300
                pha                     ; (QD needs 3 pages)
                PushWord #20            ; queue size
                PushWord #0             ; X clamp low
                PushWord #MaxX          ; X clamp high
                PushWord #0             ; Y clamp low
                PushWord #200           ; Y clamp high
                PushWord MyID
                _EMStartup

*** LOAD SOME TOOLS FROM RAM ***

LoadEmUp        PushLong #ToolTable
                _LoadTools

*** WINDOW MANAGER ***

                PushWord MyID
                _WindStartup

                PushLong #$0000
                _Refresh

*** CONTROL MANAGER ***

                PushWord MyID
                lda DPPointer           ; DP to use = qd dp + $400
```

```
                clc
                adc #$400
                pha
                _CtlStartup

*** MENU MANAGER ***

                PushWord MyID
                lda DPPointer              ; DP to use = qd dp + $600
                clc
                adc #$600
                pha
                _MenuStartup

                _ShowCursor

                rts

                END

*
*  THE ROUTINE THAT ENDS THE PROGRAM
*

EndIt            START

                 Using QuitData

                 _Quit QuitParams

                 END

*
*  SHUT DOWN ALL THE TOOLS WE STARTED UP
*

ShutDown        START
                Using GlobalData

                _MenuShutDown
                _CtlShutDown
                _WindShutDown
                _EMShutDown
                _QDShutDown
                _MTShutDown

                PushLong DPHandle
                _DisposeHandle
```

```
                PushWord MyID
                _MMShutDown
                _TLShutDown

                rts

                END

*
*  ROUTINE THAT SETS THE QUIT FLAG
*

doQuit          START
                Using QuitData

                lda #$8000
                sta QuitFlag
                rts

                END

*
*  A USEFUL AND CONVENIENT WAY NOT TO DO ANYTHING
*

Ignore          START

                rts

                END

*
*  DATA SEGMENTS
*

*
* Menu Data
*

MenuData        DATA

Return          equ 13

Menu1           dc c'>L@\XN1',i1'RETURN'
                dc c' LAn Apple Menu\N257',i1'RETURN'
                dc c'.'

Menu2           dc c'>L File    \N2',i1'RETURN'
```