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

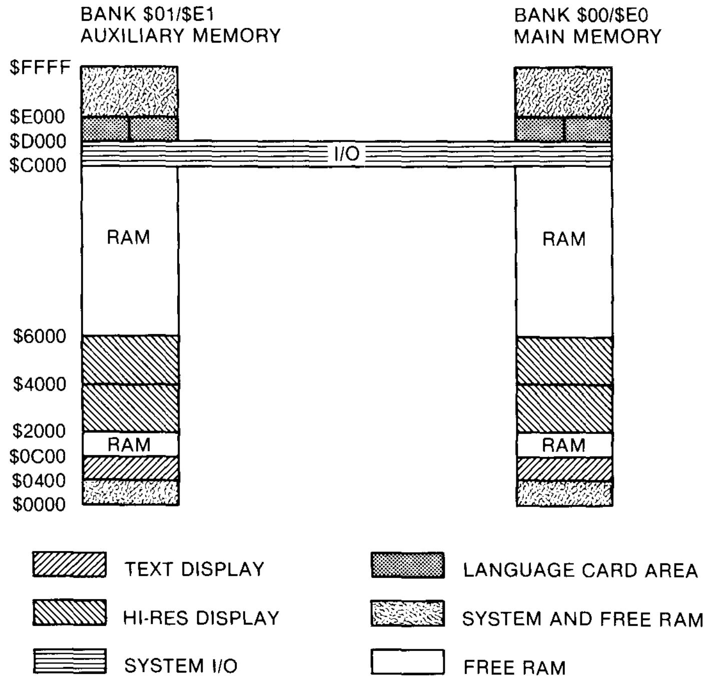

Figure 1–1
Memory map of the IIGS in IIc/IIe emulation mode

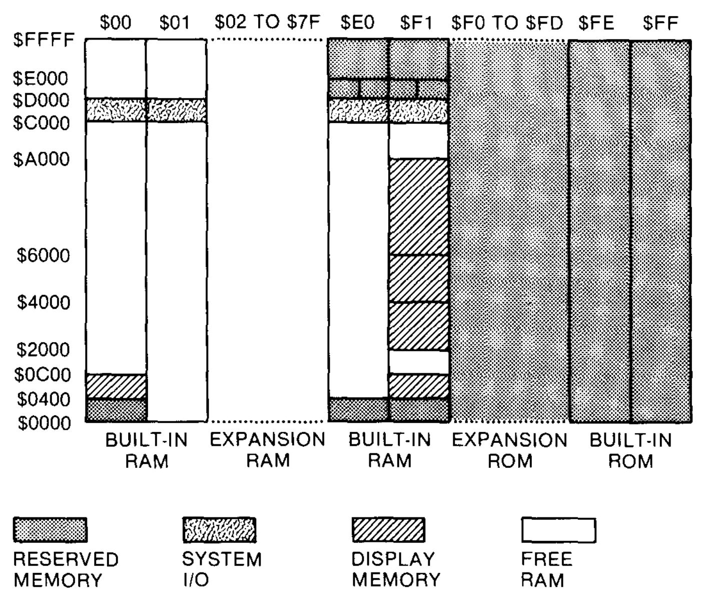

Figure 1–2
Memory map of the IIGS in 16-bit native mode

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.

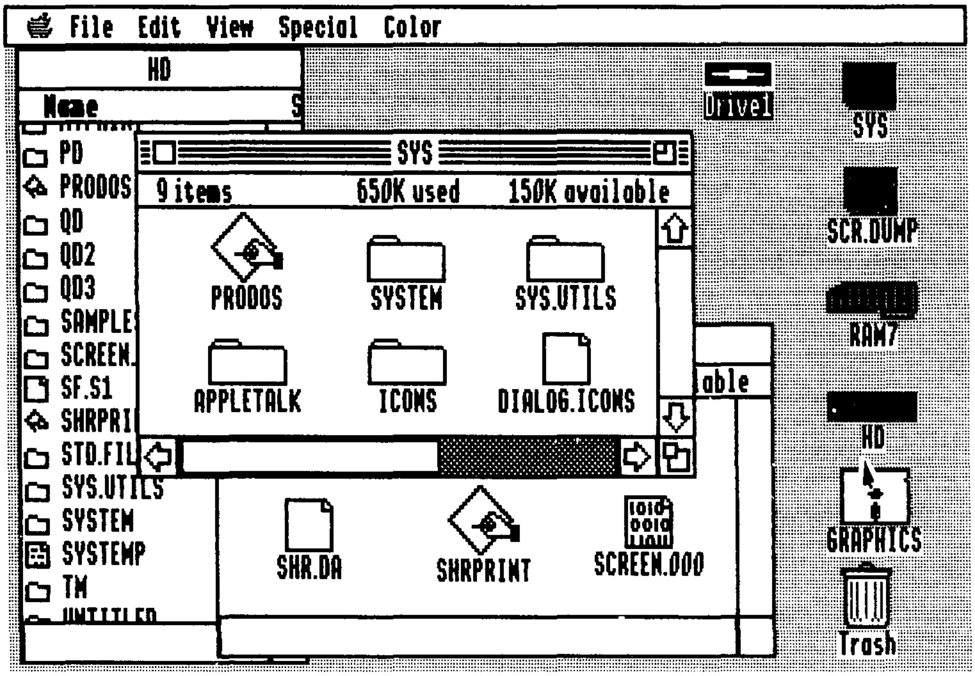

Figure 1–3
Finder disk screen display

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

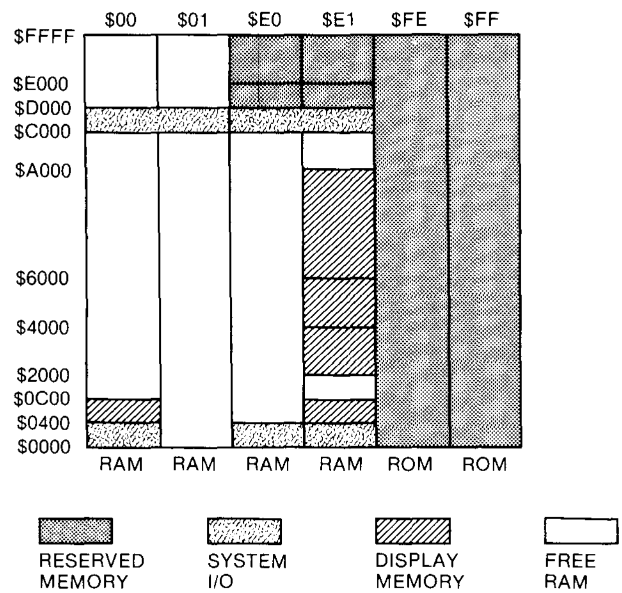

Figure 4–1
Memory map of an unexpanded Apple IIGS

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

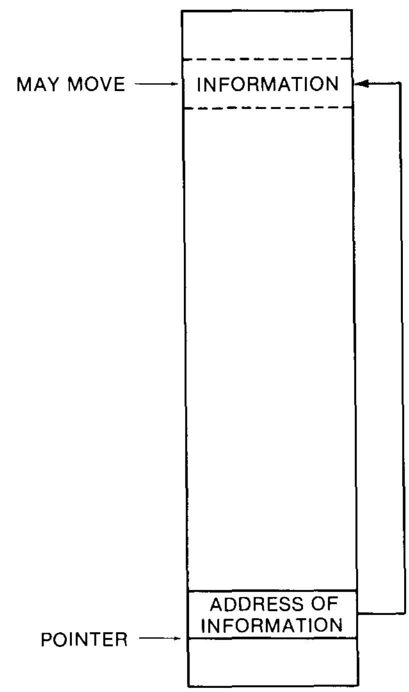

Figure 4–2
Using a pointer in an assembly language program

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

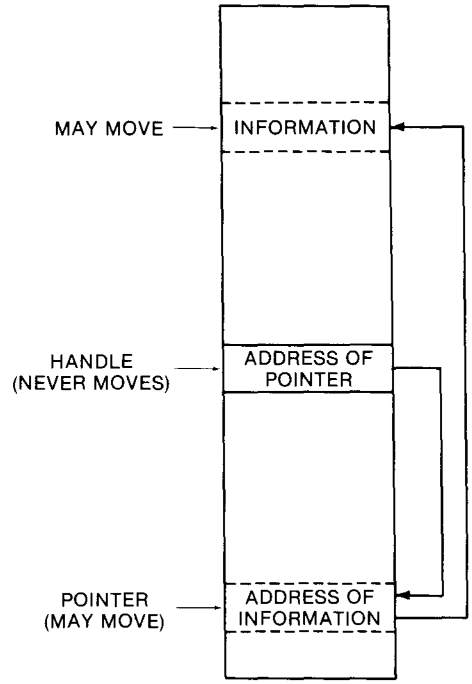

Figure 4–3
Using a handle in an assembly language program

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.

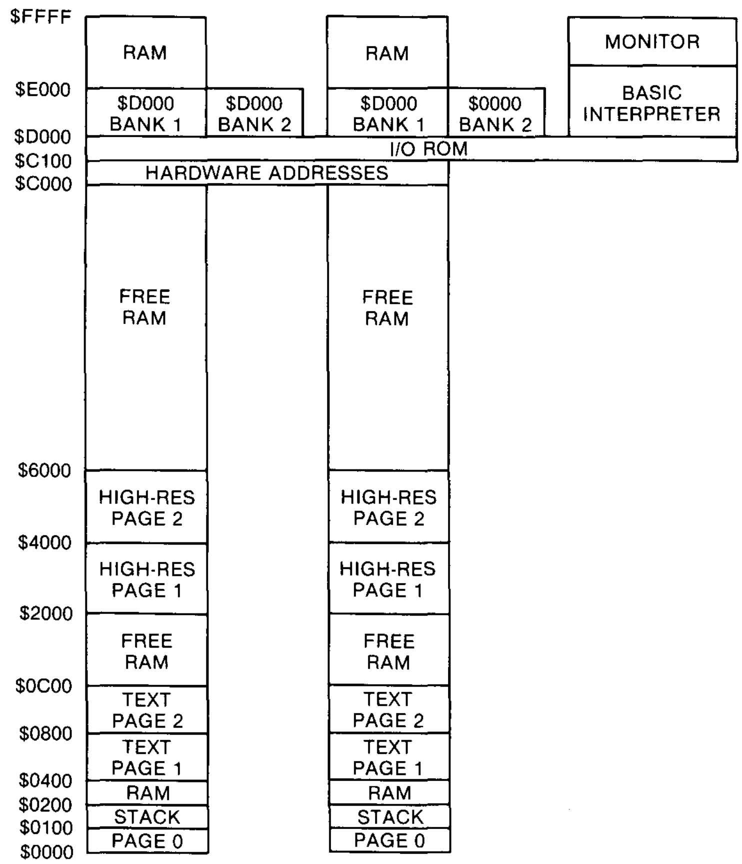

Figure 4–4
A map of the IIGS in emulation mode

- 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.

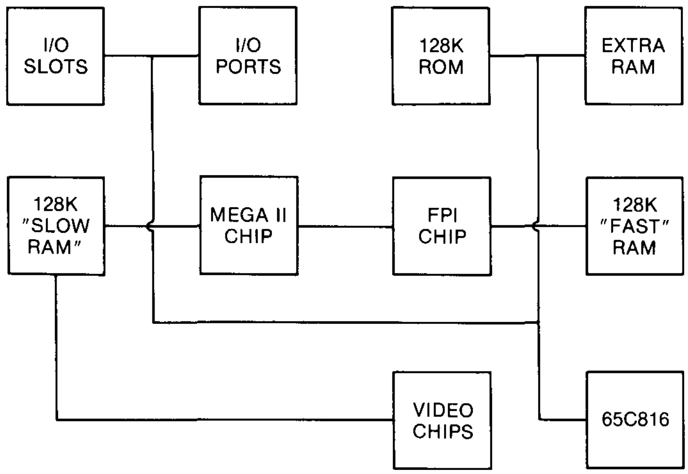

Figure 4–5
Incorporating the Mega II chip into the IIGS's design

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.*

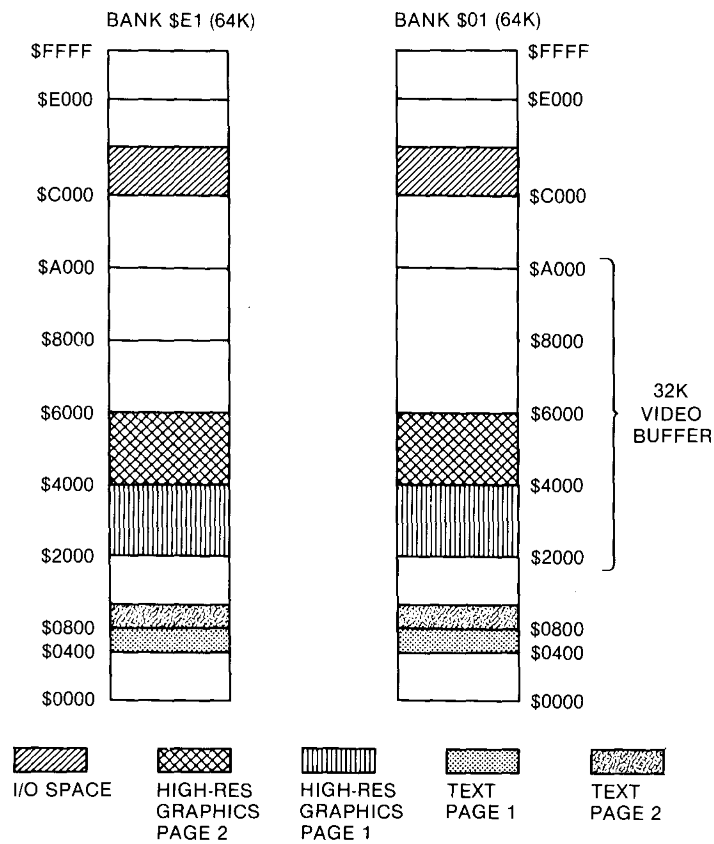

Figure 4–6
Memory shadowing in the Apple IIGS

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

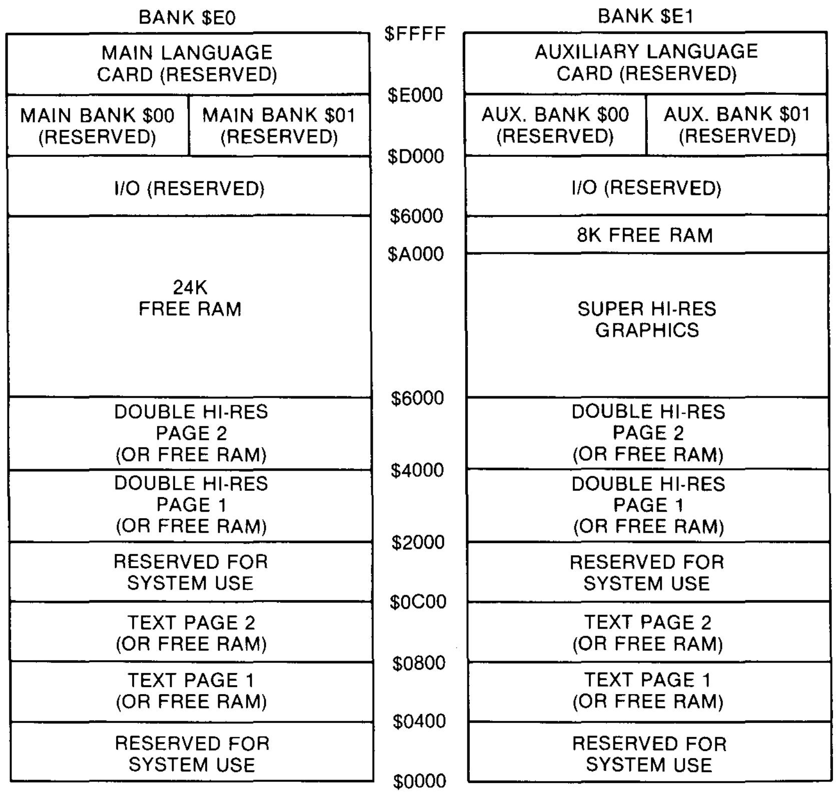

Figure 4–7
Detailed memory map of banks $E0 and $E1

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

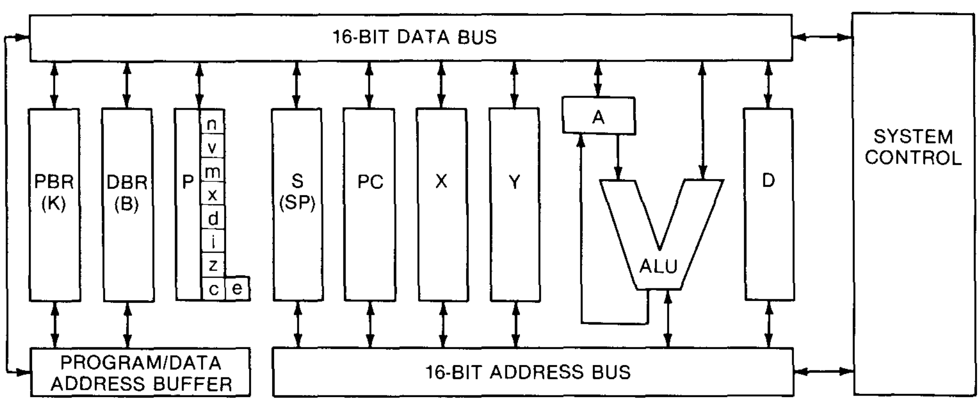

Figure 5–1
Simplified block diagram of the 65C816

- 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.

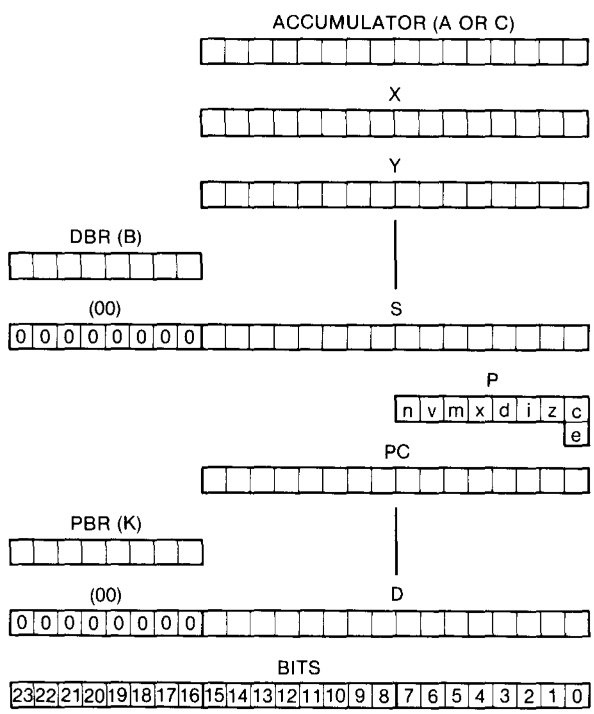

Figure 5–2
65C816 register configuration in native mode

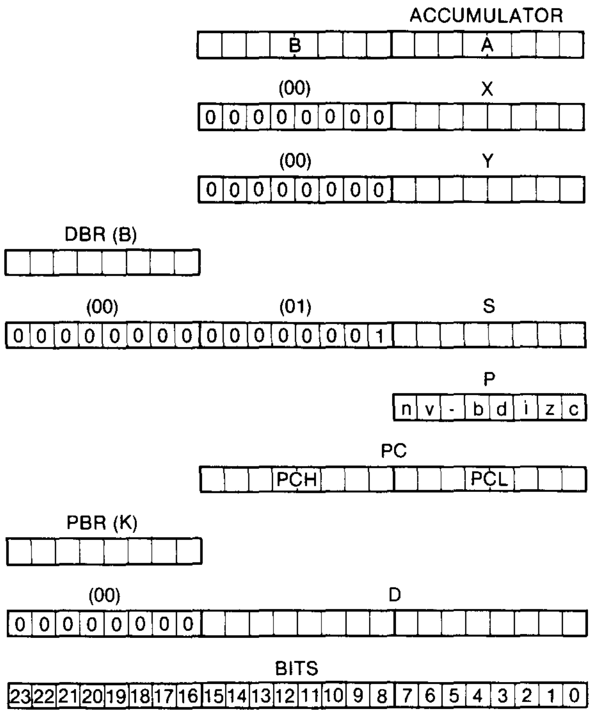

Figure 5–3
65C816 register configuration in emulation mode

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 concatenated 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 visualized 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

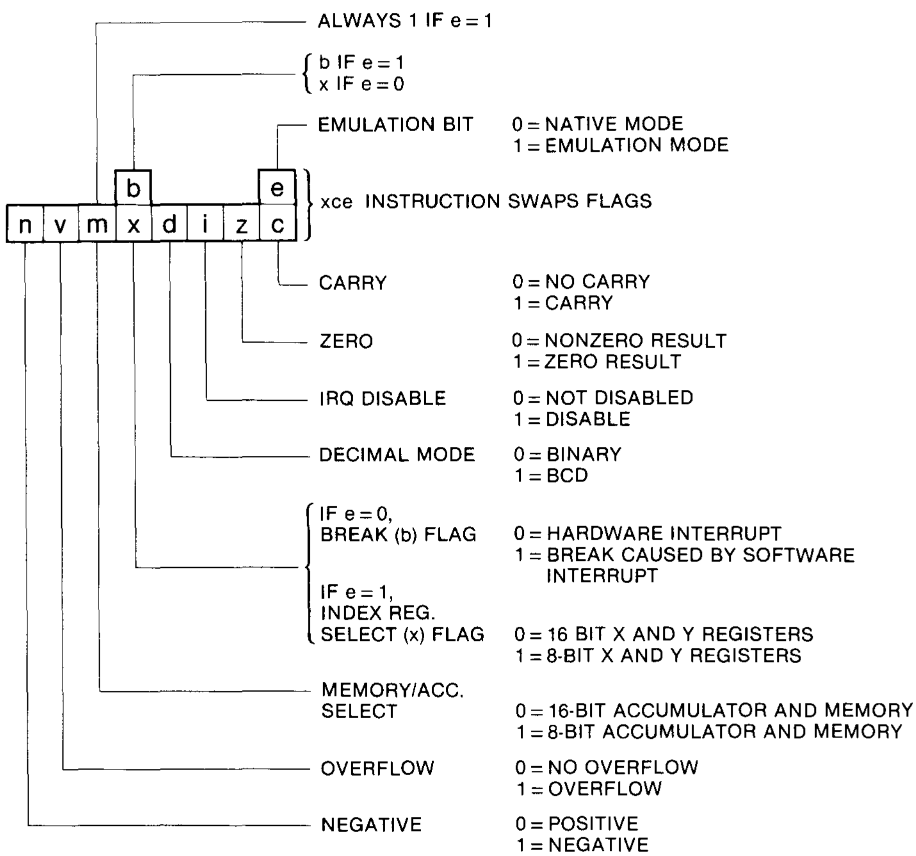

Figure 5–4
Processor status register

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 hexadecimal 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

Addressing Mode	Example	Identifier
Simple Addressing		
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 Addr-Demo 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.

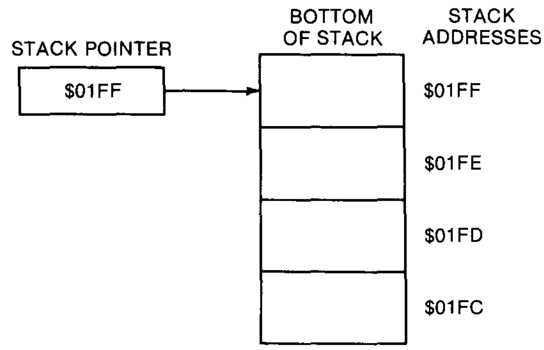

Figure 6–1
How the stack pointer works

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.

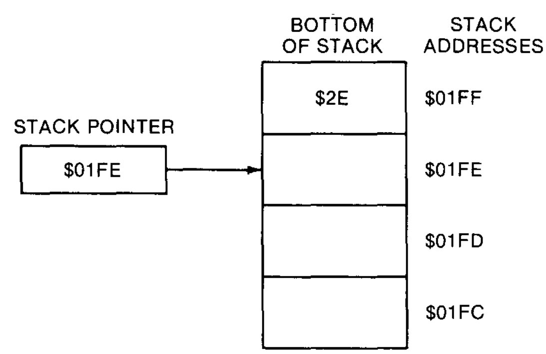

Figure 6–2
Placing a number on the stack

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

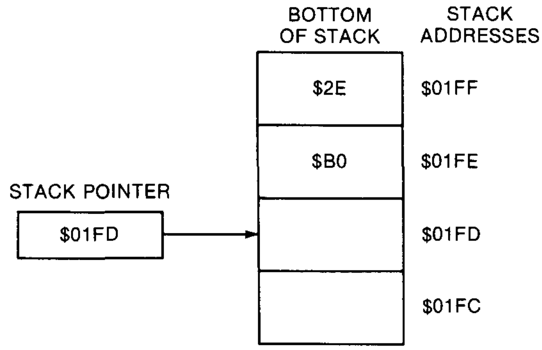

Figure 6–3
Placing another number on the stack

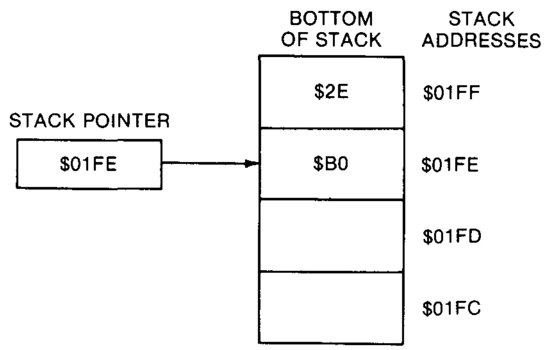

Figure 6–4
Pulling a number off the stack

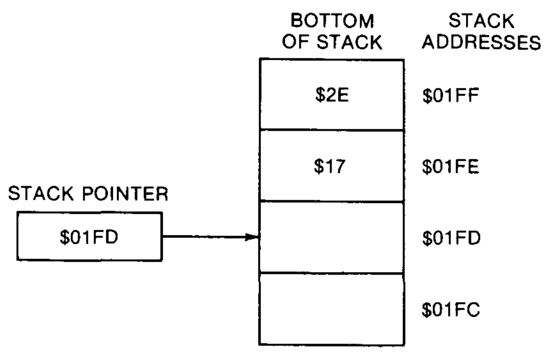

Figure 6–5
One last stack manipulation

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.

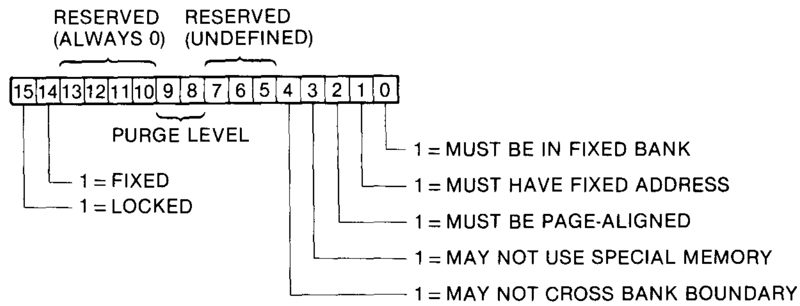

Figure 7–1
Attributes word used by the Memory Manager

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

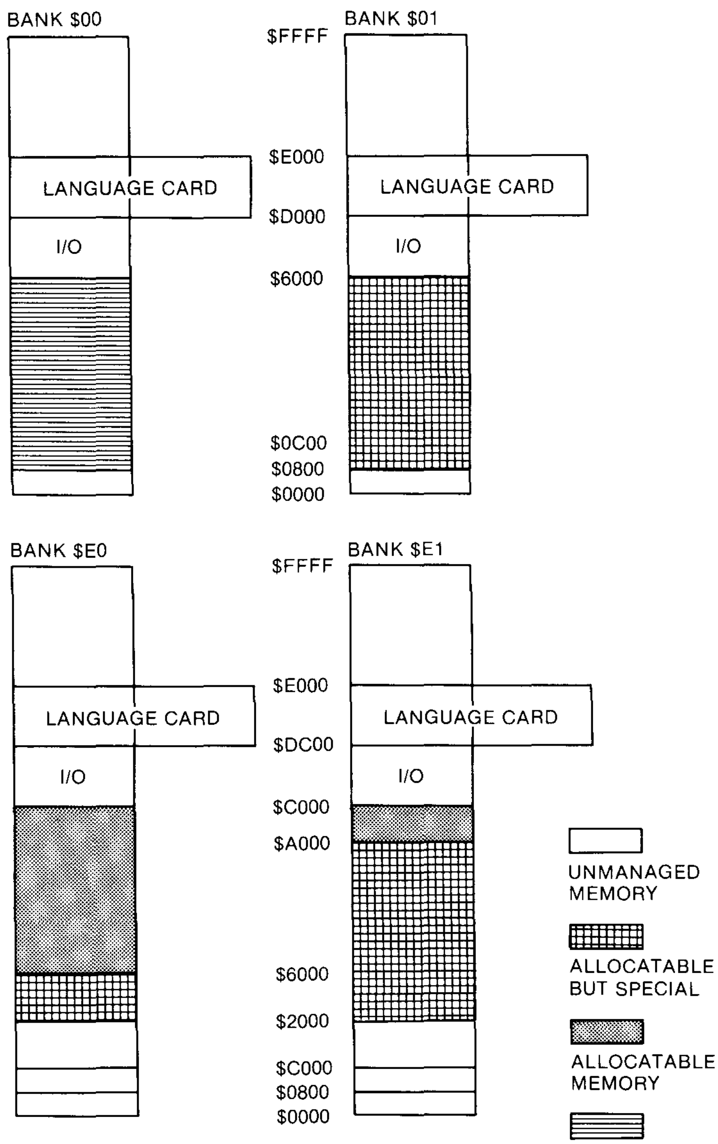

Figure 7–2
Allocatable, special, and unmanaged memory

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

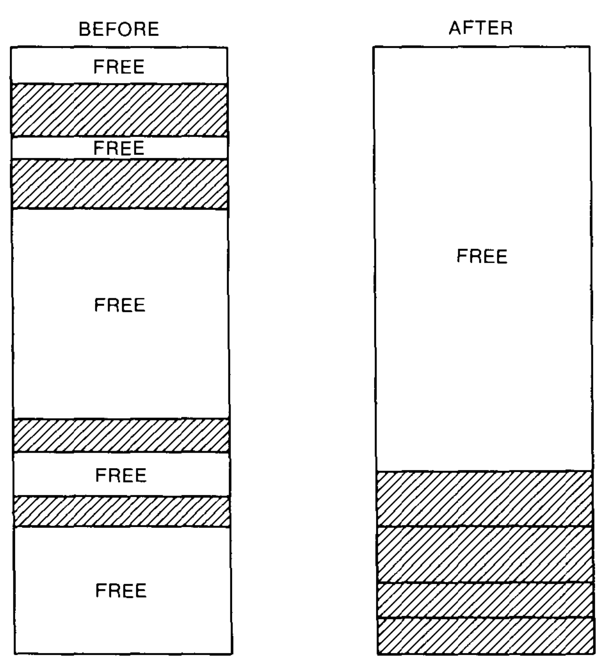

Figure 7–3
How the Memory Manager compacts memory

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

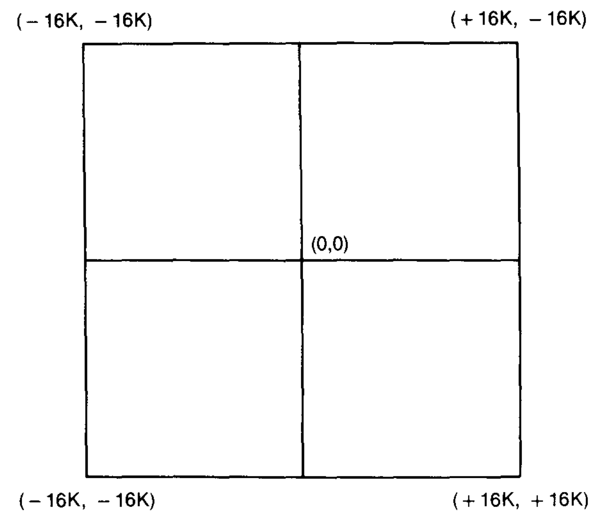

Figure 8–1
QuickDraw II conceptual drawing plane

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.

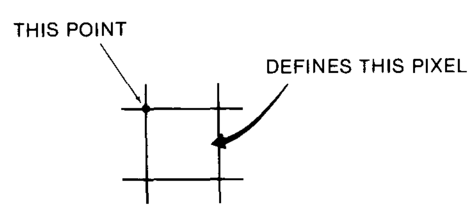

Figure 8–2
Coordinates of a pixel

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,

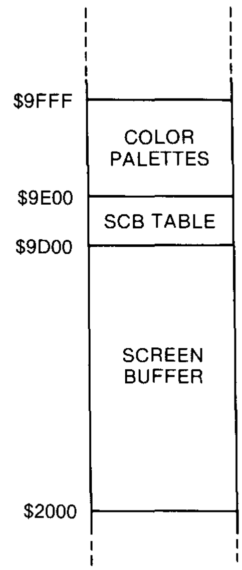

Figure 8–4
Memory map of screen buffer, SCB table, and color palettes

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.

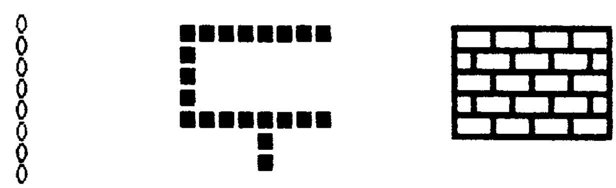

Figure 8–5
Pen pattern in memory and on the screen

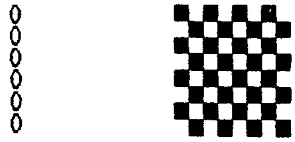

Figure 8–6
Pen mask

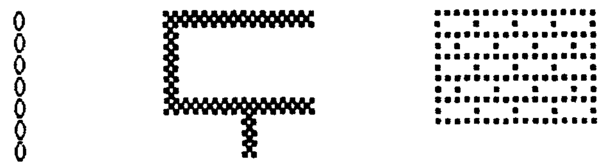

Figure 8–7
Effect of using a pen mask

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

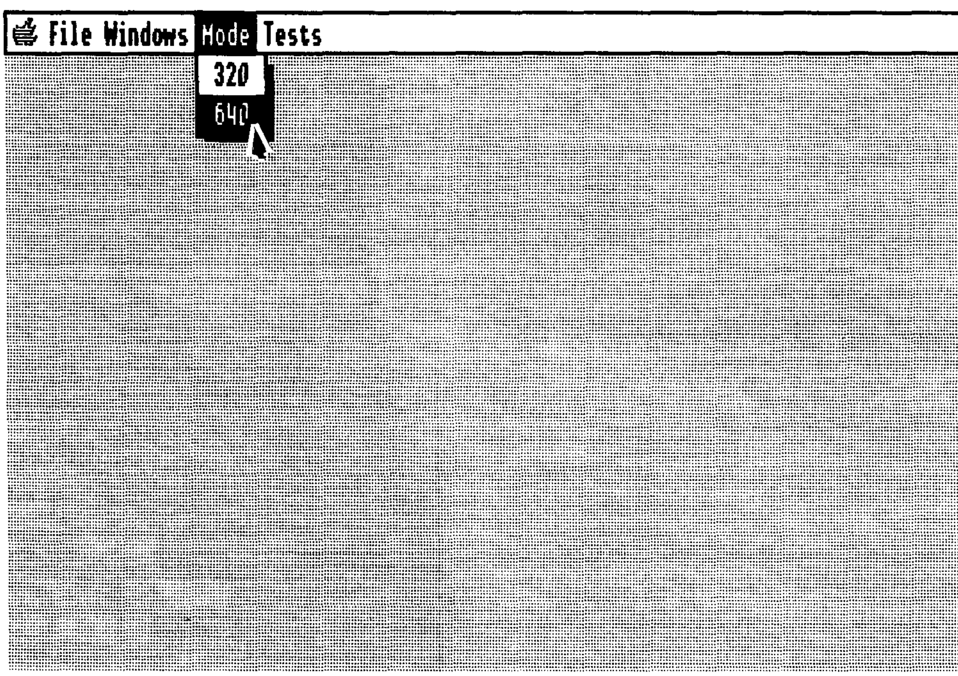

Figure 9–1
Menu in 640 mode

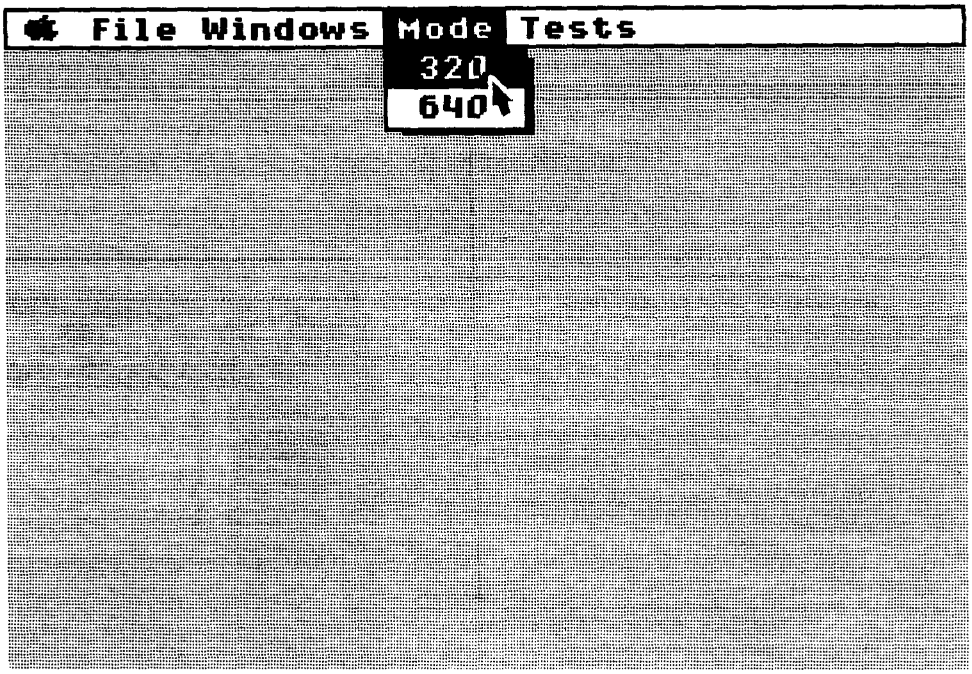

Figure 9–2
Menu in 320 mode

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

                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         ; 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'
```

```
                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

***

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

***

TaskTable   DATA

            dc i'ignore'             ; 0 null
            dc i'ignore'             ; 1 mouse down
            dc i'ignore'             ; 2 mouse up
            dc i'ignore'             ; 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

*
* TaskMaster events
*

            dc i'DoMenu'             ; 1 in menu bar
            dc i'ignore'             ; 2 in system window
            dc i'ignore'             ; 3 in content of window (Move It)
            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

***

ToolTable       DATA

                dc i'5'                  ; number of tools in table
                dc i'$04,$0100'          ; QuickDraw
                dc i'$06,$0100'          ; Event Manager
                dc i'$0E,$0000'          ; Window Manager
                dc i'$0F,$0100'          ; Menu Manager
                dc i'$10,$0100'          ; Control Manager

                END

***

EventData       DATA

EventRecord     anop                      ; table for Event Manager
EventWhat       ds 2
EventMessage    ds 4
EventWhen       ds 4
EventWhere      ds 4
EventModifiers  ds 2
TaskData        ds 4
TaskMask        dc i4'$0FFF'

                END

***

QuitData        DATA

QuitFlag        ds 2

QuitParams      dc  i4'0'
                dc  i4'0'
                dc  i4'0'

                END

***
```

```
GlobalData      DATA

MyID            dc  i'0'            ; program ID word
MyDP            ds  2

                END
```

Listing 9–10
MENU.C program

```
/***********************************/
/* Data and routine to create menus */
/***********************************/

/*   Set up menu strings. Because C uses \ as an escape character,
we use two when we want a \ as an ordinary character. The \
at the end of each line tells C to ignore the carriage return. This lets
us set up our items in an easy-to-read vertical alignment. */

char *menu1 = "\
>L@\\XN1\r\
 LAn Apple Menu\\N257\r\
.";

char *menu2 = "\
>L File   \\N2\r\
 LQuit \\N258*Qq\r\
.";

char *menu3 = "\
>L Appetizers   \\N3\r\
 LApple Salad \\N259\r\
 LApple Jello \\N260\r\
 LApple Slices \\N261\r\
 LApple Juice \\N262\r\
.";

char *menu4 = "\
>L Entrees   \\N4\r\
 LApple Duckling \\N263\r\
 LApple Dumplings \\N264\r\
.";

char *menu5 = "\
>L Beverages   \\N5\r\
 LApple Shake \\N265\r\
 LApple Cola \\N266\r\
 LApple Wine \\N267\r\
.";
```

```
char *menu6 = "\
>L Desserts   \\N6\r\
 LApples \\N268\r\
 LApple Pie \\N269\r\
 LApple Turnover \\N270\r\
.";

#define QUIT_ITEM 258  /* these will help us check menu item numbers */
#define LAST_ITEM 270

BuildMenu()
{
   InsertMenu(NewMenu(menu6),0);
   InsertMenu(NewMenu(menu5),0);
   InsertMenu(NewMenu(menu4),0);
   InsertMenu(NewMenu(menu3),0);
   InsertMenu(NewMenu(menu2),0);
   InsertMenu(NewMenu(menu1),0);
   FixMenuBar();
   DrawMenuBar();
}

/********************************/
/* Main routine and event loop  */
/********************************/

WmTaskRec   myEvent;
Boolean done = false;

main()
{
   StartTools();
   BuildMenu();
   EventLoop();
   ShutDown();
}

/*   When a menu bar event is returned, test the item number for a
checkable item. Use the logical inverse of the value returned by
GetMItemMark as a parameter to CheckMItem. This will toggle the check
mark for each item.  */

EventLoop()
{
Word *data = (Word *)&myEvent.wmTaskData; /* address of item id  */

   myEvent.wmTaskMask = 0x0FFF;
   while (!done)
```

```
      if ( TaskMaster(everyEvent,&myEvent) == wInMenuBar ) {
         if (*data == QUIT_ITEM)
            done = true;
         else if ((*data > QUIT_ITEM) && (*data <= LAST_ITEM))
            CheckMItem (!GetMItemMark(*data), *data);
         HiliteMenu(false,*(data + 1)); /* data + 1 is address of menu
          id  */
      }
}
```

CHAPTER 10

Doing Windows

Using the Window Manager

Yes, the Apple IIGS does windows—and with real class, too! To make sure they're done properly, the IIGS employs a Window Manager. The Window Manager—like the Event Manager, which was introduced in chapter 7—is a very important toolkit in the Apple IIGS Toolbox. It is the Window Manager's job to handle all windows placed on the IIGS desktop. It can create, draw, shrink, expand, scroll, and move windows. When you've finished working with a window, the Window Manager can remove it from your screen. When you're through with a window for good, the Window Manager can dispose it and deallocate its memory.

The Window Manager takes care of all kinds of windows, not just picture windows and document windows, but also dialog windows, alert windows, and windows custom-tailored for specific programs. Want a round window or a triangular window? The Window Manager can make one. How about a window that seems to explode when you click the mouse in its go-away box or a window with custom-designed controls? No problem for the Apple IIGS Window Manager. It's a toolkit that can do just about any kind of window.

Kinds of Windows

The kinds of windows the Window Manager can manage are divided into three categories:

- *Document windows.* Most of the windows used in IIGS programs are in this category. A window doesn't have to contain text to be classified as a document window. Windows that contain pictures drawn with programs like PaintWorks Plus are also document windows.
- *Dialog windows.* There are three kinds of dialog windows: modal dialogs, modeless dialogs, and alert windows. Although low-level operations for all three types of windows can be handled by the Window Manager, they are mostly the responsibility of the Dialog Manager. So we won't go into detail about them until chapter 11, which is all about the Dialog Manager.
- *Custom-designed windows.* You can design custom windows using the Window Manager, but that is beyond the scope of this book. If you'd like to design your own windows, you can find some tips on how to do it in the *Apple IIGS Toolbox Reference.*

Window Frames

There are two kinds of predefined window frames: *alert window frames* and *document window frames.* An alert window frame is a double black line. A document frame is a single black line or includes controls.

A window does not have to be an alert window to have an alert window frame; document windows can have alert window frames, too. A standard document window frame and an alert window frame are illustrated in figure 10–1.

Controls

The screen of the IIGS represents a working desktop. Various graphic objects appear on this desktop and are manipulated with a mouse. A window is a

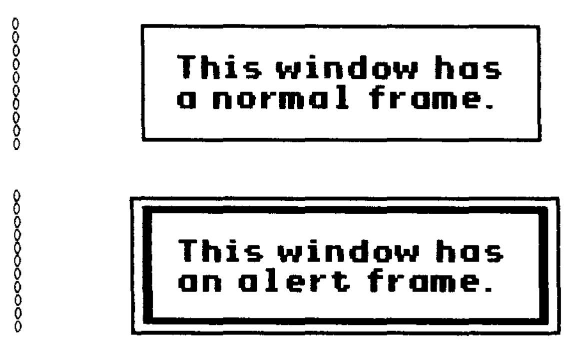

Figure 10–1
Document frame and alert frame

desktop object that presents information; it can contain a document, a picture, a message, or other items. Windows can be almost any size or shape, and one or more of them can be on the desktop at any time.

Windows owe their name to the fact that they can show you more information than the IIGS screen can display at one time. When a window is on the screen, you can look through it into a larger area. The information displayed through a window can be pictures, text, data, or all three. When you look at something through a window—for example, a picture—the window can be moved around over the picture with a control called a *scroll bar.*

Most document windows have two scroll bars: a horizontal scroll bar, which scrolls the window horizontally, and a vertical scroll bar, which scrolls the window vertically. You'll learn how to use both kinds of scroll bars before you finish this chapter.

A document window can also have the following controls:

- A *title bar,* which is a horizontal bar that displays the window's title, if there is one. A title bar can contain a close box, which makes the window disappear from the screen, and a zoom box, which changes the window's size. A title bar can be used as a *drag region* for moving the window.
- A *grow region,* which is a small box in the lower right corner of a window that changes the window's size.
- An *information bar,* another horizontal bar in which an application can display information that won't be affected by the movements of scroll bars.

Information bars may have their uses, but they are not popular in programs written for the IIGS. A standard document window, without an information bar, is illustrated in figure 10–2. The controls in the title bar of a document window are used as follows:

- Clicking the mouse anywhere in an inactive window highlights its title bar and makes it the active window, the window in which drawing and other activities take place. The title bars of all other windows become unhighlighted. Although these windows remain on the screen, they become inactive windows. According to Apple's *Human Interface Guidelines,* there should never be more than one active window on the screen.
- Clicking the mouse in the close box, or go-away region, closes the window. Usually, when you click the mouse in the close box, an application program calls the Window Manager routine `HideWindow`, which makes the window disappear from the screen.
- Pressing the mouse button in the window's drag region (title bar) and then dragging the window pulls an outline of the window across the screen. Holding the mouse button down and releasing it in a new location moves the window there. Unless the Apple key is held

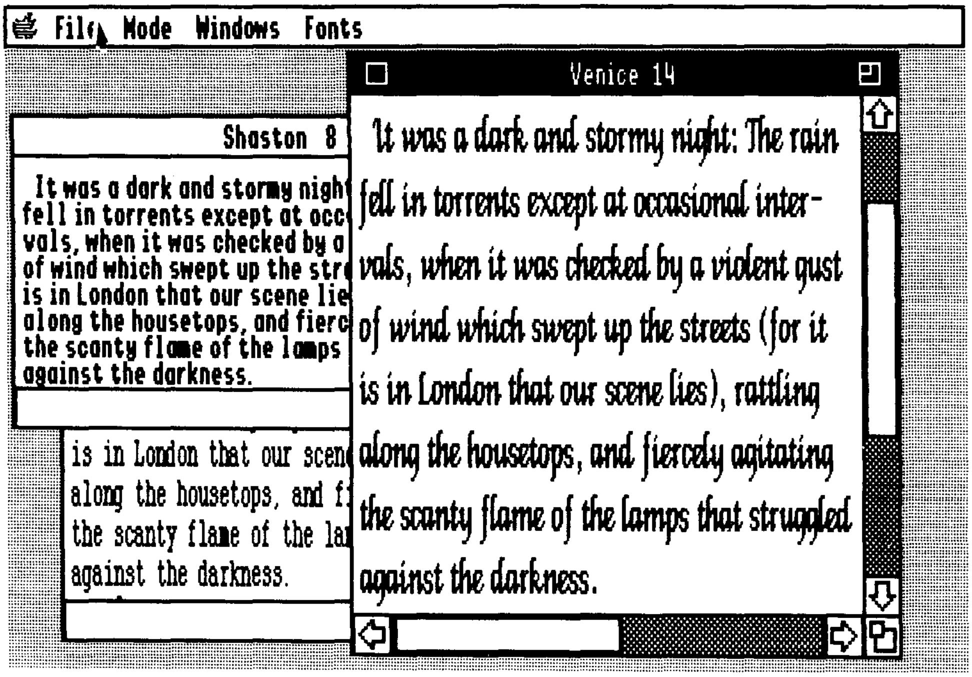

Figure 10–2
Standard document window

down when the mouse button is released, the moved window becomes the active window.

- Clicking the mouse inside the grow box and then dragging the grow box changes the window's size.

To keep windows from getting lost, the Window Manager prevents them from being dragged completely across the screen. The title bar can never be moved to a point where the visible area of the title bar is less than four pixels square.

Some windows are created by application programs and others are created by tools in the Toolbox. (For example, the Dialog Manager can create dialog windows.) Windows created by application programs and by tools in the Toolbox are known collectively as *application windows*. Another class of windows, called *system windows*, display desk accessories.

What the Window Manager Does

The Window Manager draws windows using QuickDraw II and the Control Manager, and it disposes them with the help of the Memory Manager. After a window is drawn on the screen, the Window Manager's main function is to keep track of overlapping windows. The Window Manager handles tasks so that you can draw in any window without running into windows in front of it. You can move a window to a different place on the screen, change its size, or change its plane (front-to-back order), and you don't have to worry about details, such as how parts of various windows cover parts of other windows. The Window Manager redraws windows as needed and ensures that they overlap properly.

Window Regions

Every window is made up of two regions:

- A *content region,* which is the area that lies inside the window's frame. An application can draw objects and text in this portion of a window.
- A *frame region,* which is the outline of the entire window, including its title bar and standard window controls.

A window's content region and frame region make up what is known as the *structure region* of the window.

Every window also has a *data area*: a block of memory that includes all the data that can be viewed through the window. If the window has scroll bars, they can be used to move the window over its data area.

If a window has a grow box, a zoom box, or both, they can be used to increase or decrease the size of the window, causing more or less of its data area to be displayed. When the window is scrolled, it moves over the data area. But when the window is moved from one part of the screen to another, the data area is moved with it, so the view remains the same.

Initializing the Window Manager

Before the Window Manager can be started up, it must be loaded into memory, and QuickDraw and the Event Manager must be loaded and initialized. The Window Manager call `WindStartup` can then be issued to initialize the Window Manager. Then you can use the Window Manager call `NewWindow` to create any windows needed in a program.

TaskMaster

In programs that use the Window Manager, there are two ways to handle user input. One way is to use the Event Manager call `GetNextEvent`. The other is to use the Window Manager call `TaskMaster`.

The easiest way to use the Window Manager is with TaskMaster. As you may recall from chapter 9, TaskMaster can handle events related to menus as well as events that involve windows. The interaction between TaskMaster and menus is covered in chapter 9. In this chapter, you see how to use TaskMaster in programs that make use of windows.

WINDOW.S1 shows how an assembly language program can handle windows using TaskMaster. WINDOW.C is a C language version of the same program. Both programs are at the end of this chapter.

When TaskMaster is used in a program, it does the following. First, TaskMaster makes the Event Manager call `GetNextEvent`. If an event isn't ready, TaskMaster returns a task code of 0. If an event is ready, TaskMaster looks at it and tries to handle it. If TaskMaster can't handle the event, it returns the event code to the application. The application can then handle the event as if its event code had been returned by `GetNextEvent`.

If TaskMaster can handle the event, it calls standard functions to try to complete the task. For example, if you press the mouse button in an active window's zoom box, TaskMaster makes the Window Manager call `TrackZoom` until the mouse leaves the zoom box or the mouse button is released. If you release the mouse button while the mouse is in the zoom box, TaskMaster calls `ZoomWindow` to zoom the window either in or out, as appropriate. This takes care of the complete zoom operation selected by the user, so TaskMaster returns no event.

If TaskMaster can handle only part of an event, it does what it can and then returns control to the calling program. For example, if you press the mouse in the active window's content region, TaskMaster can detect it, but it can't do anything further. In this case, TaskMaster returns a task code of $0013 (`wInContent`). That lets an application program know that the mouse button has been pressed in the active window's content region, but it is up to the application to determine what to do next.

The operation of TaskMaster is covered in detail in chapter 9, but here's a brief review. A call to TaskMaster takes three paramaters: a word to save a space on the stack, an event mask, and a pointer to a task record.

The event mask passed to TaskMaster is like an event mask used by the Event Manager. The task record used by TaskMaster is like an event record used by the Event Manager, but with two extra fields. Each time TaskMaster makes a `GetNextEvent` call, `GetNextEvent` fills in the first seventeen fields of the task record being used by TaskMaster. Then TaskMaster handles any events it can handle, fills in the last two fields of the task record, and returns.

Listing 10–1 is a task record used in this chapter's example program, WINDOW.S1. The WINDOW.S1 program, listed in its entirety at the end of this chapter, is a sketcher program that allows the user to draw into a window with a mouse. When a sketch is drawn, each dot in it is actually drawn twice: once into the window on the screen and once into a pixel image that paints the window's contents each time the window is updated. Thus, sketches drawn using the WINDOW.S1 program do not disappear from memory when a window is removed from the screen. They remain in memory and can show up in a window again when it is redrawn on the screen. In later chapters, the WINDOW.S1 program becomes even more sophisticated.

Listing 10–1
Task record in WINDOW.S1

```
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'

                END
```

As you may recall from chapter 9, 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 that is passed to the Event Manager in the `GetNextEvent` call.

A task mask is a word used by an application to tell TaskMaster what kinds of events it should look for and what kinds of events it should ignore. The high word of a task mask—bits 16 through 31–should always be clear. In the low word of a task mask, each bit corresponds to a task; a set bit causes TaskMaster to look for an event, and a cleared bit tells TaskMaster to ignore an event. For TaskMaster to look for every type of event it can handle, the task mask should be $0000FFFF. The bit layouts of an event mask and a task mask are listed in chapter 9.

Window Records

For each window used in an application program, the Window Manager maintains a *window record.* A window record contains a number of fields, but only the first seven are directly accessible to application programs. The rest of the fields in a window record can be accessed only through calls to the Window Manager. Table 10–1 shows the seven window record fields accessible to application programs.

When the Window Manager is active, it maintains a window list: a list of all windows currently open. It is important to note that a window can be open but hidden, and thus not visible on the screen.

As table 10–1 shows, the first field in a window record is a pointer to the Window Manager's window list. The second field is the window's GrafPort—the GrafPort itself, not a pointer to it. Thus, the length of the GrafPort field is the length of a GrafPort; the field is 186 bytes.

When a window is created using the Window Manager call `NewWindow`, the call returns a pointer to the new window's GrafPort. Thus, the value returned by `NewWindow` is also a pointer to the second field of a window

Table 10–1
Window Record Fields Accessible to Application Programs

Name	Length	Function
wNext	Long	Pointer to next window in the window list
wport	186 bytes	Window's port; returned window pointers point to here
wStrucRgn	Handle	Handle of window's structural region (frame plus content)
wContRgn	Handle	Handle of window's content region
wUpdateRgn	Handle	Handle of update regions (regions that needs redrawing)
wControl	Handle	Handle of application's first control in content region
wFrame	Word	Bit array that describes window's frame

record. So the value returned by NewWindow, as well as being a pointer to a GrafPort, can also calculate the addresses of the other six fields of a window record.

In addition to a GrafPort and a pointer to the next window in the window list, a window record contains a pointer to the window's title. A window's title is a bit array that provides details about the window's frame and the handles of four regions used to draw the window. The bit array in the wFrame field of a window record is shown in table 10–2.

NewWindow Call

Every window used in an application program must be set up with a call to the Window Manager routine NewWindow. A call to NewWindow takes two parameters: 2 null words (zeros) to save spaces on the stack and a pointer to a parameter block. The call returns with a pointer to a window pushed onto the stack. Listing 10–2 is a NewWindow call used in the WINDOW.S1 program.

Listing 10–2
Call to NewWindow

```
PushLong #0              ; space for result
PushLong #Win0ParamBlock
_NewWindow


pla
sta Win0Ptr
pla
sta Win0Ptr+2
```

Table 10–2
Bits in the wFrame Field

Bit	Name of Field	Value
0	F_HILITED	1 = Frame highlighted
		0 = Frame not highlighted
1	F_ZOOMED	1 = Currently zoomed
		0 = Frame not zoomed
2	F_ALLOCATED	1 = Record was allocated
		0 = Record was provided by application
3	F_CTRL_TIE	1 = Control's state is independent
		0 = Inactive window has inactive controls
4	F_INFO	1 = Information bar
		0 = No information bar
5	F_VIS	1 = Window is currently visible
		0 = Window is invisible
6	F_QCONTENT	1 = Return wInContent even if window is inactive
		0 = Don't return wInContent if window is inactive
7	F_MOVE	1 = Title bar is a drag region
		0 = No drag region
8	F_ZOOM	1 = Zoom box on title bar
		0 = No zoom box (zoom box must have title bar)
9	F_FLEX	1 = GrowWindow and ZoomWindow won't change the origin
		0 = GrowWindow and ZoomWindow will affect the origin
10	F_GROW	1 = Grow box
		0 = No grow box (grow box must have at least one scroll bar)
11	F_BSCRL	1 = Window frame horizontal scroll bar
		0 = No horizontal scroll bar
12	F_RSCRL	1 = Window frame vertical scroll bar
		0 = No vertical scroll bar
13	F_ALERT	1 = Alert type frame (don't set grow box, close box, info bar, title bar, or scrolls)
		0 = Standard frame
14	F_CLOSE	1 = Close box
		0 = No close box (close box must have title bar)
15	F_TITLE	1 = Title bar
		0 = No title bar

Parameter Blocks Before an application makes a NewWindow call, it must set up a parameter block that spells out many details about the window. Listing 10–3 is a NewWindow parameter block used in the WINDOW.S1 program. The fields in a window's parameter block are described in table 10–3.

Listing 10–3
Parameter block for a NewWindow call

```
WinOParamBlock anop
        dc    i'WinOEnd-WinOParamBlock'
        dc    i2'%1101110111000000' ; Bits describing frame
        dc    i4'WinOTitle'         ; Pointer to title
        dc    i4'0'                 ; RefCon
        dc    i2'26,0,188,308'      ; Full size (0=default)
        dc    i4'0'                 ; Color table pointer
        dc    i2'0'                 ; Vertical origin
        dc    i2'0'                 ; Horizontal origin
        dc    i2'200'               ; Data area height
        dc    i2'320'               ; Data area width
        dc    i2'200'               ; Max cont height
        dc    i2'320'               ; Max cont width
        dc    i2'2'                 ; No. of pixels to scroll vertically
        dc    i2'2'                 ; No. of pixels to scroll horizontally
        dc    i2'20'                ; No. of pixels to page vertically
        dc    i2'32'                ; No. of pixels to page horizontally
        dc    i4'0'                 ; Information bar text string
        dc    i2'0'                 ; Info bar height
        dc    i4'0'                 ; DefProc field
        dc    i4'0'                 ; Routine to draw info bar
        dc    i4'PaintO'            ; Routine to draw content
        dc    i'26,0,188,308'       ; Size/position of content
        dc    i4'$FFFFFFFF'         ; Plane to put window in
        dc    i4'0'                 ; Address for record (0=to allocate)

WinOEnd    anop
```

Windows and GrafPorts

Before the NewWindow call returns, it creates a GrafPort for the window being set up and pushes a pointer to that GrafPort onto the stack. From that point, the application that created the window can treat it as a GrafPort. The application can draw into the window using QuickDraw II routines.

When the NewWindow call sets up a window, it uses the information passed in the window's parameter block to create the window's attributes. For example, the first field in the parameter block describes the window's frame—using the bit layout illustrated in table 10–2—and the second field

Table 10–3
Fields in a Window Parameter Block

Field	Name	Length	Description
1	paramLength	Word	Number of bytes in parameter table
2	wFrame	Word	Bit array describing window frame
3	wTitle	Pointer	Pointer to window's title
4	wRefCon	Long	Reserved for application's use
5	wZoom	Rect	Size and position of window when zoomed (0 = screen size)
6	wColor	Pointer	Pointer to window's color table
7	wYOrigin	Word	Content's vertical origin
8	wXOrigin	Word	Content's horizontal origin
9	wDataH	Word	Height of entire document or pixel image
10	wDataW	Word	Width of entire document or pixel image
11	wMaxH	Word	Maximum height of content allowed by GrowWindow
12	WMaxW	Word	Maximum width of content allowed by GrowWindow
13	wScrollVer	Word	Number of pixels to scroll document vertically using scroll bar arrows
13	wScrollHor	Word	Number of pixels to scroll document horizontally using scroll bar arrows
14	wPageVer	Word	Number of pixels to scroll vertically using page control
14	wPageHor	Word	Number of pixels to scroll horizontally using page control
15	wInfoRefCon	Long	Value passed to information bar draw routine
16	wInfoHeight	Word	Height of information bar
17	wFrameDefProc	Pointer	Address of standard window definition procedure
18	wInfoDefProc	Pointer	Address of routine that draws information bar interior
19	wContDefProc	Pointer	Address of routine that draws content region interior
20	wPosition	Rect	Window's starting position and size
21	wPlane	Long	Window's starting plane (FFFFFFFF = frontmost)
22	wStorage	Pointer	Address of memory to use for window record (0 = don't care)
23	paramLength	Word	Total number of bytes in parameter table, including this field

contains the window's title. In subsequent fields, the width and height of the window's data areas and content areas are defined. A data area is a rectangle that encloses all the data a window can work with (for example, a pixel map). A content area is a rectangle enclosing the largest portion of the data area that may be displayed on the screen.

Some fields in a window's parameter block duplicate fields in the window's window record. When a window is created using a `NewWindow` call, the call uses information provided in the window's parameter block to fill in the corresponding fields of the window's window record.

One very important field in a window parameter block is the fourth field from the end of the block. This field contains a pointer to a routine that is used to draw the contents of the window each time the window is displayed on the screen. In the WINDOW.S1 program, the field looks like this:

```
dc      i4'Paint0'              ; Routine to draw content
```

The routine that paints a window must be written according to a specific format, and must end with the assembly language mnemonic `rtl`.

In the `Paint0` segment of the WINDOW.S1 program, the QuickDraw call `PPToPort` copies the contents of a specific pixel map into the window used in the program. This pixel map is set up in a program segment called `MakeWin0` and is accessed in the program by the pointer `Pic0Ptr`.

The program segments `MakeWin0` and `Paint0` are in listing 10–5, the complete listing of the WINDOW.S1 program at the end of this chapter. Here is what happens in the segment of code labeled `Paint0`.

First, the Memory Manager call `NewHandle` reserves a 32K block of RAM—enough memory to hold a pixel map that is the size of one screen. The call returns with a handle to the requested block of data pushed onto the stack. This handle is then pulled off the stack and stored in a variable called `Win0Handle`. Later in the program, the `Paint0` routine uses the block of data pointed to by `Win0Handle` to draw the contents of the program's window on the screen.

When the handle called `Win0Handle` is assigned, a segment of code labeled `Deref` dereferences the handle (converts it into a pointer). The `Deref` routine also locks the handle being dereferenced so the Memory Manager can't move the handle's block of memory in the middle of an important operation, which could crash the program. Later, when the important operation is over, the `Unlock` routine unlocks the handle, enabling the Memory Manager to manage it again.

When `Win0Handle` is dereferenced, the pointer thus obtained is stored in a `LocInfo` data structure at the end of the WINDOW.S1 program in a field labeled `Pic0Ptr`. Then a `NewWindow` call creates a new window. To set the new window's attributes, the `NewWindow` call uses the parameter block in listing 10–3.

As explained previously, the WINDOW.S1 program allows you to draw into a screen window and, at the same time, to draw into the pixel map that paints the window on the screen each time it is updated or redrawn. This is why sketches drawn with the WINDOW.S1 program do not vanish from memory when a window is removed from the screen. Instead, they remain in RAM and can be redrawn into a window when it shows up again on the screen.

To make this technique possible, the WINDOW.S1 program creates a GrafPort that can be used to draw into the pixel map from which the program's window is drawn. This GrafPort is set up in the `NewPort` program segment. For its `LocInfo` data, the new GrafPort uses the `PicOLocInfo` data structure in the `PortData` data segment at the end of the program.

When the GrafPort that points to a pixel image is created, the WINDOW.S1 program clears the area of memory used for the pixel image with the `BlkFill` program segment. In this segment, the pen color is set to white and the QuickDraw call `PaintRect` clears the bit image to white. Later in the program, when the user asks for a new blank screen by making the menu choice `New`, the program uses the `BlkFill` routine to clear both the window port and the bit image port to white.

(Incidentally, the `PaintRect` call can be used to fill any block of RAM with any value, even in a nongraphics program. To "stuff" a block of memory, just pass to `PaintRect` the size of the area you want filled and the value you want it filled with. `PaintRect` does the rest—and you save the time and effort it would take to write a 65C816 block fill program.)

Window Manager's GrafPort

The WINDOW.S1 program, like every program that uses windows, has another GrafPort that is created by the Window Manager. When you use the Window Manager in a program, it always creates a special GrafPort that has the entire screen as its port rectangle. In all programs that use the Window Manager, this port is known as the Window Manager port. The Window Manager uses it to draw all windows, along with their scroll bars and other controls, on the IIGS screen.

How a Window Is Drawn

When the Window Manager draws or redraws a window, it always draws the window's frame first. Then it draws the window's contents.

During this process, the Window Manager manipulates regions of the Window Manager port as necessary to ensure that only what should be drawn is drawn. The Window Manager generates an *update event* to draw a window's contents. But before an update event can take place, the Window Manager must accumulate, in the update region, the areas of the window's content region that need updating.

In programs that use either TaskMaster or the Event Manager, the Event Manager periodically calls a routine called `CheckUpdate` to see if there is a window on the screen whose update region is not empty. If it finds one, it reports that an update event has occurred and passes a pointer to the window that needs updating in the event message field of its event record. If TaskMaster is used, it then updates the window as required. Programs that don't use TaskMaster have to do the updating themselves. Obviously, it's easier to use TaskMaster.

Some Window Manager routines can change the state of a window from inactive to active or from active to inactive. For each change, the Window Manager generates an *activate event,* passing along the window pointer in the event message. The `activeFlag` bit in the `modifiers` field of the event record is set if the window becomes active and cleared if it becomes inactive.

When the Event Manager finds out from the Window Manager that an activate event has been generated, it passes the event to the application or TaskMaster through its `GetNextEvent` routine. An activate event has the highest priority of any type of event, so when the Event Manager detects one it gets immediate action.

Usually, activate events are generated in pairs, because when one window becomes active another usually becomes inactive, and vice versa. Occasionally, however, a single activate event is generated, for example, when there is only one window in the window list or when an active window is closed permanently.

When a pair of activate events comes along, the Window Manager first generates the event for the window becoming inactive. It then generates the event for the window becoming active. In most applications, pairs of activate events are handled competently by TaskMaster. Rarely does an application program have to intervene.

Coordinates and the Window Manager

When `NewWindow` is called to create a window, it takes the window's bounds rectangle from the `LocInfo` field of the window's GrafPort. Thus, a window's local coordinates begin in the upper left corner of the bounds rectangle specified in the `LocInfo` field of the window's GrafPort. In a window's global coordinate system, coordinate 0,0 is always assigned to the pixel in the upper left corner of the window's bounds rectangle.

Global Coordinates in WINDOW.S1

In the WINDOW.S1 program, the `LocInfo` record that defines the window's bounds rectangle is titled `Pic0LocInfo`. This record is in a data segment labeled `PortData`, which appears at the end of the program. The bounds rectangle defined in the `Pic0LocInfo` record appears in the `Pic0Frame` field. In the WINDOW.S1 program, therefore, the bounds rectangle assigned to the program's window is the rectangle 0,0,200,320.

The global coordinates of a window are always based on a pixel image, specifically, the pixel image pointed to by the second field of the window's `LocInfo` record. In a window's global coordinate system, coordinate 0,0 is always assigned the pixel in the upper left corner of the window's pixel image.

Local Coordinates in WINDOW.S1

The pointer to the pixel image used in the WINDOW.S1 program is `Pic0Ptr`. This pointer is the second field in a `LocInfo` record called `Pic0LocInfo`. The `Pic0LocInfo` record is in a data segment called `PortData`, which appears at the end of the program.

Port Rectangle in WINDOW.S1

The port rectangle of a window is a rectangle outlining the maximum portion of the window that can be displayed on the screen at any given time. If a window is partially hidden (for example, partly covered by another window or partly off the screen), the window's visible region (`VisRgn`) is also used to determine how much of the window is visible on the screen. In the WINDOW.S1 program, the Window Manager takes care of VisRgns automatically. But, as you shall see shortly, the program has to perform a few manipulations using port rectangles.

Coordinate Conversions in WINDOW.S1

In programs like WINDOW.S1, coordinates often have to be converted from one system to another. Some QuickDraw and Window Manager routines use global coordinates, but others use local coordinates. For example, in the segment of the WINDOW.S1 program labeled MoveIt, TaskMaster returns mouse coordinates in global coordinates, and the Event Manager call GetMouse and the QuickDraw II call LineTo require local coordinates. For this reason, the QuickDraw call GlobalToLocal is used to convert the global coordinates returned by TaskMaster to the local coordinates required by other calls.

The MoveIt segment of the WINDOW.S1 program is the heart of the program. In this section, mouse movements are tracked and lines are drawn on the screen. TaskMaster detects the location of the IIGS mouse and returns it, in global coordinates, in the EventWhere field of its task record. The mouse location is then converted into local coordinates in these two lines:

```
PushLong #EventWhere
_GlobalToLocal
```

The GlobalToLocal call converts the global coordinates in the EventWhere record to local coordinates. After this conversion, the EventWhere field contains local coordinates, which can then be used by calls that require them. In the MoveIt segment, other conversions are taken care of by the StartDrawing and SetOrigin calls.

When a window is created, the upper left coordinate of its bounds rectangle are usually set to 0,0. Thus, in the local coordinate system used by a new window, the first pixel in its bounds rectangle is generally assigned the coordinate 0,0.

As you have seen, every window has both a port rectangle and a bounds rectangle. The intersection of a window's bounds rectangle and port rectangle make up the largest possible area of the window that can be displayed on the screen.

Suppose a window has a bounds rectangle that starts at local coordinate 0,0 and is the same size as the screen. Let's also suppose the window has a port rectangle that covers a smaller area in the middle of the screen. The coordinates of this port rectangle are 65,50 (the vertical coordinate is listed first). A bounds rectangle and a port rectangle that fit this description are illustrated in figure 10–3.

Now let's assume you want to use the WINDOW.S1 program to draw a sketch in the window (that is, in the port rectangle) shown in figure 10–3. You first have to convert the mouse location returned by TaskMaster from global coordinates to local coordinates. But, because of the way the IIGS Window Manager works, you also have to reset the origin of the window's

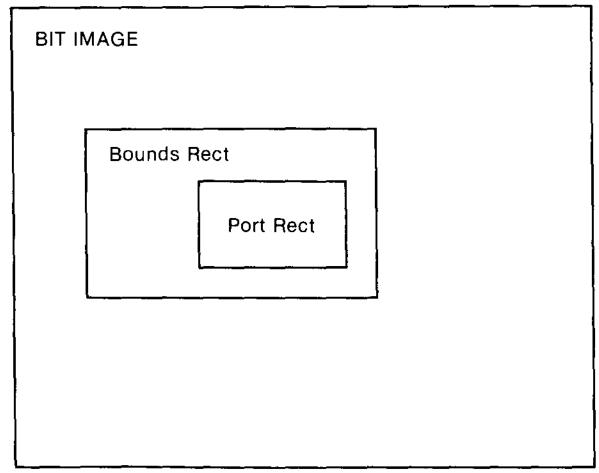

Figure 10–3
Relationship between a bit image, BoundsRect, and PortRect

port rectangle; you have to change the value of the upper left corner of the port rectangle, as expressed in local coordinates.

This is why the port rectangle's origin must be reset. When the Window Manager draws all the windows on a screen—complete with scroll bars, title bars, and all other necessary features—it uses a GrafPort that has the whole screen as its bounds rectangle. But before the Window Manager can draw the content region of a single window (for example, when the window has to be updated or redrawn), it has to switch to that window's GrafPort and change the origin of the window's port rectangle from its usual value of 0,0 to the value it had when it was a port rectangle in the Window Manager's GrafPort, which uses the whole screen as its bounds rectangle.

The logic of this procedure is a little difficult to follow. After the origin of a window's port rectangle is changed, the Window Manager can draw into the window, and the drawing ends up in the proper location on the screen.

When the Window Manager has finished drawing in a window, it must set the window's origin back to 0,0 before it can leave the window's port and return to its own GrafPort, so that it can regain the capability of drawing anywhere on the screen.

When the Window Manager has to draw in a window, it automatically carries out all the procedures just outlined. But when an application wants to draw in a window, it has to perform the same kinds of operations the Window Manager performs when it draws in a window.

To start drawing in a window, an application can use one of two approaches. It can either

- Make the QuickDraw call `SetPort` to make the window's port the current port and then make the QuickDraw call `Set Origin` with the proper parameters
- Make the Window Manager call `StartDrawing`, which carries out both of the previous steps automatically

The simpler approach is to use the StartDrawing call—and that is what is done in the WINDOW.S1 program.

After an application has finished drawing into a window, it must return the origin of the window's port rectangle to its original state by making the QuickDraw call SetOrigin using parameters 0,0.

Running the WINDOWS.S1 Program

After the procedure for drawing into a window is understood, the operation of the WINDOW.S1 program becomes straightforward. The main part of the WINDOWS.S1 program is MainProgram. In this section, the tools used by the program are initialized, a menu is constructed, and the MakeWin0 subroutine is called to create a window.

Next, the NewPort subroutine is called to set up a GrafPort used by the window's pixel map. Then the BlkFill subroutine is called to clear the pixel map to white. (You could clear the screen to another color by simply replacing the color code $FF in the BlkFill routine with a different color code.)

When the window's pixel image is cleared, the WINDOW.S1 program jumps to the EventLoop subroutine. This is the main event loop of the program. While the event loop is running, TaskMaster continuously looks for button down events. If TaskMaster detects a button down event, the program uses a jump table labeled TaskTable to determine what should be done.

If TaskMaster reports a menu event, the table called TaskTable sends the program to the doMenu subroutine. It is up to doMenu to carry out an appropriate response to the user's menu selection. Depending upon the menu choice, the doMenu routine can either call the Repaint subroutine to draw a new window, call the doWin0 subroutine to redraw a window, or jump to the doQuit subroutine to end the program.

If a window event is detected, TaskMaster takes care of all routine window-related operations, such as scrolling the window or changing its size. If TaskMaster detects a button down event in the window's go-away box, the program jumps to a short subroutine titled doGoAway, which hides the window. If TaskMaster reports a button down event in the window's content region, the program jumps to the MoveIt subroutine, which enables the user to draw in the window.

The MoveIt routine, as noted, is the heart of the WINDOW.S1 program. In this segment of code, as long as the mouse is inside a window and the mouse button is down, the QuickDraw call LineTo draws a line on the screen tracing the mouse's movements. When the mouse button is released, the mouse's movements are still followed, but the tracing is done using the MoveTo call rather than the LineTo call, so no line is drawn on the screen.

You can clear the window at any time by making the menu selection New. You can temporarily hide the window being drawn by clicking the mouse in the window's go-away region. If a window is hidden, but is not erased with a click in the menu item New, you can bring the window back into view by making the menu selection Untitled (for now, the title of the window). After New is selected, however, the window is permanently erased and cannot be retrieved from memory.

Other Features of WINDOW.S1

The WINDOW.S1 program has some new features that should be mentioned before you conclude this chapter. One is the `InsertSysDisk` subroutine, which is called from the `ToolInit` program segment. The other new and noteworthy feature is a macro called `ErrorCheck`, which is also called from the `ToolInit` segment of the program.

The `InsertSysDisk` subroutine is called when the WINDOW.S1 program tries to load the tools it needs and finds that the IIGS system disk—on which some tools are stored—is not currently in the computer's disk drive. When this condition is detected, `InsertSysDisk` is called and prints a message on the screen asking the user to insert the system disk in the disk drive.

The `ErrorCheck` macro is called following several critical routines, such as the loading of essential tools. If the calling of a vital routine is aborted by an error, the `ErrorCheck` macro ends the program. A system failure message—a rolling-Apple symbol accompanied by an error message and an error number—is displayed on the screen.

InsertSysDisk Routine

To see how the `InsertSysDisk` routine works, look through the `ToolInit` segment for the label `LoadEmUp`. Study the code that follows the labels `LoadEmUp` and `DoInsertDisk`, and you'll see that this section of code forms a loop. When the program comes to the `LoadEmUp` label, it makes the Tool Locator call `LoadTools` to load all the tools used in the program. The `LoadTools` call, like most Toolbox calls, uses a specific convention for detecting errors. If the call is completed successfully, without an error, it returns with the P register's carry flag clear and a value of 0 in the accumulator. If an error is encountered in making the call, however, the call returns with the carry bit set and an error number in the accumulator.

In the WINDOW.S1 program, if the `LoadTools` call returns without an error, the program jumps a few lines to a section of code labeled `ToolsLoaded` and the tools that have been loaded start up normally. If the call returns with the carry set and the number 45 in the accumulator, however, the program jumps to the `DoInsertDisk` subroutine, which prints a message on the screen asking the user to insert the IIGS system disk (which contains some of the tools used by the computer). If the user complies and the necessary tools are found, the program proceeds normally. If this doesn't solve the problem, the program ends and a system failure message is displayed.

ErrorCheck Macro

To end programs and display system-death messages after fatal errors occur, the WINDOW.S1 program uses the ErrorCheck macro. Several calls to the ErrorCheck macro appear in the ToolInit segment of the WINDOW.S1 program.

The ErrorCheck macro appears in listing 10–4. To use it in your programs, type it into a macro file and add it to your library of macros using APW's MACGEN shell command.

Listing 10–4
ErrorCheck

```
 MACRO
&lab ErrorCheck &msg
&lab bcc end&syscnt
 pea x&syscnt|-16
 pea x&syscnt
 ldx #$1503
 jsl $E10000
x&syscnt str "&msg"
end&syscnt anop
 MEND
```

The WINDOW.S1 and INITQUIT.S1 Programs

The WINDOW.S1 program, like the C language programs in the last few chapters, is divided into two parts: WINDOW.S1 and INITQUIT.S1. The WINDOW.S1 program, listing 10–5, and the INITQUIT.S1 program, listing 10–6, are at the end of this chapter.

Splitting a program into two or more parts can save a considerable amount of typing. For example, INITQUIT.S1—the portion of the program that loads, starts up, and shuts down tools—is also used in sample programs in chapters 11 and 12.

In programs written using the APW assembler-editor package, it's easy to divide a program into sections and then put all the sections together again at assembly time. All you have to do is type each section, save it as a separate source code file, and then combine the files you have saved using the APW assembler directive COPY. Look at the end of the WINDOW.S1 program in listing 10–5, and you'll see that the last line of the listing is

```
COPY INITQUIT.S1
```

When the APW assembler reaches that line, it starts assembling INITQUIT.S1 and adds it to WINDOW.S1, just as if the two listings were a single listing. Furthermore, any number of COPY directives can appear at the end of a source code listing. So you can add many modules to an APW program by using the COPY directive.

The WINDOW.C and INITQUIT.C Programs

The WINDOW.C program, listing 10–7, is a C language version of WINDOW.S1. It is designed to be used with the INITQUIT.C program, listing 10–8, which performs the same functions as INITQUIT.S1 and was introduced in chapter 9. THe WINDOW.C and INITQUIT.C programs appear at the end of this chapter.

WINDOW.C and INITQUIT.C are combined into one program with the statement

```
#include "initquit.c"
```

This statement is in the first line of the WINDOW.C program.

There are significant differences between WINDOW.C and its assembly language equivalent, WINDOW.S1. In WINDOW.C, for example, the `Sketch()` function, which draws on the screen, is simplified. It uses the function `StartDrawing()` just once, then it uses `SetPort()` thereafter. This is a more streamlined way to write the `Sketch()` routine in C, but the method used in WINDOW.S1 works better in assembly language. Experiment and you'll see why.

In WINDOW.C, the `ErasePicO()` function, which is called `repaint` in WINDOW.S1, is also simplified. Instead of completely dismantling a window environment and then rebuilding it (the technique used in WINDOW.S1) the `ErasePicO()` function keeps the window's environment, but simply erases what is in it. Because of differences in the way in which WINDOW.S1 and WINDOW.C work, this is another approach that works well in C, but the technique used in WINDOW.S1 works better in assembly language.

WINDOW.S1 and INITQUIT.S1 Listings

Listing 10–5
WINDOW.S1 program

```
*
*  WINDOW.S1
*

*** A FEW ASSEMBLER DIRECTIVES ***

                Title 'Window'
                ABSADDR on
                LIST off
                SYMBOL off
                65816 on
                mcopy window.macros

                KEEP window
```

```
*
*  EXECUTABLE CODE STARTS HERE
*

Begin           START
                Using QuitData

                jmp MainProgram          ; skip over data

                END

*
*  SOME DIRECT PAGE ADDRESSES AND A FEW EQUATES
*

DPData          START

DPTemp          gequ    $00
DPPointer       gequ    DPTemp+4
DPHandle        gequ    DPPointer+4

ScreenMode      gequ $00                ; 320 mode
MaxX            gequ 320                ; X clamp high

True            gequ $8000
False           gequ $00

                END

*
*  MAIN PROGRAM LOOP
*

MainProgram     START
                Using GlobalData
                Using PortData

                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 MakeWin0            ; create empty window
```

```
*** OPEN A PORT SO WE CAN DRAW IN WINDOW'S PIXEL MAP ***

                jsr NewPort

                lda #PicOPort
                sta BlkToFill
                lda #^PicOPort
                sta BlkToFill+2

                jsr BlkFill

*** LINE THAT JUMPS TO THE EVENT LOOP ***

                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           anop
                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
```

```
*
*  ROUTINE TO DRAW SKETCHES ON THE SCREEN
*
MoveIt          START
                Using EventData
                Using GlobalData
                Using PortData

                PushLong TaskData
                _StartDrawing
                PushLong #RectPtr
                _GetPortRect
                PushLong #EventWhere        ; convert them to
                _GlobalToLocal              ; local coordinates
                PushLong EventWhere         ; move cursor to mouse location
                _MoveTo

                pea 0
                pea 0
                _SetOrigin

                PushLong #Pic0Port
                _SetPort

                PushLong #RectPtr
                _ClipRect

                PushLong EventWhere
                _MoveTo

loop            pea 0                       ; space for return
                pea 0                       ; check button zero
                _StillDown
                pla
                beq out

                lda TaskData+2
                pha
                lda TaskData
                pha
                _StartDrawing

                lda #^EventWhere
                pha
                lda #EventWhere
                pha
                _GetMouse
```

```
                lda EventWhere+2
                pha
                lda EventWhere
                pha
                _LineTo

                pea 0
                pea 0
                _SetOrigin

                lda #^PicOPort
                pha
                lda #PicOPort
                pha
                _SetPort

                lda EventWhere+2
                pha
                lda EventWhere
                pha
                _LineTo

                brl loop

out             anop
                rts

RectPtr         ds 8

                END

*
*  REPAINT: MAKE NEW EMPTY WINDOW
*

Repaint          START
                 Using PortData
                 Using WindowData
                 Using GlobalData

                 PushLong #0
                 _GetPort
                 PullLong ThisPortPtr

                 PushLong #PicOPort
                 _SetPort

                 PushLong #ScreenRect
```

```
               _ClipRect

               lda #PicOPort
               sta BlkToFill
               lda #^PicOPort
               sta BlkToFill+2

               jsr BlkFill

               PushLong ThisPortPtr
               _SetPort

               PushLong WinOPtr
               _HideWindow

               PushLong WinOPtr
               _CloseWindow

               PushLong PicOHandle
               _DisposeHandle

               jsr MakeWinO
               jsr doWinO

               rts

ThisPortPtr    ds 4
ScreenRect     dc i'0,0,200,320'

               END

NewPort        START
               Using GlobalData
               Using PortData

               PushLong #0             ; space for result
               _GetPort
               PullLong OrigPortPtr    ; save pointer to current port

               PushLong #PicOPort      ; pointer to new port
               _OpenPort               ; open a port for pixel map

               PushLong #PicOPort      ; make new port the current
               _SetPort                ; port (temporarily)

               PushLong #ScreenRect
               _ClipRect
```

```
              PushLong #Pic0LocInfo   ; set up loc info for new port
              _SetPortLoc

              PushLong OrigPortPtr    ; make original port
              _SetPort                ; the current port again

              rts

ScreenRect    dc i'0,0,200,320'

              END

*
*  CREATE AND DRAW A WINDOW
*

MakeWin0       START
               using GlobalData
               using WindowData
               using PortData

*** SET HANDLE FOR PIC 0 (new) ***

               PushLong #$00
               PushLong #$8000        ; 32K (one screen)
               PushWord MyID
               PushWord #$C000        ; locked and fixed
               PushLong #0
               _NewHandle

               ErrorCheck 'Could not get handle.'

               pla
               sta Pic0Handle
               pla
               sta Pic0Handle+2

*** DEREF HANDLE, CLEAR MEMORY, AND CREATE POINTER ***

              lda Pic0Handle          ; lock and deref Pic0Handle
              ldx Pic0Handle+2        ; while we do our thing with it
              jsr Deref

              sta Pic0Ptr             ; deref gives us a pointer
              stx Pic0Ptr+2           ; to Pic0Handle's pixel map
*                                     ; so we'll save it
```

```
*** SET UP WINDOW 0 ***

               PushLong #0             ; space for result
               PushLong #Win0ParamBlock
               _NewWindow

               pla
               sta Win0Ptr
               pla
               sta Win0Ptr+2

               rts

               END

* DoWin0
* Selects and shows window 0 (blank) in response to menu selection.

DoWin0         START
               using GlobalData
               using WindowData

               PushLong Win0Ptr
               _SelectWindow

               PushLong Win0Ptr
               _ShowWindow

               rts

               END

*
* Paint0
* Draws empty window when TaskMaster calls.
*

Paint0         START
               using GlobalData
               Using PortData
               using WindowData

               phb
               phk
               plb

               phd
               lda MyDP
```

```
                tcd

                PushLong #Pic0LocInfo
                PushLong #Pic0Frame
                PushWord #0
                PushWord #0
                PushWord #0
                _PPToPort

                pld
                plb
                rtl

                END

***  BLOCK FILL ROUTINE ***

BlkFill         START
                Using GlobalData
                Using WindowData
                Using PortData

                PushLong #0
                _GetPort
                PullLong OrigPortPtr

                PushLong BlkToFill
                _SetPort

                PushWord #$FF
                _SetSolidPenPat

                PushLong #ARect
                _PaintRect

                _PenNormal

                PushLong OrigPortPtr
                _SetPort

                rts

OrigPortPtr     ds 4
ARect           dc i'0,0,200,320'

                END
```

```
*
*  CREATE AND DRAW MENU
*

BuildMenu       START
                using MenuData          ; proceeding from back to front

                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 #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
                cmp #256
                bcc GiveUp                ; this should never happen
```

```
                and #$00FF
                asl a
                tax

                jsr (MenuTable,x)

GiveUp          anop
                PushWord #False          ; draw normal
                PushWord TaskData+2      ; which menu
                _HiliteMenu

                rts

                END

*
* InsertSysDisk
* This routine is called when tools need to be loaded and the
* system disk is offline.  Routine asks user to insert system disk.
*

InsertSysDisk    START

                _SetPrefix SetPrefixParams
                _GetPrefix GetPrefixParams

                PushWord #0             ; space for result
                PushWord #195           ; x pos
                PushWord #30            ; y pos
                PushLong #PromptStr     ; prompt string
                PushLong #VolStr        ; vol string
                PushLong #OKStr
                PushLong #CancelStr
                _TLMountVolume

                pla

                rts

PromptStr       str 'Please insert the disk'

VolStr          ds 16

OKStr           str 'OK'

CancelStr       str 'Shutdown'

GetPrefixParams dc i'7'
```

```
                dc i4'VolStr'

SetPrefixParams dc i'7'
                dc i4'BootStr'

BootStr         str '*/'

                END

*
*  WINDOW GO-AWAY ROUTINE
*

doGoaway        START
                Using EventData

                PushLong TaskData
                _HideWindow
                rts

                END

*
*  A USEFUL AND CONVENIENT WAY NOT TO DO ANYTHING
*

Ignore          START

                rts

                END

*
* Deref
* Derefs the handle passed in a and x registers.
* Result passed back in a and x registers.
*

Deref           START
                sta DPTemp
                stx DPTemp+2
                ldy #2
                lda [DPTemp],y
                tax
                lda [DPTemp]
                rts

                END
```

```
*
*  DATA SEGMENTS
*

*
* Menu Data
*

MenuData        DATA

Return          equ 13

Menu1           dc c'>L@\XN1',i1'RETURN'
                dc c' LA Window Program \N257',i1'RETURN'
                dc c'.'

Menu2           dc c'>L File \N2',i1'RETURN'
                dc c' LNew \N258V',i1'RETURN'
                dc c' LQuit \N259',i1'RETURN'
                dc c'.'

Menu3           dc c'>L Windows \N3',i1'RETURN'
                dc c' LUntitled \N260',i1'RETURN'
                dc c'.'

                END
***

MenuTable       DATA

*               Menu 1 (apple)
                dc i'ignore'            ; one for the NDAs
                dc i'ignore'            ; 'a window program'

*               Menu 2 (file)
                dc i'Repaint'           ; 'doWin0' (new window)
                dc i'doQuit'            ; quit item selected

*               Menu 3 (windows)
                dc i'doWin0'            ; 'untitled'

                END
```

```
***

TaskTable       DATA

                dc i'ignore'            ; 0 null
                dc i'ignore'            ; 1 mouse down
                dc i'ignore'            ; 2 mouse up
                dc i'ignore'            ; 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

*
* TaskMaster events
*

                dc i'DoMenu'            ; 1 in menu bar
                dc i'ignore'            ; 2 in system window
                dc i'MoveIt'            ; 3 in content of window (MoveIt)
                dc i'ignore'            ; 4 in drag
                dc i'ignore'            ; 5 in grow
                dc i'doGoAway'          ; 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

***

ToolTable       DATA

                dc i'8'                 ; number of tools in table
                dc i'$04,$0100'         ; quickdraw
```

```
                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'$14,$0000'
                dc i'$15,$0000'
                dc i'$17,$0000'         ; std file manager

                END

***

EventData       DATA

EventRecord     anop                    ; table for Event Manager
EventWhat       ds 2
EventMessage    ds 4
EventWhen       ds 4
EventWhere      ds 4
EventModifiers  ds 2
TaskData        ds 4
TaskMask        dc i4'$0FFF'

                END

***

QuitData        DATA

QuitFlag        ds 2

QuitParams      dc  i4'0'
                dc  i4'0'
                dc  i4'0'

                END

***

WindowData      DATA

Pic0Handle      ds  4
Win0Ptr         ds  4
Win0Title       str 'Untitled'
```

```
WinOParamBlock anop

         dc     i'WinOEnd-WinOParamBlock'
         dc     i2'%1101110111000000' ; Bits describing frame
         dc     i4'WinOTitle'         ; Pointer to title
         dc     i4'0'                 ; RefCon
         dc     i2'26,0,188,308'      ; Full Size (0= default)
         dc     i4'0'                 ; Color Table Pointer
         dc     i2'0'                 ; Vertical origin
         dc     i2'0'                 ; Horizontal origin
         dc     i2'200'               ; Data Area Height
         dc     i2'320'               ; Data Area Width
         dc     i2'200'               ; Max Cont Height
         dc     i2'320'               ; Max Cont Width
         dc     i2'2'                 ; No. of pixels to scroll vertically
         dc     i2'2'                 ; No. of pixels to scroll horizontally
         dc     i2'20'                ; No. of pixels to page vertically
         dc     i2'32'                ; No. of pixels to page horizontally
         dc     i4'0'                 ; Infomation bar text string
         dc     i2'0'                 ; Info bar height
         dc     i4'0'                 ; DefProc
         dc     i4'0'                 ; Routine to draw info bar
         dc     i4'PaintO'            ; Routine to draw content
         dc     i'26,0,188,308'       ; Size/position of content
         dc     i4'$FFFFFFFF'         ; Plane to put window in
         dc     i4'0'                 ; Address for window record (0 to
                                      ; allocate)
*
WinOEnd    anop

              END

***

GlobalData     DATA

BlkToFill      ds 4

MyID           dc   i'0'              ; program ID word
MyDP           ds   2

BlockSize      ds   4
FilVal         ds   2

               END
```

```
***
PortData        DATA

OrigPortPtr     ds 4                        ; pointer to original port

PicOPort        ds $AA

PicOLocInfo     dc i'$00'                   ; 320 mode
PicOPtr         ds 4                        ; MakeWinO fills this in
                dc i'160'                   ; width
PicOFrame       dc i'0,0,200,320'           ; pic image frame rect

                END

*
* IOData
*
IOData          DATA

ReplyRecord     anop
GoodFlag        ds 2
FType           dc i'193'                   ; $c1
AuxFType        dc i'0'                     ; #0
FName           ds 15
FullPathName    ds 128

CreateParams    anop
NameC     dc    i4'0'
          dc    i2'$00C3'                   ; DRNWR
CType     dc    i2'$00C1'                   ; super high-res graphics
CAux      dc    i4'$00000000'               ; Aux
          dc    i2'$0001'                   ; type
          dc    i2'$0000'                   ; create date
          dc    i2'$0000'                   ; create time

DestParams      anop
NameD           dc     i4'0'

OpenParams      anop
OpenID          ds 2
NamePtr         ds 4
                ds 4

ReadParams      anop
ReadID          ds 2
PicDestIN       ds 4
                dc i4'$8000'          ; this many bytes
                ds 4                  ; how many xfered
```

```
WriteParams    anop
WriteID        ds 2
PicDestOUT     ds 4
               dc i4'$8000'        ; this many bytes
               ds 4                ; how many xfered

CloseParams    anop
CloseID        ds 2

               END

***

               COPY INITQUIT.S1
```

Listing 10–6
INITQUIT.S1 program

```
*
*  INITQUIT.S1: WHERE WE INITIALIZE OUR TOOLS
*

ToolInit       START
               Using GlobalData
               Using ToolTable

*** START UP TOOL LOCATOR ***

               _TLStartup                 ; Tool Locator

*** INITIALIZE MEMORY MANAGER ***

               PushWord #0
               _MMStartup
               ErrorCheck 'Could not init Memory Manager.'

               pla
               sta MyID

*** INITIALIZE MISC. TOOL SET ***

               _MTStartup
               ErrorCheck '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

               ErrorCheck '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
               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               ; X clamp low
               PushWord #MaxX            ; X clamp high
               PushWord #0               ; Y clamp low
               PushWord #200             ; Y clamp high
               PushWord MyID
               _EMStartup
               ErrorCheck 'Could not start Event Manager.'

*** LOAD SOME TOOLS FROM RAM ***

LoadEmUp       PushLong #ToolTable
               _LoadTools
               bcc ToolsLoaded

               cmp #$45                  ; prodos error: vol not found
               beq doInsertDisk
```

```
               sec
               ErrorCheck 'Could not load tools.'

DoInsertDisk   anop
               jsr InsertSysDisk
               cmp #1
               beq LoadEmUp
               sec
               ErrorCheck 'Tool loading aborted.'

*** WINDOW MANAGER ***

ToolsLoaded    PushWord MyID
               _WindStartup
               ErrorCheck 'Could not Start Window Manager.'

               PushLong #$0000
               _Refresh

*** CONTROL MANAGER ***

               PushWord MyID
               lda DPPointer            ; DP to use = qd dp + $400
               clc
               adc #$400
               pha
               _CtlStartup
               ErrorCheck 'Could not start Control Manager.'

*** MENU MANAGER ***

               PushWord MyID
               lda DPPointer            ; DP to use = qd dp +$500
               clc
               adc #$500
               pha
               _MenuStartup
               ErrorCheck 'Could not start Menu Manager.'

               _ShowCursor

*** LINE EDIT ***

               PushWord MyID
               lda DPPointer
               clc
               adc #$600                ; qd dp + $600
               pha
```

```
               _LEStartup
               errorcheck 'Could not start up Line Edit.'

*** DIALOG MANAGER ***

               PushWord MyID
               _DialogStartup
               errorcheck 'Could not start Dialog Manager.'

*** STANDARD FILE MANAGER ***

               PushWord MyID
               lda DPPointer
               clc
               adc #$700                 ; qd dp + $700
               pha
               _SFStartup
               errorcheck 'Could not start up SF Manager.'

               rts

               END

*
*  THE ROUTINE THAT ENDS THE PROGRAM
*

EndIt          START

               Using QuitData

               _Quit QuitParams

*** A QUIT CALL SHOULDN'T RETURN; IF IT DOES, WE'RE FINI ***

               ErrorCheck 'We returned from a quit call!'

               END

*
*  SHUT DOWN ALL THE TOOLS WE STARTED UP
*

ShutDown       START
               Using GlobalData
               Using WindowData

               _SFShutdown
```

```
                _DialogShutdown
                _LEShutdown
                _MenuShutDown
                _CtlShutDown
                _WindShutDown
                _EMShutDown
                _QDShutDown
                _MTShutDown

                PushLong DPHandle
                _DisposeHandle

                PushLong PicOHandle
                _DisposeHandle

                PushWord MyID
                _MMShutDown
                _TLShutDown

                rts

                END

*
*  ROUTINE THAT SETS THE QUIT FLAG
*

doQuit          START
                Using QuitData

                lda #$8000
                sta QuitFlag
                rts

                END
```

WINDOW.C and INITQUIT.C Listings

Listing 10–7
WINDOW.C program

```
#include "initquit.c"

*************************************/
/* Data and routine to create menus */
/*************************************/
```

```
/*  Set up menu strings. Because C uses \ as an escape character, we use
two when we want a \ as an ordinary character. The \ at the end of each
line tells C to ignore the carriage return. This lets us set up our items
in an easy-to-read vertical alignment.  */

char *menu1 = "\
>L@\XN1\r\
 LA Window Program \\N257\r\
.";

char *menu2 = "\
>L File \\N2\r\
 LNew  \\N258V\r\
 LQuit \\N259\r\
.";

char *menu3 = "\
>L Windows \\N3\r\
 LUntitled \\N260\r\
.";

#define QUIT_ITEM 259   /* these will help us check menu item numbers */
#define QUIT_ITEM 259   /* these will help us check menu item numbers */
#define NEW_ITEM 258
#define UNTIT_ITEM 260

BuildMenu()
{
   InsertMenu(NewMenu(menu3),0);
   InsertMenu(NewMenu(menu2),0);
   InsertMenu(NewMenu(menu1),0);
   FixMenuBar();
   DrawMenuBar();
}


/*********************************************************************/
* Data structures and routine to set up offscreen drawing environment */
/*********************************************************************/

LocInfo pic0LocInfo = { mode320,
                          NULL, /* space for pointer to pixel image */
                      160,  /* width of image in bytes = 320 pixels */
                      0,0,200,320  /* frame rect */
                    };

Rect screenRect = {0,0,200,320};
GrafPort pic0Port;
```

```
#define IMAGE_ATTR attrLocked+attrFixed+attrNoCross+attrNoSpec+attrPage

Pic0Setup()  /* called once by MakeWindow at start of program */
{
GrafPortPtr thePortPtr;

   pic0LocInfo.ptrToPixImage = *(NewHandle(0x8000L,myID,IMAGE_ATTR,NULL));
   thePortPtr = GetPort();
   OpenPort(&pic0Port);
   SetPort(&pic0Port);
   SetPortLoc(&pic0LocInfo);
   ClipRect(&screenRect);
   EraseRect(&screenRect);
   SetPort(thePortPtr);
}

/***************************************************/
/* Data structures and routine to create window    */
/***************************************************/

/* Initialize template for NewWindow  */

#define FRAME fQContent+fMove+fZoom+fGrow+fBScroll+fRScroll
+fClose+fTitle

ParamList template = { sizeof (ParamList),
                        FRAME,
                        "\pUntitled",    /* pointer to title */
                        0L,             /* RefCon           */
                        26,0,188,308, /* full size (0=default)  */
                        NULL,           /* use default ColorTable */
                        0,0,            /* origin           */
                        200,320,        /* data area height & width */
                        200,320,        /* max cont height & width  */
                        2,2,    /* ver & hor scroll increment */
                        20,32,  /* ver & hor page increment  */
                        NULL,   /* no info bar text string  */
                        0,      /* info bar height = none   */
                        NULL,   /* default def proc         */
                        NULL,   /* no info bar draw routine */
                        NULL,   /* draw content must be filled in
                          at run time */
                        26,0,188,308, /* starting content rect */
                        -1L,    /* topmost plane */
                        NULL    /* let window manager allocate record */
                      };
```

```
/* Window's draw content routine */
pascal void DrawContent()
{
   PPToPort(&pic0LocInfo,&(pic0LocInfo.boundsRect),0,0,modeCopy);
}

GrafPortPtr win0Ptr;

MakeWindow() /* complete template, make (the window,
 and setup offscreen port */
{
   template.wContDefProc = DrawContent;
   win0Ptr = NewWindow(&template);
   Pic0Setup(); /* create offscreen image for use by DrawContent */
}

/*******************************************************************/
/* Main routine. Set up environment, call eventloop, and shut down */
/*******************************************************************/
main()
{
   StartTools();
   BuildMenu();
   MakeWindow();
   EventLoop();
   DisposeHandle(FindHandle(pic0LocInfo.ptrToPixImage));
   ShutDown();
}

/**************************************/
/* Event loop and supporting routines */
/**************************************/
WmTaskRec   myEvent;
Boolean done = false;

EventLoop()
{
   myEvent.wmTaskMask = 0x0FFF;
   while(!done)
      switch ( TaskMaster(everyEvent,&myEvent)) {
         case wInMenuBar:
            DoMenus();
            break;
         case wInGoAway:
            HideWindow(win0Ptr);
            break;
         case wInContent:
            Sketch();
```

```
    }
}

DoMenus()
{
Word *data = (Word *)&myEvent.wmTaskData; /* address of item id  */

    switch(*data) {
      case QUIT_ITEM:
         done = true;
         break;
      case NEW_ITEM:
         ErasePic0();
         HideWindow(win0Ptr);
         CloseWindow(win0Ptr);
         win0Ptr = NewWindow(&template);
      case UNTIT_ITEM:
         SelectWindow(win0Ptr);
         ShowWindow(win0Ptr);
         break;
    }
    HiliteMenu(false,*(data + 1)); /* data + 1 is address of menu id */
}

ErasePic0()
{
GrafPortPtr oldPortPtr;

   oldPortPtr = GetPort();
   SetPort(&pic0Port);
   ClipRect(&screenRect);
   EraseRect(&screenRect);
   SetPort(oldPortPtr);
}

Sketch() /* sketch into current port and into offscreen port */
{
Point mouseLoc;
GrafPortPtr thePortPtr = (GrafPortPtr)myEvent.wmTaskData;
Rect theRect;

   mouseLoc = myEvent.wmWhere;

   StartDrawing(thePortPtr);  /* set up correct drawing coordinate
    system */
   GetPortRect(&theRect);     /* copy current Port Rect           */
   GlobalToLocal(&mouseLoc);  /* get cursor pos in local coordinates */
```

```
   MoveTo(mouseLoc);         /* set pen position to mouse loc   */
   SetPort(&picOPort);       /* switch to offscreen port       */
   ClipRect(&theRect);       /* clip offscreen drawing to window's
    port rect */
   MoveTo(mouseLoc);         /* set offscreen pen to same location  */
   SetPort(thePortPtr);      /* switch back to window's port    */

   while (StillDown(0)) {
      GetMouse(&mouseLoc);     /* get new mouse coordinates */

      LineTo(mouseLoc);        /* draw line in both ports */
      SetPort(&picOPort);
      LineTo(mouseLoc);
      SetPort(thePortPtr);
   }
   SetOrigin(0,0);             /* restore normal coordinates */
}
```

Listing 10–8
INITQUIT.C program

```
#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>
#include <LINEEDIT.H>
#include <DIALOG.H>

#define MODE mode320 /* 640 graphics mode def. from quickdraw.h */
#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 */
Handle zp;           /* handle for page 0 space for tools */

int ToolTable[] = {7,
                   4, 0x0100,  /* QD        */
                   6, 0x0100,  /* Event     */
                  14, 0x0100,  /* Window    */
                  16, 0x0100,  /* Control   */
                  15, 0x0100,  /* Menu      */
```

```
                20, 0x0100,  /* Line Edit */
                21, 0x0100,  /* Dialog    */
                };


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(0x700L,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));
   LEStartUp(myID, (int) (*zp + 0x600));
   DialogStartUp(myID);
   ShowCursor();
}


ShutDown()             /* shut down all of the tools we started */
{
   GrafOff();
   DialogShutDown();
   LEShutDown();
   MenuShutDown();
   CtlShutDown();
   WindShutDown();
   EMShutDown();
   QDShutDown();
   MTShutDown();
```

```
    DisposeHandle(zp); /* release our page 0 space */
    MMShutDown(myID);
    TLShutDown();
}
```

CHAPTER 11

Dialog with a IIGS

Using the Dialog Manager

The main channel of communication between the Apple IIGS and its user is handled by a tool set known as the Dialog Manager. When a program needs to inform the user of something important or give the user guidance—or when a program needs to obtain information from the user—the Dialog Manager provides the interface between computer and user.

The Dialog Manager communicates with the IIGS user through *dialog windows*—boxes that are usually programmed to appear on the screen when they are needed. Dialog windows can display messages, obtain user input, or both. They can contain icons, pictures, text, and user-operated controls. Some icons can stay on the screen for a long time and can be moved around. Others remain in one spot until they are deactivated and then go away as quickly as they appeared.

In this chapter, you'll take a look at various kinds of dialog windows, and you'll see how dialogs can be used in IIGS programs.

What Dialog Windows Look Like

Dialog windows resemble ordinary document windows, but they have controls that ordinary windows usually do not have. A dialog window usually appears near the top of the screen, in the center of the screen and slightly below the menu bar, and is somewhat narrower than the screen. Figure 11–1 shows a typical dialog window.

As figure 11–1 shows, a dialog window looks something like a printed form. Like a paper form, a dialog can contain messages, illustrations, and blanks to be filled in by the user. These features can be presented in many formats, such as

- Messages designed to provide the user with information, instructions, or alerts.
- Controls such as buttons, scroll bars, and squares that can be checked off by the user. Text messages may or may not be supplied along with these controls.
- Rectangles in which the user may type in text. These rectangles, called edit lines, may be blank when they appear on the screen or they may contain default text that can be edited by the user.
- Graphic symbols: either icons or pictures drawn using QuickDraw. Icons are easier to manage than QuickDraw pictures and are thus more commonly used. But there is no reason why a QuickDraw picture can't appear in a dialog window.
- Any other types of items an application can define.

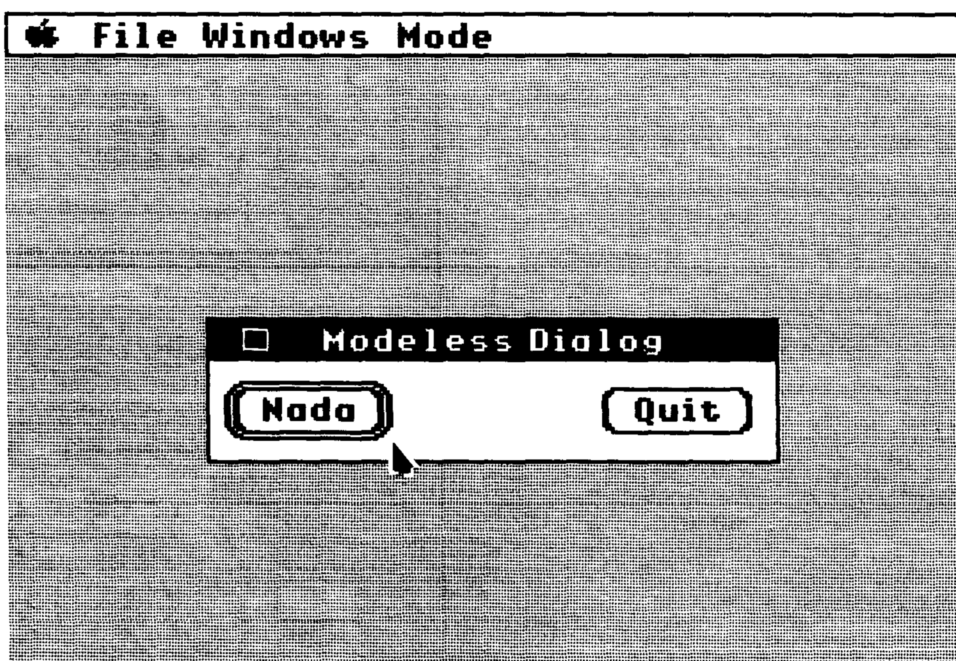

Figure 11–1
Typical dialog window

Dialog I/O

The simplest kind of dialog window is one that requires no response at all. Such a noninteractive dialog might be created to print a message on the screen while an application is performing a time-consuming process. When the operation is finished, the dialog could be removed from the screen.

Another simple type of dialog is one that contains just two items: a printed message and one button, often labeled OK, that the user can press after reading the message. In most cases, the dialog in which the message appears then disappears from the screen.

The button that makes the dialog disappear does not have to be labeled OK. It could be labeled Start or Proceed, or it could have another name. But, for simplicity, we call this button the OK button throughout this chapter.

Many kinds of dialog windows can be used in IIGS programs. Some dialog windows display more than one message on the screen, some display different messages at different times, and some accept input from the user. For example, if a dialog window appears on the screen as the result of some action by the user, it might contain a button labeled Cancel that is clicked to cancel the action that caused the dialog to appear. Or there could be a button labeled Help that is used to request additional information.

Dialog Items

In Dialog Manager jargon, buttons with labels like OK, Cancel, and Help are known as *dialog items*. There are many kinds of dialog items, and each is designed to be used in a slightly different way. Some dialog items provide information to the user, some obtain information from the user, and some do both. The items that can be used in dialog windows can be divided into the following categories:

- Button items. A button item is a simulated pushbutton that contains a label such as OK, Help, or Cancel. A button item usually has round corners and usually contains a label displayed in the standard IIGS type font, or system font. When the user clicks the IIGS mouse inside a button item, an application program can carry out whatever response is appropriate.
- Check items. A check item is a small square box that is empty or contains an *X*. When the user clicks the mouse in an empty check item, an *X* appears. When the user clicks the mouse in a check item that contains an *X*, the *X* disappears.

  A dialog box can contain any number of check items. When a dialog with a user ends, the application using the dialog can check to see which boxes have been checked and which have been left unchecked, and take the appropriate actions.

- Radio items. A radio item is a small circle that is empty or contains a still smaller circle. The inner circle in a radio item is usually black. When the user clicks the mouse in an empty radio item, an inner circle appears. When the user clicks the mouse in a radio item that contains an inner circle, the inner circle disappears.

- Scroll bar items. A scroll bar item is a special scroll bar used only in dialogs. A scroll bar item can be used to display the progress of an operation. For example, the white square, or ''thumb'' of a scroll bar, can move down the bar as files are printed to show the user how the operation is progressing.

- Static text items. A static text item, usually abbreviated StatText item, consists only of a Pascal-type string (a length byte followed by a string of ASCII characters). StatText items only display information; they cannot accept input from the user. Text in a StatText item does not have to be enclosed in a visible rectangle, and it cannot be edited.

- Long static text items. A long static text item, abbreviated LongStatText item, consists only of a block of text. The text in a LongStatText item is not preceded by a length byte, so its length must be passed to the Dialog Manager as a parameter when the item is created with a `NewDItem` call. More about this call is provided later in this chapter. LongStatText items only display messages; they cannot accept input from the user. Text in a LongStatText item does not have to be enclosed in a visible rectangle, and it cannot be edited.

- Edit line items. An edit line item contains space for one line of text that is entered or edited by the user. The text usually appears inside a visible rectangle. When an edit line item appears on the screen, it is empty or contains default text. If it is empty, you can fill it in by typing information, and you can edit the information after it has been typed. If the item contains default text when it appears on the screen, that text can be edited by the user.

- Icon items. An icon item contains an icon. Icons used in dialog windows are stored in memory in a specific format and appear in the dialog window when it is displayed on the screen. When the user clicks the mouse in an icon item, the application using the dialog can take whatever action is appropriate.

- Picture items. A picture item contains a picture drawn with QuickDraw II. When the user clicks the mouse in a picture item, the application using the dialog can take whatever action is appropriate.

- User items. Any item that is not in any of the previous categories is called a user item. User items are defined by application programs.

Types of Dialog Windows

There are three kinds of dialog windows: modal dialogs, modeless dialogs, and alert dialogs. Let's take a closer look at each of these types of dialog windows.

Modal Dialogs

Modal dialogs require the user to respond to a dialog message before taking any other action. Modal dialogs derive their name from the fact that they put a program in a state, or mode, of being unable to take any action outside a dialog window. A modal dialog usually has at least one button item that is clicked to perform some action and a Cancel button that is clicked to make the dialog box go away. Normally, clicking the mouse anywhere outside the dialog window only makes the IIGS speaker beep.

In programs written according to Apple's *Human Interface Guidelines,* one button item in a dialog window may be outlined in bold; that is, it may have a double outline. If such a button appears in a dialog box, it is usually the OK button, the button that ends the dialog by initiating some action and making the dialog window go away. When a button has a double outline, the Return key on the keyboard can always be pressed as an alternative to clicking the outlined button. In short, a button with a double outline is the dialog's default button—the safest button to use in the current situation. If there is no boldly outlined button, pressing the Return key will have no effect on the dialog. A typical modal dialog window is illustrated in figure 11–2.

Modeless Dialogs

A dialog cannot be modal and modeless at the same time; different routines create these two types of dialogs. When a program is running, however, it can be difficult to distinguish between a modal dialog and a modeless dialog because they often look alike.

A *modeless dialog,* like a modal dialog, usually has an OK button (often doubly outlined) and a Cancel button. And, just like a modal dialog, a modeless dialog can contain other controls that do not erase the dialog window and do not result in any change in a program until an OK button is pressed to make the dialog go away.

But modeless dialogs do not put a program into any special state, or mode, and thus do not require the user to respond to a dialog before taking any other action. When a modeless dialog is on the screen, it can stay there while the user performs actions unrelated to the dialog. For example, the user might be permitted to work in various windows on the desktop before clicking a button in the dialog window.

Because a modeless dialog can remain on the screen while document windows (or even other dialog windows) are in use, you can create a modeless dialog window that has a title bar and thus can be moved on the screen. Because of this feature—and because they can stay on the screen while various operations take place—modeless dialogs are used as desk accessories. Clocks, calculators, notepads, and other desk accessory items are often incorporated into programs in the form of modeless dialogs.

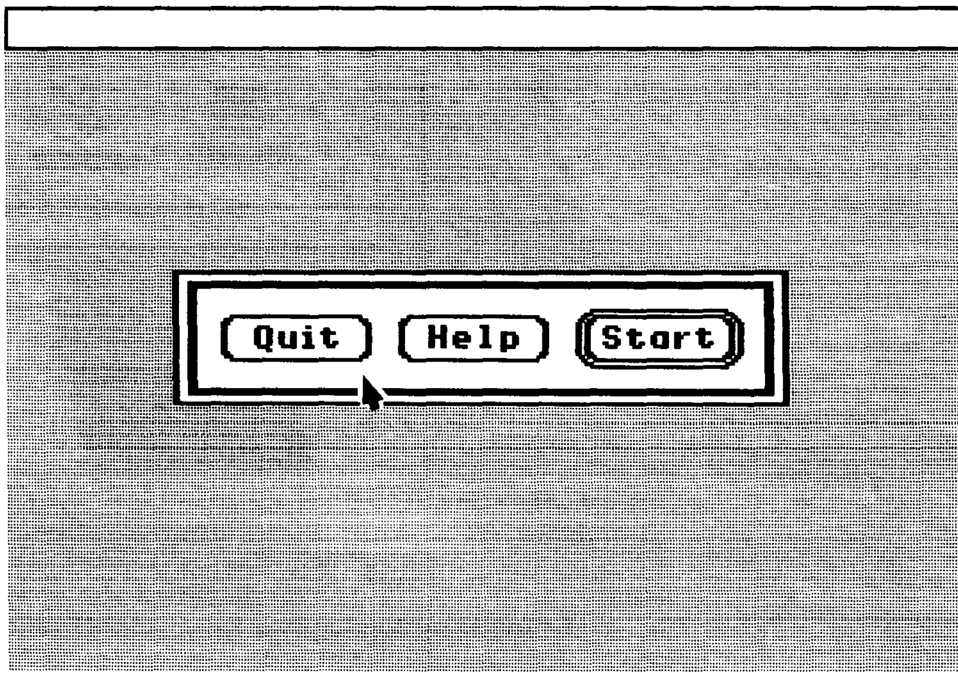

Figure 11–2
Modal dialog window

Figure 11–3 shows a modeless dialog box that is similar to a document window. Like a standard document window, it has both a title bar and a close box. So it can be moved, hidden, closed, and opened again, like any other similarly equipped window.

Alert Dialogs

An *alert dialog* looks much like a modal dialog (or a modeless dialog without a title bar). But an alert dialog has a special function. It appears only when something has gone wrong or when something important must be brought to the user's attention. Alert dialogs can provide a program with a convenient method for reporting errors or issuing warnings.

An alert window is usually placed slightly farther below the menu bar than a modal or modeless dialog. And an alert dialog often contains an icon that gives the user a visual clue about the nature of the alert. There are three standard types of alert icons: Stop, Note, and Caution. You can also design other kinds of icons. An alert dialog can also be programmed to beep or make other sounds when it is activated.

To help the user who isn't sure how to proceed when an alert box appears, the button used most often in the current situation is displayed with a double outline. This button is also the alert's default button. If the user presses the Return key, the effect is the same as clicking the alert's default button.

One special feature of an alert dialog is that it can behave in a different way each time it is activated. This feature can give the user increasingly

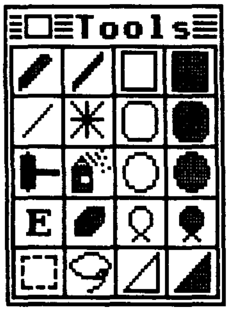

Figure 11–3
Modeless dialog window

severe warnings each time an error is made or a dangerous situation becomes more dangerous. For example, the first time an error is made, the error might beep the speaker but generate no alert box. Thereafter, each successive error might cause an alert dialog to be displayed, and each alert might carry an increasingly severe warning.

Furthermore, the sound produced by an alert dialog does not have to be a beep. It can be any sequence of tones, which may occur either by themselves or with an alert dialog. Figure 11–4 is an illustration of a typical alert dialog window.

Manipulating Dialog Windows

After a modal or modeless dialog is created, it can be manipulated like any other window. With the help of routines provided by the Window Manager and QuickDraw, an application can do just about anything to a dialog window: show, hide, or move it, change its size or plane, or close and discard it when it is no longer needed. The Dialog Manager even recognizes the `ClipRgn` field of the dialog window's GrafPort, so the QuickDraw II `SetClipRgn` and `ClipRect` routines can keep portions of a window from being displayed on the screen.

When an alert window is designed, however, the Dialog Manager takes care of most details, so that all alert windows have a standard appearance and behavior. The size and location of the box are supplied as part of the definition of the alert and are not changed easily. You do not have to specify an alert window's plane because an alert always appears in front of all other windows. After an alert window is on the screen, the application that uses it never has to manipulate it. That's because an alert window requires the user to respond before doing anything else, and the user's response makes the box disappear.

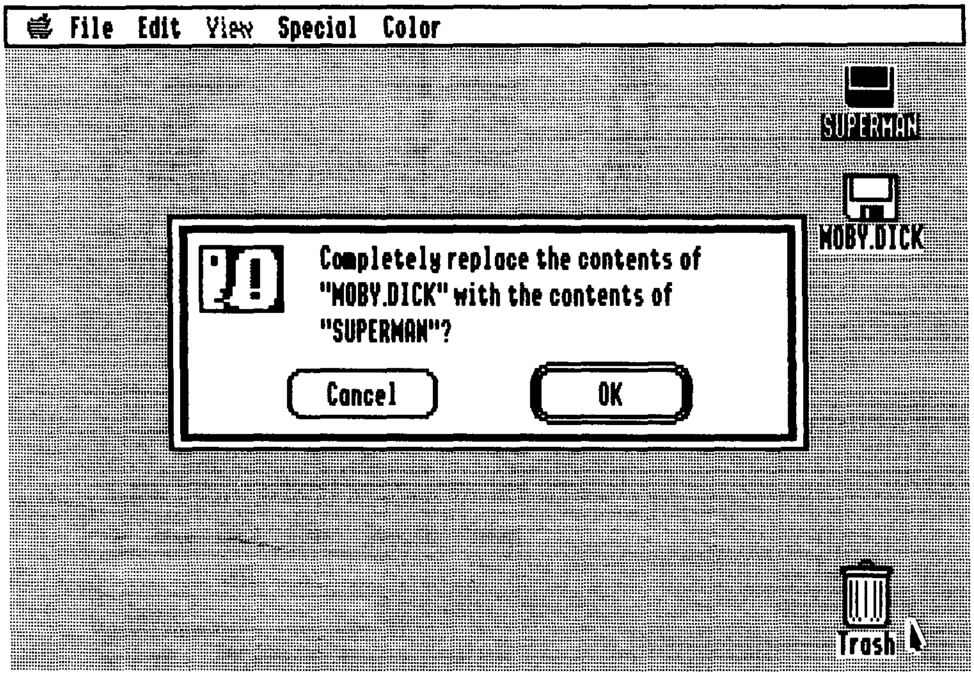

Figure 11–4
Alert dialog window

Initializing the Dialog Manager

Before the dialog is started, the following tool sets must be loaded and started:

- Tool Locator (always loaded and active)
- Memory Manager
- Miscellaneous Tool Set
- QuickDraw II
- Event Manager
- Window Manager
- Control Manager
- LineEdit Tool Set

After these tools are loaded and initialized, the `DialogStartUp` call can be made to start up the Dialog Manager. If you want the type font used in your dialog and alert windows to be something other than the system font, you can make the Dialog Manager call `SetDAFont`.

When the Dialog Manager is loaded and started up, the `NewModalDialog`, `NewModelessDialog`, and `GetNewModalDialog` calls can be used to create dialog windows. `NewModelessDialog` creates a dialog using a special kind of dialog record, and `GetNewModelessDialog` creates a dialog using a template that can be accessed by more than one dialog window.

After a dialog is set up, the `NewDItem` and `GetNewDItem` calls can be used to create the items that appear in each dialog. The `CloseDialog` call can be used to close and dispose of any dialogs.

Creating a Dialog Window

The Dialog Manager requires the same kind of information to create a dialog that the Window Manager requires to create a document window. These are the steps that are usually used to set up a dialog window:

1. The application calls `NewModalDialog`, `GetNewModalDialog`, or `NewModelessDialog`. In addition to creating a dialog window, these calls determine how the window looks and behaves.
2. The Dialog Manager must be supplied with a rectangle that becomes the port rectangle of the window's GrafPort.
3. The Dialog Manager must be told whether the window will be visible or invisible when it is created. If it is created as a visible window, it appears on the screen immediately. If it is created as an invisible window, the Window Manager calls `SelectWindow` and `ShowWindow` must be made each time the window appears on the screen.

If a modeless dialog is created, the plane in which it appears in relation to other windows must also be specified. By convention, a newly created window always appears in the frontmost plane.

The example program in this chapter, DIALOG.S1, uses the call `NewModalDialog` to create a modal dialog window. Listing 11–1 shows how `NewModalDialog` is used in the program. Instructions for typing and compiling the DIALOG.S1 program in both assembly language and C are at the end of this chapter.

Listing 11–1
Calling the NewModalDialog routine

```
PushLong #0              ; output
PushLong #DRect
PushWord #True           ; visible
PushLong #0              ; refcon
_NewModalDialog

pla
sta MDialogPtr
pla
sta MDialogPtr+2
```

As listing 11–1 shows, the `NewModalDialog` call takes four parameters:

- 2 null words (zeros), which provide a space on the stack for a 2-word result.
- A pointer to a rectangle that defines the location of the dialog window on the screen.
- A 1-word space for a Boolean value. If the value is nonzero, or true, the dialog is displayed on the screen as soon as it is created. If the value is zero, the window is not displayed until a specific command, such as `ShowWindow`, is called to display it on the screen.

When a `NewModalDialog` call returns, a pointer to the dialog window which it created is on the stack. In the DIALOG.S1 program, this pointer is stored in the `MDialogPtr` variable.

Creating an Item List

Before a dialog window can be displayed on the screen, the `NewDItem` call must be used to create each item that will appear in the window. The dialog window in the DIALOG.S1 program contains three buttons: Start, Quit, and Help. Listing 11–2 shows how the `NewDItem` call creates the Start button.

Listing 11–2
NewDItem call

```
PushLong MDialogPtr       ; item belongs to this window
PushWord #1               ; item ID number
PushLong #ButtonRect1     ; pointer to button's rect
PushWord #ButtonItem      ; item type
PushLong #ButtonText1     ; item descriptor
PushWord #0               ; item's initial value
PushWord #0               ; visible/invis flag
PushLong #0               ; color table pointer
_NewDItem
```

As listing 11–2 shows, the `NewDItem` call takes eight parameters:

- A pointer to the window to which the item belongs.
- A 1-word identification number that will be used in all dialog-related items to identify the item being created.
- A pointer to a rectangle that defines where the item will appear inside its dialog window. Note that this rectangle is expressed not in screen coordinates, but in local coordinates that treat the dialog window as a bounds rectangle.

- A 1-word parameter identifying the type of item being created. This parameter is a constant that can be found in APW's LIBRARIES/AINCLUDE file, under the filename E16.DIALOG. In the DIALOG.S1 program, the constants for item types are listed in the `DialogData` data segment.

  By convention, the OK button in an alert's item list is always assigned an ID of 1, and the Cancel button should always have an ID of 2. The Dialog Manager provides predefined constants equal to the item ID for OK and Cancel as follows:

```
OK        equu    1
Cancel    equ     2
```

  In a modal dialog's item list, the item whose ID is 1 is generally assumed to be the dialog's default button. If the user presses the Return key, the Dialog Manager normally returns the ID of the default button, just as when that item is actually clicked.

  To conform with Apple's *Human Interface Guidelines,* the Dialog Manager automatically prints a double outline in bold around the default button, unless there is no default button—that is, no button item with an ID number of 1. So, if you don't want a dialog to have a default button, you should not assign any button an ID number of 1. The item types listed in the DIALOG.S1 program are shown in listing 11–3.

- A two-word parameter called a dialog item descriptor. The function of this parameter can vary, depending upon the type of item being created. Table 11–1 shows the functions the item descriptor parameter can have when used with different kinds of items.
- A one-word parameter setting the initial value of the item descriptor, if applicable.
- A flag determining whether the item being created should be visible or invisible when the window is first displayed. This parameter can also include item-specific information, for example, the family number of a radio button or whether a scroll bar is horizontal or vertical. Further information on item-specific data in this parameter is in the *Apple IIGS Toolbox Reference.*
- A pointer to a color table, which can be used to change the standard colors used to draw items in a dialog. Custom color tables can be used for standard or custom-designed controls. But make sure your use of color conforms to Apple's *Human Interface Guidelines.*

Listing 11–3
Item types in DIALOG.S1

```
DialogData      DATA

ButtonItem      equ   10
CheckItem       equ   11
RadioItem       equ   12
```

```
ScrollBarItem  equ   13
UserCtlItem    equ   14
StatText       equ   15
EditText       equ   16
EditLine       equ   17
IconItem       equ   18
PicItem        equ   19
UserItem       equ   20

               END
```

Table 11–1
Item Descriptor Parameter in a NewDItem Call

Type	Function of Descriptor	Value
ButtonItem	Pointer to a string containing item's label	N/A
CheckItem	N/A	0 = not checked 1 = checked
RadioItem	N/A	0 = not checked 1 = checked
ScrollBarItem	Pointer to dialog scroll bar action procedure	0 or default value if ItemDescr = 0
UserCtlItem	Pointer to control definition procedure	Initial value of control
UserCtlItem2	Pointer to parameter block	Initial value of control
StatText	Pointer to static string	Application use
LongStatText	Pointer to the beginning of text	Length of text (0 to 32,767)
EditLine	Pointer to default string	Maximum length allowed (0 to 255)
IconItem	Handle to the icon	Application use
PicItem	Handle to the picture	Application use
UserItem	Pointer to item definition procedure	Application use

Using a Dialog Window in a Program

When a modal dialog is created, the ModalDialog call can be used to accept user input. Listing 11–4 shows how the ModalDialog call is used in the DIALOG.S1 program. Let's take a look now at how the routine in listing 11–4 works. Then we'll see how the routine is used in the DIALOG.S1 program.

Listing 11–4
ModalDialog call

```
Again       PushWord #0             ; space for result
            PushLong #0             ; filter procedure pointer
            _ModalDialog
            pla

next        cmp #3
            beq Again

            cmp #1
            beq noquit

button2     lda #$FFFF              ; button 2 was pressed

            sta QuitFlag

noquit      PushLong MDialogPtr     ; use this exit for #1 or #3
            _CloseDialog

            rts
```

The `ModalDialog` call takes two parameters: a 1-word null (zero) value that saves a space on the stack and a pointer to a user-written filter procedure, if there is one. A filter procedure, usually abbreviated `FilterProc`, is a routine that an application can call to filter out unwanted responses by the user (for example, to ignore non-numeric characters typed in an `EditLine` item that calls for numeric characters only). If a 0 is passed to `ModalDialog` in the `FilterProc` parameter, it means no filter process is set up by the application using the dialog. In that case, `ModalDialog` will not look for one.

In the DIALOG.S1 program, `ModalDialog` is called with two 0 parameters: a null word to save a space on the stack and a null pointer because there is no filter procedure in the program.

When a `ModalDialog` call returns, a 1-word value—the ID number of the item selected by the user—is pushed on the stack. In the DIALOG.S1 program, this value is pulled off the stack and compared with the literal values 3 and 1. If the value is 3—the item ID number for the Help button—the program loops back to the line labeled `Again`. That's because no help function is written for the DIALOG.S1 program. If you expand the program, you may want to write a help function.

If the `ModalDialog` call returns a value of 1—the item ID number of the Start button—the dialog is erased from the screen with a `CloseDialog` call and the DIALOG.S1 program continues, as though there had never been a dialog window on the screen.

If the routine in listing 11–4 discovers that the user has clicked a button that is neither item 1 nor item 3, it is smart enough to determine that the user

If the routine in listing 11–4 discovers that the user has clicked a button that is neither item 1 nor item 3, it is smart enough to determine that the user has made the only other choice, item 2. This is the Quit button, which ends the program by storing a nonzero value in the program's `quit` flag before returning.

The DIALOG.S1 Program

DIALOG.S1 is an expanded version of the WINDOW.S1 program in chapter 10, so you can save yourself a lot of work by modifying WINDOW.S1 instead of typing the entire DIALOG.S1 program. To convert WINDOW.S1 into DIALOG.S1, the following modifications are necessary:

1. Replace the heading of the WINDOW.S1 program with the heading shown in listing 11–5.
2. Add three lines to the main program segment of the WINDOW.S1 program so that the segment looks like the one in listing 11–6.
3. Following the program segment labeled `EventLoop`, insert the segment that appears in listing 11–7. This segment displays a dialog window on the screen.
4. In the data segment labeled `MenuData`, change the line

   ```
   dc c' LA Window Program \N257',i1'RETURN'
   ```

   to

   ```
   dc c' LA Dialog Program \N257',i1'RETURN'
   ```

5. At the end of the program, add the data segment that appears in listing 11–8. This segment provides the item codes used in the DIALOG.S1 program.
6. Before you assemble DIALOG.S1, make sure you have the latest version of INITQUIT.S1 saved on the same disk that holds your DIALOG.S1 source code. Then the `COPY` directive at the end of DIALOG.S1 will combine the DIALOG.S1 and INITQUIT.S1 programs.

When you've typed, assembled, and executed DIALOG.S1, you'll be ready to examine the portion that creates a dialog on the screen. Starting from the beginning of the DIALOG.S1 program, move down the listing until you see the label `Main Program`. Below that label look for this line:

```
jsr doDialog1
```

If you have typed and run the program, you should have no trouble figuring out what this line does. After all tools are initialized and an empty menu bar appears on the screen, the line `jsr doDialog1` simply places a modal dialog on the screen and waits for the user's input. The user can do one of three things: click Start, which erases the dialog box and resumes execution of the DIALOG.S1 program, click Help, which won't do anything because there is no help routine, or click Quit, which ends the program.

Listing 11–5
Heading segment

```
*
*  DIALOG.S1
*

*** A FEW ASSEMBLER DIRECTIVES ***

                Title 'Dialog'
                ABSADDR on
                LIST off
                SYMBOL off
                65816 on
                mcopy dialog.macros

                KEEP dialog
```

Listing 11–6
Main loop segment

```
*
*
*   MAIN PROGRAM LOOP

MainProgram     START
                Using GlobalData
                Using PortData

                phk
                plb
                tdc                     ; get current direct page
                sta MyDP                ; and save it for the moment

                jsr ToolInit            ; start up all tools we'll need

*** PUT DIALOG NO. 1 ON THE SCREEN ***

                jsr doDialog1

                jsr BuildMenu           ; create and draw menu bar
                jsr MakeWin0            ; create empty window
```

```
*** OPEN A PORT SO WE CAN DRAW IN WINDOW'S PIXEL MAP ***

                jsr NewPort

                lda #PicOPort
                sta BlkToFill
                lda #^PicOPort
                sta BlkToFill+2

                jsr BlkFill

*** LINE THAT JUMPS TO THE EVENT LOOP ***

                jsr EventLoop          ; check for key & mouse events

*** WHEN EVENT LOOP ENDS, WE'LL SHUT DOWN ***

                jsr Shutdown
                jmp Endit

                END
```

Listing 11–7
Dialog window segment

```
*
*   DODIALOG1: PRINT DIALOG NO. 1 ON THE SCREEN
*

doDialog1      START
               using GlobalData
               using WindowData
               using DialogData
               using QuitData

               PushLong #0             ; output
               PushLong #DRect
               PushWord #True          ; visible
               PushLong #0             ; refcon
               _NewModalDialog

               pla
               sta MDialogPtr
               pla
               sta MDialogPtr+2

               PushLong MDialogPtr     ; item belongs to this window
               PushWord #1             ; item ID number
```

```
                PushLong #ButtonRect1     ; pointer to button's rect
                PushWord #ButtonItem      ; item's id number
                PushLong #ButtonText1     ; item descriptor
                PushWord #0               ; item's initial value
                PushWord #0               ; visible/invis flag
                PushLong #0               ; color table pointer
                _NewDItem

                PushLong MDialogPtr
                PushWord #2
                PushLong #ButtonRect2
                PushWord #ButtonItem
                PushLong #ButtonText2
                PushWord #0
                PushWord #0
                PushLong #0
                _NewDItem

                PushLong MDialogPtr
                PushWord #3
                PushLong #ButtonRect3
                PushWord #ButtonItem
                PushLong #ButtonText3
                PushWord #0
                PushWord #0
                PushLong #0
                _NewDItem

Again           PushWord #0                ; space for result
                PushLong #0                ; filter procedure pointer
                _ModalDialog
                pla

next            cmp #3
                beq Again

                cmp #1
                beq noquit

button2         lda #$FFFF                 ; button 2 was pressed
                sta QuitFlag

noquit          PushLong MDialogPtr        ; use this exit for #1 or #3
                _CloseDialog

                rts

DRect           dc i'84,63,114,252'         ; screen coordinates
```

```
ButtonRect1     dc i'8,129,22,179'          ; local coordinates using
ButtonRect2     dc i'8,8,22,58'             ; dialog window's frame
ButtonRect3     dc i'8,67,22,117'           ; as a bounds rectangle

ButtonText1     str 'Start'
ButtonText2     str 'Quit'
ButtonText3     str 'Help'

MDialogPtr      ds 4

                END
```

Listing 11–8
DialogData segment

```
DialogData     DATA

ButtonItem      equ    10
CheckItem       equ    11
RadioItem       equ    12
ScrollBarItem   equ    13
UserCtlItem     equ    14
StatText        equ    15
EditText        equ    16
EditLine        equ    17
IconItem        equ    18
PicItem         equ    19
UserItem        equ    20

               End
```

The DIALOG.C Program

Listing 11–9, DIALOG.C, is a C language version of the DIALOG.S1 program. It is designed to be used with the `include` file INITQUIT.C, and it works just like DIALOG.S1.

Listing 11–9
DIALOG.C program

```
#include "initquit.c"

Boolean done = false;
WmTaskRec   my Event;
```

```
/************************************/
/* Data and routine to create menus   */
/************************************/
/* Set up menu strings. Because C uses \ as an escape character, we use
two when we want a \ as an ordinary character. The \ at the end of each
line tells C to ignore the carriage return. This lets us set up our items
in an easy-to-read vertical alignment. */

char *menu1 = "\
>L@\\XN1\r\
 LA Window Program \\N257\r\
.";

char *menu2 = "\
>L File \\N2\r\
 LNew \\N258V\r\
 LQuit \\N259\r\
.";

char *menu3 = "\
>L Windows \\N3\r\
 LUntitled \\N260\r\
.";

#define QUIT_ITEM 259   /* these will help us check menu item numbers */
#define NEW_ITEM 258
#define UNTIT_ITEM 260

BuildMenu()
{
   InsertMenu(NewMenu(menu3),0);
   InsertMenu(NewMenu(menu2),0);
   InsertMenu(NewMenu(menu1),0);
   FixMenuBar();
   DrawMenuBar();
}

/*********************************************************************/
/* Data structures and routine to set up offscreen drawing environment */
/*********************************************************************/
LocInfo pic0LocInfo = { mode320,
                        NULL, /* space for pointer to pixel image */
                        160,  /* width of image in bytes = 320 pixels */
                        0,0,200,320  /* frame rect */
                      };

Rect screenRect = {0,0,200,320};
GrafPort pic0Port;
```

```
#define IMAGE_ATTR attrLocked+attrFixed+attrNoCross+attrNoSpec+attrPage


PicOSetup()  /* called once by MakeWindow at start of program */
{
GrafPortPtr thePortPtr;

   picOLocInfo.ptrToPixImage = *(NewHandle(0x8000L,myID,IMAGE_ATTR,NULL));
   thePortPtr = GetPort();
   OpenPort(&picOPort);
   SetPort(&picOPort);
   SetPortLoc(&picOLocInfo);
   ClipRect(&screenRect);
   EraseRect(&screenRect);
   SetPort(thePortPtr);
}

/***************************************************/
/* Data structures and routine to create window    */
/***************************************************/

/* Initialize template for NewWindow  */

#define FRAME fQContent+fMove+fZoom+fGrow+fBScroll+fRScroll+fClose+fTitle

ParamList template = { sizeof (ParamList),
                         FRAME,
                         "\"Untitled",    /* pointer to title */
                         0L,            /* RefCon           */
                         26,0,188,308, /* full size (0=default)  */
                         NULL,          /* use default ColorTable */
                         0,0,           /* origin           */
                         200,320,       /* data area height & width */
                         200,320,       /* max cont height & width  */
                         2,2,     /* vertical & horizontal scroll increment */
                         20,32,   /* vertical & horizontal page increment */
                         NULL,    /* no info bar text string  */
                         0,       /* info bar height = none   */
                         NULL,    /* default def proc         */
                         NULL,    /* no info bar draw routine */
                         NULL,    /* draw content must be filled in at run time */
                         26,0,188,308, /* starting content rect */
                         -1L,     /* topmost plane */
                         NULL     /* let Window Manager allocate record */
                       };
```

```
/* Window's draw content routine */

pascal void DrawContent()
{
  PPToPort(&pic0LocInfo,&(pic0LocInfo.boundsRect),0,0,modeCopy);
}

GrafPortPtr win0Ptr;

MakeWindow() /* complete template, make window, and set up offscreen port */
{
   template.wContDefProc = DrawContent;
   win0Ptr = NewWindow(&template);
   Pic0Setup(); /* create offscreen image for use by DrawContent */
}

/************************************************/
/* Data and routine to set up and display dialog */
/************************************************/

ItemTemplate item1 = {1,{8,129,22,179},buttonItem,"\pStart\r",0,0,NULL };
ItemTemplate item2 = {2,{8,8,22,58},buttonItem,"\pQuit\r",0,0,NULL };
ItemTemplate item3 = {3,{8,67,22,117},buttonItem,"\pHelp\r",0,0,NULL};

DialogTemplate dtemp = {{84,63,114,252},true,0L,&item1,&item2,&item3,NULL} ;

DoDialog()   /* Create and display an opening dialog box */
{
GrafPortPtr dlgPtr;
Word hit;

  dlgPtr = GetNewModalDialog(&dtemp);

  while ((hit = ModalDialog(NULL)) == 3);
  done = (hit == 2);
  CloseDialog(dlgPtr);
}

/*******************************************************************/
/* Main routine. Set up environment, call event loop, and shut down */
/*******************************************************************/

main()
{
   StartTools();
   DoDialog();
   BuildMenu();
```

```
   MakeWindow();
   EventLoop();
   DisposeHandle(FindHandle(pic0LocInfo.ptrToPixImage));
   ShutDown();
}


/*************************************/
/* Event loop and supporting routines */
/*************************************/

EventLoop()
{
   myEvent.wmTaskMask = 0x0FFF;
   while(!done)
      switch ( TaskMaster(everyEvent,&myEvent)) {
         case wInMenuBar:
            DoMenus();
            break;
         case wInGoAway:
            HideWindow(win0Ptr);
            break;
         case wInContent:
            Sketch();
     }
}

DoMenus()
{
Word *data = (Word *)&myEvent.wmTaskData; /*address of item id  */

    switch(*data) {
     case QUIT_ITEM:
         done = true;
         break;
      case NEW_ITEM:
         ErasePic0();
         HideWindow(win0Ptr);
         CloseWindow(win0Ptr);
         win0Ptr = NewWindow(&template);
      case UNTIT_ITEM:
         SelectWindow(win0Ptr);
         ShowWindow(win0Ptr);
         break;
    }
    HiliteMenu(false,*(data + 1)); /* data + 1 is address of menu id  */
}
```

```
ErasePic0()
{
GrafPortPtr oldPortPtr;

   oldPortPtr = GetPort();
   SetPort(&pic0Port);
   ClipRect(&screenRect);
   EraseRect(&screenRect);
   SetPort(oldPortPtr);
}

Sketch() /* sketch into current port and into offscreen port */
{
Point mouseLoc;
GrafPortPtr thePortPtr = (GrafPortPtr)myEvent.wmTaskData;
Rect theRect;

   mouseLoc = myEvent.wmWhere;

   StartDrawing(thePortPtr); /* set up correct drawing coordinate system */
   GetPortRect(&theRect); fr /* copy current Port Rect            */
   GlobalToLocal(&mouseLoc); /* get cursor pos in local coordinates */

   MoveTo(mouseLoc);         /* set pen position to mouse loc  */
   SetPort(&pic0Port);       /* switch to offscreen port      */
   ClipRect(&theRect);       /* clip offscreen drawing to window's Port Rect */
   MoveTo(mouseLoc);         /* set offscreen pen to same location  */
   SetPort(thePortPtr);      /* switch back to window's port   */

   while (StillDown(0)) {
      GetMouse(&mouseLoc);   /* get new mouse coordinates */

      LineTo(mouseLoc);      /* draw line in both ports */
      SetPort(&pic0Port);
      LineTo(mouseLoc);
      SetPort(thePortPtr);
   }
   SetOrigin(0,0);           /* restore normal coordinates */
}
```

CHAPTER 12

The Standard File Operations Tool Set

And ProDOS 16

Until the advent of the Apple IIGS, it could be difficult to incorporate disk drive operations into assembly language programs. Today, in programs written for the IIGS, the job is much easier. Here are four major reasons.

The Apple IIGS has new features that earlier Apple II computers do not have. For example, the Memory Manager tool set relieves the programmer of the responsibility of dealing with absolute addresses. It also has a new kind of I/O port, a SmartPort, which keeps track of the locations of disk drives and supports named devices and multiple, user-defined file prefixes.

The disk operating system in the IIGS is ProDOS 16—a 16-bit descendent of ProDOS 8, which was designed for the Apple IIe and the Apple IIc. ProDOS 16 is faster, more powerful, and easier to use than its 8-bit predecessor. And, unlike ProDOS 8, ProDOS 16 makes use of several new features of the IIGS.

The APW assembler-editor has a library of ProDOS macros that simplify the job of making ProDOS calls. In this chapter, you'll see how those macros are used.

The Standard File Operations Tool Set, which is included in the IIGS Toolbox, makes the task of working with ProDOS 16 even easier. When the Standard File Operations Tool Set is used in a program, a special dialog box is created every time a file is loaded or saved. You can load or save the file by either clicking the mouse inside a button item or typing the name of the file in a line edit control. You can also search through directories using the

Standard File Tool Set's dialog boxes, and you can even switch disks and change directories. The tool set gives the programmer the option of using predesigned dialog boxes or creating custom-designed boxes. Application programs can select the types of files that will or will not be listed on the screen.

In this chapter, you'll see how easy it is to create, load, save, and edit files using ProDOS 16, the ProDOS macros in the APW assembler-editor package, and the IIGS Standard File Operations Tool Set. These techniques are demonstrated using a sample program called SF.S1, which is listed at the end of this chapter. A C language version, SF.C, is also listed at the end of this chapter. Figure 12–1 shows the Standard File Tool Set screen display.

Introducing ProDOS 16

If you have written Apple II programs using ProDOS 8, you probably won't have any trouble understanding ProDOS 16. ProDOS 16 calls are made in the same way as ProDOS 8 calls: by filling in a block of parameters, pushing the address of the parameter block onto the stack, and jumping to a fixed entry point.

There are two important differences in the way calls are made in ProDOS 8 and ProDOS 16. In ProDOS 16, a program must jump to the ProDOS entry point with a `jsl` instruction rather than a `jsr` instruction, and the entry point is in bank $E1 rather than bank $00. In programs written using the APW

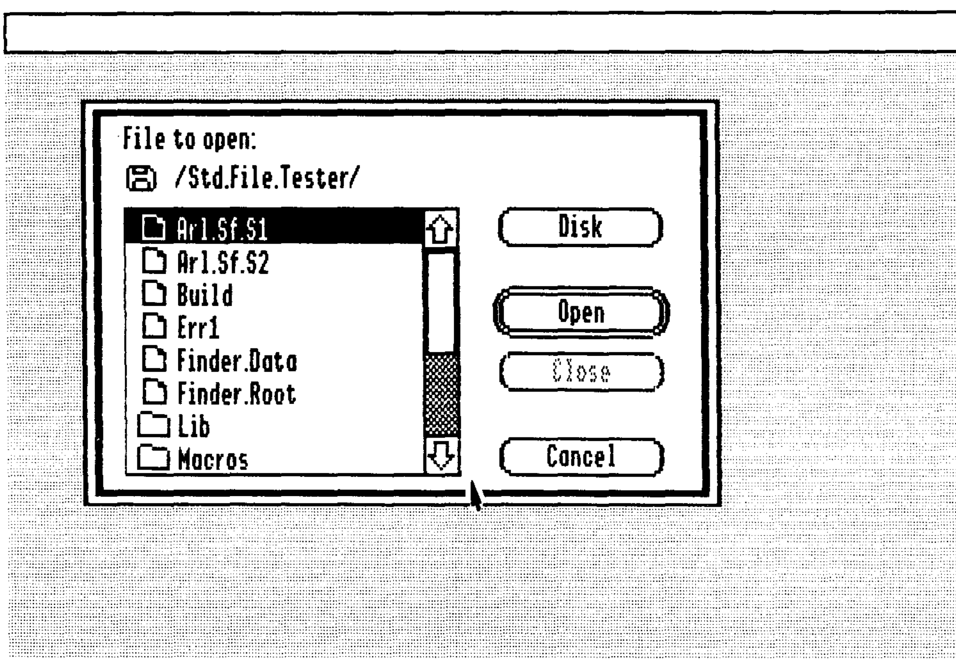

Figure 12–1
Standard File Operations Tool Set screen display

library of ProDOS macros, neither of these details makes any difference; the macros take care of them.

The kernel (or central part) of the Apple IIGS operating system is ProDOS 16, which is covered in detail in the *Apple IIGS ProDOS 16 Reference*. ProDOS accesses the disk drive or disk devices on which files are stored and retrieved and manages the creation and modification of files. ProDOS 16 also controls certain features of the IIGS operating environment, such as pathname prefixes and procedures for quitting programs and starting new ones.

ProDOS 16 can communicate with various disk drives, including hard disk drives, 5.25-inch floppy disk drives, and 3.5-inch disk drives. Because the IIGS has an intelligent disk port called a SmartPort, programs that use ProDOS 16 do not have to specify a disk's slot number or drive number to access the disk. Under ProDOS 16, a disk can also be accessed by its volume name or device name.

In ProDOS 16, just as in ProDOS 8, disks are also known as volumes, and information on a volume is divided into files. A file is an ordered sequence of bytes that has several attributes, including a name and a file type.

There are two primary types of files in ProDOS 16: standard files and directory files. Directory files contain the names and disk locations of other files. When a volume is formatted, a volume directory file is placed on it. The volume directory has the same name as the volume and usually contains the names and disk locations of other directory files.

ProDOS 16 supports a hierarchical file system. In a hierarchical file structure, volume directories can contain the names of other directories, called subdirectories, and subdirectories can, in turn, contain the names of other files or subdirectories.

In ProDOS 16, a file is identified by its pathname: a sequence of filenames starting with the name of the volume directory and ending with the name of the file. A pathname that begins with the name of a volume is a full pathname and is always preceded by a slash (/). If the name of the volume in which a file is stored is known, the file can be referenced by a partial pathname: a pathname that is not preceded by a slash and does not include a volume name.

Whether a pathname is preceded by a slash or not, the names of the directories, subdirectories, and files in the pathname are all separated by slashes. More details about pathnames are in the *Apple IIGS ProDOS 16 Reference*.

Loading a File with ProDOS 16

The SF.S1 program contains three code segments that make calls to ProDOS 16: `EndIt`, `LoadOne`, and `SaveOne`. `EndIt` makes the ProDOS call `Quit` to end the program. `LoadOne` appears in listing 12–1. `SaveOne` is explained shortly.

Listing 12–1
Loading a file using ProDOS 16

```
LoadOne         START
                using IOData

                _Open OpenParams
                bcc cont1
                ErrorCheck 'Could not open picture file.'

cont1           anop
                lda OpenID
                sta ReadID
                sta CloseID

                _Read ReadParams
                bcc cont2
                ErrorCheck 'Could not read picture file.'

cont2           anop
                _Close CloseParams

                clc
                rts

OpenParams      anop
OpenID          ds 2
NamePtr         ds 4
IOBuffer        ds 4

ReadParams      anop
ReadID          ds 2
PicDestIN       ds 4
                dc i4'$8000'            ; this many bytes
                ds 4                    ; how many xfered

CloseParams     anop
CloseID         ds 2

                END
```

In listing 12–1, the APW macro `Open` opens a file, the `Read` macro copies it into memory, and the `Close` macro closes it. In each of these calls, a label that identifies a parameter block is used as an operand. The parameter blocks used in the program appear at the end of the listing.

In the source code listing of the SF.S1 program, only one parameter—the number of bytes to be read into RAM—is filled in. When you run the program, a segment of code called ReadIt fills in the other parameters. You'll examine the ReadIt segment later in this chapter.

As listing 12–1 shows, the ProDOS call Open takes three parameters:

- A 1-word file identification number that ProDOS assigns to the file being called when the Open call is made.
- A pointer to a string that contains the name of the file to be loaded. The string must be provided by the program using the Open call.
- A pointer to a 1,024-byte I/O buffer that ProDOS allocates when the call is made.

The ProDOS Read call takes four parameters:

- A 1-word file identification number. This is the ID number ProDOS assigns to the file when it is opened using an Open call.
- A pointer to a block of memory in which the file is stored. This block of memory must be provided by the application program making the Read call. In the SF.S1 program, the block is allocated using the Memory Manager call NewHandle in the segment of code labeled MakeWin0.
- A long word containing the number of bytes read into memory. In the SF.S1 program, \$8000 bytes (or 32K) of memory are loaded into memory. This number was chosen because it is the length of the IIGS screen buffer and is thus the number of bytes required by one screenful of data.
- A long word that ProDOS fills in with the number of bytes actually transferred after the Read call is made.

When the file is read, a Close call should be issued to close the file. A Close call takes one parameter: the 1-word ID number assigned to the file when it is opened.

Saving a File with ProDOS 16

In the SF.S1 program, the code segment labeled SaveOne also makes a call to ProDOS 16. Listing 12–2 shows how ProDOS 16 can be used to save a program.

Listing 12–2
Saving a file using ProDOS 16

```
SaveOne         START
                using IOData
                _Destroy DestParams
                _Create CreateParams
                bcc cont0
                ErrorCheck 'Could not create pic file.'
```

```
cont0           _Open OpenParams
                bcc cont1
                ErrorCheck 'Could not open pic file.'

cont1           anop
                lda OpenID
                sta WriteID
                sta CloseID

                _Write WriteParams
                bcc cont2
                ErrorCheck 'Could not write to pic file.'

cont2           anop
                _Close CloseParams

                clc
                rts
DestParams      anop
NameD           dc    i4'0'

CreateParams    anop
NameC     dc    i4'0'
          dc    i2'$00C3'               ; DRNWR
CType     dc    i2'$00C1'               ; super high-res graphics
CAux      dc    i4'$00000000'           ; Aux
          dc    i2'$0001'               ; type
          dc    i2'$0000'               ; create date
          dc    i2'$0000'               ; create time

OpenParams      anop
OpenID          ds 2
NamePtr         ds 4
                ds 4

WriteParams     anop
WriteID         ds 2
PicDestOUT      ds 4
                dc i4'$8000'            ; this many bytes
                ds 4                    ; how many xfered

CloseParams     anop
CloseID         ds 2

                END
```

Five ProDOS 16 calls appear in listing 12–2. `Destroy`, `Create`, `Open`, `Write`, and `Close`. Let's take a closer look at each of these calls.

The `Destroy` call deletes a file. It is used in the SF.S1 program to delete one file so that another file can be created and placed in the RAM space left by the first one. The `Destroy` call takes just one paramater: the name of the file being deleted.

The `Create` call takes seven parameters:

- A pointer to a string that contains the name of the file being created. The string must be provided by the program using the `Create` call.
- A word whose bits contain information about how the file can be accessed. Only the low-order byte of this word is significant, and bits 2 through 4 are not used. The meanings of the other five bits are listed in table 12–1.
- A word identifying the file's file type. ProDOS 16 file types are listed in table 12–2.
- A long word identifying the file's auxiliary file type. Many applications use this field. For example, APW source files (file type $B0) use the auxiliary file type parameter to identify the language of a file—that is, whether it is a 65C816 assembly language file, a C file, an exec file, and so on. ProDOS 16 applies no restrictions to this parameter, however, and user-written applications may use it to distinguish between subtypes of files.
- A word identifying the file's storage type. This parameter identifies the level in the ProDOS hierarchy in which a file falls. Values that can be stored in this parameter, and their meanings, are listed in table 12–3. The values most commonly used in this parameter are $01 and $0D. More information on file storage types can be found in the *Apple IIGS ProDOS 16 Reference*.
- Create date: a word specifying the date on which a file was created. Bits 0 through 4 hold the day of the month, bits 5 through 8 hold the number of the month, and bits 9 through 15 hold the year. If no date is specified when a file is created, ProDOS 16 supplies the date from the system clock.
- Create time: a word specifying the time a file was created. Bits 0 through 5 hold the minute and bits 8 through 12 hold the hour. Bits 6, 7, and 13 through 15 are not used. If no date is specified when a file is created, ProDOS 16 supplies the date from the system clock.

An `Open` call must be issued before a file can be saved on a disk. You saw the parameters of an `Open` call previously, when you examined listing 12–1.

The ProDOS 16 call `Write` takes four parameters:

- A 1-word file ID number assigned when the file is opened.
- A pointer to the memory address of the information to be saved as a file.

Table 12–1
Access Byte in the Create Call

Bit	Name	Function	Value
7	D	Destroy enable bit	0 = File can't be destroyed 1 = File can be destroyed
6	RN	Rename enable bit	0 = File can't be renamed 1 = File can be renamed
5	B	Backup needed bit	0 = File backup is required 1 = Backup not required
4		Reserved	
3		Reserved	
2		Reserved	
1	W	Write enable bit	0 = File can't be written to 1 = File can be written to
0	R	Read enable bit	0 = File can't be read 1 = File can be read

- A long word holding the number of bytes to be saved.
- A long word in which ProDOS stores the number of bytes that have actually been transferred after the call is completed.

When you have finished saving a file, a `Close` call should be issued to close the file. A `Close` call takes one parameter: the 1-word ID number assigned to the file when it is opened.

Using the Standard File Tool Set

The Standard File Operations Tool Set, as noted, offers the IIGS user an easy and convenient method for loading and saving files—a collection of dialog boxes that can be programmed to appear on the screen when needed. These dialog boxes make loading and saving files as easy as clicking the mouse button. The Standard File Tool Set is even more of a timesaver for the IIGS programmer than it is for the IIGS user!

Before the Standard File Operations Tool Set is started up, the following tool sets must be loaded and initialized:

- Tool Locator (always loaded and active)
- Window Manager
- Control Manager
- Menu Manager
- LineEdit Tool Set
- Dialog Manager

When these tool sets are loaded and started up, the Standard File Tool Set can be initialized with the `SFStartup` call. Before a program that uses the tool set ends, `SFShutdown` should be called.

Table 12–2
ProDOS 16 File Types

Type	Name	Description
$00		Uncategorized file
$01	BAD	Bad block file
$02–03		Used by SOS (Apple III)
$04	TXT	ASCII text file
$05		Used by SOS (Apple III)
$06	BIN	Binary file
$07		Used by SOS (Apple III)
$08	FOT	Apple II graphics screen file
$09-$0E		SOS (Apple III) reserved
$0F	DIR	Directory file
$10-$18		Used by SOS (Apple III)
$19	ADB	AppleWorks database file
$1A	AWP	AppleWorks word-processor file
$1B	ASP	AppleWorks spreadsheet file
$1C-$AF		Reserved
$B0	SRC	APW source file
$B1	OBJ	APW object file
$B2	LIB	APW library file
$B3	S16	ProDOS 16 application program file
$B4	RTL	Run-time library
$B5	EXE	ProDOS 16 shell application file
$B6		ProDOS 16 permanent initialization file
$B7		ProDOS 16 temporary initialization file
$B8		New desk accessory (NDA)
$B9		Classic desk accessory (CDA)
$BA		Tool set file
$BB-$BE		Reserved for ProDOS 16 load files
$BF		ProDOS 16 document file
$C0-$EE		Reserved
$EF	PAS	Pascal area on a partitioned disk
$F0	CMD	ProDOS 8 CI added command file
$F1-$F8		ProDOS 8 user-defined files 1–8
$F9		ProDOS 8 reserved
$FA	INT	Integer BASIC program file
$FB	INV	Integer BASIC variable file
$FC	BAS	Applesoft BASIC program file
$FD	VAR	Applesoft BASIC variables file
$FE	REL	Relocatable code file (EDASM)
$FF	SYS	ProDOS 8 system program file

Table 12–3
File Storage Types

Value	Meaning
$00	Inactive entry
$01	Seedling file
$02	Sapling file
$03	Tree file
$04	Apple II Pascal region on a partitioned disk
$05	Directory file

Loading a File with the Standard File Tool Set

The easiest way to load a file using the Standard File Tool Set is with the `SFGetFile` call. The `SFGetFile` routine displays a standard, predesigned dialog box and allows the IIGS operator to use the dialog to open and load the selected file. With `SFGetFile`, the calling program can specify where the dialog box will be placed on the screen and the prompt that appears at the top of the box. The calling program can also filter the types of files to be displayed in the box. But the routine does not allow an application program to modify the appearance of the box. Programs that use a custom-designed dialog box must use another Standard File routine, `SFPGetFile`.

In the SF.S1 program, the `SFGetFile` call loads files into memory. Listing 12–3 shows the section of the program that uses the `SFGetFile` call.

The SFGetFile Call

As listing 12–3 illustrates, the `SFGetFile` call takes five parameters:

- A 1-word integer that specifies the horizontal screen coordinate of the upper left corner of the dialog box.
- Another 1-word integer that specifies the vertical screen coordinate of the upper left corner of the dialog box.
- A pointer to a Pascal-style string that is printed as a prompt inside the dialog box.
- A pointer to a "filter process" that can provide special instructions to the Dialog Manager about the handling of files. If such a process is used, it must be defined by the calling program. Instructions for designing a filter process are in the *Apple IIGS Toolbox Reference.* No filter process is used in the SF.S1 program.
- A pointer to a reply record, a specially designed record that the `SFGetFile` call fills with information before it returns. Listing 12–4 shows the reply record used in the SF.S1 program.

Listing 12–3
SFGetFile call in SF.S1

```
LoadIt          START
                using WindowData
                using IOData

                jsr Repaint

                PushWord #20              ; upper x coordinate
                PushWord #20              ; upper y coordinate
                PushLong #PromptPtr
                PushLong #0               ; no filter process
                PushLong #TypeListPtr     ; file types to display
                PushLong #ReplyRecord     ; defined in iodata
                _SFGetFile

                lda GoodFlag
                bne cont
                jmp return                ; user canceled operation

cont            lda #FName
                sta NamePtr
                lda #^FName
                sta NamePtr+2

                lda Win0Handle
                ldx Win0Handle+2
                jsr Deref
                sta PicDestIn
                stx PicDestIn+2
                jsr LoadOne

                PushLong NamePtr
                PushLong Win0Ptr
                _SetWTitle                ; update window title

                lda Win0Handle
                ldx Win0Handle+2
                jsr Unlock

                PushLong NamePtr          ; update 'title' menu item
                PushWord #262             ; menu item number
                _SetMItemName             ; update name of item
                PushWord #0
                PushWord #0
                PushWord #3               ; menu number
                _CalcMenuSize             ; update width of items
```

```
return        rts

PromptPtr     str 'Load Picture:'

TypeListPtr   anop
NumEntries    dc i1'1'
Filetype1     dc h'c1'

              END
```

Listing 12–4
Reply record used by SFGetFile call

```
ReplyRecord    anop
GoodFlag       ds 2
FType          dc ds 2          ; in SF.S1, will always be $C1
AuxFType       dc i'0'          ; #0
FName          ds 15
FullPathName   ds 128
```

An SFGetFile reply record has five fields:

- A 1-word flag, called GoodFlag in the SF.S1 program, that holds a Boolean value. The flag is cleared to 0 if the user aborts the SFGetFile operation by pressing a Cancel button inside the dialog box. If the user does not press the Cancel button, the flag is set.
- A 1-word parameter that contains the type of file selected by the user. This parameter, like all other parameters in a reply record, is filled in by the SFGetFile call.
- A 1-word parameter that contains the auxiliary file type of the file selected by the user.
- A Pascal-style string that contains the name of the file selected by the user. The length of this parameter can be set by the application that calls SFGetFile. The most common length for this parameter is 15 bytes.
- Another Pascal-style string that contains the full pathname of the file selected by the user. The length of this parameter must be set by the application that calls SFGetFile. The recommended length for the parameter is 128 bytes.

All the information returned by the SFGetFile call is placed in its reply record; it does not push any values onto the stack.

In the SF.S1 program, a pointer to the file name returned by SFGetFile is loaded into the NamePtr variable. The handle of the screen buffer used in the program is then dereferenced (coverted into a pointer), and the LoadOne subroutine loads the file chosen by the user into the screen buffer.

Next, the program makes the Window Manager call SetWTitle to update the name of the window being displayed on the screen. Then the Menu Manager routines SetMItemName and CalcMenuSize replace the menu item Untitled with a menu item that displays the name of the selected window.

The SFPutFile Call

The simplest way to save a file using the Standard File Tool Set is with the call SFPutFile. The SFPutFile routine, like the SFGetFile routine, displays a standard, predesigned dialog box. The IIGS operator can then use the dialog to save the selected file on a disk. With SFPutFile, like SFGetFile, the calling program can specify the location of the dialog box on the screen, the prompt that appears at the top of the box, and the types of files to be displayed in the box. But it does not permit an application program to modify the design of the box. Programs that use a custom-tailored dialog box must use another Standard File routine, SFPPutFile.

In the SF.S1 program, files are saved using the SFPutFile call. Listing 12–5 shows how the call is used in the program.

Listing 12–5
SFPutFile call in SF.S1

```
SaveIt          START
                Using WindowData
                Using IOData

                PushWord #20              ; upper X coordinate
                PushWord #20              ; upper Y coordinate
                PushLong #TopMsg
                PushLong #Win0Title
                PushWord #15              ; max length of filename
                PushLong #ReplyRecord     ; defined in iodata
                _SFPutFile

                lda GoodFlag
                bne cont
                jmp return                ; user canceled operation

cont            lda #FName
                sta NamePtr
                lda #^FName
                sta NamePtr+2

                lda Win0Handle
                ldx Win0Handle+2
                jsr Deref
                sta PicDestOut
                stx PicDestOut+2
```

```
                jsr SaveOne

                PushLong NamePtr
                PushLong WinOPtr
                _SetWTitle              ; update window title

                lda WinOHandle
                ldx WinOHandle+2
                jsr Unlock

                PushLong NamePtr        ; update 'title' menu item
                PushWord #262           ; menu item number
                _SetMItemName           ; update name of item

                PushWord #0
                PushWord #0
                PushWord #3             ; menu number
                _CalcMenuSize           ; update width of items

return          rts

TopMsg          str 'Type name of picture:'

                END
```

SFPutFile, like SFGetFile, takes five parameters. There are some differences, however, between the parameter sequences used by the two calls. The parameters that must be passed to the SFPutFile call are

- A 2-byte integer that specifies the horizontal screen coordinate of the upper left corner of the dialog box.
- Another 2-byte integer that specifies the vertical screen coordinate of the upper left corner of the dialog box.
- A pointer to a Pascal-style string that is printed as a prompt inside the dialog box.
- A pointer to a Pascal-type string that can be used to specify a default file name. If a pointer is specified, the string that is pointed to is printed in a line edit item inside the default box. You can then save that file by clicking the mouse button inside an OK box or pressing the Return key. If you want to save another file, the default string can be erased or edited using standard line edit techniques. If a 0 is passed in this parameter, a default string is not printed on the screen.
- A pointer to the same kind of five-field reply record used by the SFGetFile call.

After the `SFPutFile` routine is called in the SF.S1 program, the `LoadOne` subroutine loads the file selected by the user into the program's window buffer. The name of the window is updated, and the menu is modified so that it displays the new window's name.

The SF.S1 Program

The sample program in this chapter, SF.S1, is an expanded version of the DIALOG.S1 program created in chapter 11. To convert DIALOG.S1 into SF.S1, the following modifications are necessary:

1. Edit the heading of the program so that it looks like the one shown in listing 12–6.
2. Following the program segment labeled `EventLoop`, insert the segments shown in listing 12–7. These segments are the heart of the SF.S1 program. They load and save files and control the Standard File Tool Set.
3. Replace the data segment labeled `MenuData` with the segment shown in listing 12–8.
4. At the end of the program, add the data segment shown in listing 12–9.
5. Make sure that the latest version of INITQUIT.S1 is on the same disk that holds your SF.S1 source code. The `COPY` directive at the end of the SF.S1 combines the SF.S1 program and the INITQUIT.S1 program.

Listing 12–6
SF.S1 heading segment

```
*
*  SF.S1
*

*** A FEW ASSEMBLER DIRECTIVES ***

                Title 'SF'

                ABSADDR on
                LIST off
                SYMBOL off
                65816 on
                mcopy SF.macros

                KEEP SF
```

Listing 12–7
SF.S1 new segments

```
*
*  LOADIT: ROUTINE TO LOAD A PICTURE FROM DISK
*

LoadIt          START
                using WindowData
                using IOData

                jsr Repaint

                PushWord #20              ; upper x coordinate
                PushWord #20              ; upper y coordinate
                PushLong #PromptPtr
                PushLong #0               ; no filter process
                PushLong #TypeListPtr     ; file types to display
                PushLong #ReplyRecord     ; defined in iodata
                _SFGetFile

                lda GoodFlag
                bne cont
                jmp return                ; user canceled operation

cont            lda #FName
                sta NamePtr
                lda #^FName
                sta NamePtr+2

                lda Win0Handle
                ldx Win0Handle+2
                jsr Deref
                sta PicDestIn
                stx PicDestIn+2
                jsr LoadOne

                PushLong NamePtr
                PushLong Win0Ptr
                _SetWTitle                ; update window title

                lda Win0Handle
                ldx Win0Handle+2
                jsr Unlock

                PushLong NamePtr          ; update 'title' menu item
                PushWord #262             ; menu item number
                _SetMItemName             ; update name of item
```

```
               PushWord #0
               PushWord #0
               PushWord #3              ; menu number
               _CalcMenuSize            ; update width of items

return         rts

PromptPtr      str 'Load Picture:'

TypeListPtr    anop
NumEntries     dc i1'1'
Filetype1      dc h'c1'

               END

*
*  SAVEIT: ROUTINE TO SAVE A PICTURE TO DISK
*

SaveIt         START
               Using WindowData
               Using IOData

               PushWord #20              ; upper X coordinate
               PushWord #20              ; upper Y coordinate
               PushLong #TopMsg
               PushLong #Win0Title
               PushWord #15              ; max length of file name
               PushLong #ReplyRecord     ; defined in iodata
               _SFPutFile

               lda GoodFlag
               bne cont
               jmp return                ; user canceled operation

cont           lda #FName
               sta NamePtr
               lda #^FName
               sta NamePtr+2

               lda Win0Handle
               ldx Win0Handle+2
               jsr Deref
               sta PicDestOut
               stx PicDestOut+2
```

```
                jsr SaveOne

                PushLong NamePtr
                PushLong WinOPtr
                _SetWTitle                ; update window title

                lda WinOHandle
                ldx WinOHandle+2
                jsr Unlock

                PushLong NamePtr          ; update 'title' menu item
                PushWord #262             ; menu item number
                _SetMItemName             ; update name of item

                PushWord #0
                PushWord #0
                PushWord #3               ; menu number
                _CalcMenuSize             ; update width of items

return          rts

TopMsg          str 'Type name of picture:'

                END

*
* LoadOne
* Loads the picture whose pathname is passed in NamePtr to address
* passed in PicDestIN
*

LoadOne         START
                using IOData

                _Open OpenParams
                bcc cont1
                ErrorCheck 'Could not open picture file.'

cont1           anop
                lda OpenID
                sta ReadID
                sta CloseID

                _Read ReadParams
                bcc cont2
                ErrorCheck 'Could not read picture file.'
```

```
cont2           anop
                _Close CloseParams

                clc
                rts
                END

*
* SaveOne
* Saves the picture whose pathname is passed in NamePtr from address
* passed in PicDestOUT
*

SaveOne         START
                using IOData

                lda NamePtr
                sta NameC
                sta NameD
                lda NamePtr+2
                sta NameC+2
                sta NameD+2

                _Destroy DestParams

                lda #$c1                    ; SuperHiRes picture type
                sta CType
                lda #$0                     ; standard type = 0
                sta CAux

                _Create CreateParams
                bcc cont0
                ErrorCheck 'Could not create pic file.'

cont0           _Open OpenParams
                bcc cont1
                ErrorCheck 'Could not open pic file.'

cont1           anop
                lda OpenID
                sta WriteID
                sta CloseID

                _Write WriteParams
                bcc cont2
                ErrorCheck 'Could not write to pic file.'
```

```
cont2           anop
                _Close CloseParams

                clc
                rts

                END
```

Listing 12–8
SF.S1 new MenuData segment

```
*
* Menu Data
*

MenuData        DATA

Return          equ 13

Menu1           dc c'>L@\XN1',i1'RETURN'
                dc c' LA Window Program \N257',i1'RETURN'
                dc c'.'

Menu2           dc c'>L File \N2',i1'RETURN'
                dc c' LNew \N258V',i1'RETURN'
                dc c' LLoad \N259',i1'RETURN'
                dc c' LSave \N260V',i1'RETURN'
                dc c' LQuit\N261',i1'RETURN'
                dc c'.'

Menu3           dc c'>L Windows \N3',i1'RETURN'
                dc c' LUntitled \N262',i1'RETURN'
                dc c'.'

                END

MenuTable       DATA

*               Menu 1 (apple)
                dc i'ignore'            ; one for the NDAs
                dc i'ignore'            ; 'a window program'

*               Menu 2 (file)
                dc i'Repaint'           ; 'doWinO' (new window)
                dc i'LoadIt'
                dc i'SaveIt'
                dc i'doQuit'            ; quit item selected
```

```
*               Menu 3 (windows)
                dc i'doWin0'              ; 'untitled'

                END

***
```

Listing 12-9
SF.S1 IOData segment

```
*
* IOData
*

IOData          DATA

ReplyRecord     anop
GoodFlag        ds 2
FType           dc i'193'                 ; $c1
AuxFType        dc i'0'                   ; #0
FName           ds 15
FullPathName    ds 128

CreateParams    anop
NameC     dc    i4'0'
          dc    i2'$00C3'                 ; DRNWR
CType     dc    i2'$00C1'                 ; super high-res graphics
CAux      dc    i4'$00000000'             ; Aux
          dc    i2'$0001'                 ; type
          dc    i2'$0000'                 ; create date
          dc    i2'$0000'                 ; create time

DestParams      anop
NameD           dc   i4'0'

OpenParams      anop
OpenID          ds 2
NamePtr         ds 4
                ds 4

ReadParams      anop
ReadID          ds 2
PicDestIN       ds 4
                dc i4'$8000'              ; this many bytes
                ds 4                      ; how many xfered
```

```
WriteParams     anop
WriteID         ds 2
PicDestOUT      ds 4
                dc i4'$8000'           ; this many bytes
                ds 4                   ; how many xfered

CloseParams     anop
CloseID         ds 2

                END
```

The SF.C Program

Listing 12–10 is a C language version of the SF.S1 program. Designed to be used with the `include` file INITQUIT.C, it works almost exactly like the SF.S1 program.

In the C version of the SF program, files are not loaded and saved using ProDOS calls, as they are in the assembly language version. Instead, SF.C uses four C library routines: `Open`, `Close`, `Read`, and `Write`. These routines are called in the `LoadIt` and `SaveIt` segments of the program.

The `Open` function returns an integer, known as a file descriptor, for each file successfully opened. If the call fails, it returns −1. In the SF.C program, you test the value returned by `Open`. If the value is −1, a dialog window appears on the screen and tells the user an I/O error has occurred. Then the user can try to continue or quit. This dialog is created and displayed in the `BadIO` segment of the program.

The event loop of the program is the same as the one that appeared in the DIALOG.C program in chapter 11. The `DoMenus` section is expanded to accommodate some new menu choices, but the changes need little explanation.

There are also changes in the way window titles are selected and displayed. These modifications are necessary because window titles can change in the SF.S1 program. Although there may be a more elegant way to accommodate the shifting of window titles, calling `HideWindow` and then `ShowWindow` does the job.

Also, the File menu selection in SF.S1 does not conform strictly to the usual conventions for saving and loading files. For example, in the SF.S1 program, you can use the menu selections New, Load, or Quit without saving first—and you can thus wipe out the picture currently on the screen without warning. Because SF.S1 is a tutorial program, we decided to forego fixing that bug to avoid adding more complexity to the program.

One feature we did add was to disable the menu selection Save when no window is open. Disabling an item lets the user know "that can't be done right now," and ensures that TaskMaster does not return the constant that represents the disabled item in the `wmTaskData` field.

Listing 12–10
SF.C program

```
#include "initquit.c"
#include <prodos.h>
#include <string.h>
#include <fcntl.h>

Boolean done = false;
WmTaskRec   myEvent;

/************************************/
/* Data and routine to create menus */
/************************************/

/*  Set up menu strings. Because C uses \ as an escape character, we use
two when we want a \ as an ordinary character. The \ at the end of each
line tells C to ignore the carriage return. This lets us set up our items
in an easy-to-read vertical alignment. */

char *menu1 = "\
>L@\\XN1\r\
 LA Standard File Program \\N257\r\
.";

char *menu2 = "\
>L File \\N2\r\
 LNew \\N258V\r\
 LOpen #\\N259\r\
 LSave \\N260V\r\
 LQuit #\\N261\r\
.";

char *menu3 = "\
>L Windows \\N3\r\
 LUntitled \\N262\r\
.";

#define NEW_ITEM 258
#define OPEN_ITEM 259
#define SAVE_ITEM 260
#define QUIT_ITEM 261   /* these will help us check menu item numbers */
#define TITLE_ITEM 262

BuildMenu()
{
   InsertMenu(NewMenu(menu3),0);
   InsertMenu(NewMenu(menu2),0);
```

```
   InsertMenu(NewMenu(menu1),0);
   FixMenuBar();
   DrawMenuBar();
   DisableMItem(SAVE_ITEM);/* save is disabled until a window is drawn */
}

/*********************************************************************/
/* Data structures and routines to set up and refresh               */
/* offscreen drawing environment                                     */
/*********************************************************************/

LocInfo pic0LocInfo = { mode320,
                        NULL, /* space for pointer to pixel image */
                        160,  /* width of image in bytes = 320 pixels */
                        0,0,200,320  /* frame rect */
                      };

Rect screenRect = {0,0,200,320};
GrafPort pic0Port;

#define IMAGE_ATTR attrLocked+attrFixed+attrNoCross+attrNoSpec+attrPage

Pic0Setup()  /* called once by MakeWindow at start of program */
{
GrafPortPtr thePortPtr;

   pic0LocInfo.ptrToPixImage = *(NewHandle(0x8000L,myID,IMAGE_ATTR,NULL));
   thePortPtr = GetPort();
   OpenPort(&pic0Port);
   SetPort(&pic0Port);
   SetPortLoc(&pic0LocInfo);
   ClipRect(&screenRect);
   EraseRect(&screenRect);
   SetPort(thePortPtr);
}

ErasePic0()
{
GrafPortPtr oldPortPtr;

   oldPortPtr = GetPort();
   SetPort(&pic0Port);
   ClipRect(&screenRect);
   EraseRect(&screenRect);
   SetPort(oldPortPtr);
}
```

```
/************************************************************************/
/* Data and routines for handling Open and Save calls                  */
/************************************************************************/

#define O_PICLOAD O_RDONLY+O_BINARY
#define O_PICSAVE O_WRONLY+O_CREAT+O_BINARY+O_TRUNC

SFReplyRec file = {0,193}; /* intit 2 fields, rest are 0'd  */
char curpath[130] ;  /* place for C string version of pathname */
Byte typelist[2] = {1,193}; /* we only want to open hi-res pictures */
FileRec fileInfo = {file.fullPathname}; /* initialize first field  */

LoadIt()
{
int filedes;
char oldTitle[16];

   strncpy(oldTitle,file.filename,16); /* save title in case load fails */

   SFGetFile(20,20,"\pLoad Picture:",NULL,typelist,&file);
   if(file.good) {
      p2cstr(strncpy(curpath,file.fullPathname,(int)*file.fullPathname+ 1 ) ) ;

      if((filedes = open(curpath,O_PICLOAD)) != -1) {
         read(filedes,pic0LocInfo.ptrToPixImage,0x8000);
         close(filedes);

         SetMItemName(file.filename,262);
         CalcMenuSize(0,0,3);
         RenewWind();
      }
      else {
         BadIO();      /* load failed, put up message and restore title */
         strncpy(file.filename,oldTitle,16);
      }
   }
}

SaveIt(winPtr)
GrafPortPtr winPtr;
{
int filedes;
char oldTitle[16];

   strncpy(oldTitle,file.filename,16); /* save title in case save fails */

   SFPutFile(20,20,"\pType name of picture:",file.filename,15,&file);
```

```
   if(file.good){
      p2cstr(strncpy(curpath,file.fullPathname,(int)*file.fullPathname+
1));
      if((filedes = open(curpath,O_PICSAVE)) ! 1= -1){
         write(filedes,pic0LocInfo.ptrToPixImage,0x8000);
         close(filedes);

         GET_FILE_INFO(&fileInfo);  /* make file's type a hires picture */
         fileInfo.fileType = 0xC1;
         SET_FILE_INFO(&fileInfo);

         SetMItemName(file.filename,TITLE_ITEM);
         CalcMenuSize(0,0,3);
      }
      else {        /* save failed, put up message and restore title */
         BadIO();
         strncpy(file.filename,oldTitle,16);
      }
   }
   else strncpy(file.filename,oldTitle,16);
}

/*****************************************************/
/* Data structures and routines to create window     */
/*****************************************************/

/* Initialize template for NewWindow  */

#define FRAME fQContent+fMove+fZoom+fGrow+fBScroll+fRScroll+fClose+fTitle

ParamList template = { sizeof(ParamList),
                       FRAME,
                       file.filename,  /* Pointer to title in SFReplyRec */
                       0L,             /* RefCon              */
                       26,0,188,308,   /* Full size (0=default)  */
                         NULL,         /* use default ColorTable */
                       0,0,            /* origin              */
                       200,320,        /* data area height & width */
                       200,320,        /* max cont height & width  */
                       2,2,     /* vertical & horizontal scroll increment */
                       20,32,   /* vertical & horizontal page increment   */
                       NULL,    /* no info bar text string  */
                       0,       /* info bar height = none   */
                       NULL,    /* default def proc         */
                       NULL,    /* no info bar draw routine */
                       NULL,    /* draw content must be filled in at run time */
                       26,0,188,308, /* starting content rect */
```

```
                        -1L,    /* topmost plane */
                        NULL    /* let window manager allocate record */
                           };

/* Window's draw content routine */

pascal void DrawContent()
{
   PPToPort(&picOLocInfo,&(picOLocInfo.boundsRect),0,0,modeCopy);
}

GrafPortPtr winOPtr;

MakeWindow() /* Set default title str, complete template, make the window */
{
   strncpy(file.filename,"\pUntitled",9); /* default name for new window   */
   template.wContDefProc = DrawContent;
   winOPtr = NewWindow(&template);
}

RenewWind()  /* a way to restore a window to its default size and position */
{        /* will not affect the contents unless ErasePicO is called first */

   EnableMItem(SAVE_ITEM);
   HideWindow(winOPtr);
   CloseWindow(winOPtr);
   winOPtr = NewWindow(&template);
   SelectWindow(winOPtr);
   ShowWindow(winOPtr);
}

/********************************************************************/
/* Data and routines to set up and display dialogs                  */
/********************************************************************/

char prompt[40] = "\pUnable to load or save ";

ItemTemplate item1 = { 1,{8,129,22,179},buttonItem,"\pStart\r",0,0,NULL };
ItemTemplate item2 = { 2,{8,8,22,58},buttonItem,"\pQuit\r",0,0,NULL };
ItemTemplate item3 = { 3,{8,67,22,117},buttonItem,"\pHelp\r",0,0,NULL };
ItemTemplate item4 = { 4,{30,8,55,259},statText,prompt,0,0,NULL };
ItemTemplate item5 = { 1,{8,129,22,179},buttonItem,"\pOK",0,0,NULL };

DialogTemplate dtemp = {{84,63,114,252},true,0L,&item1,&item2,&item3,NULL};
DialogTemplate iotemp = {{84,23,144,292},true,0L,&item5,&item2,&item4,NULL};

DoDialog()   /* Create and display an opening dialog box */
{
```

```
GrafPortPtr dlgPtr;
Word hit;

  dlgPtr = GetNewModalDialog(&dtemp);

  while ((hit = ModalDialog(NULL)) == 3);
  done = (hit == 2);
  CloseDialog(dlgPtr);
}

BadIO()
{
GrafPortPtr dlgPtr;

  strncat(prompt,file.filename + 1, *file.filename);
  *prompt = 23 + *file.filename;
  dlgPtr = GetNewModalDialog(&iotemp);

  done = (ModalDialog(NULL) == 2);
  CloseDialog(dlgPtr);
}

/********************************************************************/
/* Main routine. Set up environment, call eventloop, and shut down  */
/********************************************************************/

main()
{
   StartTools();
   DoDialog();
   BuildMenu();
   MakeWindow();
   Pic0Setup();
   EventLoop();
   DisposeHandle(FindHandle(pic0LocInfo.ptrToPixImage));
   ShutDown();
}

/**************************************/
/* Event loop and supporting routines */
/**************************************/

EventLoop()
{
   myEvent.wmTaskMask = 0x0FFF;
   while(!done)
      switch ( TaskMaster(everyEvent,&myEvent)) {
         case wInMenuBar:
```

```
            DoMenus();
            break;
         case wInGoAway:
            DisableMItem(SAVE_ITEM);
            HideWindow(win0Ptr);
            break;
         case wInContent:
            Sketch();
     }
}

DoMenus()
{
Word *data = (Word *)&myEvent.wmTaskData; /* address of item id  */

    switch(*data) {
      case QUIT_ITEM:
         done = true;
         break;
      case OPEN_ITEM:
         LoadIt();
         break;
      case SAVE_ITEM:
         SaveIt();
         HideWindow(win0Ptr);  /* Make sure the title gets updated */
         ShowWindow(win0Ptr);
         break;
      case NEW_ITEM:
         ErasePic0();
         strncpy(file.filename,"\pUntitled",9);
         RenewWind();
         break;
      case TITLE_ITEM:
         EnableMItem(SAVE_ITEM);
         SelectWindow(win0Ptr);
         ShowWindow(win0Ptr);
         break;
    }
    HiliteMenu(false,*(data + 1)); /* data + 1 is address of menu id  */
}

Sketch() /* sketch into current port, and into offscreen port */
{
Point mouseLoc;
GrafPortPtr thePortPtr = (GrafPortPtr)myEvent.wmTaskData;
Rect theRect;
```

```
   mouseLoc = myEvent.wmWhere;

   StartDrawing(thePortPtr);  /* set up correct drawing coordinate system */
   GetPortRect(&theRect);     /* copy current port rect */
   GlobalToLocal(&mouseLoc);  /* get cursor pos in local coordinates */

   MoveTo(mouseLoc);         /* set pen position to mouse loc */
   SetPort(&picOPort);       /* switch to offscreen port */
   ClipRect(&theRect);       /* clip offscreen drawing to window's port rect */
   MoveTo(mouseLoc);         /* set offscreen pen to same location */
   SetPort(thePortPtr);      /* switch back to window's port   */

   while (StillDown(0)) {
      GetMouse(&mouseLoc);    /* get new mouse coordinates */

      LineTo(mouseLoc);       /* draw line in both ports */
      SetPort(&picOPort);
      LineTo(mouseLoc);
      SetPort(thePortPtr);
   }
   SetOrigin(0,0);            /* restore normal coordinates */
}
```

CHAPTER 13

The Sound of Music

The IIGS as a Sound and Music Synthesizer

One of the most remarkable features of the IIGS is its ability to synthesize music and sounds. Some reviewers have declared that the IIGS offers the finest sound-synthesizing capabilities of any computer in its class. So it's no wonder that the *s* in IIGS stands for *sound.*

You don't have to be a musician or an audio engineer to understand how the synthesizer built into the IIGS works. To write sound and music programs for the Apple IIGS, however, it doesn't hurt to know a little bit about how a music synthesizer produces sound. So, in the first part of this chapter, you take a brief look at some important facts about the science of sound and how the IIGS produces sound and music. Then you type, assemble, and run a program that turns your IIGS keyboard into a music synthesizer capable of producing an almost limitless variety of sounds.

The Characteristics of Sound

When you hear a sound from a musical instrument, four characteristics are combined to create the sound you perceive. These four characteristics are

- Volume, or loudness
- Frequency, or pitch
- Timbre, or sound quality

- Dynamic range, or the difference in level between the loudest sound that can be heard and the softest sound that can be heard during a given period of time. This time period can range between the time it takes to play a single note and the length of a much longer listening experience, such as a musical performance or a complete musical recording.

Sound Hardware in the IIGS

To produce sounds that have these four characteristics—volume, frequency, timbre, and dynamic range—the IIGS is equipped with a pair of special-purpose sound chips. One is the *digital oscillator chip,* or DOC, and the other is the *general logic unit,* or GLU. Let's take a closer look at these two processors.

The Digital Oscillator Chip

The digital oscillator chip, or DOC, is a sound-generating microprocessor designed by the Ensoniq sound synthesizer company. DOCs are used in Ensoniq synthesizers as well as in the IIGS.

The basic sound-generating unit used by the DOC is a component called an *oscillator.* To produce a sound, an oscillator must step through a table of sound samples stored as digital numbers. This table must be supplied by the application program using the oscillator. It can be created while a program is running, or it can be stored on a disk and loaded into memory in advance.

The DOC contains thirty-two oscillators, but two are unavailable for use in application programs. One is always used as a clock, and another is reserved for future use. That leaves thirty oscillators, each of which can function independently. In practice, however, the DOC's oscillators are used in pairs because it takes at least two oscillators to produce a continuous instrumental voice.

When two oscillators are used together to produce a sound, they form a functional unit called a *generator.* So, in normal use, the DOC has fifteen generators and thus is a 15-voice chip.

The DOC also has a component called an *analog-to-digital converter,* or ADC. The ADC makes it possible for the DOC to record a digital sample of an actual sound, so that the sound can be played back later from its digital sample. More information about this capability is in the *Apple IIGS Hardware Reference.*

The General Logic Unit

The general logic unit, or GLU, is a chip that interfaces the DOC processor and the IIGS system. It also enables the IIGS to produce sound in the same way as older Apple IIs: by toggling a single-bit switch that can make a speaker vibrate at various rates of speed. But thanks to the GLU, this method of producing sound is improved; its volume can now be software controlled.

In addition to its DOC and GLU chips, the IIGS has 64K of dedicated RAM used only for storing sound samples. Because this area of memory is used only by the DOC, it is sometimes referred to as DOC RAM.

Sound Tools in the Toolbox

The IIGS Toolbox contains three tool kits that make it possible to write sound and music programs without accessing the sound registers used by the DOC and the GLU directly. These three tool sets are the

- Sound Tool Set, which starts and stops sounds, sets sound volumes, performs read and write operations to and from DOC registers, and reads and writes data to and from DOC RAM.
- Note Synthesizer, a higher-level tool set that produces and controls musical notes. The Note Synthesizer can emulate the sound of virtually any musical instrument and can produce unique musical sounds with almost any characteristics desired.
- Note Sequencer, a still higher-level tool set that makes it easier to combine various notes, chords, note patterns, and rhythms into musical performances and compositions.

The sample program in this chapter, MUSIC.S1, uses the Sound Tool Set and the Note Synthesizer. It does not use the Note Sequencer because it is an interactive program. The MUSIC.S1 program appears at the end of this chapter.

More About the Science of Sound

Now that you know something about how the IIGS produces music and sound, you're ready to take a closer look at the four primary characteristics of every sound: volume, frequency, timbre, and dynamic range.

Volume

If you've ever turned a volume knob on a radio, you know just about all you'll need to know about volume to write sound and music programs for the IIGS.

In programs written using the Sound Tool Set, the volume of a sound is controlled using the Sound Tool call `SetSoundVolume`. In programs that use the Note Synthesizer, volume is expressed as a value ranging from 0 to 127 and is controlled by passing a parameter to the Note Synthesizer call `NoteOn`.

As you shall see later, the `NoteOn` call must be made every time a note is produced by the Note Synthesizer. In the MUSIC.S1 program, volume is controlled using the `NoteOn` call. You'll see how this is done later in this chapter.

Frequency

The pitch of a musical note is determined by its frequency. In programs written using the IIGS Note Synthesizer, frequency is measured in semitones, or halftones. A semitone value ranges from 0 to 127, with 60 representing middle C.

The frequency of a note, like the note's volume, can be established by passing a parameter to the Note Synthesizer call `NoteOn`. An example is provided later in this chapter.

Timbre

Timbre, or note quality, is sometimes illustrated with the help of a waveform. There are four basic varieties of waves: sine wave, square wave (or pulse wave), triangle wave, and sawtooth wave. But these four types of waves can be combined with each other, and with irregular wave patterns, in endless varieties.

To understand how waveforms work, you need to know a little about musical harmonics. So here is a crash course in music theory.

With the help of an electronic instrument, you can generate a tone that has just one pure frequency. But when a note is played on a musical instrument, more than one frequency is usually produced. In addition to a primary frequency, or a fundamental, there is usually a set of secondary frequencies called harmonics. It is this total harmonic structure that determines the timbre of a sound.

When a tone containing only a fundamental frequency is viewed on an oscilloscope, the pattern produced on the screen is a pure sine wave. When a flute is played, the waveform produced is very close to that of a pure sine wave. The waveform of a sine wave is shown in figure 13–1.

When harmonics are added to a tone, the result is a richer sound that produces what is sometimes called a triangle wave. Triangle waveforms, or waves that are close to triangle waveforms, are produced by instruments such as xylophones, organs, and accordians. Figure 13–2 is a triangle wave.

When still more harmonics are added to a note, other kinds of waves are formed. Harpsichords and trumpets, for example, produce a type of wave sometimes called a sawtooth wave. A piano generates a squarish kind of wave called a square wave or a pulse wave. A sawtooth wave is illustrated in figure 13–3, and a pulse wave is shown in figure 13–4.

Another kind of waveform that the DOC can produce is a noise waveform. A noise waveform creates a random sound output that varies with a frequency proportionate to that of an oscillator built into Voice 1. Noise waveforms are often used to imitate the sound of explosions, drums, and other nonmusical noises.

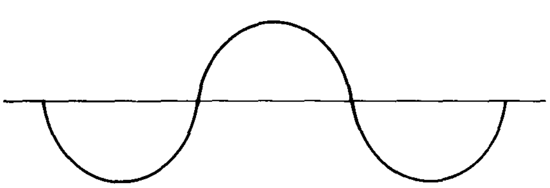

Figure 13–1
Sine waveform

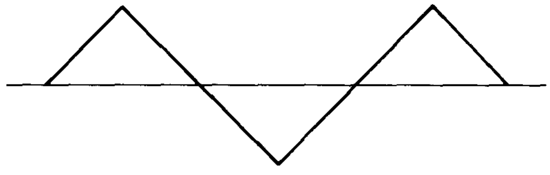

Figure 13–2
Triangle waveform

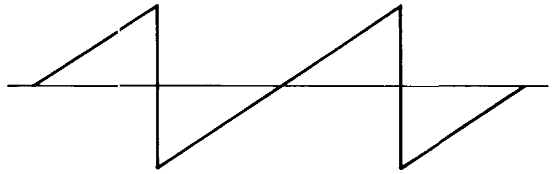

Figure 13–3
Sawtooth waveform

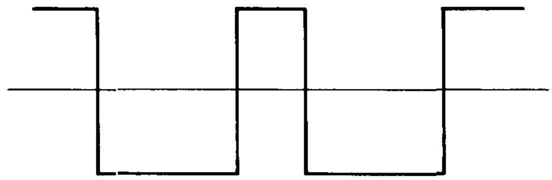

Figure 13–4
Pulse waveform

In programs written for the IIGS, waveforms can be created when needed—as they are in the MUSIC.S1 program—or they can be created and loaded into memory in advance. No matter how a waveform is created, though, it must be moved into DOC RAM before it can be used to produce a sound.

Dynamic Range

The dynamic range of a note—the difference in volume between its loudest sound level and its softest sound level—can be illustrated in many ways. To illustrate and control the dynamic ranges of notes, audio engineers sometimes use a device called an *ADSR envelope,* or attack-decay-sustain-release envelope. An ADSR envelope illustrates four distinct stages in the life of a note: four phases every note undergoes between the time it starts and the time it fades away. These four phases—attack, decay, sustain, and release—are shown in the ADSR envelope illustrated in figure 13–5.

A Close Look at an ADSR Envelope

As figure 13–5 shows, every note starts with an attack. The attack phase of a note is the length of time it takes for the volume of the note to rise from a level of zero to the note's peak volume.

As soon as a note reaches its peak volume, it begins to decay. The decay phase of a note is the length of time it takes for the note to decay from its peak volume to a predefined sustain volume.

When the decay phase of a note ends, the note is usually sustained for a certain period of time at a certain volume. Then a release phase begins. During this final phase, the volume of the note drops from its sustain level back down to zero.

When the IIGS Note Synthesizer is used in a program, the ADSR envelope of each sound in the program can be set up by creating a data structure called an instrument record. Then, when a note is played, the address of this record can be passed as a parameter to the Note Synthesizer call `NoteOn`.

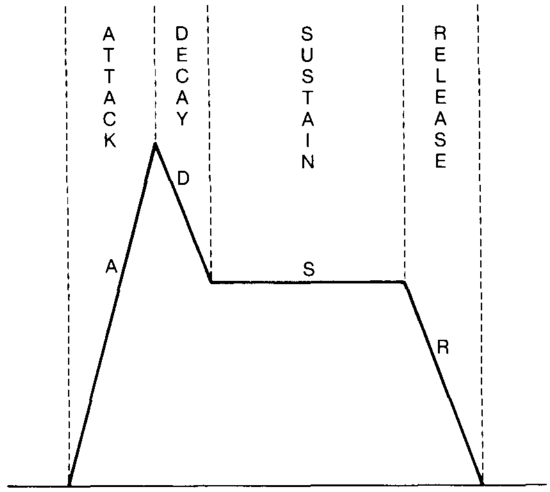

Figure 13–5
ADSR envelope

Initializing the Sound Tool Set and the Note Synthesizer

The Sound Tool Set and the Note Synthesizer, like most tools in the IIGS Toolbox, must be loaded and started before they can be used in a program. In programs that use both tool kits, the Sound Tool Set must be started first because the Note Synthesizer uses part of the Sound Tool Set's direct page.

In the MUSIC.S1 program, the Sound Tool Set is initialized in a program segment labeled `SoundStartUp`, and the Note Synthesizer is started in a segment labeled `NoteStartUp`.

`SoundStartUp`, the call that initializes the Sound Tool Set, is quite straightforward. It takes one parameter—a pointer to a direct page workspace—and returns with the carry clear if there is no error.

`NSStartup`, the call that initializes the Note Synthesizer, takes two parameters. The first parameter is a 2-byte update rate, which determines the rate at which sound envelopes are generated. Update rates are expressed in units of .4 cycles per second, or hertz. In the MUSIC.S1 program, the update rate passed to the `NSStartup` call is the decimal number 70, so the sound envelope used in the program is updated at a rate of 60 times a second, or 60 hertz.

The second parameter passed to the `NSStartup` call is a pointer to an interrupt-driven routine that can be used for note sequencing. No interrupts are used in the MUSIC.S1 program, so the value for this parameter is zero.

How the Note Synthesizer Works

When the Note Synthesizer is used in an application program, a sound generator must be allocated for each voice used in the program. The call to allocate a generator is `AllocGen`.

The `AllocGen` call takes two parameters: a 2-byte space to return a result on the stack and a 1-word value to establish the priority of the generator being allocated.

This is how generator priorities work. Generator priorities can range from 0 to 128. When a generator has a priority of 0, it is free and thus can be allocated. If there are no free generators when a generator is to be allocated, the Note Synthesizer looks for the lowest-priority generator and "steals" it—if it has a priority of less than 128. If a generator has a priority of 128, it cannot be stolen.

When the `AllocGen` call returns, a generator number ranging from 0 to 13 is pushed onto the stack. Then, when a note is to be played by one of the DOC's fifteen generators, the generator can be referred to by its assigned number.

NoteOn Call

When all the generators needed by a program are allocated, the `NoteOn` call can be made each time a note is to begin, and the `NoteOff` call can be made each time a note is to end.

The `NoteOn` call takes four parameters:

- A 1-word generator number (the identification number assigned by the `AllocGen` call)
- A 1-word semitone number (a number ranging from 0 to 127, with the value 60 representing middle C)
- A 1-word volume parameter (a number ranging from 0 to 127)
- A 2-word pointer to an instrument record

The structure of an instrument record is described in the next section. The `NoteOn` call does not return any parameters.

The Structure of an Instrument Record

When the `NoteOn` call is used in a program, one of the parameters passed to it is an instrument record. The instrument record used in the MUSIC.S1 program is shown in table 13–1. The routine that plays notes using the instrument record is in the `PlayNote` segment of the program. The following paragraphs describe each of the fields shown in table 13–1.

The `Envelope` field of an instrument record is composed of up to eight linear segments. Each of these segments has a breakpoint value and an increment value, or slope. During each segment, the volume of the note being played ramps (increases or decreases) from its current value to its breakpoint value. The time that this process takes is determined by the increment value of the note's envelope.

The value of a breakpoint can range from 0 to 127. This range of values represents the level of a sound on a logarithmic scale, with each 16 steps changing the note's amplitude by 6 decibels (dB). The last breakpoint used in an envelope should have a value of 0.

Each increment value in the envelope field can range from 0 through 127. An increment is a value that is added to or subtracted from a note's current level at the update rate passed to the `NoteOn` call, thus changing its

Table 13–1
Instrument Record

Field Number	Field Name	Field Length
1	`Envelope`	24 bytes
2	`ReleaseSegment`	1 byte
3	`PriorityIncrement`	1 byte
4	`PitchBlendRange`	1 byte
5	`VibratoDepth`	1 byte
6	`VibratoSpeed`	1 byte
7	Spare	1 byte
8	`AWaveCount`	1 byte
9	`BWaveCount`	1 byte
10...	`WaveLists`	6 bytes each

frequency at a rate determined by its update rate. The sustain level of an envelope is created by setting an increment value to 0.

An increment is a 2-byte, fixed-point number, that is, a number that represents a fraction. Specifically, the fraction represented by an increment value is the value over 256. Thus, if an increment value is 1, it represents the fraction 1/256 and has to be added to a note's current volume 256 times—over a total elapsed time of 2.56 seconds—to cause the volume of the note to go up by 1.

The `ReleaseSegment` field of an instrument record is a number ranging from 0 to 7. This number determines how many segments it takes for the release of a note to go down to 0. When the release phase diminishes to 0, the note ends.

The `PriorityIncrement` field of an envelope is a number subtracted from the envelope's generator priority when the envelope reaches its sustain phase. Then, when the note reaches its release phase, its priority is cut in half. The priority of each allocated generator is also decremented by 1 each time a new generator is allocated. The purpose of this process is to ensure that the ''oldest'' active generators are ''stolen'' first when a new generator needs to be allocated.

The `PitchBlendRange` of an envelope is the number of semitones that a pitch is raised when its pitchwheel—a constantly incrementing value—reaches 127. The `PitchBlendRange` field controls a sound's vibrato effect. There are only three valid values for this field: 1, 2, and 4.

The `VibratoDepth` field defines the initial depth of a note's vibrato. Vibrato depth can range from 0 to 127, with a value of 0 meaning no vibrato will be used. The `VibratoSpeed` field, a value ranging from 0 to 255, controls the rate of vibrato oscillation. The next field, field 7, is reserved for future expansion.

Each of the digital oscillator chip's generators is made up of a pair of oscillators. Each oscillator in a pair can be used to synthesize as many different kinds of sound waves as desired. In an instrument record, field 8, `AWaveCount`, tells how many kinds of waves are defined for the first oscillator in a pair. Field 9, `BWaveCount`, tells how many kinds of waves are defined for the second oscillator.

In an instrument record, a `WaveList` is a variable length array. Each element in a `WaveList` array has 6 bytes, divided into four fields. Fields 8 and 9 of an instrument record—the `AWaveCount` and `BWaveCount` fields—determine how many `WaveList` arrays the record contains.

The five fields in a `WaveList` array are:

- `TopKey` (1 byte). The highest semitone (ranging from 0 to 127) that a waveform will play. When a note is played by an instrument, the Note Synthesizer examines the `TopKey` field in each of the instrument's waveforms until it finds one that will play the requested note. Therefore, the waveforms listed in each wavelist should be arranged in an order of increasing `TopKey` values, and the last TopKey value in a wavelist should be 127.
- `WaveAddress` (1 byte). This field contains the high byte of the address of a waveform. This value is placed directly into a DOC register that holds a pointer to a waveform address. The waveform stored at the indicated address must be supplied by the program being executed.
- `WaveSize` (1 byte). This 1-byte field is placed directly in a DOC register that defines the size of the wave being accessed.
- `DOCMode` (1 byte). This field determines what mode the DOC uses to play the waveform listed. The most commonly used DOC mode is swap mode, in which two oscillators are used together to form a generator. DOC mode 0 is swap mode. More information on DOC modes are in the *Apple IIGS Hardware Reference*.
- `RelPitch` (2 bytes). This field is a 2-byte word that tunes the waveform in which it appears. The high byte of the word (the second byte of the field) is expressed in semitones, but can be a signed number. The low byte (the first byte of the field) is a value expressed in increments representing 1/256 of a semitone.

The MUSIC Program

Listing 13–1 is a complete listing of the MUSIC.S1 program. Listing 13–2, MUSIC.C, is a C language version of the program. INITQUIT.C, listing 13–3, is an `include` file that handles disk input and output for MUSIC.C. All three listings appear at the end of this chapter.

Type, assemble, and run the MUSIC program, and it will turn your IIGS keyboard into the keyboard of a real sound synthesizer. The keys on the Tab row are the synthesizer's white keys, and the keys on the numbers row are the black keys. The keyboard layout of the MUSIC synthesizer is illustrated in figure 13–6.

After you know how the IIGS produces sound, it isn't difficult to figure out how the MUSIC.S1 program works. It loads and starts up the Sound Tool

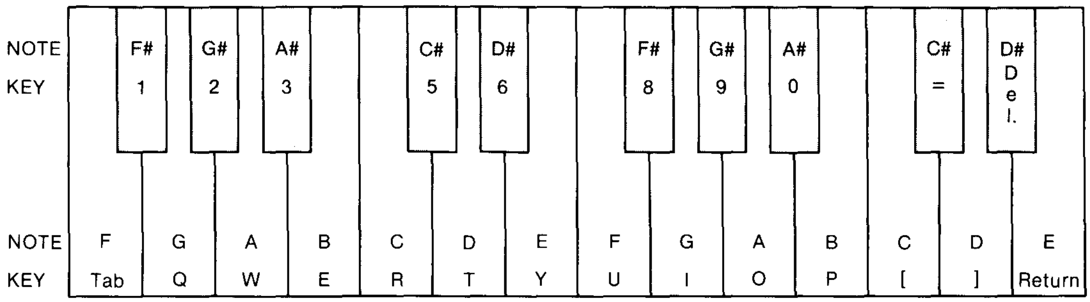

Figure 13–6
Key layout of the MUSIC synthesizer

Set and the Note Synthesizer, and then enters a loop that reads characters typed on the IIGS keyboard. In a segment labeled `GetKey`, the program constantly checks to see if the user has pressed a key on either of the top two rows of the keyboard. If such a key is pressed, the ASCII code of the typed character is converted into a musical semitone, and the program segment labeled `PlayNote` produces the appropriate musical sound. MUSIC.C is a fairly straightforward translation of the program into C.

Not the End

This brings us to the end of this book, but we have barely begun to explore the amazing capabilities of the Apple IIGS. If you have typed, assembled, and executed the Name Game program, and the programs designed to demonstrate the capabilities of the IIGS graphics and sound tools, you have all the supplies to hack your way into the IIGS jungle and see what lies beyond that first row of trees. So happy hunting!

MUSIC.S1, MUSIC.C, and INITQUIT.C Listings

Listing 13–1
MUSIC.S1 program

```
*
*  MUSIC.S1: Creating a Mini-Synthesizer
*

         keep music
         65816 on
         absaddr on
         mcopy music.macros
         longi on
         longa on
```

```
Music    START

         phk
         plb

         jsr SoundStartup

         jsr LoadSound

         jsr NoteStartup

         cli                           ; this seems to be necessary

         PrintLn ' '
         PrintLn '   Your computer is now a mini-synthesizer.'
         PrintLn ' '
         PrintLn '   The white keys are on the TAB row.'
         PrintLn '   The black keys are on the number row.'
         PrintLn ' '
         PrintLn '   Keep shift lock down; press space bar to quit.'

loop     PushWord #0
         PushWord #0                   ; no echo
         _ReadChar                     ; read key the user typed
         pla
         and #$7F                      ; clear high bit
         cmp #$20                      ; space bar?
         beq exit

         jsr GetKey                    ; convert ASCII to a note
         bcs loop                      ; if carry set, no action

         jsr PlayNote                  ; call Note Synthesizer

         bra loop

exit     jsr Shutdown
         _Quit QuitParams

QuitParams anop
         dc i4'0'
         dc i2'0'

         END
```

```
***

GetKey         START

               short m,i
               ldx #23 ; 24 keys, starting from zero
loop           cmp Key,x  ; look for key in table
               beq foundit
               dex
               bpl loop
               jmp nonote ; search over--no note found

foundit        anop
               txa
               adc #$2A ; convert X reg content to a note
               clc      ; found note--clear carry
               jmp fini

nonote         sec      ; no note found--set carry
fini           long m,i
               rts

Key            dc h'09 31 51 32 57 33 45 52 35 54' ; ascii codes
               dc h'36 59 55 38 49 39 4f 30 50 5b'
               dc h'3d 5d 7f 0d'

               END

*
* Start up the tools we'll need
*

SoundStartup START

             _TLStartup
             PushWord #0
             _MMStartup
             ErrorCheck 'Could not call Memory Manager'

             pla
             sta MyID
             _MTStartup
             ErrorCheck 'Could not call Misc Tools'

             PushLong #ToolTable
             _LoadTools
             ErrorCheck 'Could not load sound tools'
```

```
*** GET SOME DIRECT PAGE SPACE AND START UP SOUND TOOLS ***

            PushLong #0                   ; room for handle
            PushLong #$100                ; one page
            PushWord  MyID
            PushWord #$C001               ; type: locked, fixed
            PushLong #0
            _NewHandle
            ErrorCheck 'Not enough memory!'
            pla
            sta    0                      ; using addresses $0000
            pla
            sta    2                      ; and $0002
            lda    [0]                    ; on direct page
            pha
            _SoundStartup
            ErrorCheck 'Could not start up sound tool'
            rts

MyID        ds 2
ToolTable dc i'1,25,0'                    ; one tool: #25, version 0
X
             end

*
*  Load Sound
*

LoadSound     START

*
* This routine creates a square wave, which approximates the
* waveform created by a piano.
*

            ldx #0
            lda #$40

            SetMode8                      ; use 8-bit accumulator

topedge     sta WaveForm,x                ; draw top edge of wave
            inx
            cpx #128
            bne topedge
```

```
lowedge      anop                        ; draw low edge of wave
             lda #$C0
             sta WaveForm,x
             inx
             cpx #256
             bne lowedge

             SetMode16                   ; restore 16-bit accumulator

*
* Now we'll move the wave over to the DOC, using the sound tools.
*

             PushLong #WaveForm          ; arg1: src ptr
             PushWord #0                 ; doc start address
             PushWord #$100              ; byte count
             _WriteRamBlock
             ErrorCheck 'writing wave'
             rts

WaveForm     ds 256

             END

*
*  NoteStartup
*

NoteStartup  START
             PushWord #70                ; 60 Hz updates
             PushLong #0                 ; no IRQ routine for me
             _NSStartup
             ErrorCheck 'Could not start up note synthesizer'
             rts

             END

*
* Now we play the note
*

PlayNote     START
             using NoteData
             sta   SemiTone

             PushWord #0                 ; space for result
             PushWord #64                ; medium priority
             _AllocGen
```

```
            ErrorCheck 'Could not allocate generator'
            pla
            sta GenNum

            PushWord GenNum
            PushWord SemiTone
            PushWord #112              ; medium volume
            PushLong #Piano            ; ptr to piano definition
            _NoteOn
            ErrorCheck 'Problem with NoteOn call'

*
* Normally, we would wait a while before issuing a note off. But
* because a piano has a fast attack and a long release, that
* isn't necessary in this case.
*

            PushWord GenNum
            PushWord SemiTone
            _NoteOff
            ErrorCheck 'Problem with NoteOff calloscillators
            rts

SemiTone    ds 2
GenNum      ds 2

            END

***

NoteData    DATA

Piano       dc i1'127,0,127'           ; env: sharp attack
            dc i1'112,20,1'            ; come down more slowly
            dc i1'0,48,0'              ; slow decay to 0
            dc i1'0,20,5'              ; and release in 112 steps
            dc i1'0,0,0'
            dc i1'0,0,0'
            dc i1'0,0,0'
            dc i1'0,0,0'               ; fill out 8 stages with 0's
            dc i1'3'                   ; release segment
            dc i1'32'                  ; priority inc
            dc i1'2,0,0,0,1,1'         ; pbrange,vibdep,vibf,spare3
```

```
*
* Multi-sampled piano waveforms.
* First oscillator does the attack; second does loop.
*

AWavelist dc i1'127,0,0,0,0,12'          ; topkey,addr,size,ctrl,pitch

BWavelist dc i1'127,0,0,0,0,12'

            END

*
*  Routine that shuts down tools.
*

Shutdown START

            _NSShutdown
            ErrorCheck 'Problem with Note Synthesizer shutdown'
            _SoundShutdown
            rts

            END
```

Listing 13–2
MUSIC.C program

```
#include "initquit.c"

#define space ' '

EventRecord myEvent;
Word waveForm[257];
Instrument piano = {127,0x7F00, /* envelope */
                    112,0x0114,
                    0,0x0030,
                    0,0x0514,
                    0,0x0000,
                    0,0x0000,
                    0,0x0000,
                    0,0x0000,    /* end envelope */
                    3,32,
                    2,0,0,0,1,1,
                    127,0,0,0,0x0C00,    /* aWaveForm */
                    127,0,0,0,0x0C00     /* bWaveForm */
                   };

/* Keys contain the letters in "piano keyboard "#order. The backslashes */
/* are followed by the octal ASCII values of Tab, Delete, and Return.   */
```

```
char keys[] = "\0111Q2W3ER5T6YU8I9OOP[=]\177\015";

main()
{
  StartTools();
  Prompt();
  LoadSound();
  asm {         /*it is necessary to clear interrupts */
      cli;
      }
  NSStartUp(70,nil);
 err("\pUnable to start up Note Synthesizer.\r\r";

  EventLoop ();
  NSShutDown();
  Shutdown();
}

LoadSound() /* it is more efficient to store words instead of bytes */
{
int i;

  for (i=0;i<64;i++)
    waveForm[i] = 0x4040;
  for (i=64;i<128;i++)
    waveForm[i] = 0xC0C0;

  WriteRamBlock (waveForm,0,0x100);
    err("Error in WriteRamBlock");
}

Prompt()
{
  GrafOff();
  printf("Your computer is now a mini-synthesizer\n\n");
  printf("The white keys are on the tab row.\n");
  printf("The black keys are on the number row.\n\n");
  printf("Keep shift lock down: Press space bar to quit.\n");
}

#define KEYMASK keyDownMask+autoKeyMask

EventLoop()
{
Boolean done = false;
Word i;
char theKey;
```

```
  while (!done)
    if(GetNextEvent(KEYMASK,&myEvent)) {
      theKey = (char)(myEvent.message);
      if (theKey == space)
        done = true;
      else
        if((i=findChar(keys,theKey)) < 24)
          PlayNote(i + 0x2A);
    }
}

int findChar(str,c) /* position of c in str, strlen if not present */
char *str;
char c;
{
int i=0;
  while(*str != c) {
    if(*(str++))
      i++;
    else
      break;
  }
  return i;
}

PlayNote(semiTone)
Word semiTone;
{
Word genNum;
  NoteOn((genNum = AllocGen(0,64)),semiTone,112,&piano);
  NoteOff(genNum,semiTone);
}
```

Listing 13–3
INITQUIT.C program

```
#include <TYPES.H>
#include <PRODOS.H>
#include <LOCATOR.H>
#include <MEMORY.H>
#include <MISCTOOL.H>
#include <QUICKDRAW.H>
#include <EVENT.H>
#include <SOUND.H>
#include <NOTESYN.H>
```

```
#define MODE mode320 /* 640 graphics mode def. from quickdraw.h */
#define MaxX 320       /* max X for cursor (for Event Mgr) */
#define dpAttr attrLocked+attrFixed+attrBank /* for
allocating direct page space */

#define err(str) if(_toolErr) SysFailMgr(_toolErr,str)

int  myID;             /* for Memory Manager. */
Handle zp;             /* handle for page 0 space for tools */
QuitRec qParms = {NULL,0};

int toolTable[] = {4,
                    4, 0x0100,  /* QD      */
                    6, 0x0100,  /* Event   */
                    8, 0x0100,  /* Sound   */
                   25, 0x0000   /* NoteSyn */
                   };

StartTools()          /* start up these tools: */
{
   TLStartUp();              /* Tool Locator */
   LoadEmUp();   /* load tools from disk */
   myID = MMStartUp();       /* Mem Manager  */
      err("\pUnable to start up Memory Mgr.\r\r");
   MTStartUp();              /* Misc Tools   */
      err("\pUnable to start up Misc. Tools\r\r");
   ToolInit();  /*start up the rest */
}

LoadEmUp()  /*Load tools, prompt for boot disk if not present */
{
Word response;
Pointer volName;

   GET_BOOT_VOL(&volName);

   LoadTools(toolTable);
   while(_toolErr == volumeNotFound) {
      response = TLMountVolume
(0,195,30," \pPlease insert the disk",volName," \pOK","\pCancel");
      if(response == 1)
         LoadTools(toolTable);
     else {
         TLShutDown();
         QUIT(&qParms);     /* try to exit gracefully */
      }
   }
```

```
   err("\pUnable to load tools\r\r");
}

ToolInit()      /* init the rest of needed tools */
{
   zp = NewHandle(0x500L,myID,dpAttr,0L);   /* reserve 6 pages */
     err("\pUnable to allocate DP space\r\r");

   QDStartUp((int) *zp, MODE, 160, myID);    /* uses 3 pages */
      err("\pUnable to start up QuickDraw.\r\r");
   EMStartUp((int) (*zp + 0x300), 20, 0, MaxX, 0, 200, myID);
      err("\pUnable to start up Event Mgr.\r\r");
   SoundStartUp((int) (*zp + 0x400));
      err("\pUnable to start up Sound Mgr.\r\r");
}

ShutDown()          /* shut down all of the tools we started */
{
   GrafOff();
   SoundShutDown();
   EMShutDown();
   QDShutDown();
   MTShutDown();
   DisposeHandle(zp); /* release our page 0 space */
   MMShutDown(myID);
   TLShutDown();
}
```

PART 3

Appendix

APPENDIX A

The 65C816 Instruction Set

This section is a complete listing of the 65C816 instruction set. It does not include pseudo-operations (also known as pseudo-ops, or directives), which vary from assembler to assembler. Tables A-1, A-2, and A-3 list the abbreviations used in this appendix.

Table A-1
Processor Status (P) Register Flags

Abbreviation	Flag
n	Negative (sign)
v	Overflow
b	Break
d	Decimal
i	Interrupt
z	Zero
c	Carry
m	Memory/accumulator select
e	Emulation

Table A-2
65C816 Registers

Abbreviation	Register
A	Accumulator or 8-bit accumulator
B	B register (high-order byte of 16-bit accumulator)
C	16-bit accumulator
X	X register
Y	Y register
P	Program counter
S	Stack pointer
M	Memory register
D	Direct page register
DBR (or B)	Data bank register
PBR (or K)	Program bank register

Table A-3
Addressing Modes

Abbreviation	Mode
#	Immediate
(a)	Absolute indirect
(a,x)	Absolute indexed indirect
(d)	Direct indirect
(d),y	Direct indirect indexed
(d,x)	Direct indexed indirect
(r,s),y	Stack relative indirect indexed
a	Absolute
a,x	Absolute indexed with X
a,y	Absolute indexed with Y
Acc	Accumulator
al	Absolute long
al,x	Absolute indexed long
d	Direct
d,x	Direct indexed with X
d,y	Direct indexed with Y
i	Implied
r	Program counter relative
r,s	Stack relative
rl	Program counter relative long
s	Stack
xya	Block move
[d]	Direct indirect long
[d],y	Direct indirect indexed long

adc — add with carry — 6502, 65C02, 65C816

Adds the contents of the accumulator to the contents of the effective address specified by the operand. If the P register's carry flag is set, a carry is also added to the result. The sum is stored in the accumulator.

If the accumulator is in 8-bit mode when the `adc` instruction is issued, two 8-bit numbers will be added, and the result of the operation is also 8 bits long. If the operation results in a carry, the carry flag is set.

If the accumulator is in 16-bit mode when the instruction is issued, two 16-bit numbers are added, and the result of the operation is also 16 bits long. If this operation results in a carry, the P register's carry flag is set.

The 65C816 has no instruction for adding without a carry. The `adc` instruction is the only addition instruction available. The carry flag can be cleared, however, with a `clc` instruction prior to an addition operation, and then no carry is added to the result.

It is considered good programming practice to issue a `clc` instruction before beginning any addition sequence. Then a carry bit will not be added to the result by mistake. If the first operation in an addition sequence results in a carry, the carry is added to the next higher-order operation, and each intermediate result correctly reflects the carry from the previous operation.

If the decimal flag is set when an `adc` instruction is issued, the addition operation is carried out in binary coded decimal (BCD) format. If the decimal flag is clear, binary addition is performed.

In emulation mode, `adc` is an 8-bit operation. In native mode, it is a 16-bit operation, with the high-order byte situated in the effective address plus 1.

Flags affected: n, v, z, c

Registers affected: A, P

Addressing Mode	Bytes	Opcode (hex)
`adc (d)`	2	72
`adc (d),y`	2	71
`adc (d,x)`	2	61
`adc (r,s),y`	2	73
`adc d`	2	65
`adc d,x`	2	75
`adc r,s`	2	63
`adc [d]`	2	67
`adc [d],y`	2	77
`adc #`	2 (3)	69
`adc a`	3	6D
`adc a,x`	3	7D

adc a,y	3	79
adc al	4	6F
adc al,x	4	7F

and — logical AND — 6502, 65C02, 65C816

Performs a binary logical AND operation on the contents of the accumulator and the contents of the effective address specified by the operand. See figure A-1. Each bit in the accumulator is ANDed with the corresponding bit in the operand. The result of the operation is stored in the accumulator.

```
      0        0        1        1
AND   0  AND   1  AND   0  AND   1
-------  -------  -------  -------
      0        0        0        1
```

Figure A-1
Truth table for AND

The and instruction is often used as a mask, to clear specified bits in a memory location. When used as a mask, the instruction compares each bit in a memory location with the corresponding bit in the accumulator. Each bit cleared in the memory location clears the corresponding bit in the accumulator. Bits set in the memory location have no effect on their corresponding bits in the accumulator. For example, the sequence

```
lda #$00FF
and MEMLOC
sta MEMLOC
```

clears the high-order byte in MEMLOC, while leaving the low-order byte unchanged.

The and instruction conditions the P register's n and z flags. The n flag is set if the most significant bit of the result of the AND operation is set; otherwise, it is cleared. The z flag is set if the result is 0; otherwise, it is cleared.

In emulation mode, and is an 8-bit operation. In native mode, it is a 16-bit operation, with the high-order byte situated in the effective address plus 1.

Flags affected: n, z

Registers affected: A, P

Addressing Mode	Bytes	Opcode (hex)
and (d)	2	32
and (d),y	2	31
and (d,x)	2	21
and (r,s),y	2	33
and d	2	25
and d,x	2	35

and r,s	2	23
and [d]	2	27
and [d],y	2	37
and #	2 (3)	29
and a	3	2D
and a,x	3	3D
and a,y	3	39
and al	4	2F
and al,x	4	3F

asl — arithmetic shift left — 6502, 65C02, 65C816

Shifts each bit in the accumulator or the effective address specified by the operand one position to the left. See figure A-2. A 0 is deposited into the bit 0 position, and the leftmost bit of the operand is forced into the carry bit of the P register. The result of the operation is left in the accumulator or the affected memory register. The `asl` instruction is often used in assembly language programs as an easy method for dividing by 2.

In emulation mode, `asl` is an 8-bit operation. In native mode, it is a 16-bit operation, with the high-order byte situated in the effective address plus 1.

Flags affected: n, z, c

Registers affected: A, P, M

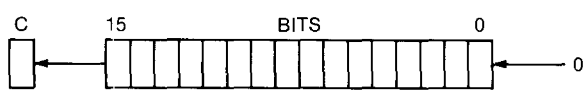

Figure A-2
ASL operation

Addressing Mode	Bytes	Opcode (hex)
asl Acc	1	0A
asl d	2	06
asl d,x	2	16
asl a	3	0E
asl a,x	3	1E

bcc — branch if carry clear — 6502, 65C02, 65C816

(Alias: `blt`.) Tests the P register's carry flag. Executes a branch if the carry flag is clear. Results in no operation if the carry flag is set.

The destination of the branch must be within a range of −128 to +127 memory addresses from the instruction immediately following the `bcc` instruction.

The `bcc` instruction is used for three main purposes:

- To test the carry flag after an arithmetic operation
- To test a bit that has been moved into the carry flag using a rotate, shift, or transfer operation
- To make a programming decision based on a comparison of two values

When `bcc` tests the result of a comparison operation, it comes after a comparison instruction (`cmp`, `cpx`, or `cpy`). When two values are compared with a comparison instruction, data in memory is subtracted from data in the accumulator. This does not affect the value of the accumulator, but it conditions the carry flag as a result of the comparison. The carry flag can then be tested using `bcc`. If the value in the accumulator is less than the value of the operand, the carry is clear and a branch is taken.

If `bcc` results in a branch, a 1-byte signed displacement, fetched from the second byte of the instruction, is sign-extended to 16 bits and added to the program counter. When the address of the branch is calculated, the result is loaded into the program counter, transferring control to that location.

Because the meaning of `bcc` is not intuitively clear when the instruction 9is used as the result of a branch-after-compare operation, the APW assembler also accepts an alias: `blt`, which stands for *branch on less than* and assembles into the same machine language opcode as `bcc`.

Flags affected: None

Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
r	2	90

bcs — branch if carry set — 6502, 65C02, 65C816

(Alias: `bge`.) Tests the P register's carry flag. Executes a branch if the carry flag is set. Results in no operation if the carry flag is clear.

The destination of the branch must be within a range of −128 to +127 memory addresses from the address immediately following the `bcs` instruction.

The `bcs` instruction is used for three main purposes:

- To test the carry flag after an arithmetic operation
- To test a bit that has been moved into the carry flag using a rotate, shift, or transfer operation
- To make a programming decision based on a comparison of two values

When `bcs` tests the result of a comparison operation, it comes after a comparison instruction (`cmp`, `cpx`, or `cpy`). When two values are compared using a comparison instruction, data in memory is subtracted from data in the accumulator. This does not affect the value of the accumulator, but it conditions the carry flag as a result of the comparison. The carry flag can then be tested using bcs. If the value in the accumulator is greater than or equal to the value of the operand, the carry is set and a branch is taken.

If bcs results in a branch, a 1-byte signed displacement, fetched from the second byte of the instruction, is sign-extended to 16 bits and added to the program counter. When the address of the branch is calculated, the result is loaded into the program counter, transferring control to that location.

Because the meaning of bcs is not intuitively apparent when the instruction is used as the result of a branch-after-compare operation, the APW assembler also accepts an alias: bge, which stands for *branch on greater than or equal to* and assembles into the same machine language opcode as bcs.

Flags affected: None

Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
r	2	B0

beq — branch if equal — 6502, 65C02, 65C816

Tests the P register's zero flag. Executes a branch if the zero flag is set—that is, if the result of the last operation which affected the zero flag was 0. Results in no operation if the zero flag is clear.

The destination of the branch must be within a range of -128 to $+127$ memory addresses from the address immediately following the beq instruction.

The beq instruction is used for several purposes:

- To test whether a value that has been pulled, shifted, incremented, or decremented is equal to 0
- To test the value of an index register to determine whether a loop has been completed
- To make a programming decision based on a comparison of two values

When beq tests the result of a comparison operation, it comes after a comparison instruction (cmp, cpx, or cpy). When two values are compared using a comparison instruction, data in memory is subtracted from data in the accumulator. This does not affect the value of the accumulator, but it conditions the carry flag as a result of the comparison. The zero flag can then be tested using beq. If the value in the accumulator is equal to the value of the operand, the zero flag is set and a branch is made.

If beq results in a branch, a 1-byte signed displacement, fetched from the second byte of the instruction, is sign-extended to 16 bits and added to the program counter. When the address of the branch is calculated, the result is loaded into the program counter, transferring control to that location.

Flags affected: None
Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
r	2	F0

bge

branch if carry set

bge is not a 65C816 instruction, but an alias recognized by the APW assembler. When assembled, it generates the same machine language opcode as the assembly language instruction bcs. For further details, see bcs.

bit

test memory bits against accumulator — 6502, 65C02, 65C816

Performs a binary logical AND operation on the contents of the accumulator and the contents of a specified memory location. The contents of the accumulator are not affected, but three flags in the P register are affected.

If any bits in the accumulator and the value being tested match, the z flag is cleared. If no match is found, the z flag is set. Thus, a bit instruction followed by a bne instruction can determine if there is a bit match between the accumulator and the value of the operand. Similarly, a bit instruction followed by a beq instruction detects a no-match condition.

Another result of the bit instruction, in all of its addressing modes except immediate, is that bits 6 and 7 of the value in memory being tested are transferred directly into the v and n flags of the P register. This feature of the bit instruction is often used in signed binary arithmetic. If a bit operation results in the setting of the n flag, the value tested is negative. If the operation results in the setting of the v flag, that indicates an overflow condition when signed numbers are used.

In the immediate addressing mode, the only P register flag affected by the bit instruction is the z flag.

Flags affected in all modes except immediate addressing mode: n, v, z
Flags affected in immediate addressing mode: z
Registers affected: P

Addressing Mode	Bytes	Opcode (hex)
bit d	2	24
bit d,x	2	34

`bit #`	2 (30)	89
`bit a`	3	2C
`bit a,x`	3	3C

blt — branch if less than

`blt` is not a 65C816 instruction, but an alias recognized by the APW assembler. When assembled, it generates the same opcode as the assembly language instruction `bcc`. For further details, see `bcc`.

bmi — branch on minus — 6502, 65C02, 65C816

Tests the P register's n flag. Executes a branch if the n flag is set. Results in no operation if the n flag is clear.

The destination of the branch must be within a range of −128 to +127 memory addresses from the address immediately following the `bmi` instruction.

In operations involving two's complement arithmetic, `bmi` is often used to determine whether a value is negative. In logical operations, it is used to determine if the high bit of a value is set. It is sometimes used to detect whether short loops have counted down past 0.

If `bmi` results in a branch, a 1-byte signed displacement, fetched from the second byte of the instruction, is sign-extended to 16 bits and added to the program counter. When the address of the branch is calculated, the result is loaded into the program counter, transferring control to that location.

Flags affected: None

Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
`r`	2	30

bne — branch if not equal — 6502, 65C02, 65C816

Tests the P register's zero flag. Executes a branch if the zero flag is clear (if the result of the last operation which affected the zero flag was not zero). Results in no operation if the zero flag is set.

The destination of the branch must be within a range of −128 to +127 memory addresses from the address immediately following the `bne` instruction.

The `bne` instruction is used for several purposes:

- To test whether a value that has been pulled, shifted, incremented, or decremented is equal to zero

- To test the value of an index register to determine whether a loop has been completed
- To make a programming decision based on a comparison of two values

When bne tests the result of a comparison operation, it is used after a comparison instruction (cmp, cpx, or cpy). When two values are compared using a comparison instructions, data in memory is subtracted from data in the accumulator. This does not affect the value of the accumulator, but it conditions the carry flag as a result of the comparison. The zero flag can then be tested using bne. If the value in the accumulator is not equal to the value of the operand, the zero flag is set and a branch is made.

If bne results in a branch, a 1-byte signed displacement, fetched from the second byte of the instruction, is sign-extended to 16 bits and added to the program counter. When the address of the branch is calculated, the result is loaded into the program counter, transferring control to that location.

Flags affected: None
Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
r	2	D0

bpl — branch on plus — 6502, 65C02, 65C816

Tests the P register's n flag. Executes a branch if the n flag is clear. Results in no operation if the n flag is set.

The destination of the branch must be within a range of −128 to +127 memory addresses from the address immediately following the bmi instruction.

In operations involving two's complement arithmetic, bpl is often used to determine whether a value is negative. In logical operations, it is used to determine if the high bit of a value is clear.

If bpl results in a branch, a 1-byte signed displacement, fetched from the second byte of the instruction, is sign-extended to 16 bits and added to the program counter. When the address of the branch is calculated, the result is loaded into the program counter, transferring control to that location.

Flags affected: None
Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
r	2	10

bra — branch always — 65C02, 65C816

The bra instruction always results in a branch; no testing is done. There are three major differences between bra and the unconditional jump instruction jmp.

Because signed displacements are used, a statement that uses the `bra` instruction is only 2 bytes long, compared with the 3-byte length of a statement containing a `jmp` instruction. Second, the `bra` instruction uses displacements from the program counter and is thus relocatable. Last, the destination of the branch must be within a range of −128 to +127 memory addresses from the address immediately following the `bra` instruction.

When the branch instruction is used, a 1-byte signed displacement, fetched from the second byte of the instruction, is sign-extended to 16 bits and added to the program counter. After the branch address is calculated, the result is loaded into the program counter, transferring control to that location.

Flags affected: None

Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
r	2	80

brk — break, or software interrupt — 6502, 65C02, 65C816

Forces a software interrupt, usually passing control of the Apple IIGS to the monitor. In programs written for the 65C816, a `brk` instruction can be handled in two ways, depending on whether the processor is in native mode or emulation mode.

If the 65C816 is in native mode, the program bank register is pushed onto the stack. Next, the program counter is incremented by 2 and pushed onto the stack. This incrementation takes place so that a break instruction can be followed by a signature byte identifying which break in a program caused the program to halt.

After the program counter is incremented by 2 and placed on the stack, the program bank register is cleared to 0, and the program counter is loaded from a special `brk` vector situated at $00FFE6 and $00FFE7. (This vector exists only in native mode, not in emulation mode, and that is why there is no need for the P register to have a break flag when the 65C816 is configured for emulation mode. In emulation mode, a `brk` instruction sends a program to vector $00FE6-$00FE7 instead of setting a special flag.) After the break is executed, the P register's decimal flag is cleared to 0.

If the 65C816 is in emulation mode when a `brk` instruction is given, the program counter is incremented by 2 and then pushed onto the stack, just as in native mode. Next, the processor status register, with the b (break) flag set, is pushed onto the stack. The interrupt disable flag is then set, and the program counter is loaded from an interrupt vector at $FFFE and $FFFF.

This is a different interrupt vector from the one the `brk` instruction uses when the 65C816 is in native mode. In native mode, the `brk` instruction does not have its own interrupt vector, as it does in emulation mode, but shares one with hardware interrupts (IRQs). This shared vector is at memory addresses $FFFE and $FFFF. So, after an interrupt occurs, the interrupt handling routine at $FFFE-$FFFF must pull the processor status register off the stack and check the b flag to determine whether program processing was halted by a software interrupt (`brk`) instruction or a hardware interrupt.

If the break was caused by a software interrupt, the flag is set. But hardware IRQs push the P register onto the stack with the b flag clear. So, if the b flag is not set, the program was halted by a hardware IRQ.

When the 65C816 is in native mode, the P register's decimal flag is not modified by the `brk` instruction.

Flags affected in native mode: b and i
Flags affected in emulation mode: b, d, and i
Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
i	2†	00

†`brk` is a 1-byte instruction, but increments the program counter by 2 before pushing it onto the stack.

brl — branch always long — 65C816

The `brl` instruction, like the `bra` instruction, always causes a branch. But `brl` is a 3-byte instruction. The 2 bytes immediately following the opcode form a 16-bit signed displacement from the program counter. Thus, the destination of a `brl` instruction can be anywhere within the current 64K program bank.

After the destination address of the branch is calculated, the result is loaded into the program counter, transferring control to that address.

There are two major differences between the `brl` instruction and the jump instruction `jmp`. The `brl` instruction (like any other branch instruction) is relocatable, but the `jmp` instruction is not. Also, `jmp` executes one cycle faster than `brl`.

Flags affected: None
Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
r	3	82

bvc — branch if overflow clear — 6502, 65C02, 65C816

Tests the overflow (v) flag in the 65C816 P register. Executes a branch if the overflow flag is clear. Results in no operation if the overflow flag is set. This instruction is used primarily in operations involving signed numbers.

The destination of the branch must be within a range of − 128 to + 127 memory addresses from the address immediately following the `bvc` instruction.

The most common use for `bvc` is to detect whether there is an overflow from bit 6 to bit 7 in a calculation involving signed numbers. The instruction can also test bit 6 in a value that has been moved into the v flag by the `bit` instruction.

The v flag can be altered by the instructions `adc`, `sbc`, `clv`, `bit` (in all but immediate mode), `sep`, and `rep`. It is also one of the flags restored from the stack by the `plp` and `rtl` instructions.

Flags affected: None

Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
r	2	50

bvs branch if overflow set 6502, 65C02, 65C816

Tests the overflow (v) flag in the 65C816 P register. Executes a branch if the overflow flag is set. Results in no operation if the overflow flag is clear. This instruction is used primarily in operations involving signed numbers.

The destination of the branch must be within a range of − 128 to + 127 memory addresses from the address immediately following the `bvs` instruction.

The most common use for `bvs` is to detect if there is an overflow from bit 6 to bit 7 in a calculation involving signed numbers. The instruction can also test bit 6 in a value that is moved into the v flag by the `bit` instruction.

The v flag can be altered by the instructions `adc`, `sbc`, `clv`, `bit` (in all but immediate mode), `sep`, and `rep`. It is also one of the flags restored from the stack by the `plp` and `rti` instructions.

Flags affected: None

Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
r	2	70

clc clear carry 6502, 65C02, 65C816

Clears the carry bit of the processor status register. The `clc` instruction should be used prior to addition (`adc`) operations to make sure that the carry flag is clear before addition begins. It should also be used prior to the `xce` (exchange carry flag with emulation bit) instruction when the intent of the instruction is to put the 65C816 into native mode.

Flags affected: c

Registers affected: P

Addressing Mode	Bytes	Opcode (hex)
i	1	18

cld clear decimal mode

Puts the computer into binary mode (its default mode) so that binary operations (the kind most often used) can be carried out properly. When the decimal flag is set, `adc` and `sbc` calculations are carried out in binary coded decimal (BCD) mode.

It is a good practice to clear the decimal flag before beginning arithmetic operations that should be carried out in binary mode, in case the flag has been left in decimal mode following some previous decimal mode operation.

Flags affected: d
Registers affected: P

Addressing Mode	Bytes	Opcode (hex)
i	1	D8

cli clear interrupt disable flag — 6502, 65C02, 65C816

Enables hardware interrupts (IRQs) by clearing the P register's interrupt disable (i) flag. (If the i flag is set, hardware interrupts are ignored.) When the 65C816 starts servicing an interrupt, it finishes the instruction currently executing and then pushes the program counter and the P register on the stack. It then sets the i flag and jumps to one of ten interrupt vectors on page $FF of bank 0. The routine that it finds there must determine the nature of the interrupt and handle it accordingly.

When the interrupt service routine ends with `rti`, the `rti` instruction pulls the P register off the stack and returns to the instruction following the one that was executed just before the interrupt began. The restored P register contains a cleared i flag, so `cli` is ordinarily not necessary. However, if the interrupt service routine is designed to service interrupts that occur while a previous interrupt is still being handled, other interrupt handling routines must be reenabled with a `cli` instruction.

The `cli` instruction is also used to reenable interrupts if they have been disabled to allow the execution of time critical code or other code that cannot be interrupted.

Flags affected: i
Registers affected: P

Addressing Mode	Bytes	Opcode (hex)
i	1	58

clv — clear overflow flag — 6502, 65C02, 65C816

Clears the P register's overflow (v) flag by setting it to 0. Because the v flag is cleared by a nonoverflow result of an `adc` instruction, it is not usually necessary to clear it before an addition operation. So, until the advent of the `bra` (branch always) and `brl` (branch always—long) instructions, the most common use of the `clv` instruction was to force an unconditional branch with a sequence of code such as

```
clv
bvc SOMEPLACE
```

Now, the `bra` and `brl` instructions have made such sequences as this one unnecessary. It is up to you to find some useful function for the `clv` instruction.

Incidentally, there is no specific instruction for setting the v flag. It can, however, be set with the 65C02/65C816 instruction `rep` or by using a `bit` instruction with a mask that has bit 6 set.

Flags affected: v

Registers affected: P

Addressing Mode	Bytes	Opcode (hex)
i	1	B8

cmp — compare with accumulator — 6502, 65C02, 65C816

Compares a specified literal number or the contents of a specified memory location with the contents of the accumulator. The n, z, and c flags of the status register are affected by this operation, and a branch instruction usually follows. The result of the operation thus depends on what branch instruction is used and whether the value in the accumulator is less than, equal to, or more than the value tested.

When a `cmp` instruction is issued, the contents of the specified memory location are subtracted from the accumulator. The result is not stored in the acummulator, but the n, z, and c flags are conditioned as follows.

The z flag is set if the result of the comparison is 0 and cleared otherwise. The n flag is set or cleared by the condition of the sign bit (bit 7) of the result. The c flag is set if the value in the accumulator is greater than or equal to the value in memory. A `bcc` instruction can then be used to detect if the value in the accumulator is greater than the value in memory. The `beq` instruction can detect if the two values are equal. The `bcs` instruction can detect if the value in the accumulator is greater than or equal to the value in memory. A `beq` followed by `bcs` can detect if the value in the accumulator is greater than the value in memory.

Flags affected: n, z, c
Registers affected: P

Addressing Mode	Bytes	Opcode (hex)
cmp (d)	2	D2
cmp (d),y	2	D1
cmp (d,x)	2	C1
cmp (r,s),y	2	D3
cmp d	2	C5
cmp d,x	2	D5
cmp r,s	2	C3
cmp [d]	2	C7
cmp [d],y	2	D7
cmp #	2 (3)	C9
cmp a	3	CD
cmp a,x	3	DD
cmp a,y	3	D9
cmp al	4	CF
cmp al,x	4	DF
cmp i	2	02

cop — coprocessor enable — 65C816

The `cop` instruction allows the 65C816 to turn control over to another processor, such as a math, graphics, or music chip. When the coprocessor completes its assignment, it can return control to the 65C816.

The `cop` instruction, much like a `brk` instruction, causes a software interrupt, but through a different vector: $00FFF4 and $00FFF5.

When a `cop` instruction is issued, the program counter is incremented by 2 and pushed onto the stack. This operation allows the programmer to follow `cop` with a signature byte that specifies which coprocessor handling routine to execute. Unlike the `brk` instruction, which makes a signature byte optional, the `cop` instruction requires a signature byte. Signature bytes from $80 through $FF are reserved by the Western Design Center, which designed the 65C816. Signature bytes in the range $00 through $7F are available for use in application programs.

There are some differences between the way `cop` works in emulation mode and native mode. When a `cop` instruction is used in emulation mode, the program counter is incremented by 2 and pushed onto the stack, the status register is pushed onto the stack, the interrupt disable flag is set, and the program counter is loaded from the emulation mode coprocessor vector at \$FFF4-FFF5. Then, after the command is executed, the P register's d (decimal) flag is cleared.

When a `cop` instruction is issued in native mode, the program counter bank register is pushed onto the stack, the program counter is incremented by 2 and pushed onto the stack, the status register is pushed onto the stack, the interrupt disable flag is set, the program bank register is cleared to 0, and the program counter is loaded from the native mode coprocessor vector at \$00FFE4–00FFE5. Then, after the instruction is issued, the d (decimal) flag is cleared.

Flags affected: d, i
Registers affected: P

Addressing Mode	Bytes	Opcode (hex)
i	2†	02

†`cop` is a 1-byte instruction, but the program counter is incremented by 2 before it is pushed onto the stack, allowing (in fact requiring) a signature byte to be used following the instruction.

cpa

`cpa` is not a 65C816 instruction, but an alias that the APW assembler recognizes as an alternate for the assembly language statement `cmp a`. For further details, see `cmp`.

cpx

compare with X register 6502, 65C02, 65C816

Compares a specified literal number or the contents of a specified memory location with the contents of the X register. The n, z, and c flags of the status register are affected by this operation, and a branch instruction usually follows. The result of the operation thus depends upon what branch instruction is used and whether the value in the X register is less than, equal to, or more than the value tested.

When a `cpx` instruction is issued, the contents of the specified memory location are subtracted from the value of the X register. The result is not stored in the X register, but the n, z, and c flags are conditioned as follows.

The z flag is set if the result of the comparison is 0 and cleared otherwise. The n flag is set or cleared by the condition of the sign bit (bit 7) of the result. The c flag is set if the value in the X register is greater than or equal to the value in memory. A `bcc` instruction can then be used to detect if the value in the X register is greater than the value in memory. A `beq` instruction can detect if the two values are equal. A `bcs` instruction can detect if the value in the X register is greater than or equal to the value in memory. A `beq` instruction followed by bcs can detect if the value in the X register is greater than the value in memory.

Flags affected: n, z, c

Registers affected: P

Addressing Mode	Bytes	Opcode (hex)
cpx d	2	E4
cpx #	2 (3)	E0
cpx a	3	EC

cpy compare with Y register 6502, 65C02, 65C816

Compares a specified literal number or the contents of a specified memory location with the contents of the Y register. The n, z, and c flags of the status register are affected by this operation, and a branch instruction usually follows. The result of the operation thus depends upon what branch instruction is used and whether the value in the Y register is less than, equal to, or more than the value tested.

When a cpy instruction is issued, the contents of the specified memory location are subtracted from the value of the Y register. The result is not stored in the Y register, but the n, z, and c flags are conditioned as follows.

The z flag is set if the result of the comparison is 0 and cleared otherwise. The n flag is set or cleared by the condition of the sign bit (bit 7) of the result. The c flag is set if the value in the Y register is greater than or equal to the value in memory. A bcc instruction can then be used to detect if the value in the Y register is greater than the value in memory. A beq instruction can detect if the two values are equal. A bcs instruction can detect if the value in the Y register is greater than or equal to the value in memory. A beq instruction followed by bcs can detect if the value in the Y register is greater than the value in memory.

Flags affected: n, z, c

Registers affected: P

Addressing Mode	Bytes	Opcode (hex)
cpy d	2	C4
cpy #	2 (3)	C0
cpy a	3	CC

dea

dea is not a 65C816 instruction, but an alias that the APW assembler recognizes as an alternate for the assembly language statement dec a. For further details, see dec.

dec

decrement a memory location 6502, 65C02,65C816

Decrements the contents of a specified memory location by 1. It is important to note that `dec` does not affect the carry flag. Thus, if the value to be decremented is $00, the result of the `dec` operation is $FF.

Because `dec` does not change the carry flag, the carry flag cannot be used to test the outcome of a `dec` operation. A `dec` instruction does condition the n and z flags, however, so they can be used to test a value decremented by `dec`.

Flags affected: n, z
Registers affected: P

Addressing Mode	Bytes	Opcode (hex)
`dec Acc`	1	3A
`dec d`	2	C6
`dec d,x`	2	D6
`dec a`	3	CE
`dec a,x`	3	DE

dex

decrement the X register 6502, 65C02, 65C816

Decrements the contents of the X register by 1. It is important to note that `dex` does not affect the carry flag. Thus, if the value to be decremented is $00, the result of `dex` is $FF.

Because `dex` does not change the carry flag, the carry flag cannot be used to test the outcome of a `dex` operation. The `dex` instruction does condition the n and z flags, however, so they can be used to test a value decremented by `dex`.

Flags affected: n, z
Registers affected: P, M

Addressing Mode	Bytes	Opcode (hex)
`i`	1	CA

dey

decrement the Y register 6502, 65C02, 65C816

Decrements the contents of the Y register by 1. It is important to note that `dey` does not affect the carry flag. Thus, if the value to be decremented is $00, the result of `dey` is $FF.

Because `dey` does not change the carry flag, the carry flag cannot be used to test the outcome of a `dey` operation. The `dey` instruction does condition the n and z flags, however, so they can be used to test a value decremented by `dey`.

Flags affected: n, z
Registers affected: P, M

Addressing Mode	Bytes	Opcode (hex)
i	1	88

eor — exclusive-OR with accumulator — 6502, 65C02, 65C816

Performs an exclusive-OR operation on the contents of the accumulator and a specified literal value or memory location. Each bit in the accumulator is EORed with the corresponding bit in the operand, and the result of the operation is stored in the accumulator. See figure A-3.

The `eor` instruction is often used as a mask, to set specified bits in a memory location. When used as a mask, the instruction compares each bit in a memory location with the corresponding bit in the accumulator. If one and only one of the two bits being compared is set, the corresponding bit in the accumulator is set. Otherwise, the corresponding bit in the accumulator is cleared.

When `eor` is used with a mask consisting of all ones—that is, a mask of $FFFF in native mode or a mask of $FF in emulation mode—each bit in the operand is reversed; that is, each set bit is cleared, and each cleared bit is set. So `eor` is used quite often to reverse the settings of the bits in a word or a byte.

Here is another useful characteristic of `eor`. When it is used on a value twice in succession and with the same operand, the value is changed to another value the first time the instruction is used, and it is converted back into its original value the second time the instruction is used. Because of this characteristic, the `eor` instruction is often used to encode values and then to restore them to their original states. To encode a value using `eor`, just perform an EOR operation on it using an arbitrary 1-byte key. Later, the value can be restored to its original state by performing another EOR operation using the same key.

The `eor` instruction conditions the P register's n and z flags. The n flag is set if the most significant bit of the result of the EOR operation is set; otherwise, it is cleared. The z flag is set if the result is 0; otherwise, it is cleared.

In emulation mode, `eor` is an 8-bit operation. In native mode, it is a 16-bit operation, with the high-order byte situated in the effective address plus 1.

```
      0        0        1        1
EOR   0  EOR   1  EOR   0  EOR   1
-------  -------  -------  -------
      0        1        1        0
```

Figure A-3
Truth table for EOR

Flags affected: n, z
Registers affected: A, P

Addressing Mode	Bytes	Opcode (hex)
`eor (d)`	2	52
`eor (d),y`	2	51
`eor (d,x)`	2	41
`eor (r,s),y`	2	53
`eor d`	2	45
`eor d,x`	2	55
`eor r,s`	2	43
`eor [d]`	2	47
`eor [d],y`	2	57
`eor #`	2 (3)	49
`eor a`	3	4D
`eor a,x`	3	5D
`eor a,y`	3	59
`eor al`	4	4F
`eor al,x`	4	5F

ina

`ina` is not a 65C816 instruction, but an alias that the APW assembler recognizes as an alternate for the assembly language statement `inc a`. For further details, see `inc`.

inc

increment memory 6502, 65C02, 65C816

Increments the contents of a specified memory location by 1. The `inc` instruction neither affects nor is affected by the carry flag. So, if a value being incremented is $FF, the result of the `inc` operation is $00. Because `inc` does not affect the carry flag, the result of an `inc` operation cannot be tested by checking the carry flag. It does condition the n and z flags, however, so they can be used to test the result of an `inc` operation.

Flags affected: n, z
Registers affected: M, P

Addressing Mode	Bytes	Opcode (hex)
`inc Acc`	1	1A
`inc d`	2	E6

`inc d,x`	2	F6
`inc a`	3	EE
`inc a,x`	3	FE

inx

increment X register 6502, 65C02, 65C816

Increments the contents of the X register by 1. The `inx` instruction neither affects nor is affected by the carry flag. So, if a value being incremented is $FF, the result of the `inx` operation is $00. Because `inx` does not affect the carry flag, the result of an `inx` operation cannot be tested by checking the carry flag. It does condition the n and z flags, however, so they can be used to test the result of an `inx` operation.

Flags affected: n, z

Registers affected: X, P

Addressing Mode	Bytes	Opcode (hex)
`i`	1	E8

iny

increment Y register 6502, 65C02, 65C816

Increments the contents of the Y register by 1. The `iny` instruction neither affects nor is affected by the carry flag. So, if a value being incremented is $FF, the result of the `iny` operation is $00. Because `iny` does not affect the carry flag, the result of an `iny` operation cannot be tested by checking the carry flag. It does condition the n and z flags, however, so they can be used to test the result of an `iny` operation.

Flags affected: n, z

Registers affected: X, P

Addressing Mode	Bytes	Opcode (hex)
`i`	1	C8

jmp

jump to address 6502, 65C02, 65C816

Causes program execution to jump to the address specified. When a `jmp` instruction is issued, the program counter is loaded with the target address, causing control of the program in progress to be shifted to that address. When `jmp` is used in the absolute addressing mode, its operand can be either 16 bits or 24 bits. If a 16-bit address is used, the destination of the jump can be anywhere within the current program bank. If a 24-bit address is used, the jump is referred to as a long jump, and its destination address can be anywhere within the address space of the IIGS. When `jmp` carries out a long jump, it has the same result as the `jml` instruction.

Flags affected: None
Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
`jmp (a)`	3	6C
`jmp (a,x)`	3	7C
`jmp a`	3	4C
`jmp al`	4	5C

jsl — jump to subroutine—long — 65C816

Jumps to a subroutine using long (24-bit) addressing. The `jsl` instruction takes a 24-bit operand. It pushes a 24-bit (long) return address onto the stack, then transfers control to the subroutine at the 24-bit address that is the operand. This return address is the address of the last instruction byte (the fourth instruction byte, or the third operand byte), not the address of the next instruction. It is the return address minus 1.

When you issue a `jsl` instruction, the current program counter bank is pushed onto the stack first. Then the high-order byte and the low-order byte of the address are pushed onto the stack in standard 6502/65C816 order, low byte first. The program bank register and the program counter are then loaded with the effective address specified by the operand, and control is transferred to the specified address.

Flags affected: None
Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
`al`	4	22

jsr — jump to subroutine — 6502, 65C02, 65C816

Causes program execution to jump to the address that follows the instruction. That address should be the starting address of a subroutine that ends with the `rts` instruction. When the program reaches the `rts` instruction, execution of the program returns to the next instruction after the `jsr` instruction that caused the jump to the subroutine.

When a `jsr` instruction is issued, the high-order byte and the low-order byte of the address are pushed onto the stack in standard 6502/65C816 order, low byte first. The program counter is then loaded with the effective address specified by the operand, and control is transferred to the specified address.

Flags affected: None

Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
jsr (a,x)	3	FC
jsr a	3	20

lda — load the accumulator — 6502, 65C02, 65C816

Loads the accumulator with the contents of the effective address of the operand. The n flag is set if a value with the high bit set is loaded into the accumulator. The z flag is set if the value loaded into the accumulator is 0.

In emulation mode, `lda` is an 8-bit operation. In native mode, it is a 16-bit operation, with the high-order byte situated in the effective address plus 1.

Flags affected: n, z

Registers affected: A, P

Addressing Mode	Bytes	Opcode (hex)
lda (d)	2	B2
lda (d),y	2	B1
lda (d,x)	2	A1
lda (r,s),y	2	B3
lda d	2	A5
lda d,x	2	B5
lda r,s	2	A3
lda [d]	2	A7
lda [d],y	2	B7
lda #	2 (3)	A9
lda a	3	AD
lda a,x	3	BD
lda a,y	3	B9
lda al	4	AF
lda al,x	4	BF

ldx — load the X register — 6502, 65C02, 65C816

Loads the X register with the contents of the effective address of the operand. The n flag is set if a value with the high bit set is loaded into the X register. The z flag is set if the value loaded into the X register is 0.

In emulation mode, `ldx` is an 8-bit operation. In native mode, it is a

16-bit operation, with the high-order byte situated in the effective address plus 1.

Flags affected: n, z
Registers affected: X, P

Addressing Mode	Bytes	Opcode (hex)
`ldx d`	2	A6
`ldx d,y`	2	B6
`ldx #`	2 (3)	A2
`ldx a`	3	AE
`ldx a,y`	3	BE

ldy — load the Y register — 6502, 65C02, 65C816

Loads the Y register with the contents of the effective address of the operand. The n flag is set if a value with the high bit set is loaded into the Y register. The z flag is set if the value loaded into the Y register is 0.

In emulation mode, `ldy` is an 8-bit operation. In native mode, it is a 16-bit operation, with the high-order byte situated in the effective address plus 1.

Flags affected: n, z
Registers affected: Y, P

Addressing Mode	Bytes	Opcode (hex)
`ldy d`	2	A4
`ldy d,x`	2	B4
`ldy #`	2 (3)	A0
`ldy a`	3	AC
`ldy a,x`	3	BC

lsr — logical shift right — 6502, 65C02, 65C816

Moves each bit in the accumulator one position to the right. See figure A-4. A 0 is deposited into the leftmost position (bit 15 in native mode and bit 7 in emulation mode), and bit 0 is deposited into the carry. The result is left in the accumulator or in the affected memory register.

In emulation mode, `lsr` is an 8-bit operation. In native mode, it is a 16-bit operation, with the high-order byte situated in the effective address plus 1.

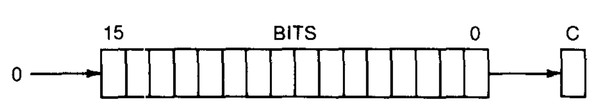

Figure A-4
LSR operation

Flags affected: n, z, c
Registers affected: A, P, M

Addressing Mode	Bytes	Opcode (hex)
`lsr Acc`	1	4A
`lsr d`	2	46
`lsr d,x`	2	56
`lsr a`	3	4E
`lsr a,x`	3	5E

mvn — move block next, or move block negative — 65C816

Copies a block of memory from one RAM address to another. Both `mvn` and the 65C816's other block move instruction, `mvp` (move block previous, or move block positive), can copy blocks from one bank to another and can copy memory blocks that overlap. When overlapping blocks are moved, however, `mvn` should be used only if the starting address of the block to be moved is higher than the starting address of the destination. If the blocks overlap and the starting address of the destination is higher than the starting address of the source, use the `mvp` instruction. Otherwise, part of the block being copied may be overwritten.

The `mvn` instruction takes two operands, each consisting of 1 byte. In programs written using the APW assembler-editor, the operands are separated by a comma. The first operand specifies the bank containing the block to be moved, and the second specifies the bank to which the block will be moved.

The source address, destination address, and length of the move are passed to the `mvn` instruction in the X, Y, and C (double accumulator) registers. The X register holds the source address, the Y register holds the destination address, and the C register holds the length of the block being moved, minus 1. For example, if the C register holds the value $00FF, 256 bytes (or $FF bytes in hexadecimal notation) are moved. The complete C register is always used, regardless of the setting of the m flag.

When you issue an `mvn` instruction, the first byte to be moved is copied from the source address stored in the X register to the destination address stored in the Y register. Then the X and Y registers are incremented. Next, the C register is decremented, and the next byte is moved. This sequence of operations continues until the number of bytes originally stored in the C register, plus 1, are moved (until the value in C is $FFFF).

When the execution of an `mvn` operation is complete, the X and Y registers point to addresses that lie 1 byte beyond the ends of the blocks to which they originally pointed. The data bank register holds the value of the destination bank value (the value of the first byte of the operand).

If the source and destination blocks do not overlap, the source block remains intact after it is copied to the destination.

The operand field of the `mvn` instruction must be coded as two addresses:

first the source, then the destination. When the instruction is assembled into machine code, however, this order is reversed.

If the 65C816 receives an interrupt while an `mvn` move is in progress, the copying of the byte being moved is completed and then the interrupt is serviced. If the interrupt handling routine restores all registers or leaves them intact and ends with an `rti` instruction, the block move is resumed automatically when the interrupt ends.

The `mvn` instruction is useful when blocks of code are moved from one bank to another. For moves that take place within one bank, however, operations that use other algorithms may be faster and more efficient.

If the 65C816 is in emulation mode or the A, X, and Y registers are in 8-bit mode when the `mvn` instruction is issued, both addresses specified in the operand must be on page 0 because the high bytes of the index registers contain zeros.

Flags affected: None
Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
xya	3	54

mvp — move block previous, or move block positive — 65C816

Copies a block of memory from one RAM address to another. Both `mvp` and the 65C816's other block move instruction, `mvn` (move block next, or move block negative), can copy blocks from one bank to another and can copy memory blocks that overlap. When overlapping blocks are moved, however, `mvp` should be used only if the starting address of the block to be moved is lower than the starting address of the destination. If the blocks overlap and the starting address of the destination is higher than the starting address of the source, use the `mvn` instruction. Otherwise, part of the block being copied may be overwritten.

The `mvp` instruction takes two operands, each consisting of 1 byte. In programs written using the APW assembler-editor, the operands are separated by a comma. The first operand specifies the bank containing the block to be moved, and the second specifies the bank to which the block will be moved.

The source address, destination address, and length of the move are passed to the `mvp` instruction in the X, Y, and C (double accumulator) registers. The X register holds the address of the last byte of the block to be moved, the Y register holds the last byte of the destination block, and the C register holds the length of the block being moved, minus 1. For example, if the C register holds the value $00FF, 256 bytes (or $FF bytes in hexadecimal notation) are moved. The complete C register is always used, regardless of the setting of the m flag.

When you issue an `mvp` instruction, the first byte to be moved is copied from the source address stored in the X register to the destination address stored in the Y register. Then the X and Y registers are decremented. Next, the C register is decremented, and the next byte is moved. This sequence of operations continues until the number of bytes originally stored in the C register, plus 1, are moved (until the value in C is $FFFF).

When the execution of an mvp operation is complete, the X and Y registers point to addresses that lie 1 byte past the starting addresses of the blocks to which they originally pointed. The data bank register holds the value of the destination bank value (the value of the first byte of the operand).

If the source and destination blocks do not overlap, the source block remains intact after it is copied to the destination.

The operand field of the mvp instruction must be coded as two addresses: first the address of the last byte of the source block, then the address of the last byte of the destination block. When the instruction is assembled into machine code, however, this order is reversed.

If the 65C816 receives an interrupt while an mvp move is in progress, the copying of the byte being moved is completed and then the interrupt is serviced. If the interrupt handling routine restores all registers or leaves them intact and ends with an rti instruction, the block move is resumed automatically when the interrupt ends.

The mvp instruction is useful when blocks of code are moved from one bank to another. For moves that take place within one bank, however, operations that use other algorithms may be faster and more efficient.

If the 65C816 is in emulation mode or the A, X, and Y registers are in 8-bit mode when the mvp instruction is issued, both addresses specified in the operand must be on page 0 because the high bytes of the index registers contain zeros.

Flags affected: None
Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
xya	3	44

nop — no operation — 6502, 65C02, 65C816

Causes the 65C816 to wait, and do nothing, for one or more cycles. The nop instruction does not affect any registers except the program counter, which is incremented once to point to the next instruction.

The nop instruction is often used to indicate spots in a program where more code may be inserted. For example, in a sequence such as

```
LAB1   nop
       lda #$FF
```

you could insert more lines of source code between the nop and lda instructions, without retyping the line containing the label LAB1.

The nop instruction can also be used to take up time. Every nop in a program takes two cycles, so nop instructions are often used in delay loops and to adjust the speeds of loops in which timing is important.

Flags affected: None

Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
i	1	EA

ora — OR accumulator with memory — 6502, 65C02, 65C816

Performs a binary inclusive-OR operation on the value in the accumulator and a literal value or the contents of a specified memory location or immediate value. See figure A-5. Each bit in the accumulator is ORed with the corresponding bit in the operand, and the result of the operation is stored in the accumulator.

The ora instruction is often used as a mask, to set specified bits in a memory location. When used as a mask, the instruction compares each bit in a memory location with the corresponding bit in the accumulator. Each bit set in the memory location sets the corresponding bit in the accumulator. Bits cleared in the accumulator have no effect on their corresponding bits in the memory location. For example, the sequence

```
lda #$00FF
ora MEMLOC
sta MEMLOC
```

sets all bits in the the low-order byte of MEMLOC, while leaving the high-order byte of MEMLOC unchanged.

The ora instruction conditions the P register's n and z flags. The n flag is set if the most significant bit of the result of the ORA operation is set; otherwise, it is cleared. The z flag is set if the result is 0; otherwise it is cleared.

In emulation mode, ora is an 8-bit operation. In native mode, it is a 16-bit operation, with the high-order byte situated in the effective address plus 1.

Flags affected: n, z

Registers affected: A, P

```
      0         0         1         1
ORA   0   ORA   1   ORA   0   ORA   1
-------   -------   -------   -------
      0         1         1         1
```

Figure A-5
Truth table for ORA

Addressing Mode	Bytes	Opcode (hex)
ora (d)	2	12
ora (d),y	2	11
ora (d,x)	2	01
ora (r,s),y	2	13
ora d	2	05
ora d,x	2	15
ora r,s	2	03
ora [d]	2	07
ora [d],y	2	17
ora #	2 (3)	09
ora a	3	0D
ora a,x	3	1D
ora a,y	3	19
ora al	4	0F
ora al,x	4	1F

pea push effective address 65C816

Pushes a 16-bit operand, always expressed in absolute addressing mode, onto the stack. This operation always pushes 16 bits of data, regardless of the settings of the m and x mode select flags, and the stack pointer is decremented twice.

Although the mnemonic pea would seem to suggest that the value pushed onto the stack must be an address, the instruction can actually be used to place any 16-bit value on the stack. For instance, the instruction

```
pea 0
```

pushes a 0 on the stack. Notice, however, that when pea places a literal value on the stack, the context is unusual. The operand of the instruction is interpreted by the assembler as a literal value. Thus, it does not require the prefix # to designate it as a literal value. So, in this example, a literal 0, not the value of memory address $0000, is pushed onto the stack.

Flags affected: None
Registers affected: S

Addressing Mode	Bytes	Opcode (hex)
s	3	F4

pei push effective indirect address 65C816

Pushes the 16-bit value located at the address formed by adding the direct page offset specified by the operand to the direct page register. Although the mnemonic `pei` may seem to suggest that the instruction's operand must be an address, it actually can be any kind of 16-bit data. The instruction always pushes 16 bits of data, regardless of the settings of the m and x mode select flags.

The first byte pushed is the byte at the direct page offset plus 1 (the high byte of the double byte stored at the direct page offset). The byte of the direct page offset itself (the low byte) is pushed next. The stack pointer then points to the next available stack location, directly below the last byte pushed.

The syntax of the `pei` instruction is that of direct page indirect. Unlike other instructions that use this syntax, however, the effective indirect address, rather than the data stored at that address, is pushed onto the stack.

Flags affected: None

Registers affected: S

Addressing Mode	Bytes	Opcode (hex)
s	2	D4

per — push effective PC relative indirect address — 65C816

Adds the current value of the program counter to the value of a 2-byte operand and pushes the result on the stack. When the program counter is added to the operand, it contains the address of the next instruction (the instruction following the `per` instruction).

After the program counter and the operand are added, the high byte of their sum is pushed onto the stack first, followed by the low byte. After the instruction is completed, the stack pointer points to the next available stack location, immediately below the last byte pushed. The `per` instruction always pushes 16 bits of data, regardless of the settings of the m and x mode select flags.

The syntax used with the `per` instruction is similar to that used with branch instructions; that is, the data to be referenced is used as an operand. The address referred to must be in the current program bank because `per`'s displacement is relative to the program counter.

The `per` instruction is useful when you write self-relocatable code in which a given address (typically the address of a data area) must be accessed. In this kind of application, the address pushed onto the stack is the run time address of the data area, regardless of where the program was loaded in memory. It could be pulled into a register, stored in an indirect pointer, or used on the stack with the stack relative indirect indexed addressing mode to access the data at that location.

The `per` instruction can also be used to push return addresses on the stack, either as part of a simulated branch-to-subroutine or to place the return address beneath the stacked parameters to a subroutine call. When `per` is used in this way, it should be noted that a pushed return address should be the desired return address minus 1.

Flags affected: None

Registers affected: S

Addressing Mode	Bytes	Opcode (hex)
s	3	62

pha push accumulator

Pushes the contents of the accumulator on the stack. The accumulator and the P register are not affected.

In emulation mode, pha is an 8-bit operation. The contents of an 8-bit accumulator are pushed on the stack, and the stack pointer is decremented by 1.

In native mode, pha is a 16-bit operation. The high byte in the accumulator is pushed first, then the low byte. The stack pointer then points to the next available stack location, directly below the last byte pushed.

Flags affected: None

Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
s	1	48

phb push data bank register 65C816

Pushes the value of the data bank register (DBR) onto the stack. The stack pointer then points to the next available stack location, directly below the byte pushed. The data bank register itself is left unchanged.

The 65C816 data bank register is an 8-bit register, so only 1 byte is pushed onto the stack, regardless of the settings of the m and x (mode select) flags.

The phb instruction allows the programmer to save the current value of the data bank register before changing the data bank's value. It is therefore useful when a program in one bank must access data in another. After the data in the other bank is accessed, the original value of the data bank register can be restored.

Flags affected: None

Register affected: S

Addressing Mode	Bytes	Opcode (hex)
s	1	8B

phd

push direct page register 65C816

Pushes the contents of the direct page register (D) onto the stack. The most important use of the phd instruction is to save the value of the D register temporarily, prior to starting an operation that may change its value. After the contents of the D register are saved, a subroutine may specify its own direct page. Then, after the subroutine ends, the original value of the D register can be restored.

Because the direct page register is always a 16-bit register, phd is always a 16-bit operation, regardless of the settings of the m and x (mode select) flags. When you use this instruction, the high byte of the direct page register is pushed first, then the low byte. The direct page register itself is unchanged. The stack pointer then points to the next available stack location, directly below the last byte pushed.

Flags affected: None
Register affected: S

Addressing Mode	Bytes	Opcode (hex)
s	1	0B

phk

push program bank register 65C816

Pushes the current value of the program bank register onto the stack. The phk instruction is often used to set the data bank register and the program bank register so that they contain the same values. A program can then access data in its own bank.

To make the program bank register and the data bank register the same, the following sequence is often used:

```
phk         ; push contents of PBR on stack
plb         ; pull PBR value into data bank register
```

When the phk instruction is used, the program bank register itself is unchanged. The stack pointer then points to the next available stack location, directly below the byte pushed. Because the program bank register is an 8-bit register, only 1 byte is pushed onto the stack, regardless of the settings of the m and x (mode select) flags.

Flags affected: None
Registers affected: S

Addressing Mode	Bytes	Opcode (hex)
s	1	48

php push processor status 6502, 65C02, 65C816

Pushes the contents of the P register on the stack. The P register itself is left unchanged, and no other registers are affected.

Because the program bank register is an 8-bit register, only 1 byte is pushed onto the stack, regardless of the settings of the m and x (mode select) flags.

Note that the P register's e flag, a "hanging flag," is not pushed onto the stack by the `php` instruction. The only way to access the e flag is with the `xce` instruction.

Flags affected: None

Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
s	1	08

phx push X register 65C02, 65C816

Pushes the contents of the X index register onto the stack. The X register itself is unchanged.

When the 65C816 is in emulation mode or when the X and Y registers are set to 8-bit lengths, the 8-bit contents of the X register are pushed onto the stack. The stack pointer then points to the next available stack location, directly below the byte pushed.

When the 65C816 is in native mode and the X and Y registers are set to 16-bit lengths, the 16-bit contents of the X register are pushed onto the stack. The high byte is pushed first, then the low byte. The stack pointer then points to the next available stack location, directly below the last byte pushed.

Flags affected: None

Registers affected: S

Addressing Mode	Bytes	Opcode (hex)
s	1	DA

phy push Y register 65C02, 65C816

Pushes the contents of the Y index register onto the stack. The Y register itself is unchanged.

When the 65C816 is in emulation mode or when the X and Y registers are set to 8-bit lengths, the 8-bit contents of the Y register are pushed onto the stack. The stack pointer then points to the next available stack location, directly below the byte pushed.

When the 65C816 is in native mode and the X and Y registers are set to 16-bit lengths, the 16-bit contents of the X register are pushed onto the stack. The high byte is pushed first, then the low byte. The stack pointer then points to the next available stack location, directly below the last byte pushed.

Flags affected: None
Registers affected: S

Addressing Mode	Bytes	Opcode (hex)
s	1	5A

pla — pull accumulator — 6502, 65C02, 65C816

Removes 1 byte from the stack and deposits it in the accumulator. The n and z flags are conditioned, just as if an `lda` operation had been carried out.

When the 6C816 is in emulation mode or when the accumulator is set to an 8-bit length, the stack pointer is first incremented. Then the byte pointed to by the stack pointer is loaded into the accumulator.

When the 65C816 is in native mode and the accumulator is set to a 16-bit length, the low-order byte of the accumulator is pulled first, followed by the high-order byte.

Flags affected: n, z
Registers affected: A, S, P

Addressing Mode	Bytes	Opcode (hex)
s	1	68

plb — pull data bank register — 6502, 65C02, 65C816

Pulls the 8-bit value on top of the stack into the data bank register (B) and changes the value of the data bank to that value. All instructions referencing data that specifies only 16-bit addresses will then get their bank address from the value pulled into the data bank register. This is the only instruction that can modify the data bank register.

The `plb` instruction is often used with `phk` (push program bank register) to set the data bank register and the program bank register so that they contain the same values. A program can then access data that is in its own bank. To make the program bank register and the data bank register the same, the following sequence is often used:

```
phk          ; push contents of PBR on stack
plb          ; pull PBR value into data bank register
```

When `plb` is used in a program, the stack pointer is incremented, then the byte pointed to by the stack pointer is loaded into the register. Because the bank register is an 8-bit register, `plb` pulls only 1 byte from the stack, regardless of the settings of the m and x (mode select) flags.

Flags affected: n, z

Registers affected: B, S, P

Addressing Mode	Bytes	Opcode (hex)
s	1	AB

pld — pull direct page register — 65C816

Pulls the 16-bit value on top of the stack into the direct page register (D), giving the D register a new value.

The most common use of `pld` is to restore the direct page register to a previous value. When a program calls a subroutine that has its own direct page, the program can save its direct page by using the instruction `phd` (push direct page) before the subroutine is called. When the subroutine ends and control returns to the program that called it, the original state of the D register can be restored with a `pld` instruction.

The direct page register is a 16-bit register, so 2 bytes are pulled from the stack, regardless of the settings of the m and x (mode select) flags. The low byte of the direct page register is pulled first, then the high byte. The stack pointer then points to where the high byte just pulled was stored, and that is the next available stack location.

Flags affected: n, z

Register affected: D, S, P

Addressing Mode	Bytes	Opcode (hex)
s	1	2B

plp — pull processor status register — 6502, 65C02, 65C816

Pulls the 8-bit value on top of the stack into the processor status register (P), changing the value of the P register. `plp` is often used to restore flag settings previously saved on the stack with a `php` (push processor status register) instruction.

It should be noted, however, that the P register's e flag (the emulation mode flag) cannot be retrieved from the stack with a `plp` instruction. That is because it is a "hanging flag" that is not pushed on the stack by the `php` instruction. The only way to set the e flag is with the `xce` instruction.

The status register is an 8-bit register, so only 1 byte is pulled from the stack by the `plp` instruction, regardless of the settings of the m and x (mode select) flags. When the instruction is used in a program, the stack pointer is first incremented. Then the byte pointed to by the stack pointer is loaded into the status register.

Flags affected: All except e
Registers affected: S, P

Addressing Mode	Bytes	Opcode (hex)
s	1	28

plx — pull X register from stack — 65C02, 65C816

Pulls the value on top of the stack into the X index register, destroying the register's previous contents. This operation conditions the n and z flags.

When the 65C816 is in emulation mode or when the X register is set to an 8-bit length, the stack pointer is first incremented. Then the byte pointed to by the stack pointer is loaded into the X register.

When the 65C816 is in native mode and the X register is set to a 16-bit length, the low-order byte of the X register is pulled first, followed by the high-order byte.

Flags affected: n, z
Registers affected: X, S, P

Addressing Mode	Bytes	Opcode (hex)
s	1	FA

ply — pull Y register from stack — 65C02, 65C816

Pulls the value on top of the stack into the Y index register, destroying the register's previous contents. This operation conditions the n and z flags.

When the 65C816 is in emulation mode or the Y register is set to an 8-bit length, the stack pointer is first incremented. Then the byte pointed to by the stack pointer is loaded into the Y register.

When the 65C816 is in native mode and the Y register is set to a 16-bit length, the low-order byte of the Y register is pulled first, followed by the high-order byte.

Flags affected: n, z
Registers affected: S, Y, P

Addressing Mode	Bytes	Opcode (hex)
s	1	7A

rep — reset status bits — 65C816

Clears flags in the status register according to the contents of an 8-bit operand. For each bit set to 1 in the operand, `rep` resets, or clears, the corresponding bit in the status register to 0. For example, if bit 5 in the operand byte is set, bit 5 in the P register is cleared to 0. Zeros in the operand byte have no effect on their corresponding status register bits.

The rep instruction allows the programmer to reset any flag or combination of flags in the status register with a single 2-byte instruction. It is the only direct means of clearing the m flag and the x flag (although instructions that pull the P register affect the m and x flags).

When the 65C816 is in emulation mode, rep does not affect the break flag or bit 5, the 6502's undefined flag bit. In native mode, however, all flags except the e flag (the ''hanging'' flag) can be cleared with the rep instruction. The only way to access the e flag is with the xce instruction.

Flags affected in native mode: All flags except e
Flags affected in emulation mode: All flags except b
Registers affected: P

Addressing Mode	Bytes	Opcode (hex)
#	2	C2

rol — rotate left — 6502, 65C02, 65C816

Moves each bit in the accumulator or a specified memory location one position to the left. See figure A-6.

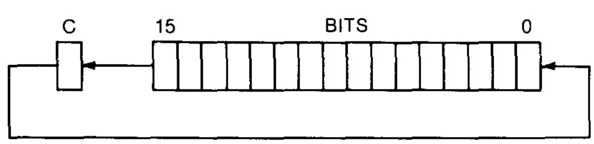

Figure A-6
ROL operation

The carry bit is deposited into the bit 0 location and is replaced by the leftmost bit (bit 15 in native mode and bit 7 in emulation mode) of the accumulator or the affected memory register. The n, z, and c flags are conditioned according to the result of the rotation operation.

Flags affected: n, z, c
Registers affected: A, P, M

Addressing Mode	Bytes	Opcode (hex)
rol Acc	1	2A
rol d	2	26
rol d,x	2	36
rol a	3	2E
rol a,x	3	3E

ror rotate right 6502, 65C02, 65C816

Moves each bit in the accumulator or a specified memory location one position to the right. See figure A-7.

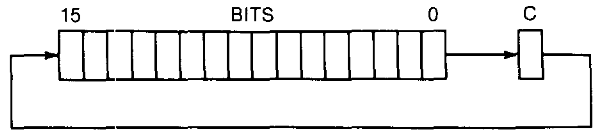

Figure A-7
ROR operation

The carry bit is deposited into the leftmost location (bit 15 in native mode and bit 7 in emulation mode) and is replaced by bit 0 of the accumulator or the affected memory register. The n, c, and z flags are conditioned according to the result of the rotation operation.

Flags affected: n, z, c

Registers affected: A, P, M

Addressing Mode	Bytes	Opcode (hex)
`ror Acc`	1	6A
`ror d`	2	66
`ror d,x`	2	76
`ror a`	3	6E
`ror a,x`	3	7E

rti return from interrupt 6502, 65C02, 65C816

The status of both the program counter and the P register are pulled from the stack and restored to their original values in preparation for resuming the routine in progress when an interrupt occurred. If the 65C816 is in native mode, the program bank register is also pulled from the stack. The `rti` instruction is used to end interrupt handling routines and return control to the program in progress when the interrupt occurred.

The `rti` instruction pulls values off the stack in the reverse order from the way they were pushed onto the stack by a hardware interrupt (IRQ) or a software interrupt (`brk` or `cop`). It is up to the interrupt handling routine to ensure that the values pulled off the stack by `rti` are valid.

When the 65C02 is in native mode, 4 bytes are pulled from the stack: the 8-bit status register, the 16-bit program counter, and the 8-bit program bank register.

In emulation mode, 3 bytes are pulled from the stack: the status register and the program counter.

Flags affected: n, v, b, d, i, z, c
Registers affected: S, P

Addressing Mode	Bytes	Opcode (hex)
s	1	40

rtl return from subroutine long 65C816

Returns to the program in progress from a subroutine that was called using the instruction `jsl` (jump to subroutine—long).

When you call a subroutine using `jsl`, the 8-bit value of the program bank register is pushed onto the stack, followed by the 16-bit value of the program counter.

When you use an `rtl` instruction to end a subroutine, the instruction pulls the value of the program counter from the stack, increments it by 1, and loads the incremented value into the program counter. Then it pulls the program bank register off the stack and loads that into the program bank register.

Flags affected: all except e
Register affected: S, P

Addressing Mode	Bytes	Opcode (hex)
s	1	6B

rts return from subroutine 6502, 65C02, 65C816

At the end of a subroutine, `rts` returns execution of a program to the next address after the `jsr` (jump to subroutine) instruction that caused the program to jump to the subroutine. At the end of an assembly language program, the `rts` instruction returns control of the IIGS to the utility that was in control before the program began.

When a subroutine is called in a 65C816 program with a `jsr` instruction, the contents of the program counter (a 16-bit value) are pushed onto the stack. When the subroutine ends with an `rts` instruction, the `rts` instruction pulls the return address from the stack, increments it, and places it in the program counter, transferring control back to the instruction immediately following the `jsr` instruction.

The instructions `jsr` and `rts` do not push or pull the contents of the program bank register. Therefore, they cannot be used to jump across bank boundaries. When a program must cross a bank boundary to jump to a subroutine, it must use the instructions `jsl` (jump to subroutine—long) and `rtl` (return from subroutine—long).

Flags affected: None
Registers affected: S

Addressing Mode	Bytes	Opcode (hex)
i	1	60

sbc subtract with carry 6502, 65C02, 65C816

Subtracts the content of the effective address of the operand from the contents of the accumulator. The opposite of the carry flag is also subtracted; because subtraction is really reverse addition, the carry flag in a subtraction operation is treated as a borrow.

Because of the way the carry flag is used in subtraction operations, you should set it before a subtraction takes place. Then, if there is a borrow by a lower-order word (or byte in emulation mode) from a higher-order word (or byte in emulation mode), the carry flag is cleared. That causes a borrow, and the result of the subtraction will be accurate.

In emulation mode, `sbc` is an 8-bit operation. In native mode, it is a 16-bit operation, with the high-order byte situated in the effective address plus 1.

The n, v, z, and c flags are all conditioned by the `sbc` instruction, and its result is deposited in the accumulator.

Flags affected: n, v, z, c
Registers affected: A, P

Addressing Mode	Bytes	Opcode (hex)
`sbc (d)`	2	F2
`sbc (d),y`	2	F1
`sbc (d,x)`	2	E1
`sbc (r,s),y`	2	F3
`sbc d`	2	E5
`sbc d,x`	2	F5
`sbc r,s`	2	E3
`sbc [d]`	2	E7
`sbc [d],y`	2	F7
`sbc #`	2 (3)	E9
`sbc a`	3	ED
`sbc a,x`	3	FD
`sbc a,y`	3	F9
`sbc al`	4	EF
`sbc al,x`	4	FF

sec set carry 6502, 65C02, 65C816

Sets the carry flag. The `sec` instruction is often used before the `sbc` instruction so that there is not an extra borrow in the subtraction operation. `sec` is also used prior to an `xce` (exchange carry flag with emulation bit) instruction if the intent of the instruction is to put the 65C816 into 8-bit emulation mode.

Flags affected: c
Registers affected: P

Addressing Mode	Bytes	Opcode (hex)
i	1	38

sed set decimal mode 6502, 65C02, 65C816

Sets the P register's d flag, taking the 65C816 out of normal binary mode and preparing it for operations using BCD (binary coded decimal) numbers. BCD arithmetic is more accurate than binary arithmetic—the usual type of 6510 arithmetic—but it is slower and more difficult to use and consumes more memory. BCD arithmetic is most often used in accounting programs, bookkeeping programs, and floating-point arithmetic.

The decimal flag can be cleared, returning the 65C816 to its default binary mode, with a `cld` (clear decimal flag) instruction.

Flags affected: d
Registers affected: P

Addressing Mode	Bytes	Opcode (hex)
i	1	F8

sei set interrupt disable 6502, 65C02, 65C816

Sets the P register's i (interrupt disable) flag, disabling the processing of hardware interrupts (IRQs). When the i bit is set, maskable hardware interrupts are ignored.

When the 65C816 begins servicing an interrupt, it sets the i flag, so interrupt handling routines that are themselves intended to be interruptable must reenable interrupts with a `cli` (clear interrupt) instruction. If other interrupts are to remain disabled during the interrupt being serviced, a `cli` instruction is not necessary, because the `rti` (return from interrupt) instruction automatically restores the status register with the i flag clear, reenabling interrupts.

Flags affected: i
Registers affected: P

Addressing Mode	Bytes	Opcode (hex)
i	1	78

sep set status bits 65C816

Sets bits in the processor status register according to the value of an 8-bit operand. For each bit set in the operand, `sep` sets the corresponding bit in the status register to 1. For example, if bit 5 is set in the operand byte, bit 5 in the status register is set to 1. Zeros in the operand byte have no effect on their corresponding bits in the P register.

The sep instruction enables the programmer to set any flag or combination of flags in the status register with a single 2-byte instruction. Also, it is the only direct means of setting the m and x (mode select) flags, although instructions that pull the P status register indirectly affect the m and x mode select flags.

When the 65C816 is in emulation mode, sep does not affect the break flag or bit 5, the 6502's non-flag bit.

Flags affected in native mode: n, v, m, x, d, i, z, c

Flags affected in emulation mode: n, v, d, i, z, c

Registers affected: P

Addressing Mode	Bytes	Opcode (hex)
sep #	2	E2

sta — store accumulator — 6502, 65C02, 65C816

Stores the contents of the accumulator in a specified memory location. The contents of the accumulator are not affected.

In emulation mode, sta is an 8-bit operation. In native mode, it is a 16-bit operation, with the high-order byte situated in the effective address plus 1.

Flags affected: None

Registers affected: M

Addressing Mode	Bytes	Opcode (hex)
sta (d)	2	92
sta (d),y	2	91
sta (d,x)	2	81
sta (r,s),y	2	93
sta dta	2	85
sta d,x	2	95
sta r,s	2	83
sta [d]	2	87
sta [d],y	2	97
sta a	3	8D
sta a,x	3	9D
sta a,y	3	99
sta al	4	8F
sta al,x	4	9F

stp — stop the processor — 6502, 65C02, 65C816

Puts the 65C816 into a dormant state until a hardware reset occurs, that is, until the processor's RES pin is pulled low.

The stp instruction is designed for use in battery-powered computers and other systems engineered to support a low-power mode. It can reduce power consumption to almost 0 by putting the 65C816 out of action while it is not actively in use.

Flags affected: None
Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
i	1	DB

stx store X register 6502, 65C02, 65C816

Stores the contents of the X register in a specified memory location. The contents of the X register are not affected.

In emulation mode, `stx` is an 8-bit operation. In native mode, it is a 16-bit operation, with the high-order byte situated in the effective address plus 1.

Flags affected: None
Registers affected: M

Addressing Mode	Bytes	Opcode (hex)
`stx d`	2	86
`stx d,y`	2	96
`stx a`	3	8E

sty store Y register 6502, 65C02, 65C816

Stores the contents of the Y register in a specified memory location. The contents of the Y register are not affected.

In emulation mode, `sty` is an 8-bit operation. In native mode, it is a 16-bit operation, with the high-order byte situated in the effective address plus 1.

Flags affected: None
Registers affected: M

Addressing Mode	Bytes	Opcode (hex)
`sty d`	2	84
`sty d,x`	2	94
`sty a`	3	8C

stz store zero to memory 65C02, 65C816

Stores a 0 in the effective address specified by the operand. The `stz` instruction does not affect any of the flags in the P register.

In emulation mode, `stz` is an 8-bit operation. In native mode, it is a

16-bit operation, with the high-order byte situated in the effective address plus 1.

Flags affected: None

Registers affected: M

Addressing Mode	Bytes	Opcode (hex)
`stz d`	2	64
`stz d,x`	2	74
`stz a`	3	9C
`stz a,x`	3	9E

tax transfer accumulator to X register 6502, 65C02, 65C816

Deposits the value in the accumulator into the X register. The n and z flags are conditioned according to the result of this operation. The contents of the accumulator are not changed.

In emulation mode, `tax` is an 8-bit operation. In native mode, it is a 16-bit operation, with the high-order byte situated in the effective address plus 1.

Flags affected: n, z

Registers affected: X, P

Addressing Mode	Bytes	Opcode (hex)
`i`	1	AA

tay transfer accumulator to Y register 6502, 65C02, 65C816

Deposits the value in the accumulator into the Y register. The n and z flags are conditioned according to the result of this operation. The contents of the accumulator are not changed.

In emulation mode, `tay` is an 8-bit operation. In native mode, it is a 16-bit operation, with the high-order byte situated in the effective address plus 1.

Flags affected: n, z

Registers affected: Y, P

Addressing Mode	Bytes	Opcode (hex)
i	1	A8

tcd — transfer 16-bit accumulator to direct page register — 65C816

Transfers the value in the 16-bit accumulator (C) to the direct page register (D). The value of C is not changed.

When the `tcd` instruction is issued, both bytes in the 16-bit accumulator are copied into the direct page register, regardless of the setting of the m flag. If the accumulator is in 8-bit mode, the low-order byte of the 16-bit accumulator (A) is transferred to the low byte of the direct page register, and the value in the accumulator's ''hidden'' high-order byte (B) is transferred to the high byte of the direct page register.

Flags affected: n, z

Registers affected: D, P

Addressing Mode	Bytes	Opcode (hex)
i	1	5B

tcs — transfer accumulator to stack pointer — 65C816

Transfers the value in the accumulator to the stack pointer. The accumulator's value is unchanged.

If the 65C816 is in native mode, `tcs` transfers both bytes in the 16-bit accumulator (C) to the stack pointer, regardless of the setting of the m flag. The accumulator's low-order byte (A) is transferred to the low byte of the stack pointer, and the value in the accumulator's ''hidden'' high-order byte (B) is transferred to the high byte of the stack pointer. If the 65C816 is in emulation mode, only the 8-bit accumulator (A) is transferred.

The `tcs` and `txs` (transfer the X register to the stack pointer) instructions are the only instructions for changing the value in the stack pointer. They are also the only two transfer instructions that do not alter the n and z flags.

Flags affected: None

Registers affected: S

Addressing Mode	Bytes	Opcode (hex)
i	1	1B

tdc

transfer direct page register to 16-bit accumulator — 65C816

Transfers the value of the direct page register (D) to the 16-bit accumulator (C). The value of the D register is not changed.

The `tdc` instruction transfers 16 bytes, regardless of the setting of the m (accumulator/memory mode) flag. If the accumulator is in 8-bit mode, the accumulator's low-order byte (A) takes the value of the low byte of the direct page register, and the accumulator's "hidden" B register takes the value of the high byte of the direct page register.

Flags affected: n, z
Registers affected: A, B, C, P

Addressing Mode	Bytes	Opcode (hex)
i	1	7B

trb

test and reset memory bits against accumulator — 65C02, 65C816

Logically ANDs the value in the accumulator with the complement of the value in a memory location. This operation clears each memory bit that corresponds to a set bit in the accumulator, while leaving unchanged each memory bit that corresponds to a cleared bit in the accumulator. The result of the operation is stored in the memory location.

In addition, the P register's z flag is conditioned by the result of the AND operation. It sets the z flag if the result of the operation is zero and clears it if the result is not zero. This is the same way that the `bit` instruction conditions the zero flag. But `trb`, unlike `bit`, is a read-modify-write instruction. It not only calculates a result and modifies a flag, but also stores the result in memory.

In emulation mode, `trb` is an 8-bit operation. In native mode, it is a 16-bit operation, with the high-order byte situated in the effective address plus 1.

Flags affected: z
Registers affected: M, P

Addressing Mode	Bytes	Opcode (hex)
trb d	2	14
trb a	3	1C

tsb — test and set memory bits against accumulator — 65C02, 65C816

Logically ORs the value in the accumulator with the value stored in a memory location. This operation sets each memory bit that corresponds to a set bit in the accumulator, while leaving unchanged each memory bit that corresponds to a cleared bit in the accumulator. The result of the operation is stored in the memory location.

In addition, the P register's z flag is conditioned by the result of the OR operation. It sets the z flag if the result of the operation is zero and clears it if the result is not zero. This is the same way that the `bit` instruction conditions the zero flag. But `tsb`, unlike `bit`, is a read-modify-write instruction. It not only calculates a result and modifies a flag, but also stores the result in memory.

In emulation mode, `tsb` is an 8-bit operation. In native mode, it is a 16-bit operation, with the high-order byte situated in the effective address plus 1.

Flags affected: z

Registers affected: M, P

Addressing Mode	Bytes	Opcode (hex)
`tsb d`	2	04
`tsb a`	3	0C

tsc — transfer stack pointer to 16-bit accumulator — 65C816

Transfers the value in the stack pointer (S) to the accumulator. The stack pointer's value is unchanged.

If the 65C816 is in native mode, `tsc` transfers both bytes in the stack pointer to the 16-bit accumulator (C), regardless of the setting of the m flag. The accumulator's low-order byte (A) takes the value of the low byte of the stack pointer, and the value in the accumulator's "hidden" high-order byte (B) takes the value of the high byte of the stack pointer. If the 65C816 is in emulation mode, B always takes a value of 1 because the stack is always page 1 in 8-bit emulation mode.

Flags affected: None

Registers affected: A, B, C

Addressing Mode	Bytes	Opcode (hex)
`i`	1	3B

tsx transfer stack to X register — 6502, 65C02, 65C816

Deposits the value in the stack pointer into the X register. The n and c flags are conditioned according to the result of this operation. The value of the stack pointer is not changed.

When the 65C816 is in emulation mode, `tsx` is an 8-bit operation. If the 65C816 is in native mode and the X register is in 16-bit mode, `tsx` is a 16-bit operation. If the 65C816 is in native mode and the X register is in 8-bit mode, only the low-order byte of the stack pointer is transferred to the X register.

Flags affected: n, c
Registers affected: X, P

Addressing Mode	Bytes	Opcode (hex)
`i`	1	BA

txa transfer X register to accumulator — 6502, 65C02, 65C816

Deposits the value in the X register into the accumulator. The n and z flags are conditioned according to the result of this operation. The value of the X register is not changed.

If the 65C816 is in native mode and the A and X registers are both in 16-bit mode, both bytes of the X register are transferred to the accumulator.

If the 65C816 is in emulation mode and the A and X registers are both in 8-bit mode, the 8-bit X register is transferred to the 8-bit accumulator.

If the 65C816 is in native mode and the accumulator is in 8-bit mode and the X register is in 16-bit mode, the low byte of the X register is moved into the accumulator's low byte (A) and the accumulator's high byte (the "hidden" register B) is not affected by the transfer.

If the 65C816 is in native mode and the accumulator is in 16-bit mode and the X register is in 8-bit mode, the X register is moved into the accumulator's low byte (A) and the accumulator's high byte (B) takes a value of 0.

Flags affected: n, z
Registers affected: A, P

Addressing Mode	Bytes	Opcode (hex)
i	1	8A

txs transfer stack to X register — 6502, 65C02, 65C816

Deposits the value in the X register into the stack pointer. No flags are conditioned by this operation. The value of the X register is not changed.

When the 65C816 is in emulation mode, `txs` is an 8-bit operation. If the 65C816 is in native mode and the X register is in 16-bit mode, `txs` is a 16-bit operation. If the 65C816 is in native mode and the X register is in 8-bit mode, the X register is transferred to the low byte of the stack pointer and the high byte of the stack pointer is zeroed.

Flags affected: None

Registers affected: S

Addressing Mode	Bytes	Opcode (hex)
i	1	9A

txy transfer X register to Y register — 6502, 65C02, 65C816

Transfers the value of the X register to the Y register. The value of the X register is not changed.

When the 65C816 is in emulation mode, `txy` is an 8-bit operation. When the 65C816 is in native mode and the X and Y registers are in native mode, `txy` is a 8-bit operation. When the 65C816 is in native mode and the X and Y registers are in 16-bit mode, `txy` is a 16-bit operation.

Flags affected: n, z

Registers affected: Y, P

Addressing Mode	Bytes	Opcode (hex)
i	1	9B

tya transfer Y register to accumulator — 6502, 65C02, 65C816

Deposits the value in the Y register into the accumulator. The n and z flags are conditioned according to the result of this operation. The value of the Y register is not changed.

If the 65C816 is in native mode and the A and Y registers are both in 16-bit mode, both bytes of the Y register are transferred to the accumulator.

If the 65C816 is in emulation mode and the A and X registers are both in 8-bit mode, the 8-bit Y register is transferred to the 8-bit accumulator.

If the 65C816 is in native mode and the accumulator is in 8-bit mode and the Y register is in 16-bit mode, the low byte of the Y register is moved into the accumulator's low byte (A) and the accumulator's high byte (the "hidden" register B) is not affected by the transfer.

If the 65C816 is in native mode and the accumulator is in 16-bit mode and the Y register is in 8-bit mode, the Y register is moved into the accumulator's low byte (A) and the accumulator's high byte (B) takes a value of 0.

Flags affected: n, z
Registers affected: A, P

Addressing Mode	Bytes	Opcode (hex)
i	1	98

tyx

transfer Y register to X register — 6502, 65C02, 65C816

Transfers the value of the Y register to the X register. The value of the Y register is not changed.

When the 65C816 is in emulation mode, `tyx` is an 8-bit operation. When the 65C816 is in native mode and the X and Y registers are in native mode, `tyx` is an 8-bit operation. When the 65C816 is in native mode and the X and Y registers are in 16-bit mode, `tyx` is an 16-bit operation.

Flags affected: n, z
Registers affected: Y, P

Addressing Mode	Bytes	Opcode (hex)
i	1	BB

wai

wait for interrupt — 65C816

The `wai` instruction puts the 65C816 in a dormant condition during an external event to reduce its power consumption or to provide an immediate response to interrupts so that the proccessor can be synchronized with the external event.

After an interrupt is received, control is generally vectored through one of the hardware interrupt vectors, and an `rti` instruction in an interrupt handling routine returns control to the instruction following the `wai` instruction. But if interrupts are disabled by setting the P register's i flag and a hardware interrupt takes place, the 65C816's wait condition is terminated and control resumes with the next instruction, rather than through the interrupt vectors. This system provides a very fast response to an interrupt, allowing synchronization with external events.

Flags affected: None

Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
i	1	CB

wdm reserved for future expansion 65C816

The letters *wdm* are the initials of William D. Mensch, Jr., the designer of the 65C02 and the 65C816. The `wdm` instruction uses opcode $42, the only one of the 65C816's 256 possible machine language opcodes that is not used. It is left unused so that it can be a gateway to any new assembly language instructions that may be added to the 65C816's instruction set. If new instructions are added, they have to take 2-byte opcodes, and the `wdm` instruction will signify that the next byte is an opcode in the processor's expanded instruction set.

If the `wdm` instruction is used in a IIGS program, it has no effect except to consume time. It behaves like a 2-byte `nop` instruction. But you should not use `wdm` in a program because it would make the program incompatible with any future 65C02 family chips.

Flags affected: None

Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
i	2†	42

†Subject to change in future processors.

xba swap the B and A accumulators

Swaps the contents of the 8-bit A register (the low-order byte of the 16-bit accumulator C) with the contents of the 8-bit B register (the high-order byte of the 16-bit accumulator C). When the 65C816 is in emulation mode, this is the only way to access the accumulator's ''hidden'' B register. The transfer conditions the P register's n and z flags.

The `xba` instruction can be used to invert the low-order, high-order arrangement of a 16-bit value or to store an 8-bit value in the B register. Because it is an exchange, the previous contents of both accumulators are changed, replaced by the previous contents of the other.

Neither the m (mode select) flag nor the e (emulation mode) flag affects this operation.

Flags affected: n, z

Registers affected: A, B, C, P

Addressing Mode	Bytes	Opcode (hex)
i	1	EB

xce

exchange carry and emulation bits 65C816

Swaps the P register's carry flag with the e (emulation mode) flag. The `xce` instruction is the only method for toggling the 65C816 between 16-bit native mode and 8-bit emulation mode.

If the processor is in emulation mode, it can be switched to native mode by clearing the carry bit and then executing the `xce` instruction. If the processor is in native mode, it can be switched to emulation mode by setting the carry bit and then executing the `xce` instruction.

Flags affected: c, e

Registers affected: P

Addressing Mode	Bytes	Opcode (hex)
i	1	FB

APPENDIX B

Apple IIGS Toolbox Calls

This appendix contains most of the calls in the Apple IIGS Toolbox. The calls are listed alphabetically.

Tool Abbreviations

Abbreviation	Meaning
ADB	Apple Desktop Bus
CM	Control Manager
DM	Desk Manager
DLM	Dialog Manager
EM	Event Manager
FM	Font Manager
IM	Integer Math Tool Set
LE	LineEdit Tool Set
LM	List Manager
MM	Memory Manager
MUM	Menu Manager
MTS	Miscellaneous Tool Set
PM	Print Manager

Tool Abbreviations (cont.)

Abbreviation	Meaning
QD	QuickDraw II
SAN	SANE Tool Set
SK	Scheduler
ST	Sound Tool Set
SF	Standard File Operations Tool Set
TT	Text Tool Set
WM	Window Manager

Toolbox Calls

Call	Tool	Call Number	Function
AbsOff	ADB	$1009	Disables automatic polling of an absolute device.
AbsOn	ADB	$0F09	Enables automatic polling from an absolute device.
AddFamily	FM	$0D1B	Allows a family to be added to the Font Manager's list of font families.
AddFontVar	FM	$141B	Allows a pre-existing family to be added to the available font list.
AddPt	QD	$8004	Adds two points and leaves their sum in the destination point.
Alert	DLM	$1715	Invokes an alert defined by a specified alert template.
ASynchADBReceive	ADB	$0D09	Receives data from a ADB device.
AutoAbsPoll	ADB	$1109	Reads flags to determine if automatic polling is on or off.
BeginUpdate	WM	$1E0E	Starts the window drawing procedure when a window is updated.
BlockMove	MM	$2B02	Copies a specified number of bytes from a source to a destination.
BringToFront	WM	$240E	Brings a window to the front and redraws other windows as necessary.
Button	EM	$0D06	Returns the current state of the specified mouse button.
CalcMenuSize	MUM	$1C0F	Sets menu dimensions, either manually or automatically.

Call	Tool	Call Number	Function
CautionAlert	DLM	$1A15	Performs functions similar to those of the Alert routine.
CharBounds	QD	$AC04	Sets a specified rectangle to be the bounds of a specified character.
CharWidth	QD	$A804	Returns the width in pixels of a specified character.
CheckHandle	MM	$1E02	Checks a handle to see if it's valid.
CheckMItem	MUM	$320F	Displays or removes a check mark to the left of a menu item.
CheckUpdate	WM	$0A0E	Checks to see if any windows need updating.
ChooseCDA	DM	$1105	Activates the Desk Manager and displays the CDA menu.
ChooserFont	FM	$161B	Displays a dialog for selection of a new font, size, and/or style.
ClrHeartBeat	MTS	$1403	Removes all tasks from the heartbeat interrupt task queue.
ClampMouse	MTS	$1C03	Sets mouse clamp values and places the mouse at the minimum values.
ClearMouse	MTS	$1B03	Sets the mouse's X and Y axis positions to $0000 or clamp minimums.
ClearScreen	QD	$1504	Sets the words in screen memory to a specified value.
ClearSRQTable	ADB	$1609	Clears the SRQ list of all entries.
ClipRect	QD	$2604	Makes the current port's clip rectangle equal to a given rectangle.
CloseAllNDAs	DM	$1D05	Closes all open NDAs.
CloseDialog	DLM	$0C15	Removes a dialog from the screen and deletes it from the window list.
CloseNDA	DM	$1605	Closes a specified new desk accessory.
CloseNDAbyWinPtr	DM	$1C05	Closes the NDA whose window pointer is passed.
ClosePoly	QD	$C204	Completes the polygon creation started with OpenPoly.
ClosePort	QD	$1A04	Deallocates the regions in a port.

Call	Tool	Call Number	Function
CloseRgn	QD	$6E04	Stops processing of a region and returns the created region.
CloseWindow	WM	$0V0E	Removes a window from the screen and deletes it from the window list.
CompactMem	MM	$1F02	Compacts memory.
CopyRgn	QD	$6904	Copies the contents of a region from one region to another.
CountFamilies	FM	$091B	Returns the number of font families available.
CountFonts	FM	$101B	Returns the number of fonts available that fit a certain description.
CountMItems	MUM	$140F	Returns the number of items in a specified menu.
CreateList	LM	$091C	Creates a list control and returns its handle.
CStringBounds	QD	4AE04	Sets a specified rectangle to be the bounds of a specified C string.
CStringWidth	QD	$AA04	Returns the width of a specified C string.
CtlBootInit	CM	$0110	Called only by the Tool Locator when the system is booted.
CtlNewRes	CM	$1210	Reinitializes resolution and mode.
CtlReset	CM	$0510	Called on system reset.
CtlShutDown	CM	$0310	Deactivates the Control Manager.
CtlStartUp	CM	$0210	Starts up the Control Manager for use by an application.
CtlStatus	CM	$0610	Checks the current status of the Control Manager.
CtlTextDev	TT	$160C	Passes a control code to a specified text device.
CtlVersion	CM	$0410	Returns the version number of the Control Manager.
Dec2Int	IM	$280B	Converts an ASCII string into a 16-bit signed or unsigned integer.
Dec2Long	IM	$290B	Converts an ASCII string into a 32-bit integer.
DefaultFilter	DLM	$3615	Calls a modal or an alert dialog's standard default filter.

Call	Tool	Call Number	Function
DeleteID	MTS	$2103	Deletes all references to a specified user ID.
DeleteMenu	MUM	$0E0F	Removes a specified menu from the menu list.
DeleteMItem	MUM	$100F	Removes a specified item from the current menu.
DelHeartBeat	MTS	$1303	Deletes a specified task from the heartbeat interrupt task queue.
DeskBootInit	DM	$0105	Internal routine called at boot time to initialize the Desk Manager.
DeskReset	DM	$0505	Resets the Desk Manager.
DeskShutDown	DM	$0305	Shuts down the Desk Manager.
DeskStartUp	CM	$0205	Starts up the Desk Manager.
DeskStatus	DM	$0605	Tells if the Desk Manager is active.
Desktop	WM	$0C0E	Keeps track of regions on the desktop and controls desktop pattern.
DeskVersion	DM	$0405	Returns the version number of the Desk Manager.
DialogBootInit	DLM	$0115	Called by the Tool Locator at initialization.
DialogReset	DLM	$0515	Resets the Dialog Manager.
DialogSelect	DLM	$1115	Handles modeless dialog events.
DialogShutDown	DLM	$0315	Shuts down the Dialog Manager.
DialogStartUp	DLM	$0215	Starts up the Dialog Manager.
DialogStatus	DLM	$0615	Indicates if the Dialog Manager is active.
DialogVersion	DLM	$0415	Returns the version number of the Dialog Manager.
DiffRgn	QD	47304	Returns a region that is the difference between two regions.
Disable Increment	ST		Disables auto-increment mode.
DisableDItem	DLM	$3915	Disables a specified item in a specified dialog.
DisableMItem	MUM	$310F	Displays an item in dimmed characters and makes it unselectable.
DisposeAll	MM	$1102	Discards all the handles belonging to a specified user ID.

Call	Tool	Call Number	Function
DisposeControl	CM	$0A10	Deletes a specified control from its window's control list.
DisposeHandle	MM	$1002	Disposes of a specified block and deallocates its handle.
DisposeMenu	MUM	$2E0F	Frees the memory allocated by NewMenu.
DisposeRgn	QD	$6804	Deallocates space for a specified region.
DlgCopy	DLM	$1315	Applies the LineEdit procedure LECopy to an EditLine item.
DlgCut	DLM	$1215	Applies the LineEdit procedure LECut to an EditLine item.
DlgDelete	DLM	$1515	Applies the LineEdit procedure LEDelete to an EditLine item.
DlgPaste	DLM	$1415	Applies the LineEdit procedure LEPaste to an EditLine item.
DoWindows	EM	$0906	Returns the address of the Event Manager's direct page work area.
DragControl	CM	$1710	Pulls a dotted outline of a control around the screen.
DragRect	CM	$1D10	Pulls a dotted outline of a rectangle around the screen.
DragWindow	WM	$1A0E	Pulls around the outline of a window, following mouse movements.
DrawChar	QD	$A404	Draws a specified character at the current pen location.
DrawControls	CM	$1010	Draws all controls currently visible in a specified window.
DrawCString	QD	$A604	Draws a specified C string at the current pen location.
DrawDialog	DLM	$1615	Draws the contents of a specified dialog box.
DrawIcon	QD	$0B12	Draws an icon on the screen.
DrawMember	LM	$0C1C	Redraws a member of the list whose state may have changed.
DrawMenuBar	MUM	$2A0F	Draws the current menu bar, along with any menu titles on the bar.
DrawOneCtl	CM	$2510	Draws a specified control.

Call	Tool	Call Number	Function
DrawString	QD	$A504	Draws a specified string at the current pen location.
DrawText	QD	$A704	Draws specified text at the current pen location.
EMBootInit	EM	$0106	Called at boot time by the Tool Locator.
EmptyRect	QD	$5204	Returns whether or not a specified rectangle is empty.
EmptyRgn	QD	$7804	Checks to see if a specified region is empty.
EMReset	EM	$0506	Returns an error if the Event Manager is active.
EMShutDown	EM	$0306	Shuts down the Event Manager and releases any workspace allocated to it.
EMStartUp	EM	$0206	Initializes the Event Manager and sets the size of the event queue.
EMStatus	EM	$0606	Indicates a nonzero value if the Event Manager is active.
EMVersion	EM	$0406	Returns the version of the Event Manager.
EnableDItem	DLM	$3A15	Enables a specified item in a specified dialog.
EnableMItem	MUM	$300F	Displays an item normally and allows it to be selected.
EndInfoDrawing	WM	$510E	Puts the Window Manager back into a global coordinate system.
EndUpdate	WM	$1F0E	Ends the window drawing procedure started by BeginUpdate.
EqualPt	QD	$8304	Indicates whether two points are equal.
EqualRect	QD	$5104	Compares two rectangles and indicates if they are equal.
EqualRgn	QD	$7704	Compares two regions and tells if they are equal.
EraseArc	QD	$6404	Erases an arc by filling it with the background pattern.
EraseControl	CM	$2410	Makes a specified control invisible.
EraseOval	QD	$5A04	Erases an oval by filling it with the background pattern.
ErasePoly	QD	$BE04	Erases a specified polygon.

Call	Tool	Call Number	Function
EraseRect	QD	$5504	Erases a rectangle by filling it with the background pattern.
EraseRgn	QD	$7B04	Fills the interior of a specified region with the background pattern.
EraseRRect	QD	45F04	Erases the interior of a round rectangle.
ErrorSound	DLM	$0915	Sets the sound procedure for alerts to a specified procedure.
ErrWriteBlock	TT	$1F0C	Writes a block of text to the error output text device.
ErrWriteChar	TT	$190C	Writes a character to the error output text device.
ErrWriteCString	TT	$210C	Writes a C-style string to the error output text device.
ErrWriteLine	TT	$1B0C	Writes a string, plus a carriage return, to the error output text device.
ErrWriteString	TT	$1D0C	Writes a string to the error output text device.
EventAvail	EM	$0B06	Accesses the next available event but leaves it in the queue.
FakeMouse	EM	$1906	Allows an application to use an alternative pointing device.
FamNum2ItemID	FM	$171B	Translates a font family number into a menu item ID.
FamNum2ItemID	FM	$1B1B	Tells if a menu item is displayed in a specified font family.
FFGeneratorStatus	ST	$1108	Reads the first 2 bytes of a block corresponding to a generator.
FFSoundDoneStatus	ST	$1408	Returns the free-form synthesizer sound-playing status.
FFSoundStatus	ST	$1008	Returns the status of all fifteen generators.
FFStartSound	ST	$0E08	Enables the DOC to start generating sound on a particular generator.
FFStopSound	ST	$0F08	Stops sound generators that may be running.
FillArc	QD	$6604	Fills the interior of an arc.
FillOval	QD	$5C04	Fills an oval with a specified pattern.

Call	Tool	Call Number	Function
FillPoly	QD	$C004	Fills a specified polygon with a specified pen pattern.
FillRect	QD	$5704	Fills the interior of a specified rectangle with a specified pattern.
FillRgn	QD	$7D04	Fills the interior of a specified region with a specified pattern.
FillRRect	QD	$6104	Fills a round rectangle with a specified pattern.
FindControl	CM	$1310	Tells in which control the mouse button was pressed.
FindDItem	DLM	$2415	Returns the ID of the item located at a specified point in a dialog.
FindFamily	FM	$0A1B	Returns the family number and name of a particular font family.
FindFontStats	FM	$111B	Places a FontID and a FontStatBits in a specified FontStat record.
FindHandle	MM	$1A02	Returns the handle of the block containing a specified address.
FindWindow	WM	$170E	Tells if the mouse was clicked inside a window, and where.
Fix2Frac	IM	$1C0B	Converts fixed to fraction.
Fix2Long	IM	$1B0B	Converts fixed to long integer.
Fix2X	IM	$1E0B	Converts fixed to extended.
FixAppleMenu	DM	$1E05	Adds the names of new desk accessories to the specified menu.
FixATan2	IM	$170B	Returns a fixed arc tangent of the coordinates of two like inputs.
FixDiv	IM	$110b	Divides two like inputs and returns a rounded fixed result.
FixFontMenu	FM	$151B	Appends the names of available font families onto a specified menu.
FixMenuBar	MUM	$130F	Computes standard sizes for the menu bar and menus.
FixMul	IM	$0F0B	Multiplies two 32-bit fixed inputs and returns a 32-bit fixed result.
FixRatio	IM	$0E0B	Returns a 32-bit fixed-number ratio of a numerator and a denominator.

Call	Tool	Call Number	Function
FixRound	IM	$130B	Takes a fixed input and returns a rounded integer result.
FlashMenuBar	MUM	$0C0F	Flashes the current menu bar using colors set by NewInvertColor.
FlushEvents	EM	$1506	Removes specified queue events until a stop mask is encountered.
FMBootInit	FM	$011B	Called at boot time by the Tool Locator.
FMGetCurFID	FM	$1A1B	Returns the FontID of the current font.
FMGetSysFID	FM	$191B	Returns the FontID of the system font.
FMReset	FM	$051B	Returns an error if the Font Manager is active.
FMSetSysFont	FM	$181B	Loads a specified font into memory, makes it unpurgeable.
FMShutDown	FM	$031B	Shuts down the Font Manager.
FMStartUp	FM	$021B	Initializes the Font Manager for use by an application.
FMStatus	FM	$061B	Returns a nonzero value if the Font Manager is active.
FMVersion	FM	$041B	Returns the version number of the Font Manager.
ForceBufDims	QD	$CC04	Works like SetBufDims, but does not pad MaxFBRExtent.
Frac2Fix	IM	$1D0B	Converts fraction to fixed.
Frac2X	IM	$1F0B	Converts fraction to extended.
FracCos	IM	$150B	Takes a fixed input and returns its fractional cosine.
FracDiv	IM	$120B	Divides two like inputs and returns a rounded fractional result.
FracMul	IM	$100B	Multiplies two fractional inputs and returns a rounded fractional result.
FracSart	IM	$140B	Takes a fractional input and returns a rounded fractional square root.
FracSin	IM	$160B	Takes a fixed input and returns its fractional sine.

Call	Tool	Call Number	Function
FrameArc	QD	$6204	Draws the boundary of an arc using the current pen state and pattern.
FrameOval	QD	$5804	Frames an oval using the current pen state and pen pattern.
FramePoly	QD	$BC04	Frames a specified polygon.
FrameRect	QD	$5304	Frames a rectangle using the current pen state and pen pattern.
FrameRgn	QD	$7904	Frames a specified region using the current pen state and pattern.
FrameRRect	QD	$5D04	Frames a round rectangle using the current pen state and pen pattern.
FreeMem	MM	$1B02	Returns the total number of free bytes in memory.
FrontWindow	WM	$150E	Returns a pointer to the first visible window in the window list.
FWEntry	MTS	$2403	Allows some Apple II entry points to be supported from native mode.
GetAbsClamp	MTS	$2B03	Returns the current values for the absolute device clamps.
GetAbsScale	ADB	$1309	Reads absolute device scaling values.
GetAddr	MTS	$1603	Returns the address of a parameter referenced by the firmware.
GetAddress	QD	$0904	Returns a pointer to a specified table.
GetAlertStage	DLM	$3415	Returns the stage of the last occurrence of an alert.
GetBackColor	QD	$A304	Returns the value of the background color field from the GrafPort.
GetBackPat	QD	$3504	Returns the current background pattern.
GetBarColors	MUM	$180F	Returns the colors for the current menu bar.
GetCaretTime	EM	$1206	Returns the time between blinks of the caret.
GetCharExtra	QD	$D504	Returns the chExtra field from the GrafPort.

Call	Tool	Call Number	Function
GetClip	QD	$2504	Copies the ClipRgn to a specified region.
GetClipHandle	QD	$C704	Returns a copy of the handle to the ClipRgn.
GetColorEntry	QD	$1104	Returns the value of a color in a specified color table.
GetColorTable	QD	$0F04	Fills a color table with the contents of another color table.
GetContentDraw	WM	$480E	Returns a pointer to the routine that draws a window's contents.
GetContentOrgin	WM	$3E0E	Returns values used to set the origin of a window's port.
GetContentRgn	WM	$2F0E	Returns a handle to a specified window's content region.
GetControlDItem	DLM	$1E15	Returns a handle to the control for a specified item.
GetCtlAction	CM	$2110	Returns the current value of a specified control's CtlAction field.
GetCtlDpage	CM	$1F10	Returns the value of the Control Manager's direct page.
GetCtlParams	CM	$1C10	Returns a specified control's additional parameter settings.
GetCtlRefCon	CM	$2310	Returns the current value of a specified control's CtlRefCon field.
GetCtlTitle	CM	$0D10	Returns the value in a specified control's CtlData field.
GetCtlValue	CM	$1A10	Returns a specified control's current CtlValue field.
GetCursorAdr	QD	$8F04	Returns a pointer to the current cursor record.
GetDAStrPtr	DM	$1405	Returns the pointer to a table of desk accessory strings.
GetDataSize	WM	$400E	Returns the height and width of the data area of a specified window.
GetDblTime	EM	$1106	Sets the time required between mouse clicks for a double click.
GetDefButton	DLM	$3715	Returns the ID of the default button item in a specified dialog.
GetDefProc	WM	$310E	Returns the address of a routine that controls a window's behavior.

Call	Tool	Call Number	Function
GetDItemBox	DLM	$2815	Returns the display rectangle of a specified item.
GetDItemType	DLM	$2615	Returns the type of a specified item.
GetDItemValue	DLM	$2E15	Returns the current value of a specified item.
GetErrGlobals	TT	$0E0C	Returns the current values for the error output device's global parameters.
GetErrorDevice	TT	$140C	Returns the type of driver installed as the error output device.
GetFamInfo	FM	$0B1B	Returns the name of a font family that has a specified family number.
GetFamNum	FM	$0C1B	Returns a family number corresponding to a given font family name.
GetFGSize	QD	$CF04	Returns the size of the font globals record.
GetFirstDItem	DLM	$2A15	Returns the ID of the first item in a specified dialog.
GetFirstWindow	WM	$520E	Returns the first window in the Window Manager's window list.
GetFont	QD	$9504	Returns a handle to the current font.
GetFontFlags	QD	$9904	Returns the current font flags.
GetFontGlobals	QD	$9704	Returns information about the current font in the specified record.
GetFontID	QD	$D104	Returns the FontID in the GrafPort.
GetFontInfo	QD	$9604	Returns information about the current font in the specified record.
GetFontLore	QD	$D904	Returns information about the current font in the specified record.
GetForeColor	QD	$A104	Returns the current foreground color from the GrafPort.
GetFrameColor	WM	$100E	Returns the color of a specified window's frame.
GetFuncPtr	TL	$0B01	Returns a pointer, less 1, to a specified tool function.

Call	Tool	Call Number	Function
GetGrafProcs	QD	$4504	Returns a pointer to the current port's GrafProcs record.
GetHandleSize	MM	$1802	Returns the size of a specified block.
GetInfoDraw	WM	$4A0E	Returns a pointer to a window's information bar drawing procedure.
GetInfoRefCon	WM	$350E	Returns a value associated with an information bar drawing routine.
GetInputDevice	TT	$120C	Returns the type of driver installed as the input device.
GetIRQEnable	MTS	$2903	Returns the interrupt enable states of certain interrupt sources.
GetIText	DLM	$1F15	Returns the text of a specified StatText or EditLine item.
GetListDefProc	LM	$0E1C	Returns a pointer to the list control's definition procedure.
GetInGlobals	TT	$0C0C	Returns current values for the input device's global parameters.
GetMasterSCB	QD	$1704	Returns a copy of the master SCB.
GetMaxGrow	WM	$420E	Returns the maximum values to which a window's content can grow.
GetMenuBar	MUM	$0A0F	Returns the handle of the current menu bar.
GetMenuFlag	MUM	$200F	Returns the menu flag for a specified menu.
GetMenuMgrPort	MUM	$1B0F	Returns a pointer to the Menu Manager's port.
GetMenuTitle	MUM	$220F	Returns a pointer to the title of a menu.
GetMHandle	MUM	$160F	Returns a handle to a menu record.
GetMItem	MUM	$250F	Returns a pointer to the name of an item.
GetMItemFlag	MUM	$270F	Returns information about a specified menu item.
GetMItemMark	MUM	$340F	Returns the character displayed to the left of a specified item.

Call	Tool	Call Number	Function
GetMItemStyle	MUM	$360F	Returns the text style for a specified menu item.
GetMouse	EM	$0C06	Returns the current mouse location.
GetMouseClamp	MTS	$1D03	Returns the current mouse clamp values.
GetMTitleStart	MUM	$1A0F	Returns the starting position for the leftmost title within the current menu bar.
GetMTitleWidth	MUM	$1E0F	Returns the width of a menu title.
GetNewDItem	DLM	$3315	Adds a new item to a specified dialog's item list using a template.
GetNewID	MTS	$2003	Creates a new user ID.
GetNewModalDialog	DLM	$3215	Creates a modal dialog and returns a pointer to its GrafPort.
GetNextDItem	DLM	$2B15	Returns the ID of the next item in a specified dialog.
GetNextEvent	EM	$0A06	Returns the next available event of a specified type or types.
GetNextWindow	WM	$2A0E	Returns a pointer to the next window in the window list.
GetNumNDAs	DM	$1B05	Returns the total number of new desk accessories currently installed.
GetOSEvent	EM	$1606	Removes a specified type of event from the queue.
GetOutGlobals	TT	$0D0C	Returns current values for the output device's global parameters.
GetOutputDevice	TT	$130C	Returns the type of driver installed as the output device.
GetPage	WM	$460E	Returns the number of pixels to be scrolled by a scroll bar's "page" region.
GetPen	QD	$2904	Returns the pen location.
GetPenMask	QD	$3304	Returns the pen mask at the specified location.
GetPenMode	QD	$2F04	Returns the pen mode from the current port.
GetPenPat	QD	$3104	Copies the current port's pen pattern into a specified location.

Call	Tool	Call Number	Function
GetPenSize	QD	$2D04	Returns the current pen size at the place indicated.
GetPenState	QD	$2B04	Returns the pen state from the GrafPort.
GetPicSave	QD	$3F04	Returns the contents of the PicSave field in the GrafPort.
GetPixel	QD	$8804	Returns the pixel below and to the right of a specified point.
GetPolySave	QD	$4304	Returns the contents of the PicSave field in the GrafPort.
GetPort	QD	$1C04	Returns a pointer to the current port.
GetPortLoc	QD	$1E04	Returns the current port's map information structure.
GetPortRect	QD	$2004	Returns the current port's port rectangle.
GetRectInfo	WM	$4F0E	Sets a rectangle in which objects can be drawn in an information bar.
GetRgnSave	QD	$4104	Returns the contents of the RgnSave field in the GrafPort.
GetRomFont	QD	$D804	Fills a specified record with information about the font in ROM.
GetSCB	QD	$1304	Returns the value of a specified scan-line control byte (SCB).
GetScrap	SK	$0D16	Copies scrap information to the specified handle.
GetScrapCount	SK	$1216	Returns the current scrap count.
GetScrapHandle	SK	$0E16	Returns a copy of the handle for the scrap of the specified type.
GetScrapPath	SK	$1016	Returns a pointer to the pathname used for the clipboard file.
GetScrapSize	SK	$0F16	Returns the size of the specified scrap.
GetScrapState	SK	$1316	Returns a flag indicating the current state of the scrap.
GetScroll	WM	$440E	Returns the number of pixels scrolled by the arrows in a scroll bar.
GetSoundVolume	ST	$0C08	Reads the volume setting for a generator.

Call	Tool	Call Number	Function
GetSpaceExtra	QD	$9F04	Returns the value of the SpExtra field from the GrafPort.
GetStandardSCB	QD	$0C04	Returns a copy of the standard SCB in the low-order byte of the word.
GetStructRgn	WM	$2E0E	Returns a handle to a specified window's structure region.
GetSysBar	MUM	$110F	Returns the handle of the current system menu bar.
GetSysField	QD	$4904	Returns the contents of the SysField in the GrafPort.
GetSysFont	QD	$B304	Returns a handle to the current system font.
GetTableAddress	ST	$0B08	Returns the jump table address for low-level routines.
GetTextFace	QD	$9B04	Returns the current text face.
GetTextMode	QD	$9D04	Returns the current text mode.
GetTextSize	QD	$D304	Not yet implemented at the time of this writing.
GetTick	MTS	$2503	Returns the current value of the tick counter.
GetTSPtr	TL	$0901	Returns a pointer to the function pointer table of a specified tool set.
GettSysWFlag	WM	$4C0E	Indicates if a specified window is a system window.
GetUpdateRgn	WM	$300E	Returns a handle to a specified window's update region.
GetUserField	QD	$4704	Returns the contents of the UserField field in the GrafPort.
GetVector	MTS	$1103	Returns the vector address for a specified vector reference number.
GetVisHandle	QD	$C904	Returns a copy of the handle to the VisRgn.
GetVisRgn	QD	$B504	Copies the contents of the VisRgn into a specified region.
GetWAP	TL	$0C01	Gets the pointer to the work area for a specified tool set.
GetWControls	WM	$330E	Returns the handle of the first control in the window's control list.

Call	Tool	Call Number	Function
GetWFrame	WM	$2C0E	Returns a bit array that describes a window's frame type.
GetWKind	WM	$2B0E	Tells if a window is a system or application window.
GetWMgrPort	WM	$200E	Returns a pointer to the Window Manager's port.
GetWRefCon	WM	$290E	Returns a value passed to NewWindow or WRefCon by an application.
GetWTitle	WM	$0E0E	Returns a pointer to a specified window's title.
GetZoomRect	WM	$370E	Returns a pointer to a rectangle representing a window's zoomed size.
GlobalToLocal	QD	$8504	Converts a point from global coordinates to local coordinates.
GrafOff	QD	$0B04	Turns off super high-resolution graphics mode.
GrafOn	QD	$0A04	Turns on super high-resolution graphics mode.
GrowSize	CM	$1E10	Returns the height and width of the grow box control.
GrowWindow	WM	$1B0E	Expands or shrinks a window, corresponding to mouse movements.
HandToHand	MM	$2A02	Copies a block of bytes from a source handle to a destination handle.
HandToPtr	MM	$2902	Copies a block of bytes from a handle to a pointer.
Hex2Int	IM	$240B	Converts a hexadecimal string into a 16-bit unsigned integer.
Hex2Long	IM	$250B	Converts a hexadecimal string into a 32-bit unsigned integer.
HexIt	IM	$2A0B	Converts a 16-bit unsigned integer into a hexadecimal string.
HideControl	CM	$0E10	Makes a specified control invisible.
HideCursor	QD	$9004	Hides the cursor, that is, decrements the cursor level.
HideDItem	DLM	$2215	Erases a specified item from a specified dialog.
HidePen	QD	$2704	Decrements the pen level.
HideWindow	WM	$120E	Makes a specified window invisible.

Call	Tool	Call Number	Function
HiLiteControl	CM	$1110	Changes the way a specified control is highlighted.
HiliteMenu	MUM	$2C0F	Highlights or unhighlights the title of a specified menu.
HiliteWindow	WM	$220E	Highlights or unhighlights a window's title bar, as appropriate.
HiWord	IM	$180B	Returns the high-order word of a long input.
HLock	MM	$2002	Locks a block specified by a handle.
HLockAll	MM	$2102	Locks all the blocks belonging to a specified user ID.
HomeMouse	MTS	$1A03	Positions mouse at the minimum clamp position.
HUnlock	MM	$2202	Unlocks a block specified by a handle.
HUnLockAll	MM	$2302	Unlocks all the blocks for a specified user ID.
IMBootInit	IM	$010B	Called at boot time by the Tool Locator.
IMReset	IM	$050B	Called when a system reset occurs.
IMShutDown	IM	$030B	Standard tool call.
IMStartUp	IM	$020B	Standard tool call.
IMStatus	IM	$060B	Returns a nonzero value indicating that the Integer Math Tool Set is active.
IMVersion	IM	$040B	Returns the version of the Integer Math Tool Set.
InflateText-Buffer	QD	$D704	Inflates the text buffer to a specified size, if necessary.
InitColorTable	QD	$0D04	Returns a copy of the standard color table for the current mode.
InitCursor	QD	$CA04	Reinitializes the cursor.
InitMouse	MTS	$1803	Sets mouse clamp values to $000 minimum and $3FF maximum.
InitPalette	MUM	$2F0F	Reinitializes the palettes used to draw the apple on the menu bar.
InitPort	QD	$1904	Initializes specified memory locations as a standard port.
InitTextDev	TT	$150C	Initializes a specified text device.

Call	Tool	Call Number	Function
InsertMenu	MUM	$0D0F	Inserts a menu into the menu list.
InsertMItem	MUM	$0F0F	Inserts an item into a menu.
InsetRect	QD	$4C04	Inserts a specified rectangle by specified displacements.
InsetRgn	QD	$7004	Shrinks or expands a specified region.
InstallCDA	DM	$0F05	Installs a specified classic desk accessory in the system.
InstallFont	FM	$0E1B	Loads a given font into memory and makes it current and unpurgeable.
InstallNDA	DM	$0E05	Installs a specified new desk accessory in the system.
Int2Dec	IM	$260B	Returns a string representing a 16-bit signed or unsigned integer.
Int2Hex	IM	$220B	Converts a 16-bit unsigned integer into a hexadecimal string.
IntSource	MTS	$2303	Enables or disables certain interrupt sources.
InvalRect	WM	$3A0E	Accumulates a rectangle into the current window port's update region.
InvalRgn	WM	$3B0E	Accumulates a region into the current window port's update region.
InvertArc	QD	$6504	Inverts the pixels inside a specified arc.
InvertOval	QD	$5B04	Inverts the pixels inside a specified oval.
InvertPoly	QD	$BF04	Inverts a specified polygon.
InvertRect	QD	$5604	Inverts the pixels in the interior of a specified rectangle.
InvertRgn	QD	$7C04	Inverts the pixels in the interior of a specified region.
InvertRRect	QD	$6004	Inverts the pixels inside a specified round rectangle.
IsDialogEvent	DLM	$1015	Determines if an event should be handled as part of a dialog.
ItemID2FamNum	FM	$171B	Translates a menu item ID into a font family number.
KillControls	CM	$0B10	Disposes of all controls associated with a specified window.

Call	Tool	Call Number	Function
KillPoly	QD	$C304	Disposes of a specified polygon.
LEActivate	LE	$0F14	Highlights current selection range in specified text.
LEBootInit	LE	$0114	Called at boot time by the Tool Locator.
LEClick	LE	$0D14	Using mouse clicks, draws a caret and highlights selected text.
LECopy	LE	$1314	Copies selected text into the LineEdit scrap.
LECut	LE	$1214	Removes selected text and places it in the LineEdit scrap.
LEDeactivate	LE	$1014	Unhighlights current selection range in specified text.
LEDelete	LE	$1514	Removes selected text and redraws the remaining text.
LEDispose	LE	$0A14	Releases the memory allocated for a specified edit record.
LEFromScrap	LE	$1914	Copies the desk scrap to the LineEdit scrap.
LeGetScrapLen	LE	$1C14	Returns the size of the LineEdit scrap in bytes.
LEGetTextHand	LE	$2214	Returns a handle to the text of a specified edit record.
LEGetTextLen	LE	$2314	Returns the length of the text of a specified edit record.
LEIdle	LE	$0C14	Places a blinking caret at the insertion point in a specified line.
LEInsert	LE	$1614	Inserts specified text into other text, and redraws the updated text.
LEKey	LE	$1114	Places a character in text and leaves an insertion point after it.
LENew	LE	$0914	Allocates text space and returns a handle to a new edit record.
LEPaste	LE	$1414	Replaces selected text with the contents of the LineEdit scrap.
LEReset	LE	$0514	Returns an error if LineEdit is active.
LEScrapHandle	LE	$1B14	Returns a handle to the LineEdit scrap.
LESetCaret	LE	$1F14	Sets the CaretHook field in the edit record to a specified pointer.

Call	Tool	Call Number	Function
LESetHilite	LE	$1E14	Sets the HiliteHook field in the edit record to a specified pointer.
LESetJust	LE	$2114	Sets up the LineEdit Tool Set record for left, right, or center justification.
LESetScrapLen	LE	$1D14	Sets the size of the LineEdit scrap to a specified number of bytes.
LESetSelect	LE	$0E14	Sets the selection range in the specified text.
LESetText	LE	$0B14	Incorporates a copy of specified text into a specified edit record.
LEShutDown	LE	$0314	Shuts down the LineEdit Tool Set and discards the LineEdit scrap.
LEStartUp	LE	$0214	Initializes the LineEdit Tool Set and allocates a handle for the LineEdit scrap.
LEStatus	LE	$0614	Indicates whether or not the LineEdit Tool Set is active.
LETextBox	LE	$1814	Draws specified text in a specified rectangle.
LETextBox2	LE	$2014	Draws specified text in a specified rectangle.
LETextBox2	LE	$2014	Draws text in a specified rectangle, with specified justification.
LEToScrap	LE	$1A14	Copies the LineEdit scrap to the desk scrap.
LEVersion	LE	$0414	Returns version number of the LineEdit Tool Set.
Line	QD	$3D04	Draws a line from the current pen location to the specified displacements.
LineTo	QD	$3C04	Draws a line from the current pen location to a specified point.
ListBootInit	LM	$011C	Called at boot time by the Tool Locator.
ListReset	LM	$051C	Called when a system reset occurs.
ListShutDown	LM	$031C	Standard tool call.
ListStartUp	LM	$021C	Standard tool call.
ListStatus	LM	$061C	Returns a nonzero value indicating that the List Manager is active.
ListVersion	LM	$041C	Returns the version of the List Manager.

Call	Tool	Call Number	Function
LLDBitMap	PM	$1C13	Prints part or all of a specified QuickDraw II bit map.
LLDControl	PM	$1B13	Resets the printer and generates linefeeds and formfeeds.
LLDShutDown	PM	$1A13	Deallocates any memory allocated by LLDStartUp.
LLDStartUp	PM	$1913	Sets up the necessary environment for low-level drivers.
LLDText	PM	$1D13	Prints a stream of text using the native facilities of the printer.
LoadFont	FM	$121B	Finds a specified font, loads it, and makes it current.
LoadOneTool	TL	$0F01	Loads a specified tool from disk and checks its version.
LoadScrap	SK	$0A16	Reads the desk scrap from the scrap file into memory.
LoadSysFont	FM	$131B	Makes the system font current without requiring its font ID.
LoadTools	TL	$0E01	Loads specified RAM-based tool sets from disk into memory.
LocalToGlobal	QD	$8404	Converts a point from local coordinates to global coordinates.
Long2Dec	IM	$270B	Returns a string representing a 32-bit signed or unsigned integer.
Long2Fix	IM	$1A0B	Converts long integer to fixed.
Long2Hex	IM	$230B	Converts a 32-bit unsigned integer into a hexadecimal string.
LongDivide	IM	$0D0B	Divides two 32-bit inputs, producing a quotient and a remainder.
LongMul	IM	$0C0B	Multiplies two 32-bit inputs and produces a 64-bit result.
LoWord	IM	$190B	Returns the low-order word of a long input.
MapPoly	QD	$C504	Maps a polygon from a source rectangle to a destination rectangle.
MapPt	QD	$8A04	Maps a point from a source rectangle to a destination rectangle.
MapRect	QD	$8B04	Maps a rectangle from a source rectangle to a destination rectangle.

Call	Tool	Call Number	Function
MapRgn	QD	$8C04	Maps a region from a source rectangle to a destination rectangle.
MaxBlock	MM	$1C02	Returns the size of the largest free block in memory.
MenuBootInit	MUM	$010F	Called at boot time.
MenuKey	MUM	$090F	Allows the user to type a character to select a menu item.
MenuNewRes	MUM	$290F	Restyles the menu after the screen resolution changes.
MenuRefresh	MUM	$0B0F	Called when the application is not using the Window Manager.
MenuReset	MUM	$050F	This call does nothing.
MenuSelect	MUM	$2B0F	Controls highlighting and pull-down action when an item is selected.
MenuShutDown	MUM	$030F	Closes the Menu Manager's port and frees any allocated menus.
MenuStartUp	MUM	$020F	Initializes the Menu Manager at application startup.
MenuStatus	MUM	$060F	Checks the current status of the Menu Manager.
MenuVersion	MUM	$040F	Returns the version of the Menu Manager.
MMBootInit	MM	$0102	Initializes the Memory Manager at boot time.
MMReset	MM	$0502	Used by the system at reset time.
MMShutDown	MM	$0302	An application makes this call when it is terminating.
MMStartUp	MM	$0202	An application makes this call when starting up.
MMStatus	MM	$0602	Returns status indicating the Memory Manager is active.
MMVersion	MM	$0402	Returns the version of the Memory Manager.
ModalDialog	DLM	$0F15	Repeatedly gets and handles events in a modal dialog's window.
ModalDialog2	DLM	$2C15	Repeatedly gets and handles events in a modal dialog's window.
Move	QD	$3B04	Moves the current pen location by specified X and Y displacements.
MoveControl	CM	$1610	Moves a specified control to a new location within its window.

Call	Tool	Call Number	Function
MovePortTo	QD	$2204	Changes the location of the current GrafPort's PortRect.
MoveTo	QD	$3A04	Moves the current pen location to the specified point.
MoveWindow	WM	$190E	Moves a window to another part of the screen, not changing its size.
MTBootInit	MTS	$0103	Called at boot time.
MTReset	MTS	$0503	Clears the heartbeat task pointer and sets the mouse flag to "not found."
MTShutDown	MTS	$0303	This call is not used in this tool set.
MTStartUp	MTS	$0203	This call is not used in this tool set.
MTSTATUS	MTS	$0603	Returns status indicating the Miscellaneous Tool Set is active.
MTVersion	MTS	$0403	Returns the version of the Miscellaneous Tool Set.
Multiply	IM	$090B	Multiplies two 16-bit inputs and produces a 32-bit result.
Munger	MTS	$2803	Manipulates bytes in a string of bytes.
NewControl	CM	$0910	Creates a control and returns a handle to it.
NewDItem	DLM	$0D15	Adds a new item to a dialog's item list.
NewHandle	MM	$0902	Creates a new block and returns the handle to the block.
NewList	LM	$101C	Resets the list control according to a specified list record.
NewMenu	MUM	$2D0F	Allocates space for a menu list and its items.
NewMenuBar	MUM	$150F	Creates a default menu bar with no menus.
NewModalDialog	DLM	$0A15	Creates a modal dialog and returns a pointer to its port.
NewModeless-Dialog	DLM	$0B15	Creates a modeless dialog and returns a handle to its port.
NewRgn	QD	$6704	Allocates space for a new region.
NewWindow	WM	$090E	Creates a window and returns a pointer to its GrafPort.
NextMember	LM	$0B1C	Searches a list record for a specified member and returns its value.

Call	Tool	Call Number	Function
NoteAlert	DLM	$1915	Performs the same functions as the Alert routine.
ObscureCursor	QD	$9204	Hides the cursor until the mouse moves.
OffsetPoly	QD	$C404	Offsets a polygon by specified X and Y displacements.
OffsetRect	QD	$4B04	Offsets a specified rectangle by specified displacements.
OffsetRgn	QD	$6F04	Moves a region a distance specified by X and Y displacements.
OpenNDA	DM	$1505	Opens a specified new desk accessory.
OpenPoly	QD	$C104	Opens a polygon structure for updating, and returns its handle.
OpenPort	QD	$1804	Initializes specified memory locations as a standard port.
OpenRgn	QD	$6D04	Allocates memory to hold information about a region being created.
OSEventAvail	EM	$1706	Accesses the next event of a given type but leaves it in the queue.
PackBytes	MTS	$2603	Packs bytes into a special format that uses less storage space.
PaintArc	QD	$6304	Paints the interior of an arc using the current pen state and pattern.
PaintOval	QD	$5904	Paints the interior of an oval using the current pen state and pattern.
PaintPixels	QD	$7F04	Transfers a region of pixels.
PaintPoly	QD	$BD04	Paints the interior of a polygon using the current pen state and pattern.
PaintRect	QD	$5404	Paints the interior of a rectangle using the current pen state and pattern.
PaintRgn	QD	$7A04	Paints the interior of a region using the current pen state and pattern.
PaintRRect	QD	$5E04	Paints the interior of a round rectangle using the current pen state and pattern.
ParamText	DLM	$1B15	Substitutes text in StatText and LongStatText items.
PenNormal	QD	$3604	Sets the pen state to the standard state.

Call	Tool	Call Number	Function
PinRect	WM	$210E	Pins a specified point inside a specified rectangle.
PMBootInit	PM	$0113	Called at boot time by the Tool Locator.
PMReset	PM	$0513	Internal routine called only at system reset.
PMShutDown	PM	$0313	Shuts down the Print Manager.
PMStartUp	PM	$0213	Initializes the Print Manager for use by an application.
PMStatus	PM	$0613	Indicates whether or not the Print Manager is active.
PMVersion	PM	$0413	Returns the version number of the Print Manager.
PosMouse	MTS	$1E03	Positions mouse at specified coordinates.
PostEvent	EM	$1406	Posts an event at the end of the event queue.
PPToPort	QD	$D604	Transfers pixels from a source pixel map to the current port.
PrChoosePrinter	PM	$1613	Displays a dialog for selecting a printer and port driver.
PrCloseDoc	PM	$0F13	Closes the GrafPort being used for printing.
PrClosePage	PM	$1113	Finishes the printing of the current page.
PrDefault	PM	$0913	Sets a print record to default values for the appropriate printer.
PrError	PM	$1413	Returns the result code left by the last Print Manager routine.
PrJobDialog	PM	$0C13	Displays a dialog for setting print quality, pages to print, and so on.
PrOpenDoc	PM	$0E13	Initializes a GrafPort for use in printing and returns its pointer.
PrOpenPage	PM	$1013	Begins a new page.
PrPicFile	PM	$1213	Prints a spooled document.
PrSetError	PM	$1513	Given an error number, performs a corresponding function.
PrStlDialog	PM	$0B13	Displays a dialog for inputting page setup information.
PrValidate	PM	$0A13	Checks if a print record is compatible with the Print Manager.

Call	Tool	Call Number	Function
Pt2Rect	QD	$5004	Creates a rectangle using an upper left point and a lower right point.
PtInRect	QD	$4F04	Detects if a specified point is in a specified rectangle.
PtInRgn	QD	$7504	Determines where a specified point is within a specified region.
PtrToHand	MM	$2802	Copies a specified number of bytes from a source to a destination.
PurgeAll	MM	$1302	Purges all of the purgeable blocks for a specified user ID.
PurgeHandle	MM	$1202	Purges a specified purgeable handle.
PutScrap	SK	$0C16	Appends specified data to data in the scrap of the same type.
QDBootInit	QD	$0104	Initializes QuickDraw II at boot time.
QDReset	QD	$0504	Resets QuickDraw II.
QDShutDown	QD	$0304	Frees up any buffers allocated for QuickDraw II.
QDStartUp	QD	$0204	Starts up QuickDraw II.
QDStatus	QD	$0604	Returns if QuickDraw II is active.
QDVersion	QD	$0404	Returns the version of QuickDraw II.
Random	QD	$8604	Returns a pseudorandom number in the range −32768 to +32767.
Read Next	ST		Reads the next address pointed to by the GLU address register.
Read RAM	ST		Reads any specified Ensoniq RAM location.
Read Register	ST		Reads any register within the DOC.
ReadASCIITime	MTS	$0F03	Reads elapsed time since 00:00:00, Jan. 1, 1904.
ReadBParam	MTS	$0C03	Reads date from a specified parameter in battery RAM.
ReadBRam	MTS	$0A03	Reads 252 bytes of data, plus 4 checksum bytes, from battery RAM.
ReadChar	TT	$220C	Reads a character from an input text device; returns it on the stack.

Call	Tool	Call Number	Function
ReadKeyMicroData	ADB	$0A09	Receive data from the microcontroller.
ReadKeyMicro-Memory	ADB	$0B09	Reads a data byte from the microcontroller ROM.
ReadLine	TT	$240C	Reads an input string and writes it to a buffer.
ReadMouse	MTS	$1703	Returns mouse position, status, and mode.
ReadRamBlock	ST	$0A08	Reads any number of locations from DOC RAM into a buffer.
ReadTimeHex	MTS	$0D03	Returns current time in hexadecimal format.
ReAllocHandle	MM	$0A02	Reallocates a block that was purged.
RectInRgn	QD	$7604	Checks whether a specified rectangle intersects a specified region.
RectRgn	QD	$6C04	Sets a specified region to a rectangle described by the input.
RefreshDesktop	WM	$390E	Redraws the entire desktop and all windows.
RemoveDItem	DLM	$OE15	Removes an item from a dialog and erases it from the screen.
ResetAlertStage	DLM	$3515	Resets a dialog so that its next stage is treated as its first stage.
ResetMember	LM	$0F1C	Searches a list record for a member and clears its select flag.
RestAll	DM	$0C05	Restores variables that were saved in calling a desk accessory.
RestoreBufDims	QD	$CE04	Restores QuickDraw's internal buffers to the sizes described in a record.
RestoreHandle	MM	$0B02	Reallocates a purged handle.
RestScrn	DM	$0A05	Restores the screen area saved by the Desk Manager.
SANEBootInit	SAN	$010	Not used in this tool set.
SANEDecStr816	SAN	$0A0A	Contains numeric scanners and formatter.
SANEDecStr816	SAN	$0B0A	Contains elementary, financial, and random number functions.
SANEFP816	SAN	$090A	Contains basic arithmetic operations and IEEE auxiliary operations.

Call	Tool	Call Number	Function
SANEReset	SAN	$050A	Not used in this tool set.
SANEShutDown	SAN	$030A	Zeros out the work area pointer for the SANE Tool Set.
SANEStartup	SAN	$020A	Starts up the SANE Tool Set for use by an application.
SANEStatus	SAN	$060A	Returns true, indicating the SANE Tool Set is active.
SANEVersion	SAN	$040A	Returns the version number of the SANE Tool Set.
SaveAll	DM	$0B05	Saves all variables preserved in activating a desk accessory.
SaveBufDims	QD	$CD04	Saves QuickDraw II's buffer sizing information in an 8-byte record.
SaveScrn	DM	$0905	Saves the 80-column text screens in banks $00, 01, EO, and E1.
ScalePt	QD	$8904	Scales a point from a source rectangle to a destination rectangle.
SchAddTask	SK	$0907	Adds a task to Scheduler's queue.
SchBootInit	SK	$0107	Initializes the flags and counters used by the Scheduler.
SchFlush	SK	$0A07	Flushes all tasks in the Scheduler's queue.
SchReset	SK	$0507	Reinitializes flags and counters.
SchShutDown	SK	$0307	Not used in this tool set.
SchStartUp	SK	$0207	Not used in this tool set.
SchStatus	SK	$0607	Returns true, indicating the Scheduler is active.
SchVersion	SK	$0407	Returns the version number of the Scheduler.
ScrapBootInit	SK	$0116	Internal routine called at load time to initialize the Scrap Manager.
ScrapReset	SK	$0516	Internal routine to reset the Scrap Manager.
ScrapShutDown	SK	$0316	Shuts down the Scrap Manager.
ScrapStartUp	SK	$0216	Starts up the Scrap Manager.
ScrapStatus	SK	$0616	Always returns true: if the Scrap Manager is loaded, it is active.

Call	Tool	Call Number	Function
ScrapVersion	SK	$0416	Returns the version number of the Scrap Manager.
ScrollRect	QD	$7E04	Scrolls a rectangle inside certain boundaries.
SDivide	IM	$0A0B	Divides two 16-bit inputs and produces two 16-bit signed results.
SectRect	QD	$4D04	Places the intersection of two rectangles in a third rectangle.
SectRgn	QD	$7104	Places the intersection of two regions in a third region.
SelectMember	LM	$0D1C	Selects a list member and scrolls the list so it is at the top.
SelectWindow	WM	$110E	Makes a specified window the active window.
SelIText	DLM	$2115	Sets the selection range or the insertion point for an EditLine item.
SendBehind	WM	$140E	Places a window behind a specified window, redrawing as appropriate.
SendInfo	ADB	$0909	Sends data to the microcontroller.
ServeMouse	MTS	$1F03	Returns the mouse interrupt status.
SetAbsClamp	MTS	$2A03	Sets clamp values for an absolute device to new values.
SetAbsScale	ADB	$1209	Sets up scaling for absolute devices.
SetAllSCBs	QD	$1404	Sets all scan-line control bytes (SCBs) to a specified value.
SetBackColor	QD	$A204	Sets a GrafPort's background color field to a specified value.
SetBackPat	QD	$3404	Sets the background pattern to a specified pattern.
SetBarColors	MUM	$170F	Sets the normal, inverse, and outline colors of the current menu bar.
SetBufDims	QD	$CB04	Sets the size of the QuickDraw II clipping and text buffers.
SetCharExtra	QD	$D404	Sets the chExtra field in the GrafPort to the specified value.
SetClip	QD	$2404	Copies a specified region into the ClipRgn.
SetClipHandle	QD	$C604	Sets the ClipRgn handle field in the GrafPort to a specified value.

Call	Tool	Call Number	Function
SetColorEntry	QD	$1004	Sets the value of a color in a specified color table.
SetColorTable	QD	$0E04	Sets a color table to specified values.
SetContentDraw	WM	$490E	Sets the pointer to a routine that redraws a window's content region.
SetContentOrigin	WM	$3F0E	Sets the origin of the window's port when handling an update event.
SetCtlAction	CM	$2010	Sets a specified control's CtlAction field to a new action.
SetCtlIcons	CM	$1810	Provides a handle to a specified new icon font.
SetCtlParams	CM	$1B10	Sets new parameters to a control's definition procedure.
SetCtlRefCon	CM	$2210	Sets a specified control's CtlReCon field to a new value.
SetCtlTitle	CM	$0C10	Sets a control's title to a given string and redraws the control.
SetCtlValue	CM	$1910	Sets a control's CtlValue field and redraws the control.
SetCursor	QD	$8E04	Sets the cursor to an image passed in a specified cursor record.
SetDAFont	DLM	$1C15	Sets the font of a given window's port to a specified font number.
SetDAStrPtr	DM	$1305	Allows a program to change the built-in classic desk accessories.
SetDataSize	WM	$410E	Sets the height and width of the data area of a specified window.
SetDefButton	DLM	$3815	Sets the ID of the default button to a specified ID.
SetDefProc	WM	$320E	Sets the address of the routine that defines a window's behavior.
SetDItemBox	DLM	$2915	Changes the display rectangle of an item to a new display rectangle.
SetDItemType	DLM	$2715	Changes the specified item to the new desired type.
SetDItemValue	DLM	$2F15	Sets the value of an item to a new value and redraws the item.
SetEmptyRgn	QD	$6A04	Sets a specified region to the empty region.

Call	Tool	Call Number	Function
`SetErrGlobals`	TT	$0B0C	Sets the global parameters for the error output device.
`SetErrorDevice`	TT	$110C	Sets the error output device to a specified type and location.
`SetEventMask`	EM	$1806	Sets the system event mask to the specified event mask.
`SetFont`	QD	$9404	Sets the current font to the specified font.
`SetFontFlags`	QD	$9804	Sets the font flags to the specified value.
`SetFontID`	QD	$D004	Sets the `FontID` field in the GrafPort.
`SetForeColor`	QD	$A004	Sets a GrafPort's foreground color field to a specified value.
`SetFrameColor`	WM	$0F0E	Sets the color of a specified window's frame.
`SetGrafProcs`	QD	$4404	Sets a GrafPort's `GrafProcs` field to a specified value.
`SetHandleSize`	MM	$1902	Changes the size of a specified block.
`SetHeartBeat`	MTS	$1203	Installs a specified task into the heartbeat interrupt task queue.
`SetInfoDraw`	WM	$160E	Sets the pointer to a window's information bar drawing procedure.
`SetInfoRefCon`	WM	$360E	Sets a value associated with a window's information bar drawing routine.
`SetInputDevice`	TT	$0F0C	Sets the input device to a specified type and location.
`SetIntUse`	QD	$B604	Tells if the cursor should be drawn using scan-line interrupts.
`SetIText`	DLM	$2015	Fetches a string for an item that contains text and redraws the item.
`SetInGlobals`	TT	$090C	Sets the global parameters for the input device.
`SetMasterSCB`	QD	$1604	Sets the master SCB to a specified value.
`SetMaxGrow`	WM	$430E	Sets the maximum values to which a window's content region can grow.
`SetMenuBar`	MUM	$390F	Sets the current menu bar.
`SetMenuFlag`	MUM	$1F0F	Sets the menu to a specified state.
`SetMenuID`	MUM	$370F	Specifies a new menu number.

Call	Tool	Call Number	Function
SetMenuTitle	MUM	$210F	Specifies the title for a menu.
SetMItem	MUM	$240F	Specifies the name for a menu item.
SetMItemBlink	MUM	$280F	Determines how many times all menu items should blink when selected.
SetMItemFlag	MUM	$260F	Controls the style of an item's highlighting and underlining.
SetMItemID	MUM	$380F	Specifies the ID number of a menu item.
SetMItemMark	MUM	$330F	Sets a specified character to display or not display to the left of a menu item.
SetMItemName	MUM	$3A0F	Specifies the name for a menu item.
SetMItemStyle	MUM	$350F	Sets the text style for a specified menu item.
SetMouse	MTS	$1903	Sets the mode value for the mouse.
SetMTitleStart	MUM	$190F	Sets the starting point for the leftmost menu on the menu bar.
SetMTitleWidth	MUM	$1D0f	Sets the width of a title.
SetOrigin	QD	$2304	Sets the upper left corner of the PortRect to a given point.
SetOriginMask	WM	$340E	Specifies the mask used to put the horizontal origin on a grid.
SetOutGlobals	TT	$0A0C	Sets the global parameters for the error output device.
SetOutputDevice	TT	$100C	Sets the output device to a specified type and location.
SetPage	WM	$470E	Sets the number of pixels that define a "page" for scrolling.
SetPenMask	QD	$3204	Sets the pen mask to the specified mask.
SetPenMode	QD	$2E04	Sets the current pen mode to the specified pen mode.
SetPenPat	QD	$3004	Sets the current pen pattern to the specified pen pattern.
SetPenSize	QD	$2C04	Sets the current pen size to the specified pen size.
SetPenState	QD	$2A04	Sets the pen state in the GrafPort to the specified values.
SetPicSave	QD	$3E04	Sets the PicSave field to a specified value.
SetPolySave	QD	$4204	Sets the PolySave field to a specified value.

Call	Tool	Call Number	Function
SetPort	QD	$1B04	Makes the specified port the current port.
SetPortLoc	QD	$1D04	Sets the current port's map information structure.
SetPortRect	QD	$1F04	Sets the current port's rectangle to the specified rectangle.
SetPortSize	QD	$2104	Changes the size of the current GrafPort's PortRect.
SetPt	QD	$8204	Sets a point to specified horizontal and vertical values.
SetPurge	MM	$2402	Sets the purge level of a block specified by a handle.
SetPurgeAll	MM	$2502	Sets the purge level of all blocks for a specified user ID.
SetPurgeStat	FM	$0F1B	Makes a specified font in memory unpurgeable or purgeable.
SetRandSeed	QD	$8704	Sets the seed value for the random number generator.
SetRect	QD	$4A04	Sets a specified rectangle to specified values.
SetRectRgn	QD	$6B04	Sets a region to a specified rectangle.
SetRgnSave	QD	$4004	Sets the RgnSave field to a specified value.
SetSCB	QD	$1204	Sets the scan-line control byte (SCB) to a specified value.
SetScrapPath	SK	$1116	Sets the clipboard file pointer to the specified value.
SetScroll	WM	$450E	Sets the number of pixels that will be scrolled by scroll bar arrows.
SetSolidBackPat	QD	$3804	Sets the background pattern to a solid pattern using a certain color.
SetSolidPenPat	QD	$3704	Sets the pen pattern to a solid pattern using the specified color.
SetSoundMIRQV	ST	$1208	Sets the entry point into the sound interrupt handler.
SetSoundVolume	ST	$0D08	Changes the DOC registers' volume setting, or the system volume.
SetSpaceExtra	QD	$9E04	Sets the spExtra field in the GrafPort to the specified value.
SetStdProcs	QD	$8D04	Sets up a record of pointers for customizing QuickDraw II operations.

Call	Tool	Call Number	Function
SetSwitch	EM	$1306	Informs the Event Manager of a pending switch event.
SetSysBar	MUM	$120F	Sets a new system bar.
SetSysField	QD	$4804	Sets the SysField in the GrafPort to a specified value.
SetSysFont	QD	$B204	Sets a specified font as the system font.
SetSysWindow	WM	$4B0E	Marks a specified window as a system window.
SetTextFace	QD	$9A04	Sets the text face to the specified value.
SetTextMode	QD	$9C04	Sets the text mode to the specified value.
SetTextSize	QD	$D204	Call is not implemented at the time of this writing.
SetTSPtr	TL	$0A01	Call used to modify tool sets by installing patches.
SetUserField	QD	$4604	Sets the UserField in the GrafPort to a specified value.
SetUserSoundIRQV	ST	$1308	Sets the entry point for an application-defined interrupt handler.
SetVector	MTS	$1003	Sets the vector address for the specified vector reference number.
SetVisHandle	QD	$C804	Sets the VisRgn handle field in the GrafPort to a specified value.
SetVisRgn	QD	$B404	Copies a specified region into the VisRgn.
SetWAP	TL	$0D01	Sets the pointer to the work area for a specified tool set.
SetWFrame	WM	$2D0E	Sets the bit vector that describes a specified window's frame type.
SetWindowIcons	WM	$4E0E	Sets the icon font for the Window Manager.
SetWRefCon	WM	$280E	Sets a window-record value reserved for an application's use.
SetWTitle	WM	$0D0E	Updates the title of a specified window.
SetZoomRect	WM	$380E	Sets the rectangle used to calculate a window's zoomed size.
SFAllCaps	SF	$0D17	Allows an application to display file names in all uppercase.

Call	Tool	Call Number	Function
SFBootInit	SF	$0117	Initializes the Standard File Operations Tool Set at boot time.
SFGetFile	SF	$0917	Displays the Standard File Operations Tool Set's standard dialog; returns data about selected files.
SFPGetFile	SF	$0B17	Displays a custom dialog and returns information on selected files.
SFPPutFile	SF	$0A17	Displays a custom dialog and returns data on files to be saved.
SFPutFile	SF	$0A17	Displays a dialog and returns data about the file to be saved.
SFReset	SF	$0517	Resets the Standard File Operations Tool Set.
SFShutdown	SF	$0317	Shuts down the Standard File Operations Tool Set.
SFStartUp	SF	$0217	Starts up the Standard File Operations Tool Set.
SFStatus	SF	$0617	Tells if the Standard File Operations Tool Set is active.
SFVersion	SF	$0417	Returns the version number of the Standard File Operations Tool Set.
ShowControl	CM	$0F10	Makes a specified control visible.
ShowCursor	QD	$9104	Shows the cursor incrementing its level to 0, if necessary.
ShowDItem	DLM	$2315	Makes visible a specified item from a specified dialog.
ShowHide	WM	$230E	Shows or hides a window, depending upon a specified parameter.
ShowPen	QD	$2804	Increments the pen level.
ShowWindow	WM	$130E	Makes a specified window visible and draws it if it was invisible.
SizeWindow	WM	$1C0E	Sizes a window's port rectangle to a specified width and height.
SolidPattern	QD	$3904	Sets a specified pattern to a solid pattern using a specified color.
SortList	LM	$0A1C	Alphabetizes a list by rearranging the array of member records.
SoundBootInit	ST	$0108	Called by the Tool Locator at initialization.

Call	Tool	Call Number	Function
SoundReset	ST	$0508	Stops all generators that may be generating sound.
SoundShutDown	ST	$0308	Shuts down the Sound Manager.
SoundStartUp	ST	$0208	Initializes a work area to be used by the sound tools.
SoundToolStatus	ST	$0608	Returns status indicating whether the Sound Tool Set is active.
SoundVersion	ST	$0408	Returns the version of the Sound Tool Set.
SRQPoll	ADB	$1409	Adds a device to the SRQ list if the device exists.
SRQRemove	ADB	$1509	Removes a device from the SRQ list.
StartDrawing	WM	$4D0E	Makes a specified window the current port and sets its origin.
StartInfoDrawing	WM	$500E	Used for drawing outside a window's information bar procedure.
StatusID	MTS	$2203	Inquires whether or not a specified user ID is active.
StatusTextDev	TT	$170C	Executes a status call to a specified text device.
StillDown	EM	$0E06	Tests if the specified mouse button is still down.
StopAlert	DLM	$1815	Performs the same functions as the Alert routine.
StringBounds	QD	$AD04	Sets a specified rectangle to be the bounds of a specified string.
StringWidth	QD	$A904	Returns the width in pixels of a specified string.
SubPt	QD	$8104	Subtracts one point from another; leaves result in destination point.
SynchADBReceive	ADB	$0E09	Receive data from an ADB device.
SysBeep	MTS	$2C03	Calls the Apple II monitor entry point BEEL1.
SysFailMgr	MTS	$1503	Displays a system failure message and ends a program.
SystemClick	DM	$1705	Called when application detects a mouse down in a system window.
SystemEdit	DM	$1805	Passes standard menu edits to system windows.

Call	Tool	Call Number	Function
`SystemEvent`	DM	$1A05	Entry point for the Event Manager into the Desk Manager.
`SystemTask`	DM	$1905	Called periodically to support desk accessory actions.
`TaskMaster`	WM	$1D0E	Calls `GetNextEvent` and then handles certain other events itself.
`TestControl`	CM	$1410	Tests which part of a control contains a specified point.
`TextBootInit`	TT	$010C	Called at boot time; sets up certain default device parameters.
`TextBounds`	QD	$AF04	Sets a specified rectangle to be the bounds of the specified text.
`TextReadBlock`	TT	$230C	Reads a block of input characters and writes it to a buffer.
`TextReset`	TT	$050C	Resets device parameters to the defaults.
`TextShutDown`	TT	$030C	A standard call that is unnecessary and performs no function.
`TextStartUp`	TT	$020C	A startup call that is unnecessary and performs no function.
`TextStatus`	TT	$060C	Returns $FFF, indicating the Text Tool Set is active.
`TextVersion`	TT	$040C	Returns the version of the Text Tool Set.
`TextWidth`	QD	$AB04	Returns the width of the specified text.
`TextWriteBlock`	TT	$1E0C	Writes a block of text to the output text device.
`TickCount`	EM	$1006	Returns the number of ticks since the system last started up.
`TLBootInit`	TL	$0101	Initializes the Tool Locator and all other ROM-based tool sets.
`TLMountVolume`	TL	$1101	Displays a simulated dialog asking the user to mount a volume.
`TLReset`	TL	$0501	Initializes the Tool Locator and other ROM-based tool sets.
`TLShutDown`	TL	$0301	Shuts down the Tool Locator when an application shuts down.
`TLStartUp`	TL	$0201	Starts up the Tool Locator when an application starts up.
`TLStatus`	TL	$0601	Returns true, indicating the Tool Locator is active.

Call	Tool	Call Number	Function
TLTextMount-Volume	TL	$1201	Displays a 40-column text window asking the user to mount a volume.
TLVersion	TL	$0401	Returns the version of the Tool Locator.
TotalMem	MM	$1D02	Returns the size of all memory, including the main 256K.
TrackControl	CM	$1510	Follows mouse movements until the mouse button is released.
TrackGoAway	WM	$180E	Removes a window from the screen when the go-away box is clicked.
TrackZoom	WM	$260E	Zooms a window when the mouse is clicked in the zoom box.
UDivide	IM	$0B0D	Divides two 16-bit inputs, producing a quotient and a remainder.
UnionRect	QD	$4E04	Places the union of two rectangles in a third rectangle.
UnionRgn	QD	$7204	Places the union of two regions in a third region.
UnLoadOneTool	TL	$1001	Unloads a specified tool from memory.
UnLoadScrap	SK	$0916	Writes the desk scrap to the scrap file, and releases its memory.
UnPackBytes	MTS	$2703	Unpacks data from the packed format used by PackBytes.
UpdateDialog	DLM	$2515	Redraws the part of a dialog that is in an update region.
ValidRect	WM	$3C0E	Removes a given rectangle from the current window's update region.
ValidRgn	WM	$3D0E	Removes a specified region from the current window's update region.
WaitMouseUp	EM	$0F06	Tests if the mouse button is still down.
WindBootInit	WM	$010E	Initializes the Window Manager at boot time.
WindDragRect	WM	$530E	Pulls around an outline of a rectangle, following mouse movements.
WindNewRes	WM	$250E	Called after the screen resolution is changed.
WindReset	WM	$050E	Resets the Window Manager.

Call	Tool	Call Number	Function
WindShutDown	WM	$030E	Shuts down the Window Manager.
WindStartUp	WM	$020E	Initializes the Window Manager.
WindStatus	WM	$060E	Returns whether or not the Window Manager is active.
WindVersion	WM	$040E	Returns the version number of the Window Manager.
Write Next	ST		Writes 1 byte of data to the next DOC register or RAM address.
Write RAM	ST		Writes a 1-byte value to any specified Ensoniq RAM location.
Write Register	ST		Writes a 1-byte parameter to any register in the DOC chip.
WriteBParam	MTS	$0B03	Writes data to a specified parameter in battery RAM.
WriteBRam	MTS	$0903	Writes 252 bytes, plus 4 checksum bytes, to the battery RAM.
WriteChar	TT	$180C	Writes a character to the output text device.
WriteCString	TT	$200C	Writes a C-style string to the output text device.
WriteLine	TT	$1A0C	Writes a string, plus a carriage return, to the output text device.
WriteRamBlkock	ST	$09080	Writes a specified number of bytes from system RAM into DOC RAM.
WriteString	TT	$1C0C	Writes a string to the output text device.
WriteTimeHex	MTS	$0E03	Sets the current time using hexadecimal format.
X2Fix	IM	$200B	Converts extended to fixed.
X2Frac	IM	$210B	Converts extended to fraction.
XorRgn	QD	$7404	Extend-ORs two regions and places the result in a third region.
ZeroScrap	SK	$0B16	Clears the contents of the scrap.
ZoomWindow	WM	$270E	Zooms a window to its maximum size when the zoom box is clicked.

Bibliography

The Apple IIe User's Guide.
New York: Macmillan, 1983.

Apple Human Interface Guidelines.
Menlo Park, CA: Addison-Wesley, 1987.

Apple Numerics Manual.
Menlo Park, CA: Addison-Wesley, 1987.

Apple IIGS Firmware Reference.
Menlo Park, CA: Addison-Wesley, 1987.

Apple IIGS Hardware Reference.
Menlo Park, CA: Addison-Wesley, 1987.

Apple IIGS ProDOS 16 Reference.
Menlo Park, CA: Addison-Wesley, 1987.

Apple IIGS Programmer's Workshop Assembler Reference.
Menlo Park, CA: Addison-Wesley, 1987.

Apple IIGS Programmer's Workshop C Reference.
Menlo Park, CA: Addison-Wesley, 1987.

Apple IIGS Programmer's Workshop Reference.
Menlo Park, CA: Addison-Wesley, 1987.

Apple IIGS Toolbox Reference.
Menlo Park, CA: Addison-Wesley, 1987.

Programmer's Introduction to the Apple IIGS.
Menlo Park, CA: Addison-Wesley, 1986.

Technical Introduction to the Apple IIGS.
Menlo Park, CA: Addison-Wesley, 1986.

Andrews, Mark

Apple Roots: Assembly Language Programming.
Berkeley: Osborne McGraw-Hill, 1986.

Eyes, David

Programming the 65816.
New York: Brady (Prentiss-Hall), 1986.

Findley, Robert

6502 Software Gourmet Guide and Cookbook.
Rochelle Park, NJ: Hayden Book Co., Inc., 1979.

Fischer, Michael

Apple IIGS Technical Reference.
Berkeley: Osborne McGraw-Hill, 1986, 1987.

Goodman, Danny

The Apple IIGS Toolbox Revealed.
New York: Bantam Books, 1986.

Hunter, Bruce H.

Understanding C.
Berkeley: Sybex, 1984.

Kerninghan, Brian W.

The C Programming Language.
Englewood Cliffs, NJ: Prentiss-Hall, 1978.

Leventhal, Lance A.

6502 Assembly Language Programming.
Berkeley: Osborne McGraw-Hill, 1979.

6502 Assembly Language Subroutines.
Berkeley: Osborne McGraw-Hill, 1982.

Maurer, W. Douglas

Apple Assembly Language.
Rockville, MD: Computer Science Press, Inc., 1984.

Mottola, Robert

Assembly Language Programming for the Apple II.
Berkeley: Osborne McGraw-Hill, 1982.

Wagner, Roger

Assembly Lines: The Book.
Santee, CA: Roger Wagner Publishing, Inc., 1984.

Waite, Mitchell

C Primer Plus.
Indianapolis, IN: Howard W. Sams & Co., Inc., 1985.

Zaks, Rodnay

Programming the Apple II in Assembly Language.
Berkeley: Sybex, 1983.

Programming the 6502.
Berkeley: Sybex, 1983.

Index

HOWARD W. SAMS & COMPANY

Programming the Apple IIGS™
in C and Assembly Language

Learning how to program the Apple IIGS is easy with this book by best-selling author Mark Andrews.

The first of its kind to include both assembly language and C, this book enables professional programmers and hobbyists alike to take advantage of the power, speed, graphics, and sound capabilities of the IIGS.

Packed with useful, entertaining type-and-run programs, *Programming the Apple IIGS in C and Assembly Language* equips you with all you need to program the Apple IIGS in C and integrate assembly language to supercharge your programs.

In this plain-English guide, you'll discover

- How to program the Apple IIGS's 65C816 chip in assembly language
- How to use the Apple IIGS Programmer's Workshop program development system
- How to create mouse-driven programs with such eye-catching graphics features as pull-down menus, multiple screen windows, icons, and dialog boxes
- How to write sound tracks for your programs using the IIGS's 15-voice, 32-oscillator sound and music synthesizer

To use this book, you need an Apple IIGS with at least two 3.5-inch disk drives, a monochrome or color monitor, and a memory expansion card with at least 512K of additional RAM.

Mark Andrews, an experienced program designer and technical writer, currently works as an assembly language programmer at Apple Computer, Inc. He has written eight books about computers and computer programming, including *Commodore 128® Assembly Language Programming* and *Commodore 64® /128 Assembly Language Programming* for Howard W. Sams & Company. He has also written hundreds of technically oriented magazine and newspaper articles and has designed and developed many commercial microcomputer programs.

0 81262 22599 8

$18.95/22599

ISBN: 0-672-22599-9

HOWARD W. SAMS & COMPANY

A Division of Macmillan, Inc.

4300 West 62nd Street

Indianapolis, Indiana 46268 USA