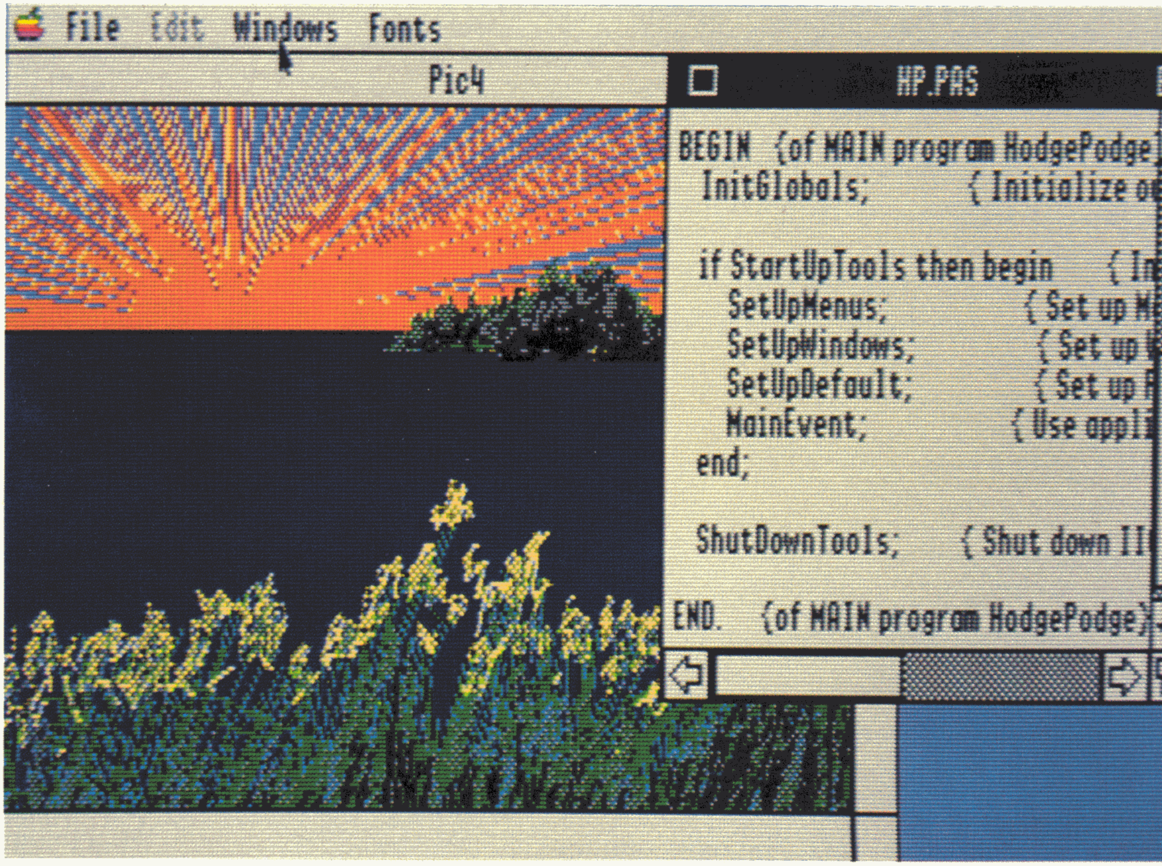
Apple® II

Programmer's Introduction to the Apple IIGS®



Apple® II Programmer's Introduction to the Apple IIGS®

Addison-Wesley Publishing Company, Inc.
Reading, Massachusetts Menlo Park, California New York
Don Mills, Ontario Wokingham, England Amsterdam Bonn
Sydney Singapore Tokyo San Juan

APPLE COMPUTER, INC.

Simultaneously published in the United States and Canada.

ISBN 0-201-17745-5
ABCDEFGHIJ-DO-898
First printing, March 1988

WARRANTY INFORMATION

ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though Apple has reviewed this manual, **APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.**

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

USE OF PARTICULAR LANGUAGE PRODUCTS FOR PURPOSES OF DEMONSTRATION DOES NOT CONSTITUTE AN ENDORSEMENT OF SUCH PRODUCTS BY APPLE COMPUTER, INC.

LICENSING REQUIREMENTS

Apple has a licensing program that allows software developers to incorporate Apple-developed oject code files into their products. A license is required for both in-house and external distribution. Before distributing any products that incorporate Apple software, please contact Software Licensing for licensing information.

Contents

Figures and tables

Chapter 6 Memory, Segments, and Files 185

Chapter 7 Creating a Segmented Application 219

Chapter 8 What Type of Program to Write? 255

Appendix C Files on an Apple IIGS System Disk 298

Appendix D HodgePodge Organization 302

Preface

Welcome to the *Programmer's Introduction*

The Apple IIGS® is a new kind of computer. It offers precise color graphics, sophisticated sound hardware, a large memory capacity, and an extensive toolbox of programming routines—giving you programming resources without precedent among personal computers. The *Programmer's Introduction to the Apple IIGS* gets you started writing programs that take advantage of these unique features.

You needn't be an expert programmer to benefit from this book, but we do assume that you know some fundamentals. Your background will most likely determine your approach.

- □ If you are familiar with programming other Apple® II computers, and wondering how different Apple IIGS programming might be…
- □ If you are familiar with programming the Macintosh® computer, and wondering how similar Apple IIGS programming might be…
- □ If you are familiar with programming other computers, and wondering how rewarding Apple IIGS programming might be…
- □ If you are familiar with *using* the Apple IIGS, and wondering how much fun Apple IIGS *programming* might be…

…this book will help get you started. It can't be a complete programming course, but it does cover the major features that set the Apple IIGS apart and make it an exciting machine to write programs for.

You should be familiar with the Apple IIGS, at least from a user's perspective, before you start this book. In particular, you should understand how to start the system and how to use the keyboard, mouse, and disk drives.

We don't teach you any programming languages here. The books listed in the next section under "Roadmap to the Apple IIGS Technical Manuals" can help you with C and 65816 assembly language. The other Apple IIGS technical manuals cover individual topics in far greater detail than we can here; please consult them as needed.

❖ *Toolbox manual:* It is not possible to write the kind of program described here without the aid of the *Apple IIGS Toolbox Reference.* We give lots of examples and general call descriptions in this book, but you'll need both volumes of the *Toolbox Reference* if you want to write your own applications.

Roadmap to the Apple IIGS technical manuals

The Apple IIGS personal computer has many advanced features, making it more complex than earlier models of the Apple II. To describe it fully, Apple has produced a suite of technical manuals. Depending on the way you intend to use the Apple IIGS, you may need to refer to a select few of the manuals, or you may need to refer to most of them.

The technical manuals are listed in Table P-1. Figure P-1 is a diagram showing the relationships among the different manuals.

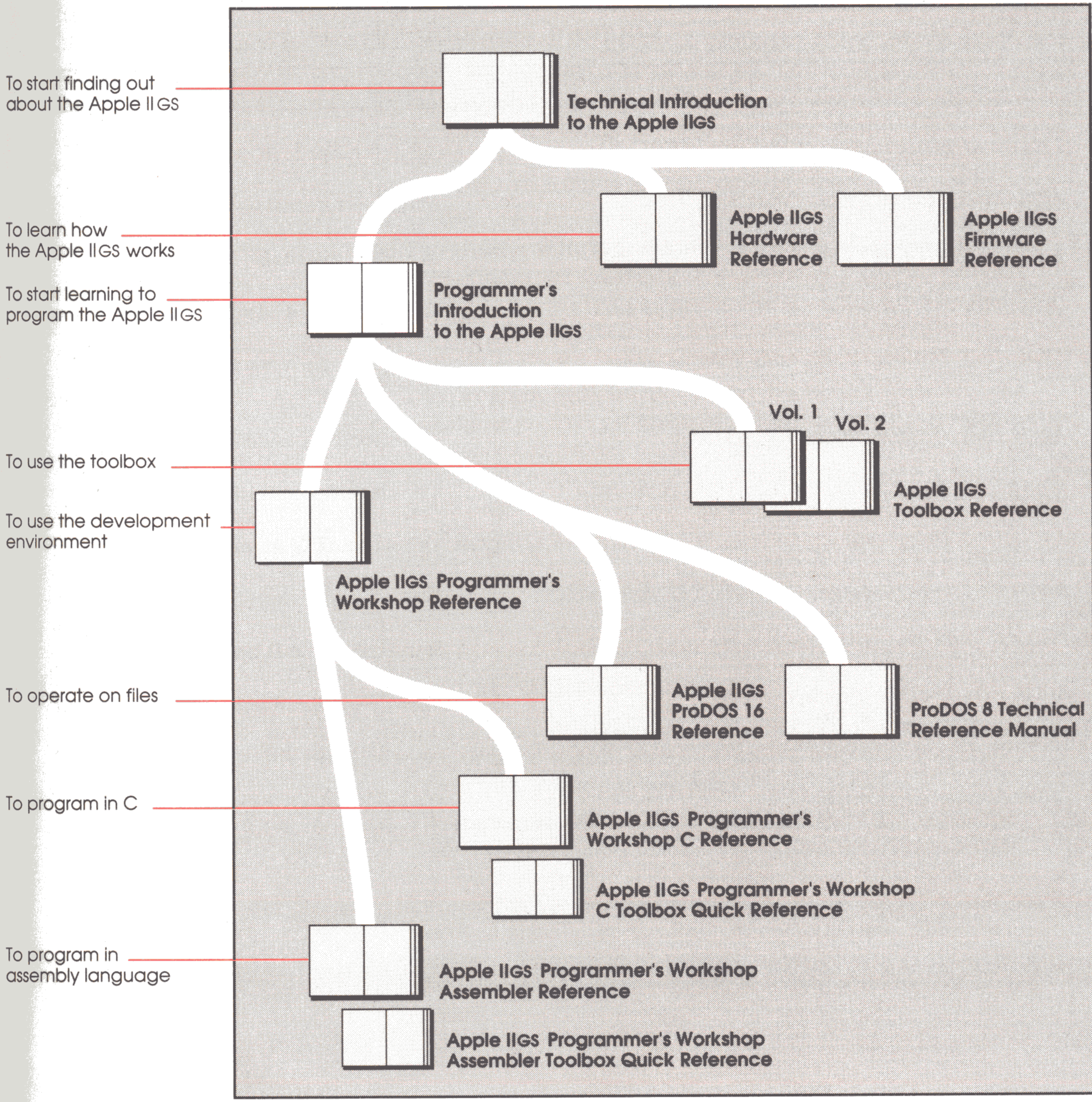


Figure P-1
Roadmap to the Apple IIGS technical manuals

Table P-1
The Apple IIGS technical manuals

Title	Subject
Technical Introduction to the Apple IIGS	What the Apple IIGS is
Apple IIGS Hardware Reference	Machine internals—hardware
Apple IIGS Firmware Reference	Machine internals—firmware
Programmer's Introduction to the Apple IIGS	Concepts and a sample program
Apple IIGS Toolbox Reference, Volume 1	How the tools work and some toolbox specifications
Apple IIGS Toolbox Reference, Volume 2	More toolbox specifications
Apple IIGS Programmer's Workshop Reference	The development environment
Apple IIGS Programmer's Workshop Assembler Reference	Using the APW Assembler
Apple IIGS Programmer's Workshop C Reference	Using C on the Apple IIGS
ProDOS 8 Technical Reference Manual	Standard Apple II operating system
Apple IIGS ProDOS 16 Reference	Apple IIGS operating system and loader
Human Interface Guidelines: The Apple Desktop Interface	Guidelines for the desktop interface
Apple Numerics Manual	Numerics for all Apple computers

Introductory manuals

The introductory manuals are for developers, computer enthusiasts, and other Apple IIGS owners who need technical information. As introductory manuals, their purpose is to help the technical reader understand the features of the Apple IIGS, particularly the features that are different from other Apple computers.

- **The technical introduction:** The *Technical Introduction to the Apple IIGS* is the first book in the suite of technical manuals about the Apple IIGS. It describes all aspects of the Apple IIGS, including its features and general design, the program environments, the toolbox, and the development environment.
- **The programmer's introduction (this book):** When you start writing Apple IIGS programs, the *Programmer's Introduction to the Apple IIGS* provides the concepts and guidelines you need. It is not a complete course in programming, only a starting point for programmers writing applications that use the Apple desktop interface (with windows, menus, and the mouse). It introduces the routines in the Apple IIGS Toolbox and includes a sample event-driven program.

Machine reference manuals

There are two reference manuals for the machine itself. They contain detailed specifications for people who want to know exactly what's inside the machine.

- **The hardware reference manual:** The *Apple IIGS Hardware Reference* is required reading for hardware developers and anyone else who wants to know how the machine works. Information for developers includes the mechanical and electrical specifications of all connectors, both internal and external. Information of general interest includes descriptions of the internal hardware and how it affects the machine's features.

- **The firmware reference manual:** The *Apple IIGS Firmware Reference* describes programs and subroutines stored in the machine's read-only memory (ROM). The *Firmware Reference* includes information about interrupt routines and low-level I/O subroutines for the serial ports, the disk port, and for the Desktop Bus interface, which controls the keyboard and the mouse. The *Firmware Reference* also describes the Monitor program, a low-level programming and debugging aid for assembly-language programs.

The toolbox manuals

Like the Macintosh, the Apple IIGS has a built-in toolbox of software routines. The two volumes of the *Apple IIGS Toolbox Reference* completely describe the calls and data structures for all tool sets, and also tell how to write and install your own tool set.

The desktop interface is described in Chapter 1.

If you are developing an application that uses the *desktop interface,* or if you want to use the Super Hi-Res graphics display, you'll find the toolbox indispensable.

The Programmer's Workshop manual

The Apple IIGS Programmer's Workshop (APW) is the development environment for the Apple IIGS computer—a set of programs that enables developers to create application programs. The *Apple IIGS Programmer's Workshop Reference* describes the APW Shell, Editor, Linker, and utility programs; these are the parts of the workshop that all developers need, regardless of which programming language they use.

The APW reference manual includes a sample program to show how to create an application. It also describes object module format, the file format used by all APW compilers to produce files loadable by the Apple IIGS System Loader.

Programming-language manuals

Apple currently provides a 65C816 assembler and a C compiler. Other compilers can be used with the workshop, provided that they follow the standards defined in the *Apple IIGS Programmer's Workshop Reference.*

There is a separate reference manual for each programming language. Each manual includes the specifications of the language and of the Apple IIGS libraries for the language, and describes how to use the assembler or compiler for that language. The manuals for the languages Apple provides are the *Apple IIGS Programmer's Workshop Assembler Reference* and the *Apple IIGS Programmer's Workshop C Reference.*

❖ *Note:* The *Apple IIGS Programmer's Workshop Reference* and the two programming-language manuals are available through the Apple Programmer's and Developer's Association (APDA).

Operating-system manuals

There are two operating systems that run on the Apple IIGS: ProDOS® 16 and ProDOS 8. Each operating system is described in its own manual: the *Apple IIGS ProDOS 16 Reference* and the *ProDOS 8 Technical Reference Manual.* ProDOS 16 uses the full power of the Apple IIGS. The ProDOS 16 manual describes its features and includes information about the System Loader, which works closely with ProDOS 16 to load program segments into memory.

ProDOS 8, previously called *ProDOS,* is the standard operating system for most Apple II computers with 8-bit CPUs. It also runs on the Apple IIGS, but it cannot access certain advanced Apple IIGS features.

All-Apple manuals

There are two manuals that apply to all Apple computers: *Human Interface Guidelines: The Apple Desktop Interface* and *Apple Numerics Manual.* If you develop programs for any Apple computer, you should know about those manuals.

The Human Interface Guidelines manual describes Apple's standards for the desktop interface of any program that runs on Apple computers. If you are writing a commercial application for the Apple IIGS, you should be familiar with the contents of this manual.

The *Apple Numerics Manual* is the reference for the Standard Apple Numeric Environment (SANE™), a full implementation of the *IEEE Standard for Binary Floating-Point Arithmetic* (IEEE Std 754-1985). If your application requires accurate or robust arithmetic, you'll probably want to use the SANE routines in the Apple IIGS.

How to use this book

The *Programmer's Introduction* is not a tutorial. Rather than ask you to type in line after line of code, we've built the book around a finished example—a sample program named *HodgePodge*. HodgePodge is a fully functioning framework of an application that demonstrates most of the programming concepts we present in this book. More than that, HodgePodge is a rather heterogeneous collection of generally useful Apple IIGS routines—hence its name. You are invited to study, copy, or incorporate any of those routines, wholesale or piecemeal, unchanged or greatly altered, into your own programs.

Start by reading Chapter 1. It introduces the basic concepts of this book—event-driven programming, the desktop user interface, the Apple IIGS Toolbox, and program segmentation.

Then run HodgePodge to see what it does. At that point you should be ready for Chapter 2, an extensively annotated set of source listings of the principal parts of HodgePodge. The listings give you the big picture on how event-driven programs are organized, demonstrate how heavily desktop programming relies on toolbox calls, and function as templates for you to use in your own programming. Complete source listings in Pascal, C, and 65816 assembly language, are in Appendixes E through G.

Chapters 3 through 8 expand further on the concepts of toolbox use, memory management, program segmentation, the development environment, and specialized program requirements. These chapters include sample source listings where appropriate, but they also discuss important Apple IIGS concepts not represented in any of the samples. They are overviews designed to give you ideas to pursue in your own programming when aided by other reference manuals.

Chapter 9 is a brief wrap-up that summarizes general program design ideas and shows where to go for further help.

Appendixes include hints on converting existing Macintosh applications to run on the Apple IIGS, and enhancing existing Apple II applications to take full advantage of the new Apple IIGS features.

❖ *Note:* Please don't feel that you need to read this book in any order. Skipping around among programming examples, explanations, and theory may be the best way to absorb the material presented here. Most important of all, experiment on the Apple IIGS as you go along. Use HodgePodge or write your own examples.

Terms and conventions

This book may define certain terms in a slightly different manner from which you are accustomed. Here are two:

- **Apple II:** A general reference to the Apple II family of computers. It includes the Apple II, Apple II Plus, Apple IIc, Apple IIe, and Apple IIGS.
- **standard Apple II:** Any Apple II computer that is *not* an Apple IIGS. Because previous members of the Apple II family share many characteristics, it is useful to distinguish them as a group from the Apple IIGS. A standard Apple II may also be called an *8-bit Apple II*, because of the 8-bit registers in its 6502 or 65C02 microprocessor.

Typographic conventions

Each new term introduced in this book is printed in **bold** type where it is first defined. That lets you know that the term has not been defined earlier, and also indicates that there is an entry for it in the glossary.

Assembly-language labels, entry points, program and subroutine names, and filenames that appear in text passages are printed in a special typeface (for example, `DoWItem` and `MENU.PAS`). There is one exception: the names of Apple IIGS system software routines, such as toolbox calls and operating system calls (for example, NewModalDialog and QUIT), are printed in normal type.

❖ *Note:* The source-code listings of the program HodgePodge follow a different, special typographic convention. See "Code-Listing Convention" in Chapter 2.

Watch for these

The following words mark special messages to you:

❖ *Note:* Text set off in this manner presents sidelights or interesting points of information.

Important Text set off in this manner—with the word **Important**—presents important information or instructions.

Warning Text set off in this manner—with the word **Warning**—indicates potential serious problems.

Chapter 1

Apple IIGS Concepts

Writing well-designed programs for the Apple IIGS computer is both an adventure and a challenge. It may require some changes in the way you approach programming, some changes that at first seem confusing. But don't worry; there are tools and resources to help you at every step, to make the shift in programming style relatively easy. And fast.

As you start, you'll want to keep several key concepts in mind. This chapter introduces those basic ideas. We'll be building on them throughout the book, and showing examples of them in action, in the sample program *HodgePodge*. They include

- *desktop applications*—programs with a user inteface based on Apple's Human Interface Guidelines
- *event-driven programming*—creating the fundamental internal structure of desktop applications
- *the Apple IIGS Toolbox*—the extensive set of programming routines that make desktop, event-driven programming practical
- *segmentation*—techniques that allow your programs to use memory more efficiently
- *development*—steps to follow in creating a running program

A more powerful Apple II

The Apple IIGS personal computer is a new Apple II with many high-performance features. Some of its highlights are

- a more powerful microprocessor with faster operation than processors used in standard Apple II computers, and with a 24-bit address bus
- 256K memory, expandable to 8 megabytes
- high-resolution RGB video display for Super Hi-Res color graphics and text
- multi-voice digital sound synthesizer
- detached keyboard with Apple Desktop Bus™ connector
- built-in I/O: clock, disk port, and serial ports with AppleTalk® interface
- slots and game I/O connectors compatible with standard Apple II computers

❖ *Note:* If you are not familiar with the Apple II family of computers, you may want to refer to the *Technical Introduction to the Apple IIGS* or your *Apple IIGS Owner's Guide* for explanations of some of the terms in this section.

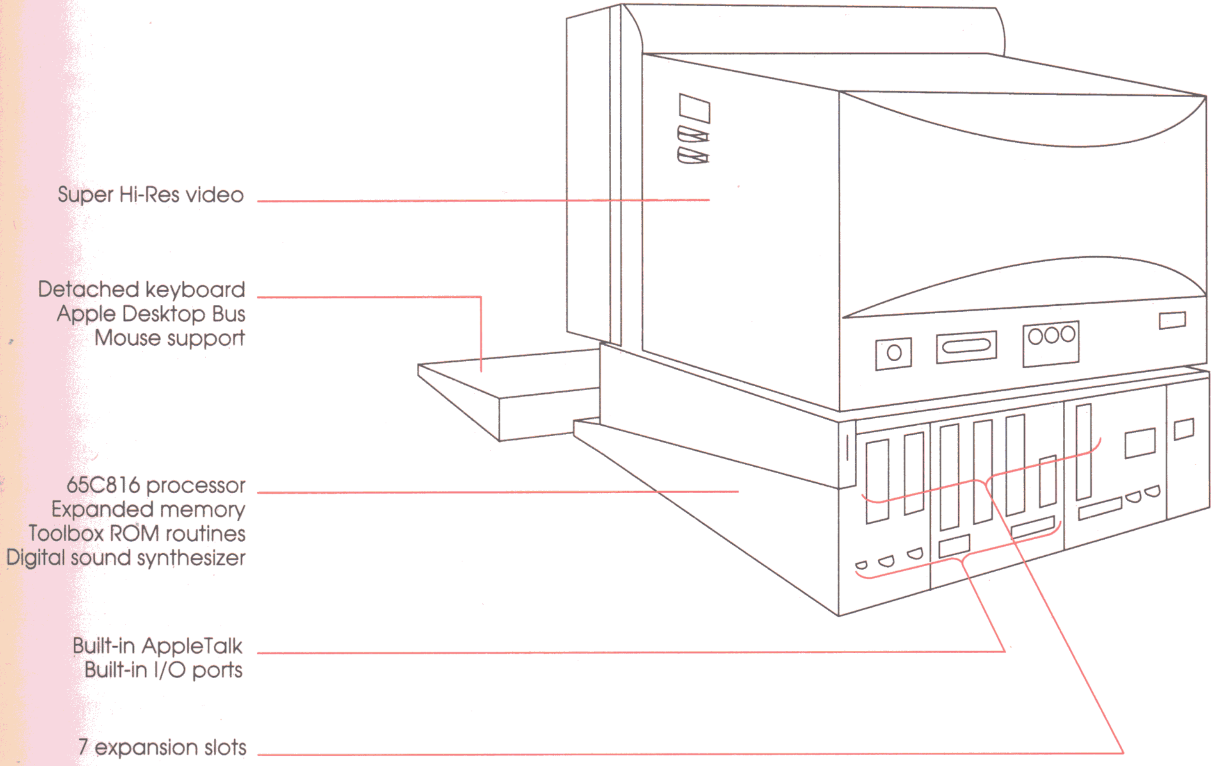


Figure 1-1
Apple IIGS features

The 65816 microprocessor

A *16-bit* processor handles data in chunks of 16 bits at a time, twice as much data per cycle as an *8-bit* processor.

The microprocessor in the Apple IIGS is a 65SC816, a *16-bit* **CMOS** design based on the 6502 processor used in previous Apple II computers. Among the features of the 65816 are

- ability to emulate 6502 and 65C02 8-bit microprocessors
- 16-bit accumulator and index registers

Stack and direct-page concepts are discussed further in Chapter 6.

- relocatable stack and direct page (zero page)
- 24-bit internal address bus, giving a 16-megabyte memory space

Two execution modes

The 65816 microprocessor can operate in two different modes: **native mode,** with all of its new features, and 6502 **emulation mode,** for running programs written for standard (8-bit) Apple II computers.

Applications written for the Apple IIGS use native mode with the accumulator and index registers 16 bits long. Also, the size of the stack and the locations of the stack and direct page within bank $00 are at the discretion of the application.

Two clock speeds

The microprocessor in the Apple IIGS can operate at either of two clock speeds: the standard Apple II speed of 1 MHz, or the faster speed of 2.8 MHz. When running programs in RAM the Apple IIGS uses a few clock cycles for refreshing memory, making the effective processing speed about 2.5 MHz. System firmware, running in ROM, runs at the full 2.8 MHz.

Transformable registers

The 65816 registers are discussed in more detail in the *Apple IIGS Hardware Reference*.

If you are an assembly-language programmer, note from Figure 1-2 how the processor's registers change size to accommodate mode changes. The accumulator and X- and Y-index registers change from 8 bits to 16 bits in going from emulation to native mode. The stack pointer also becomes 16 bits long, meaning that in native mode the stack can be anywhere in bank $00; in emulation mode it is confined to page 1 of bank $00. The direct register is not used in emulation mode; in native mode it is the base address for all *zero-page* addressing modes, meaning that in native mode the Apple IIGS can have several zero pages (called *direct pages*), located anywhere in bank $00.

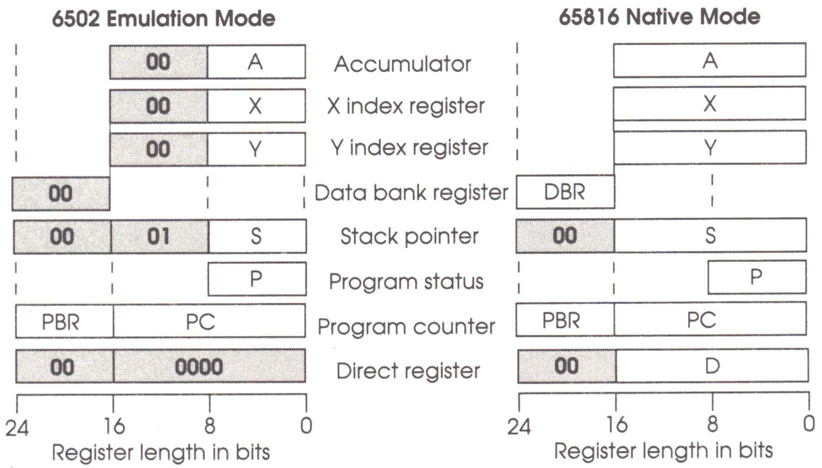


Figure 1-2
Program registers in the 65816 microprocessor

Expanded memory

Thanks to the 24-bit addresses of the 65816, the Apple IIGS can access a total memory space of 16 megabytes. Of this total, up to 8 megabytes of memory is available for RAM expansion, and one megabyte is available for ROM expansion. The rest is not used.

For additional memory concepts, see Chapter 6 of this book. For more complete information, read the *Technical Introduction to the Apple IIGS*.

The minimum memory configuration for the Apple IIGS is 256K of RAM. Programs written for the Apple IIGS—that is, programs that run the 65816 microprocessor in native mode (thereby gaining the ability to address more than 64K of memory)—can use up to about 176K of the 256K. The rest is reserved for displays and for use by the system firmware.

❖ *Note:* If your application uses the Apple IIGS Toolbox—as this book strongly recommends—your application will have less than 176K of available space on a 256K machine. So if you are writing anything other than a very small program, the program will probably require an Apple IIGS with a minimum of 512K of RAM.

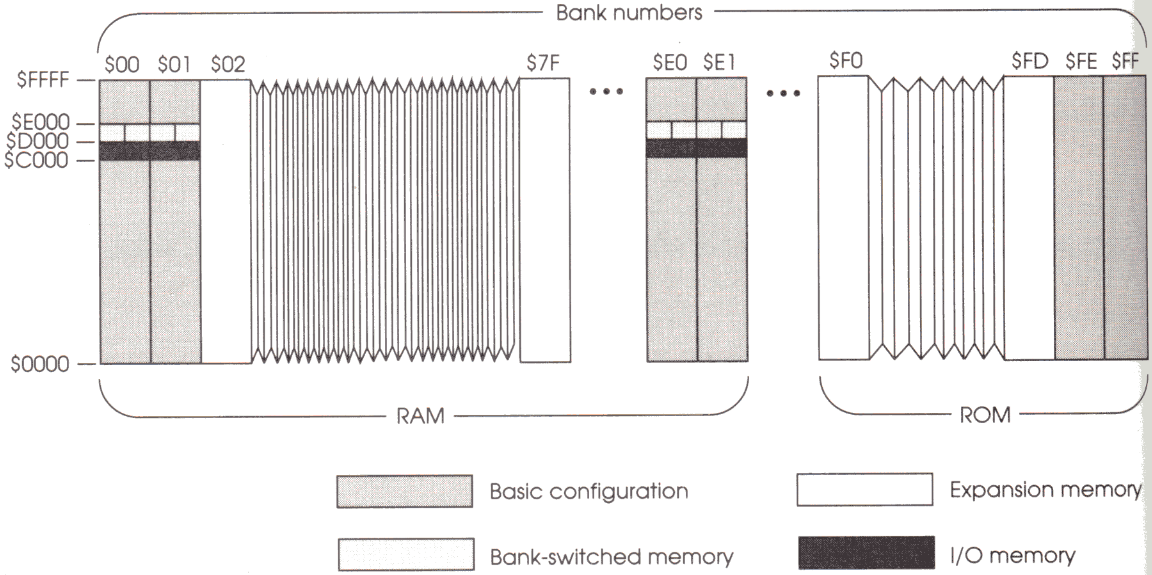


Figure 1-3
Apple IIGS memory map

The basic 256K of RAM memory is mapped as four **banks** ($00, $01, $E0, and $E1) of 64K each. As Figure 1-3 shows, portions of those banks are reserved for system use or I/O addresses, just as in other Apple II computers.

For more information on the memory configuration of standard Apple II computers, see the *Apple IIe Technical Reference Manual* or *Apple IIc Technical Reference Manual*.

The Apple IIGS has a special card slot dedicated to memory expansion. All the RAM on an Apple IIGS memory expansion card is available for Apple IIGS application programs. Expansion memory is contiguous: its address space extends without a break through all the RAM on the card. Expansion RAM on the Apple IIGS is not limited to use as data storage; program code can run in any part of RAM.

Super Hi-Res video display

The Apple IIGS gives you the most sophisticated high-resolution color display of any member of the Apple II family. Now your applications can mix dazzling color and sharp, 80-column text or precise line drawings on the same screen. And do it easily, with the help of built-in toolbox routines.

Hi-Res, Double Hi-Res, and other standard-Apple II video display modes are described in the *Apple IIe Technical Reference Manual* and *Apple IIc Technical Reference Manual.*

In addition to all the video display modes of the Apple IIc and Apple IIe, the Apple IIGS has two new Super Hi-Res display modes that look much clearer than standard Hi-Res and Double Hi-Res. Super Hi-Res is also easier to program than Hi-Res or Double Hi-Res, because it maps entire bytes onto the screen, instead of seven bits, and because its memory map is linear.

Used with an analog RGB video monitor, the Apple IIGS displays high-quality, high-resolution color graphics. Table 1-1 lists the specifications of the two new graphics modes.

Table 1-1
Super Hi-Res graphics modes

Mode	Horizontal resolution	Vertical resolution	Bits per pixel	Colors per line	Colors on screen	Colors possible
320	320	200	4 bits	16	256	4096
640	640	200	2 bits	16*	256*	4096

*Different pixels in 640 mode use different subsets of the available colors.

Pixel is short for *picture element.* A pixel corresponds to the smallest dot you can draw on the screen.

Each dot on the Super Hi-Res screen is a **pixel.** The screen image is either 320 pixels or 640 pixels across, by 200 pixels down. In memory, each pixel has either a 2-bit (640 mode) or a 4-bit (320 mode) value associated with it. The pixel values select colors from programmable *color tables.* A color table consists of 16 entries, and each entry is a 12-bit value specifying one of 4096 possible colors.

In 320 mode, each pixel consists of four bits, so it can select any one of the 16 colors in a color table. Its *palette* is all 16 colors in the color table. In 640 mode, each pixel is only two bits, so it can select from four colors only. However, the 640-mode color table is divided into four *mini-palettes* of four colors each, and successive pixels select from successive groups of four colors. Thus, even though a given pixel in 640 mode can be one of only four colors, different pixels in a line can take on any of the colors in a color table.

For more information on using color, see "Drawing to the Screen" in Chapter 3.

To further increase the number of colors available on the display, there can be up to 16 different color tables in use at the same time, giving as many as 256 different colors on the screen.

Digital sound synthesizer

Like other computers in the Apple II family, the Apple IIGS can produce simple, single-bit sounds such as clicks, beeps, and tones.

Figure 5-9 shows the major components of the sound hardware.

But it can also do a whole lot more. The Apple IIGS has a new digital sampling sound system built around a special-purpose synthesizer IC called the Digital Oscillator Chip, or DOC for short. Using the DOC, the Apple IIGS can produce 15-voice music and other complex sounds without tying up its main processor.

For more information on using sound, see the section "Making Sounds" in Chapter 5. See also the *Apple IIGS Hardware Reference* for details about the sound system and the DOC.

The sound system consists of the DOC, an audio amplifier and internal speaker, a connector for an external amplifier and speaker, 64K of independent RAM for storage of sound samples, and a custom IC called the Sound GLU (general logic unit). The Sound GLU is the system interface to the DOC, and also controls the volume of the old-style single-bit output.

Detached keyboard with Apple Desktop Bus

For more information on using the Apple Desktop Bus, see "Communicating with Files and Devices" in Chapter 5. See also the *Apple IIGS Toolbox Reference* and *Apple IIGS Hardware Reference*.

The new detached keyboard includes cursor keys and a numeric keypad. The Apple Desktop Bus, which supports the keyboard and the Apple Mouse, can also handle other input devices such as joysticks and graphics tablets.

Expansion slots and built-in I/O

In addition to the memory expansion slot mentioned earlier, the Apple IIGS has seven I/O expansion slots like those on an Apple IIe. Most peripheral cards designed for the Apple II Plus and the Apple IIe work in the Apple IIGS slots. The Apple IIGS also has game I/O connectors for joysticks and other game hardware.

Like the Apple IIc, the Apple IIGS has one built-in disk port and two serial I/O ports. The built-in AppleTalk interface uses one of the serial ports. Programs can use either the built-in ports or peripheral cards in slots to perform input/output functions.

Built-in I/O features are accessed as though they were peripheral cards in slots. For most of the expansion slots, the user can choose (on the Control Panel) between using a peripheral card or using the built-in feature associated with the slot. Table 1-2 shows the slot-equivalents for the built-in features.

Table 1-2
Apple IIGS expansion slots and internal-port equivalents

Slot	Available internal feature
1	serial port (printer)
2	serial port (communications)
3	80-column display firmware
4	mouse support
5	SmartPort (disk support)
6	Disk II® support
7	AppleTalk support

Clock-calendar and Control Panel

The Apple IIGS has a built-in real-time clock. The user sets the time and date with the Control Panel, a ROM-based program that also configures expansion slots, serial ports, display colors, sound volume and pitch, and other options.

All Control Panel settings, including the clock-calendar values, are maintained in a special battery-powered RAM that is maintained even during power interruptions.

Compatibility with standard Apple II computers

Although the Apple IIGS is more powerful than previous Apple II computers, it is still a member of the family. With the microprocessor in 6502 emulation mode, and with the ProDOS 8 operating system active, nearly any ProDOS-8-based Apple II application runs just fine on the Apple IIGS. The only noticeable difference is a 2.5-times increase in execution speed—and even that difference can be eliminated if your software must run at the 6502 clock speed. Furthermore, as just noted, most peripheral cards designed for the Apple II Plus or Apple IIe will function identically in the Apple IIGS.

ProDOS 16 is the Apple IIGS disk operating sytem. It is documented in the *Apple IIGS ProDOS 16 Reference*. See also Chapter 6 of this book.

Getting the most out of the Apple IIGS, however, requires execution in 65816 native mode under the more advanced **ProDOS 16** operating system. That's what this book is about: writing programs that take full advantage of the computer. Under those conditions, existing standard-Apple II applications cannot run without at least some modification. If you have written a standard-Apple II application, see Appendix B for suggestions on how to modify it for native-mode operation under ProDOS 16.

The Apple desktop interface

Desktop applications are programs of a particular style—a style that presents an accessible, nonthreatening, and predictably consistent interface to the user. If your programs show these qualities, they will be easier to learn and more satisfying to use.

The concepts behind this style of program constitute the Apple *Human Interface Guidelines*. This section will help you see what's involved in writing an application that follows the guidelines.

Figure 1-4 shows some of the common visual features of a desktop application. The interface is *graphics-based* rather than text-based. The screen itself represents the *desktop*, upon which documents appear in movable, scrollable, overlapping *windows*. Pull-down *menus* appear across the top of the desktop. *Icons* instead of text may represent certain concepts or objects. The user can manipulate the menus, icons, windows, and window contents with a *mouse* or other pointing device as well as with the keyboard.

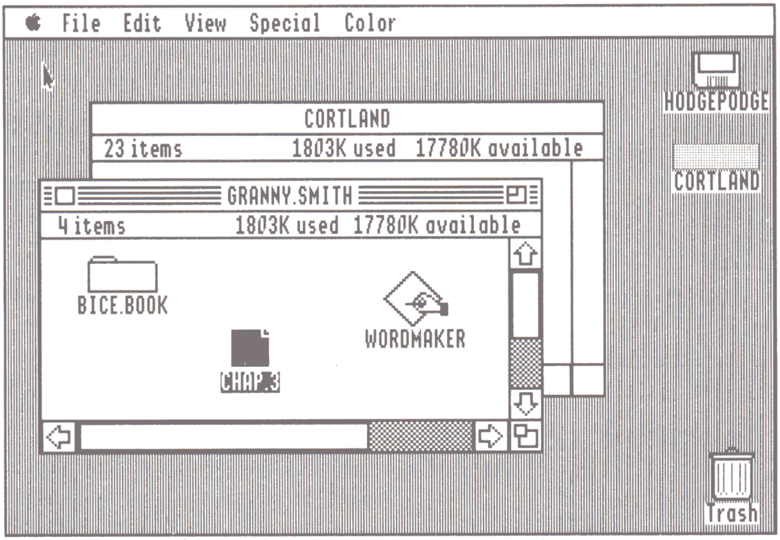


Figure 1-4
The Apple IIGS desktop

These visual features are not the real essence of a desktop application, however. The true importance of desktop applications lies in their relationship with the user, as explained next.

Human Interface Guidelines

If you are developing application programs for the Apple IIGS computer, you are strongly encouraged to follow the principles presented in *Human Interface Guidelines: The Apple Desktop Interface*. That manual describes the desktop interface through which the computer user communicates with the computer and the applications running on it. This section briefly outlines a few of the human interface concepts. The Apple Desktop Interface, first introduced with the Macintosh computer, is designed to appeal to a nontechnical audience. Whatever the purpose or structure of your application, it will communicate with the user in a consistent, standard, and nonthreatening manner if it adheres to the desktop interface standards. These are some of the basic principles:

- **Human control:** Users should feel that they are controlling the program, rather than the reverse. Give them clear alternatives to select from, and act on their selections consistently.

- **Dialog:** There should be a clear and friendly dialog between human and computer. Make messages and requests to the user in plain English.
- **Direct manipulation and feedback:** The user's physical actions should produce physical results. When a key is pressed, place the corresponding letter on the screen. Use highlighting, animation, and dialog boxes to show users the possible actions and their consequences.
- **See-and-point (instead of remember-and-type):** The user selects actions from alternatives presented on the screen. In general, the process is in the order *object* followed by *verb*—that is, one selects first the object to be acted upon, and then the action to be performed.
- **Exploration:** Give the user permission to test out the possibilities of the program without worrying about negative consequences. Keep error messages infrequent. Warn the user when risky situations are approached, but don't erect unnecessary barriers.
- **Graphic design:** Good graphic design is a key feature of the guidelines. Objects on the screen should be simple and clear, and they should have visual fidelity (that is, they should look like what they represent). Use familiar, concrete metaphors to represent aspects of the computer and program. The *desktop* is the primary metaphor in the Apple Desktop Interface.
- **Avoiding modes:** A mode is a portion of an application that the user must explicitly enter and leave, and that restricts the operations that can be performed while the mode is in effect. By restricting the user's options, modes reinforce the idea that computers are unnatural and unfriendly. Use modes sparingly.
- **Device-independence:** Make your program as hardware-independent as possible. Don't bypass the system-provided software tool sets and interfaces—your program may become incompatible with future products and features.
- **Consistency:** As much as possible, all your applications should use the same interface. Don't confuse the user with a different look for each program.
- **Evolution:** Consistency does not mean that you are restricted to using existing desktop features. New ideas are essential for the evolution of the Human Interface concept. If your application has a feature that is not described in *Human Interface Guidelines,* make sure it cannot be confused with an existing feature. It is better to do something completely different than to half-agree with the guidelines.

Why write desktop applications?

The biggest reason for programming desktop applications on the Apple IIGS is the consistent interface they present. Users spend less time learning and more time using an application if they already know their way around.

Appendix B discusses how writing Apple IIGS desktop applications differs from programming for standard Apple II computers.

There are some disadvantages to desktop applications. Apple IIGS desktop programs will not run on the Apple IIe and IIc. Because desktop applications require the use of graphics to support windows and multiple fonts, the interface can be slower than a simpler text-based command-line or menu interface. Also it takes time to learn the techniques of writing desktop applications.

On the other hand, experience with the Apple Macintosh computer has shown that an interface that is consistent from one application to another is extremely attractive to users, because it dramatically cuts down the learning time for each new application. The Apple desktop and the *Human Interface Guidelines* have been refined over several years of studies and first-hand experience by Apple and independent developers.

Appendix A discusses how writing Apple IIGS desktop applications differs from Macintosh programming.

The cost to you in development time is minor when you consider the increase in your product's appeal due to ease of use and compatibility with the Macintosh interface. In addition, if you are an Apple II developer new to the Apple desktop, the techniques you learn (although not the actual code, in most instances) are directly applicable to the Macintosh.

Event-driven programming

In the old days of programming, all programs were executed in batch mode: the entire program was put on computer cards (or worse, punched paper tape) and fed into the computer all at once. The computer executed the instructions in the same sequence every time the program was run (any conditional branching was controlled by data read in with the program), reading data and writing out results at specified points in the program.

Batch mode was fine for "crunching data", but it wasn't very useful for applications (such as word processing or drawing) that require the user to make decisions while the program is running. When computer terminals were invented, programmers began writing programs that allowed users to send commands to the computer and wait for responses—interactive programs were born.

Any interactive program is in some sense *event-driven.* That is, the computer spends much of its time waiting for some user input to occur, usually a key press. Traditional interactive programs, however, still largely control the choices and the sequence in which operations are performed. The user, who follows rather than leads, still feels that the program is in control.

With the introduction of the Apple Macintosh and Lisa® computers, Apple's Human Interface Guidelines and event-driven programming came into prominence. The basic principle of event-driven programming is that there are many choices available at any time, and that the user controls the flow of the program. In a typical Apple IIGS program, for example, the user can select choices from a half-dozen menus, open or close windows, use desk accessories, resize or move windows, or do some sort of work (such as word processing or drawing). With few exceptions, any of these operations is available at any given time.

Events that cause a response by the program include key presses and mouse-button clicks, and might also include use of game paddles, insertion of a disk in a disk drive, data coming over a communication line, or even events generated by the program itself.

The main event loop

An **event** is a notification to the application of some occurrence, internal or external, to which the application may choose to respond.

Although an event-driven program may at first appear extremely complex, its basic structure is actually quite simple. It spends most of its time waiting in a program loop called the *main event loop.* The only thing the program is waiting for is an **event**—any event. When it detects an event, it determines the type of event, takes whatever action is necessary, and returns to the main event loop to wait for the next event.

Figure 1-5 is a conceptual representation of the flow of execution in an event-driven program. For most of the time, the taxi (program execution) remains in the event loop, circulating constantly, and stopping at the taxi stand (the event queue) each time to see if there is a waiting passenger (an event). The taxi takes passengers in order, one at a time, to their respective destinations (various event-handling subroutines). The taxi waits out front while the event is being handled (execution temporarily leaves the event loop), then proceeds around the loop once more to pick up another passenger. Circulation continues until the program ends.

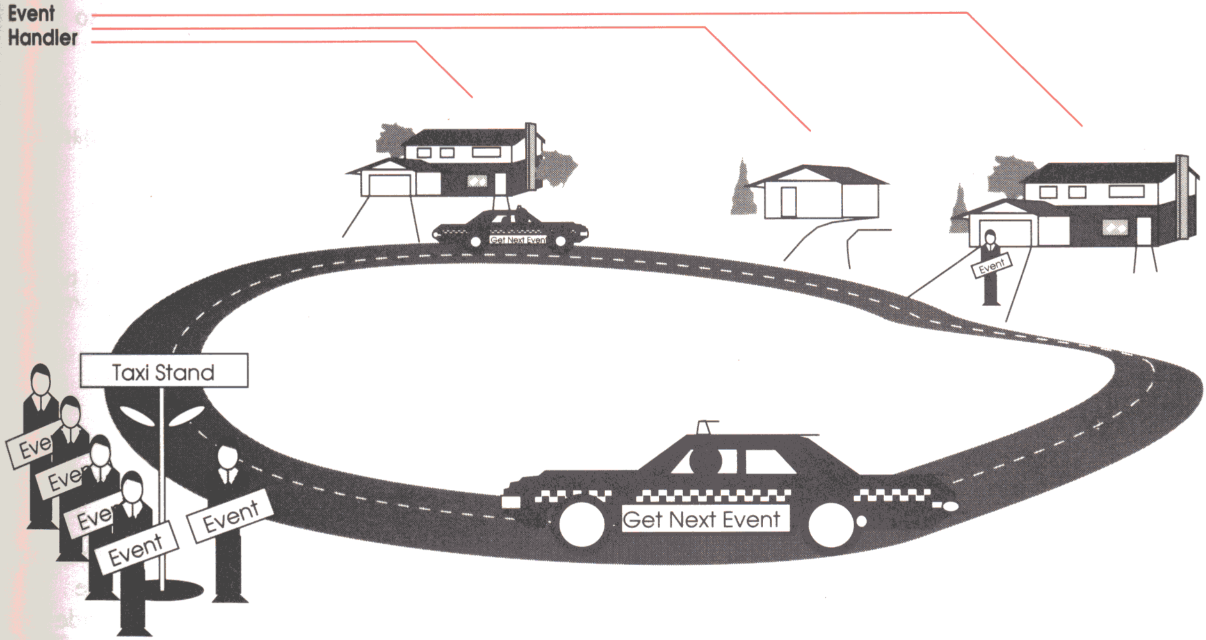


Figure 1-5
The main event loop

For a more specific example, assume that the program is a word processor. From the user's point of view, any of a large number of operations are available, from typing a character to reformatting the entire document to setting control-panel options. The main event loop, however, need wait for only two types of events: mouse-button-down and key-down.

Event handling

To illustrate how a program handles an event, let's suppose that the user decides to select an item in a window titled CORTLAND (see, for example, Figure 1-4) that is open on the screen but not currently active. The user moves the mouse to the inactive window and clicks the mouse button. When the program detects the event, it handles it like this:

1. What kind of event was it (mouse-down, key-down, and so forth)?

   Mouse-down

   (At this point execution leaves the main event loop to handle the event.)

2. Was it in a window, on the menu bar, or neither?

   Window

3. Was it the active window or an inactive window?

   Inactive

4. Which inactive window?

   CORTLAND

5. Make CORTLAND the active window.

6. Return to the main event loop.

Why return to the main event loop now instead of going to a loop that can handle events that can occur only within the active window? Because the user might change his mind and decide to open a menu, select a different window, or even quit the program. If you return to the main event loop as soon as possible, the user retains the feeling of being in control of the program.

The structure of an event-driven program can be fundamentally different from that of other types of applications. Its principal subroutines are organized by events to handle ("mouse-down," "key-down"), rather than by the specific tasks the program was written to perform ("text entry," "drawing"). Chapter 2 illustrates this difference in detail.

TaskMaster is described in Chapter 3.

The Apple IIGS provides a large number of software tools that make it easier for you to write an event-driven program. The Event Manager performs the bookkeeping that makes your program's main event loop work—it gathers events, determines their types, and places them in order, for your program to handle. A toolbox routine called *TaskMaster* automatically takes care of simple event-handling such as resizing or moving a window. Then it passes the information on to your program.

We'll look at events in much greater detail as we go along. Chapters 2 through 5 describe the sequence of tool calls and procedures that an event-driven program must execute on the Apple IIGS, and Appendixes E through G present source code for such a program in three different programming languages (assembly language, C, and Pascal).

The Apple IIGS Toolbox

Trying to write a desktop, event-driven application without the aid of some powerful system software could be quite difficult. Fortunately, the Apple IIGS comes equipped with a software *toolbox*, which contains a complete complement of *tool sets* designed to make your job easier.

Tool and *tool set* are synonymous.

The Apple IIGS tools support the standard desktop interface and provide you with building blocks to help you construct your application.

What is a tool set?

A **tool set** in the Apple IIGS environment is a collection of related software routines that provides one major capability. Each routine within a tool set performs a fundamental operation. For example, the QuickDraw II tool set provides routines that handle graphics on the Apple IIGS. Within QuickDraw II, SetPenSize and SetPenMode are routines that set the pen size and pen mode. A routine may take one or more specific parameters as input and yield one or more values as output.

Pen size and pen mode are discussed under "Drawing to the Screen," in Chapter 3.

The tool sets, then, are groups of related routines that perform many common tasks and are always available for your application's use. Taken together, the tool sets are very similar to the Macintosh toolbox. Many of the capabilities of the Apple IIGS, even those not directly related to desktop applications, are easily accessed through the tools. For example, both the Memory Manager (which allocates all Apple IIGS memory) and the Event Manager (which controls event-driven programs) are tool sets.

Manager is simply another name for *tool set*.

Why use tool sets?

Making use of tool sets allows you to concentrate on your application's specific business rather than on background work.

A number of the tools are in ROM. They are therefore available to all programs without using disk space. Additional tools are available in RAM, but you needn't worry about where a particular tool set or routine is. A tool set called the Tool Locator, which enables tool sets and applications to communicate, takes care of the necessary bookkeeping functions. All you need to know is the name of the routine and how to call it in your programming language.

You can even use the Tool Locator to access a tool set you have written yourself. See "User Tool Sets" in Chapter 8.

Tool sets insulate your program from the details of machine hardware. If the program accesses a hardware feature with a tool call, the program will remain compatible through future versions of the Apple IIGS, even if the hardware feature changes.

The tools thus provide an abundance of capabilities at a minimum cost in programming time and memory space. Their bookkeeping functions are almost automatic, the interface to them is simple, and the applications you write will not be rendered obsolete by future changes to the hardware.

❖ *Note:* Many of the Apple IIGS tool sets are independent of the operating system. They are thus available for any Apple IIGS application, regardless of the operating system it is written for.

To get an idea of the range of capabilities of the Apple IIGS Toolbox, it's useful to group the tool sets into categories. The arrangement given in Figure 1-6 is arbitrary; as you get to know the tools better, you may prefer other groupings.

Brief explanations of the tool sets within each category follow. The tool sets are described in more detail in Chapters 3 through 6.

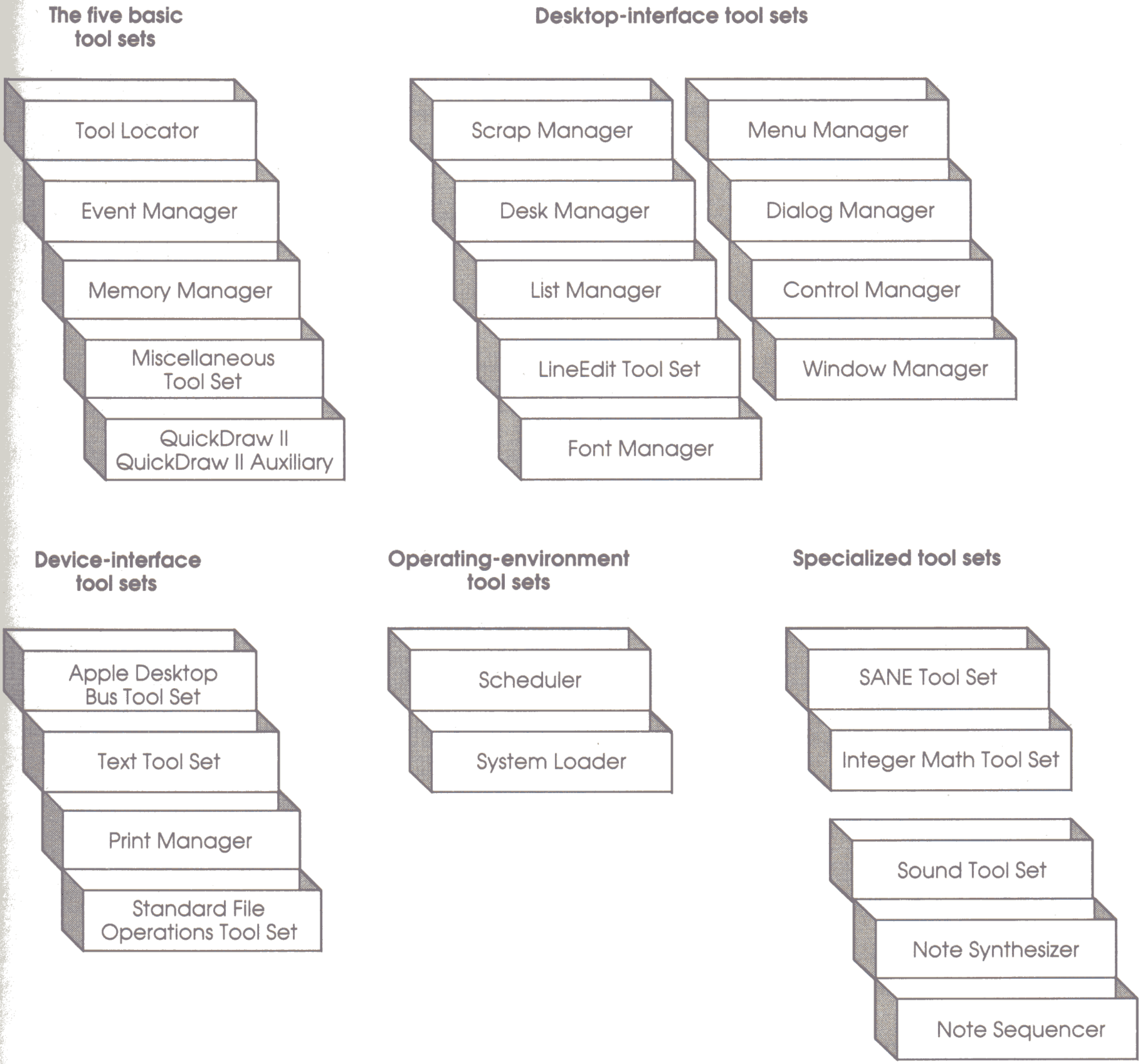


Figure 1-6
Apple IIGS tool sets

The five basic tool sets

The five tool sets listed below provide the framework upon which the other tools can build. All of these tool sets must be used in every event-driven application:

- **Tool Locator:** Handles all tool calls. This tool set relieves you of having to know where in memory any tools resides; the Tool Locator finds and passes execution to the proper routine when you make a tool call. Once you start the Tool Locator, its operation is automatic.
- **Memory Manager:** Allocates memory for use by the application. When your application needs memory, it must request it from the Memory Manager.
- **Miscellaneous Tool Set:** Includes mostly system-level routines that must be available for other tool sets to use.
- **QuickDraw II:** Controls the graphics environment and draws basic graphic objects and text on the screen. **QuickDraw II Auxiliary** is an extension to QuickDraw II. Other tool sets call QuickDraw II and QuickDraw II Auxiliary to draw such things as windows and icons.
- **Event Manager:** Receives events as they happen, maintains a queue of events, and passes events on to the application.

Desktop-interface tool sets

Tools in this group support the desktop interface. You will almost always use the Window Manager and Menu Manager in desktop programs; you should use the other tool sets if your application needs their features (for example, you need the Dialog Manager if your application uses dialog boxes). Many of these tools are also needed to support desk accessories.

The list of tool sets needed to support desk accessories is given in Table 8-1.

- **Window Manager:** Creates and updates windows, keeps track of size changes and overlapping.
- **Control Manager:** Implements *controls*—objects on the screen such as check boxes—which the user can manipulate with the mouse to cause instant action or to change settings.
- **List Manager:** Along with the Control Manager, handles ordering, display, and selection of lists of selectable items in windows.

- **Dialog Manager:** Implements *dialog boxes*, which your application should place on the screen when it needs more information to carry out a command.
- **LineEdit Tool Set:** Presents text on the screen (usually in dialog boxes), and allows the user to edit that text in limited ways.
- **Menu Manager:** Controls and maintains *pull-down menus* and the items in the menus.
- **Font Manager:** Provides fonts in a variety of sizes and styles for QuickDraw II to use when it draws text.
- **Scrap Manager:** Supports the *desk scrap*, data to be copied from one application to another (or from one place to another within an application).
- **Desk Manager:** Enables applications to support *desk accessories*, mini-applications that can be run at the same time as another application.

Device-interface tool sets

Tools in this group manage input and output between the computer and peripheral devices.

- **Print Manager:** Carries out page-setup and printing commands from the user. Provides an interface between the application and printer drivers.
- **Standard File Operations Tool Set:** Presents dialog boxes to the user when a file is to be saved or opened. Provides a standardized interface between the user and ProDOS 16.
- **Apple Desktop Bus Tool Set:** Provides access to Apple Desktop Bus commands. The Apple Desktop Bus transmits signals to and from the keyboard, mouse, and other input devices.
- **Text Tool Set:** Allows applications running in native mode to access Apple II character device drivers, which require the processor to be in emulation mode.

Operating-environment tool sets

Tool sets in this group control low-level hardware and software functions. The Memory Manager and the Miscellaneous Tool Set listed under "The Five Basic Tool Sets," can also be considered part of this group. Other members are:

- **System Loader:** Loads all program and data segments into memory.
- **Scheduler:** Allows more than one program to access system resources that normally cannot be shared.

Specialized tool sets

Tool sets in this group perform various specialized functions, as listed.

Sound generation

These tools make it easy to take advantage of the advanced sound capabilities of the Apple IIGS.

- **Sound Tool Set:** Constitutes the sound hardware's interface to the Apple IIGS Toolbox, and provides basic sound manipulation routines.
- **Note Synthesizer:** Facilitates creation of musical notes simulating a variety of instruments.
- **Note Sequencer:** Strings together notes from the synthesizer into the sequences, patterns, and phrases that make up a tune.

Mathematical computation

These tools perform mathematical functions and calculations.

- **Integer Math Tool Set:** Provides mathematical routines that manipulate integers, long integers, and signed fractional numbers. Also converts numbers to hexadecimal and decimal ASCII strings.
- **SANE Tool Set:** Implements the Standard Apple Numerics Environment, which provides extended-precision floating-point arithmetic that conforms to IEEE standard 754. Supports multiplication and division and trigonometric and other transcendental functions.

Program segmentation

Another powerful feature available to Apple IIGS programs is that they can be *segmented,* and the segments can be *relocatable* and *dynamic.* A segmented program is divided into chunks that can be loaded into memory piecemeal. A relocatable segment is a piece of code or data that needn't be put at any particular memory address in order to function correctly. A dynamic segment is one that is not loaded until it is needed during program execution.

Load files are programs in machine-executable format. See *the Apple IIGS Programmer's Workshop Reference* for information on the file format for program segments.

Segmentation of executable programs (**load files**) gives two principal advantages: (1) your program might fit into a smaller memory space to help it run in small-memory machines and under application-switching programs, and (2) it might load and start to execute more quickly. Both advantages occur because less-needed segments can be made dynamic and left on disk until they are actually called into use. Furthermore, on the Apple IIGS computer no single block of code can occupy more than 64K bytes of contiguous memory. To load a larger program than that, you must split it up into two or more load segments.

Making your load-file segments relocatable means that the available memory in the computer can be allocated efficiently among multiple programs (including system software and desk accessories).

See the *Apple IIGS ProDOS 16 Reference* for complete information on ProDOS 16 and the System Loader.

Segmentation works because the Apple IIGS Memory Manager and System Loader tool sets, work together with ProDOS 16, the Apple IIGS operating system, to execute, move, and remove program segments in a fashion that is sophisticated yet totally transparent to the program user (and in many cases to the programmer too). The Memory Manager takes care of assigning each segment to a block of memory; the System Loader keeps track of where in memory the segment has been loaded, and patches intersegment calls in each segment as it is loaded. ProDOS 16 controls execution of the programs once they are in memory.

See the *Apple IIGS Toolbox Reference* for complete information on the Memory Manager.

Chapter 6 presents a more detailed discussion of load-segment structure and how the Memory Manager, System Loader, and ProDOS 16 interact to make it all work.

Absolute and relocatable segments

To make efficient use of memory with segmented programs, the Memory Manager and System Loader need to be free to place code and data segments where they choose.

Absolute code is computer code that must be loaded at a specific address in memory and never moved. Many standard Apple II programs contain absolute code. The programmer decides where the program will sit in memory, and designs all address references and subroutine jumps accordingly.

Patching is the process of modifying code once it is in computer memory.

Relocatable code is computer code that contains relative and symbolic address references, and so can execute correctly wherever it is placed in memory. See Figure 1-7. Once it is in memory, relocatable code must be **patched** by the loader so that its address operands contain the proper values.

For efficient memory use, it is very important that as many segments as possible be relocatable. The Memory Manager must be free to place segments so they will not conflict with each other, and so that contiguous areas of free memory are maximized. None of your program's segments should be absolute.

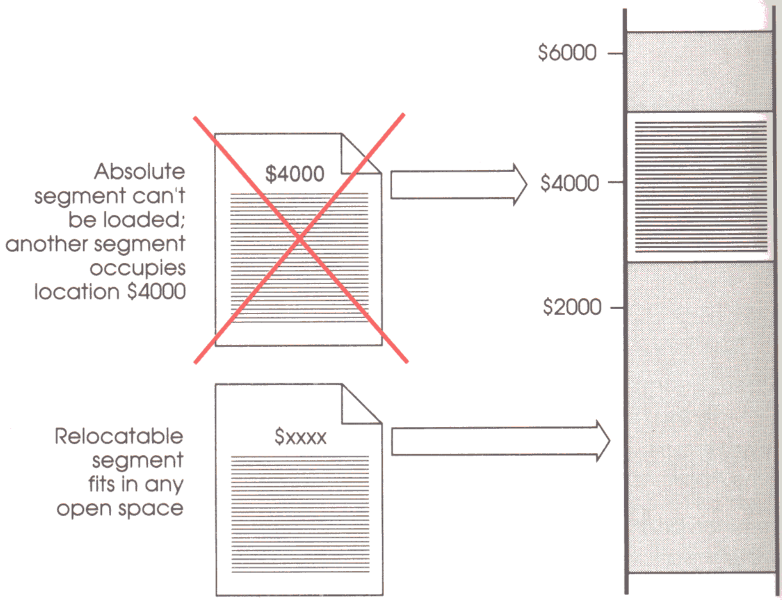


Figure 1-7
Absolute and relocatable segments

Static and dynamic segments

A *dynamic* segment is a load segment that can be loaded and run automatically during program execution. The application itself needn't do any loading—whenever the application calls a routine that is in a dynamic segment, the segment is automatically loaded and executed. Furthermore, that dynamic segment is not subsequently unloaded from memory unless the application permits it, and even then only when the memory is needed for something else; in most cases the segment remains instantly available the next time it is called.

See Chapter 6 for more information on how static and dynamic segments are loaded.

A segment that is not dynamic is *static*. A static segment is loaded at program startup, and is not unloaded or moved during execution. The main segment of any program is static; any other segments may be static or dynamic. See Figure 1-8.

The question of which segments to make static and which ones to make dynamic is not as easily answered as the question of absolute and relocatable. At least one segment in each program must be static; if the program is small, that single segment may constitute the entire program. But if the program is large or if it is designed to require little memory, many of its segments may be dynamic.

Making as many segments dynamic as possible can also decrease the time required to initially load and start up a program. On the other hand, there may then be momentary delays during execution, as the dynamic segments are loaded when called.

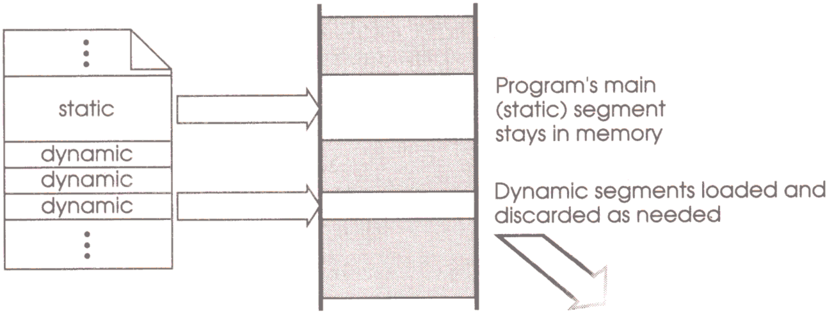


Figure 1-8
Static and dynamic segments

The Programmer's Workshop

To help you write application programs that make the most of the new Apple IIGS features, Apple has produced an integrated development environment called the Apple IIGS Programmer's Workshop (APW for short).

APW helps you create event-driven, segmented desktop applications that access the full power of the Apple IIGS Toolbox. With APW you can write modular source-code segments in a variety of high-level and low-level programming languages, and then combine them into a single functioning program.

An **object file** is a program that has passed through an assembler or compiler. It contains machine-language code.

Chapter 6 shows you how to specify object segments and load segments.

APW's **object files** and load files follow a file specification called *object module format* (OMF). OMF was developed, in part, to create a system in which program segments written in several languages could be combined and run together, because they all would have one uniform object file "language". With OMF you can optimize various routines by writing them in different languages and combining them into a single program. A routine written for a program in one language can be dropped into another program in another language, without modification.

Figure 1-9 is a simplified picture of what takes place from writing to running an application under APW.

A **source file** is a program in its original text form, as written by the programmer.

1. A program is first created as a **source file,** using a text editor appropriate for the language(s) involved. APW includes a full-featured, multi-language text editor.
2. The source file, in ASCII text form, is then either compiled or assembled to produce an *object file.* Directives in the source file control whether and how the object file is to be segmented. A single source file can be compiled into more than one object file.
3. The object file is converted by a linker into a *load file.* Directives in the original source file, as well as commands to the linker, can control segmentation in the load file. More than one object file can be combined into a single load file.
4. In the final step (if all goes well), the load file runs correctly when the loader places it in memory and it is executed. In the early stages, of course, program development usually involves at least some time with a debugger such as the Apple IIGS Debugger.

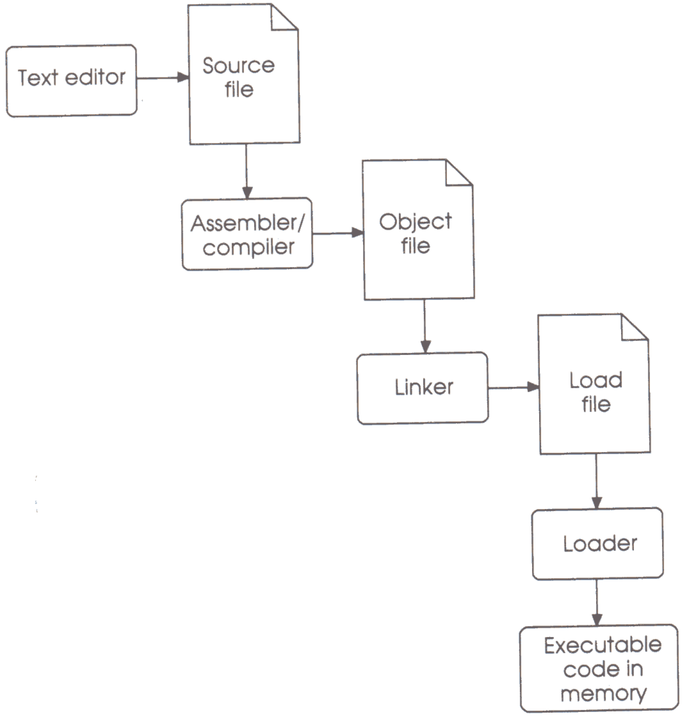


Figure 1-9
Steps in creating an application

Using APW to design and write segmented programs is covered in Chapter 7. But before we get too deeply into the *how* of Apple IIGS programming, we'd like to show you some more of the *what* and *why*. The next five chapters present an extensive programming example and give some additional background, showing what Apple IIGS programs can do and why they go about it in the ways they do.

Chapter 2

HodgePodge: A Sample Event-Driven Application

Now that you've had an overview of the Apple IIGS and programming concepts, let's plunge right into an example.

This chapter explores a demonstration application developed by Apple, called **HodgePodge.** HodgePodge has a recommended organization for event-driven, desktop applications on the Apple IIGS. We walk you through the program, presenting the code and explaining it in detail as we go along.

You may wonder why we're dissecting the sample program so soon—after all, much of its structure and most of its tool calls and parameters aren't explained until later in the book. Our hope is that, given the general concepts already presented and the extensive commentary accompanying these listings, your quickest route to understanding is to see actual code from a functioning program.

On the other hand, there is no required reading order for this book. If you want to delve deeper into toolbox concepts before looking at code samples, by all means skip ahead to Chapters 3 through 5. Come back to this chapter when you're ready.

Don't forget to look in Appendixes E, F, and G for the complete source-code listings of HodgePodge in all three languages (assembly language, C, and Pascal). And, whichever order you read things in, don't forget to try HodgePodge in action on your Apple IIGS!

What HodgePodge does

HodgePodge is a short application that loads stored graphic images (picture files) from disk and displays them in movable, scrollable, resizeable, overlapping windows on the screen. It also displays, in windows, text samples of the various fonts available on your system. See Figure 2-1.

Like a proper desktop application, HodgePodge shows menus, displays messages in dialog boxes, supports desk accessories, stores and retrieves files, prints text and graphics, and even provides an "About HodgePodge" dialog box accessible from the Apple menu.

If you have a copy of the sample program, put it in your computer and run it now. On the disk that accompanies this book, it's the application named HP, in the folder for any of the three languages. (There are three files named HP—one for each language.)

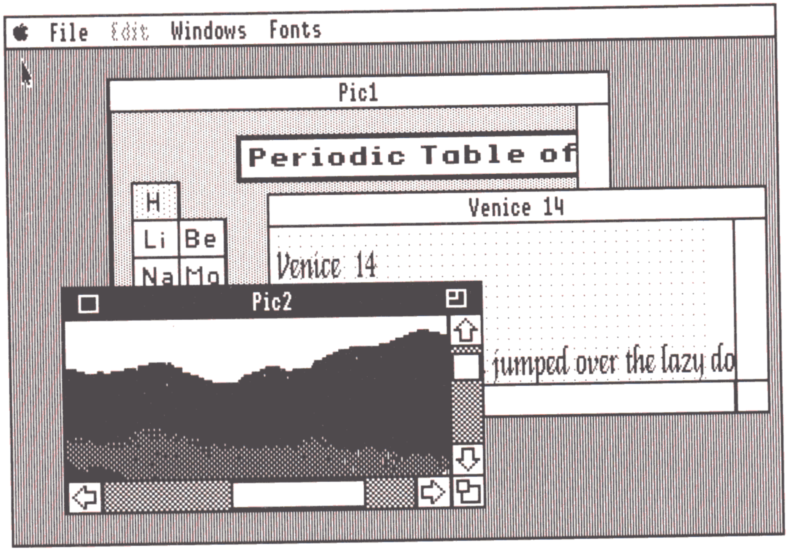


Figure 2-1
HodgePodge desktop

HodgePodge's menus

HodgePodge displays five pull-down menus from a menu bar at the top of the screen: Apple menu, File menu, Edit menu, Windows menu, and Fonts menu. Within each menu, items that the user may select appear in black; items that the user may not select are dimmed (gray). When the user selects an item on a menu, that menu's title is highlighted until the selected task is completed.

Apple menu

The Apple menu is a standard menu that all desktop applications should have. Its title is a small, colored Apple icon. The first item in the Apple menu is "About HodgePodge." Selecting it brings up a dialog box that explains a bit about the progam and its authors. "About" dialog boxes are typical of desktop programs.

New desk accessories are described under "Supporting Other Desktop Features" in Chapter 5.

The Apple menu also lists the *new desk accessories* available on the user's system.

File menu

The File menu is a standard menu that all desktop applications should have. Here it contains seven items:

- **Open:** Opens a picture file and displays it in a window.
- **Close:** Closes the frontmost or active window.
- **Save As:** Allows the user to save a picture window with its present filename or under another name.
- **Choose Printer:** Allows the user to select a printer.
- **Page Setup:** Lets the user set certain parameters for printing.
- **Print:** Prints the contents of either a picture window or a font window.
- **Quit:** Shuts down the program.

All of the items in the File menu are standard, but their implementation in some cases is specific to HodgePodge.

Edit menu

The Edit menu is a standard menu that all desktop applications should have. Here it contains five items:

- **Undo:** Allows the user to reverse the last action undertaken.
- **Cut:** Deletes the selected part of a document and places the selection in the Clipboard.
- **Copy:** Puts a copy of the selected part of a document in the Clipboard.
- **Paste:** Copies the contents of the Clipboard into a document.
- **Clear:** Deletes a selected part of a document, without affecting the Clipboard.

The Clipboard and the concepts of cut, copy, and paste are described under "Supporting Other Desktop Features" in Chapter 5.

HodgePodge itself does not use the Edit menu; however, the Edit menu must be present in case a desk accessory that needs it is activated.

Windows menu

The Windows menu is nothing but a list of HodgePodge's currently open windows. The list is arranged in the order in which the windows were opened. Selecting the name of a window from the Windows menu causes that window to be brought in front of any other open windows on the desktop.

Fonts menu

With the Fonts menu, the user can display a piece of sample text using any font on the system, in any size and with any desired styling variation (such as bold or italic). Selecting the first item on the menu brings up a dialog box with which the user selects the font to display, and then draws the text in a window. Selecting the second item toggles the display of the next-opened font window between proportionally spaced and monospaced display.

HodgePodge's picture windows

HodgePodge retrieves, displays, and stores color pixel images in a particular type of picture file. The user may open a file, view the picture, and then save the file again with the same or another name.

With picture windows, HodgePodge demonstrates how to create windows and how to display images on the screen. It also shows an example of file access and demonstrates color printing. Figure 2-2 is an example of a picture displayed in a window.

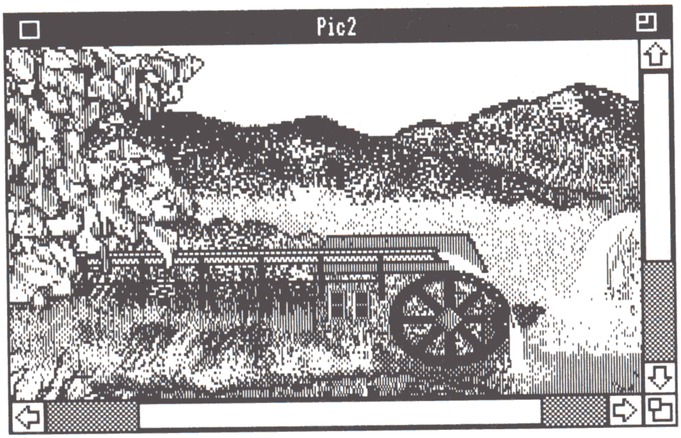


Figure 2-2
A HodgePodge picture window

HodgePodge's font windows

HodgePodge displays sample text in windows on the screen. The text may be in any point size and may have any combination of styling variations such as bold, italic, or underline. The text may be in any font available on the user's system. The actual lines of text that are displayed are specified in HodgePodge; the user cannot alter them.

Many different font windows, with different sizes and styles, may be open simultaneously. Unlike picture windows, font windows are not opened or saved as files.

With font windows, HodgePodge demonstrates how to create windows and how to draw text on the screen. Figure 2-3 is an example of a font window display.

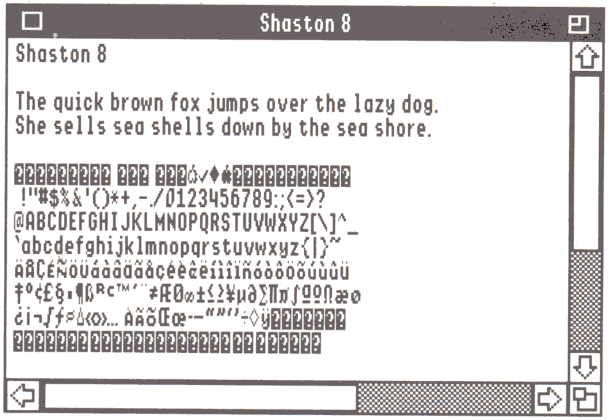


Figure 2-3
A HodgePodge font window

How to use the sample program

The sample program serves two purposes. First, it provides a real framework within which to describe how Apple IIGS applications operate and how they should be written. Second, it provides you with source code modules that you can adapt to your own purposes on your own programs. You are encouraged to use and modify any applicable parts of HodgePodge for any programs you write.

Because you may be writing programs in any of various available Apple IIGS languages, we provide the sample program in three languages—assembly language, C, and Pascal. Complete source code listings are in Appendixes E through G. The parts of the program listings reproduced in Chapters 2 through 6 are in Pascal.

❖ *HodgePodge versions:* Source code and executable forms of HodgePodge, in all three languages, are on the disk that accompanies this book. Slightly different versions of HodgePodge, with different features, have been distributed through other sources such as APDA. See Chapter 9.

Organization

See Appendix D for a complete listing of HodgePodge subroutines.

The source code for HodgePodge consists of many individual subroutines in several separate files. Figure 2-4 shows the overall organization of the principal routines. The main program (on the left) calls each of the principal subroutines (on the right) in order, from top to bottom.

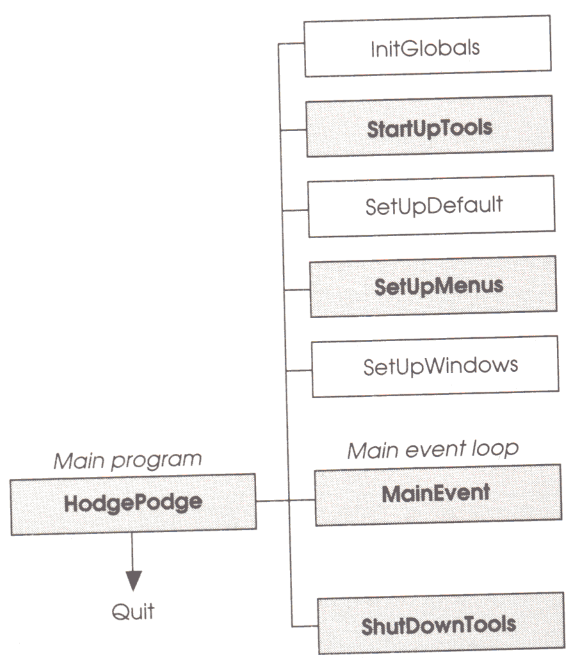


Figure 2-4
HodgePodge organization (simplified)

The most general routines, versions of which will probably appear in every desktop program you write, are more heavily shaded: `HodgePodge`, `StartUpTools`, `SetUpMenus`, `MainEvent`, and `ShutDownTools`. Most execution time is spent in the main event loop (`MainEvent`) and in the subroutines that it calls.

HodgePodge's main event loop is described under "Cycle Through the Main Event Loop," later in this chapter.

Smaller versions of Figure 2-4, highlighted to show particular subroutines, accompany discussions of the principal parts of the program. Another set of subroutine diagrams, starting with Figure 2-5, shows the flow of execution within and from the main event loop.

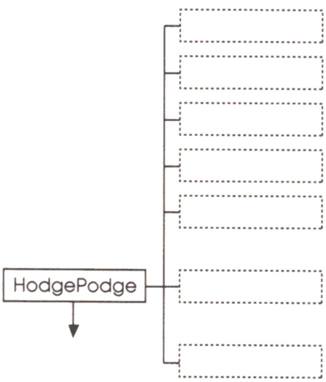
Code-listing convention

The HodgePodge source code listings in this chapter and Chapters 3 through 6 are in Pascal. In addition to the standard Pascal syntax and notation, please note the following conventions:

Pascal HodgePodge was written in TML Pascal™ for APW. See the Bibliography.

- Toolbox calls are in boldface.
- Reserved words (such as *if, then, begin, end, goto*) are in italics.
- Names of functions, procedures, types, and user-defined constants begin with capital letters.
- Names of variables, fields within records, and toolbox-defined constants begin with lowercase letters.
- Boolean values (such as *TRUE* and *FALSE*) are all capital letters.

HodgePodge at a glance: the main program



Briefly, HodgePodge (and any event-driven application) follows this sequence of operations when it executes:

1. It starts up:
   - It initializes variables and data structures.
   - It starts up the tool sets.
   - It sets up the program's menu bar.
2. It continually cycles through the main event loop.
3. As necessary, it handles application-specific events.
4. Finally, it shuts down.

The HodgePodge main program is in the source file HP.PAS.

Most of the above tasks are carried out in subroutines, but they are controlled by the main program. It is very short; this is what the Pascal version looks like:

```
program HodgePodge;                  {begin HodgePodge...}
{.}
{.                                   USES  and other declarations}
{.}

begin
  InitGlobals;                       {Initialize our globals, menus, etc.}

  if StartUpTools then               {if all tool sets loaded OK...}
    begin
      SetUpDefault;                  {Set up print record}
      SetUpMenus;                    {Set up menus}
      SetUpWindows;                  {Set up windows}
      MainEvent;                     {Use the application}
    end;

  ShutDownTools;                     {Shut down IIGS tool sets}

end.                                 {End of HodgePodge}
```

Subsequent sections lead you through the principal subroutines called from the main program. The subroutines cover the steps common to most applications—setting up, handling events, and shutting down.

The details of *how* HodgePodge performs its own specific tasks, such as displaying fonts or pictures, are mostly left for later chapters. Here we are more interested in how HodgePodge illustrates the general independence of form from function in event-driven programs. That is, from a general point of view most desktop applications look pretty much the same.

Step 0. Set the stage

The source code for a typical desktop application begins with statements that bring in needed library files, sets up the operating environment, and perhaps defines some data structures. Many of these statements control what happens when the program is assembled or compiled, rather than when it executes.

- For assembly-language programs, this category includes such tasks as selecting long or short registers, loading macro libraries, and initializing various toolbox data structures with `using` directives.
- For higher-level programming languages, this category may include defining variable types, dimensioning arrays, and loading library files.

Refer to Appendixes E through G for details.

Many constants and data structures are predefined in the interface libraries to the Apple IIGS Toolbox, and thus need not be defined within an application. They include formats and field names for toolbox records and templates, and predefined constants for values such as event codes and memory-block attributes. We'll discuss these and other data structures as we encounter them in HodgePodge.

Step 1. Start the program

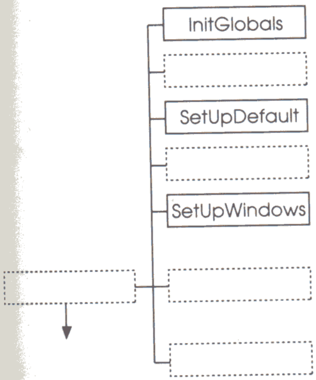
With the preliminaries out of the way, let's look at program execution. To start a desktop program off on the right foot, you need to initialize any program-specific variables and data structures you are going to use, start up the tool sets, and set up the system menu bar.

Initialize variables and data structures

Where and how you define your data and data structures depend upon your program's purpose, the language you're using, and your personal preference.

Pascal HodgePodge has three subroutines called early in program execution to set up initial values of important components of the program. Even though two of these routines are actually called *after* tool startup (as Figure 2-4 shows), all three are grouped here for simplicity. In general, your programs will do some initialization before starting tools, and some after.

Unlike several of the HodgePodge routines described in this chapter, these initialization routines are appplication-specific; your program may have very different ones.



InitGlobals is in the source file GLOBALS.PAS.

❖ *Note:* The initialization routines `InitGlobals` and `SetUpWindows` do not appear in the assembly-language and C versions of HodgePodge. In those languages, variables can be initialized as they are defined in the source file, rather than during execution.

InitGlobals

`InitGlobals` is the first routine called from the main program. It initializes several variables and text strings used later in the program; we will not describe them individually here. It also defines the text strings that constitute HodgePodge's menus. (The unusual formatting of the menu strings is explained under "Making and Modifying Menus" in Chapter 5.) In addition, `InitGlobals` creates a large colored Apple icon that is displayed in the "About HodgePodge" dialog box (Figure 4-14).

```
procedure InitGlobals;                          {begin InitGlobals…}

begin
  with plsWtTemp do                             {template for "Please wait…" dialog}
    begin                                       {--format defined by Dialog Manager}
      SetRect(dtBoundsRect,120,30,520,80);      {set its size}
      dtVisible      :=TRUE;                    {make it visible}
      dtRefCon       :=0;                       {no special info here}
      dtItemList[0] :=pointer(0);               {we'll insert this pointer later}
      dtItemList[1] :=NIL;                      {this terminates the item list}
    end;

                                                {Now define the text of HodgePodge's
                                                 menu titles and items:}

  AppleMenuStr   := concat('>>@\N300X\0',
                     '==About Hodge Podge...\N301\0',
                     '==-\N302D\0.');

  FileMenuStr    := concat('>>  File  \N400\0',
                     '==Open...\N401*Oo\0',
                     '==Close\N255D\0',
                     '==Save As...\N403D\0',
                     '==-\N404D\0',
                     '==Choose Printer...\N405\0',
                     '==Page Setup...\N406D\0',
                     '==Print...\N407*PpD\0',
                     '==-\N408D\0',
                     '==Quit\N409*Qq\0.');
```

```
EditMenuStr    := concat('>>  Edit  \N500D\0',
                     '==Undo\N250*Zz\0',
                     '==-\N501D\0',
                     '==Cut\N251*Xx\0',
                     '==Copy\N252*Cc\0',
                     '==Paste\N253*Vv\0',
                     '==Clear\N254\0.');

WindowMenuStr  := concat('>>  Window  \N600D\0',
                     '== No Windows Allocated\N601D\0.');

FontMenuStr    := concat('>>  Fonts  \N700\0',
                     '==Display Font...\N701*Ff\0',
                     '==Display Font as Mono-spaced\N702*Mm\0.');

                                         {Now initialize other variables,
                                          records & strings:}

lastWindow := NIL;                       {window pointer}

noWindStr  :=
          '==No Windows Allocated\N601D\0.';  {item for Windows menu}
monoStr    :=
          '==Display Font as Mono-spaced';     {Item for Fonts menu}
proStr     :=
          '==Display Font as Proportional';    {Item for Fonts menu}

isMonoFont := FALSE;                     {start fonts as proportional}

with desiredFont do
  begin                                  {set default font characteristics:}
   famNum := $FFFE;                      {family number}
   fontStyle  := 0;                      {plain text}
   fontSize   := 8;                      {8 pt.}
  end;
wIndex := 0;                             {WIndex is the number of open windows}

                                         {Last, define the colored Apple icon
                                          to appear in the "About..." box:}

SetRect(AppleIcon.boundsRect,0,0,64,34); {size of icon}
HPStuffHex(@AppleIcon.data[1],           {HPStuffHex puts pixel values in array}
     '0000000000000000000000000000000');
HPStuffHex(@AppleIcon.data[2],
     '0FFFFFFFFFFFFFFFFFFFFFFFFFFFFFF0');
{.}
{.
{.                                       define each pixel of the}
{.}                                       icon(see Appendix G)}
```

```
   HPStuffHex(@AppleIcon.data[33],
       '0FFFFFFFFFFFFFFFFFFFFFFFFFFFFFF0');
   HPStuffHex(@AppleIcon.data[34],
       '00000000000000000000000000000000');

end;                                          {End of InitGlobals}
```

SetUpDefault

SetUpDefault is in the source file PRINT.PAS.

`SetUpDefault` creates a default print record. (PrRecHdl is a handle-type, that references a Print Manager print record.) `SetUpDefault` must be called after tool startup because it makes Memory Manager and Print Manager calls.

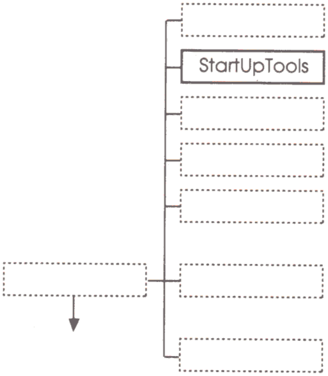
```
procedure SetUpDefault;                        {begin SetUpDefault...}

begin
  printHndl := PrRecHdl(NewHandle(140,          {allocate memory for print record}
                        myMemoryID,              {with our ID}
                        attrNoCross+attrLocked,  {and these attributes}
                        Ptr(0)));                {no location restriction}
  PrDefault(printHndl);                          {fill record with default values}
end;                                             {end of SetUpDefault}
```

SetUpWindows

SetUpWindows is in the source file WINDOW.PAS.

`SetUpWindows` sets initial window size and position on the screen. It is called after tool startup, although in this particular case it could just as easily have been part of `InitGlobals`.

```
procedure SetUpWindows;                        {begin SetUpWindows...}

begin
  wXoffset := 20;                               {set initial window position}
  wYoffset := 12;                               {from top left corner of screen}
  SetRect(iSizPos,10,20,350,80);                {the window's port rectangle}
end;                                            {End of SetUpWindows}
```



Start up the tool sets

Proper initialization, especially for the Apple IIGS Toolbox, is critical for successfully running an application. For that reason, you are urged to simply adopt the following code for your own programs. It works.

In HodgePodge, tool startup is in the subroutine `StartUpTools`, called from the main program right after `InitGlobals`. The steps are shown here in the order in which they are executed in HodgePodge. Although that is not always the precise order in which they must appear in your own source code, tool startup order is in general very important. If you change the order without knowing exactly what you are doing, your program may crash.

The tool startup subroutine peforms three essential tasks:

1. It loads the absolutely necessary tool sets—the Tool Locator, the Memory Manager, the Miscellaneous Tool Set, QuickDraw II, and the Event Manager.
2. Using a *tool table* and a single LoadTools call, it loads all the other tools HodgePodge will need.
3. It starts up those just-loaded tools, in proper order.

❖ *Note:* Many of the startup calls shown below require inputs or return results. Look at the discussions of individual tool sets in Chapters 3 through 5 for more information; see the *Apple IIGS Toolbox Reference* for complete explanations.

StartUpTools is in the source file HP.PAS.

`StartUpTools` begins by starting up the five basic tool sets. It also reserves some memory space (*direct-page space*) needed by several of the tool sets.

```
function StartUpTools : Boolean;                {begin StartUpTools...}

const   TotalDP        = $B00;                  {11 pages total direct-page space}
        DPForQuickDraw = $000;                  {offset to QuickDraw direct pages}
        DPForEventMgr  = $300;                  {offset to Event Mgr direct page}
        DPForCtlMgr    = $400;                  {offset to Control Mgr direct page}
        DPForLineEdit  = $500;                  {offset to LineEdit direct page}
        DPForMenuMgr   = $600;                  {offset to Menu Mgr direct page}
        DPForStdFile   = $700;                  {offset to Std. File direct page}
        DPForFontMgr   = $800;                  {offset to Font Mgr direct page}
        DPForPrintMgr  = $900;                  {offset to Print Mgr direct pages}

var     toolRec    :  ToolTable;                {Tool Locator record-type}
        paramBlock :  FileRec;                  {ProDOS 16 parameter block}
```

```
    baseDP     :  Integer;                  {start address of direct pages}

label  1;                                   {label used for disk-mount loop}

begin
  StartUptools:=TRUE;                       {Start by assuming all will go well}
  TLStartUp;                                {start up Tool Locator}
  CheckToolError($1);                       {check for error}

  myMemoryID := MMStartUp;                  {Start up Memory Manager: it returns
                                             a User ID for HodgePodge to use}
  MTStartUp;                                {Start up Misc Tools}
  CheckToolError($2);                       {check for error}

  toolsZeroPage :=                          {The tools need direct-page space:}
           NewHandle(TotalDP,               {allocate 11 pages, supplying...}
                 myMemoryID,                {...HodgePodge's User ID...}
                 attrBank+attrFixed+
                 attrLocked+attrPage,       {...these memory-block attributes...}
                 Ptr(0));                   {...and make it in bank $00}
  CheckToolError($3);                       {check for error}

baseDP := LoWord(toolsZeroPage^);           {get the 2-byte address of the space}

QDStartUp
           (BaseDP+DPForQuickDraw,          {address of QuickDraw's 3 dir. pages}
           ScreenMode,                      {640 mode}
           MaxScan,                         {max size of scan line}
           myMemoryID);                     {HodgePodge's User ID}
  CheckToolError($4);                       {check for error}

  EMStartUp
           (BaseDP+DPForEventMgr,           {address of Event Mgr's direct page}
           20,                              {event queue size}
           0,                               {X min clamp}
           MaxX,                            {X max clamp}
           0,                               {Y min clamp}
           200,                             {Y max clamp}
           myMemoryID);                     {HodgePodge's User ID}
  CheckToolError($5);                       {check for error}
```

Next, `StartUpTools` loads all RAM-based tools and RAM patches to ROM-based tools at once, with the LoadTools call. It first puts a simple message on the screen to notify the user that it is busy; then it constructs the tool table (the list of all tools to load); and then it loads them.

```
  MoveTo(20,20);                            {Move Pen where we want it}
  SetBackColor(0);                          {Background color = black}
  SetForeColor(15);                         {Foreground color = white}
```

```
    DrawString('One Moment Please...');          {Write the string on screen...}
    ShowCursor;                                  {...and display the arrow cursor}

                                                 {Now load RAM based tools
                                                 (and RAM patches to ROM tools)
                                                 —first, define the contents
                                                  of the Tool table:}

    toolRec.numTools := 14;                      {14 tool sets to be loaded}
    toolRec.tools[1].tsNum := 4;                 {QuickDraw //}
    toolRec.tools[1].minVersion := 0;
    toolRec.tools[2].tsNum := 5;                 {Desk Manager}
    toolRec.tools[2].minVersion := 0;
    toolRec.tools[3].tsNum := 6;                 {Event Manager}
    toolRec.tools[3].minVersion := 0;
    toolRec.tools[4].tsNum := 14;                {Window Manager}
    toolRec.tools[4].minVersion := 0;
    toolRec.tools[5].tsNum := 15;                {Menu Manager}
    toolRec.tools[5].minVersion := 0;
    toolRec.tools[6].tsNum := 16;                {Control Manager}
    toolRec.tools[6].minVersion := 0;
    toolRec.tools[7].tsNum := 18;                {QuickDraw Aux}
    toolRec.tools[7].minVersion := 0;
    toolRec.tools[8].tsNum := 19;                {Print Manager}
    toolRec.tools[8].minVersion := 0;
    toolRec.tools[9].tsNum := 20;                {Line Edit}
    toolRec.tools[9].minVersion := 0;
    toolRec.tools[10].tsNum := 21;               {Dialog Manager}
    toolRec.tools[10].minVersion := 0;
    toolRec.tools[11].tsNum := 22;               {Scrap Manager}
    toolRec.tools[11].minVersion := 0;
    toolRec.tools[12].tsNum := 23;               {Standard File}
    toolRec.tools[12].minVersion := 0;
    toolRec.tools[13].tsNum := 27;               {Font Manager}
    toolRec.tools[13].minVersion := 0;
    toolRec.tools[14].tsNum := 28;               {List Manager}
    toolRec.tools[14].minVersion := 0;

                                                 {Now load the tools we've defined:}
1:                                               {here's the label}
    paramBlock.pathname := @'*/SYSTEM/TOOLS';    {=pathname of tool directory}
    GET_FILE_INFO(paramBlock);                   {Look for that directory:}
    if toolErr<>0 then                           {If it's not there...}
      if MountBootDisk = 1 then                  {Ask user to mount boot disk...}
        goto 1;                                  {...If OK go back and try again}
      else                                       {But if user cancels...}
        begin
         StartUpTools := FALSE;                  {tool startup fails!}
         Exit;                                   {...so quit this subroutine}
        end;

                                                 {But if all is OK...}
    LoadTools(toolRec);                          {...load the tools named in Tool Table}
CheckToolError($6);                              {check for error}
```

Note that, if the disk with the needed tools isn't on line, `StartUpTools` calls the routine `MountBootDisk`, which prompts the user to remount the boot disk so tool loading can continue. `MountBootDisk` is described under "Error Handling" in Appendix D.

Once all the tool sets have been loaded, they need to be started up. `StartUpTools` now starts each one, in the proper order and with the proper input parameters as needed.

```
WindStartUp(myMemoryID);                    {start up Window Manager}
CheckToolError($7);                         {check for error}

RefreshDesktop(NIL);                        {redraw desktop}

CtlStartUp                                  {start up Control Manager}
          (myMemoryID,                      {User ID for memory blocks}
          BaseDP+DPForCtlMgr);              {address of Ctl Mgr's direct page}
CheckToolError($8);                         {check for error}

LEStartUp                                   {start up Line Edit}
          (BaseDP+DPForLineEdit,            {address of LineEdit's direct page}
          myMemoryID);                      {User ID for memory blocks}
CheckToolError($9);                         {check for error}

DialogStartUp                               {start up Dialog Manager}
          (myMemoryID);                     {User ID for memory blocks}
CheckToolError($A);                         {check for error}

MenuStartUp                                 {start up Menu Manager}
          (myMemoryID,                      {UserID for memory blocks}
          BaseDP+DPForMenuMgr);             {address of Menu Mgr's direct page}
CheckToolError($B);                         {check for error}

DeskStartUp;                                {start up Desk Manager}
CheckToolError($C);                         {check for error}

ShowPleaseWait;                             {Bring up a dialog box that says
                                            "Please wait while we..."}

SFStartUp                                   {start up Standard File}
          (myMemoryID,                      {UserID for memory blocks}
          BaseDP+DPForStdFile);             {address of Std File's direct page}
CheckToolError($D);                         {check for error}
SFAllCaps(TRUE);                            {Display file names in all caps}

QDAuxStartUp;                               {start up QuickDraw Aux}
CheckToolError($E);                         {check for error}
```

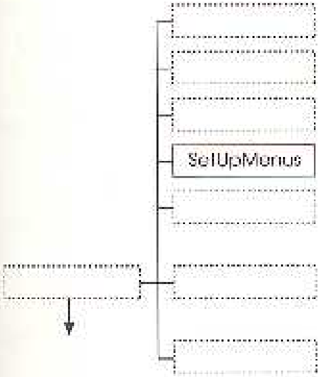
```
    WaitCursor;                             {put up watch cursor,
                                             now that it's available}
    FMStartUp                               {start up Font Manager}
            (myMemoryID,                    {UserID for memory blocks}
            BaseDP+DPForFontMgr);           {address of Font Mgr's direct page}
    CheckToolError($F);                     {check for error}

    ListStartUp;                            {start up List Manager}
    CheckToolError($10);                    {check for error}

    ScrapStartUp;                           {start up Scrap Manager}
    CheckToolError($11);                    {check for error}

    PMStartUp                               {start up Print Manager}
            (myMemoryID,                    {UserID for memory blocks}
            BaseDP+DPForPrintMgr);          {address of Print Mgr's 2 dir. pages}
    CheckToolError($12);                    {check for error}

    HidePleaseWait;                         {Remove the "Please wait…"}
    InitCursor;                             {restore normal cursor}

end;                                        {End of StartUpTools}
```

This completes toolbox initialization. The routine `StartUpTools` ends and returns control to the main program which, in addition to calling the two short initialization subroutines `SetUpWindows` and `SetUpDefault` (described earlier in this section), calls the subroutine that sets up the menu bar. That routine, `SetupMenus`, is described next.

ShowPleaseWait and HidePleaseWait are described under "Constructing Dialog Boxes and Alerts" in Chapter 4.

❖ *ShowPleaseWait:* During tool startup, the HodgePodge routine `ShowPleaseWait` is called. It puts up a dialog box that informs the user that the startup process may take a few seconds. When startup is done, `HidePleaseWait` removes the dialog box from the screen. Keeping the user informed is an important component of the Human Interface Guidelines.

❖ *Error handling:* You may have noted that, after each tool startup call, the HodgePodge subroutine `CheckToolError` is called. `CheckToolError` is a very simple error handling routine; it is described under "Error Handling" in Appendix D. It is good practice to routinely check for errors after making tool calls that can return them.



SetupMenus is in the source file MENU.PAS.

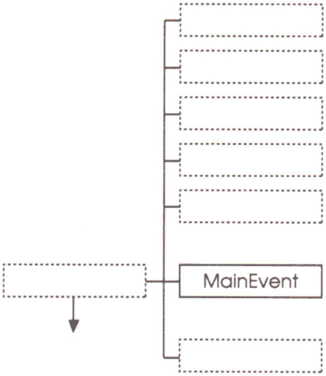
Set up the system menu bar

The routine that sets up the menu bar when HodgePodge starts up is called `SetupMenus`. `SetupMenus` is called from the main program, after `StartUpTools` and the two small initialization routines.

For each menu in turn, `SetupMenus` calls the Menu Manager routine NewMenu, passing it a pointer to a set of character strings that define the menu name and the items it contains. (The menu strings were defined in the routine `InitGlobals`.) NewMenu returns a handle to the newly created menu. `SetupMenus` then calls InsertMenu, passing it the menu handle and a position parameter (here defaulted to zero), to put the menu into the menu bar.

Finally, `SetUpMenus` adds all desk accessory names to the Apple menu (with the DeskManager call FixApplemenu), calculates the height of the menu bar, and draws the bar.

```
procedure SetUpMenus;                               {begin SetupMenus…}

var     height: Integer;                            {= height of menu font}

begin
   SetMTitleStart(10);                              {Set start position, from left edge
                                                     of menu bar, of first menu title}
   InsertMenu(NewMenu(@FontMenuStr[1]),0);          {create and insert Fonts Menu}
   InsertMenu(NewMenu(@WindowMenuStr[1]),0);        {create and insert Windows Menu}
   InsertMenu(NewMenu(@EditMenuStr[1]),0);          {create and insert Edit Menu}
   InsertMenu(NewMenu(@FileMenuStr[1]),0);          {create and insert File Menu}
   InsertMenu(NewMenu(@AppleMenuStr[1]),0);         {create and insert Apple Menu}

   FixAppleMenu(AppleMenuID);                       {Add DAs to apple menu}

   height := FixMenuBar;                            {Set sizes of menus}
   DrawMenuBar;                                     {... and draw the menu bar!}
end;                                                {End of SetUpMenus}
```



TaskMaster and GetNextEvent are further described under "Handling Events" in Chapter 3.

Step 2. Cycle through the main event loop

A desktop application spends most of its time in the main event loop, waiting for an event to handle. How an application functions is determined by what events it chooses to handle and how it handles them. The event loops for most programs are quite similar—it is in the subroutines to which the various events cause branches that the special personality of each application lies.

HodgePodge's main event loop is diagrammed in Figure 2-5. Each time through the loop, HodgePodge checks whether it's time to quit. If it isn't, HodgePodge adjusts menu items if necessary and then looks for the next event. It does this by calling the Window Manager routine *TaskMaster*. Alternatively, an application could call the Event manager routine *GetNextEvent*.

HodgePodge uses TaskMaster because TaskMaster automatically handles many events for it. TaskMaster itself calls GetNextEvent, and takes care of events that affect the size and shape of windows, such as a mouse click in the Zoom, Close, or Grow boxes. This is not a requirement; your application can ignore TaskMaster entirely and do all event-handling itself. For example, you might not use TaskMaster if you want the application to respond in an atypical manner.

If TaskMaster can't completely handle an event, it passes a *task code* (described in "Handling Events" in Chapter 3) back to the application, and the application must deal with the event specified by that code. For example, if the user selects a menu item, TaskMaster passes the information back to the application, which must find out which item was selected and take the appropriate action.

When action on an individual event is finished, the application (or TaskMaster) returns to the main event loop to wait for the next event.

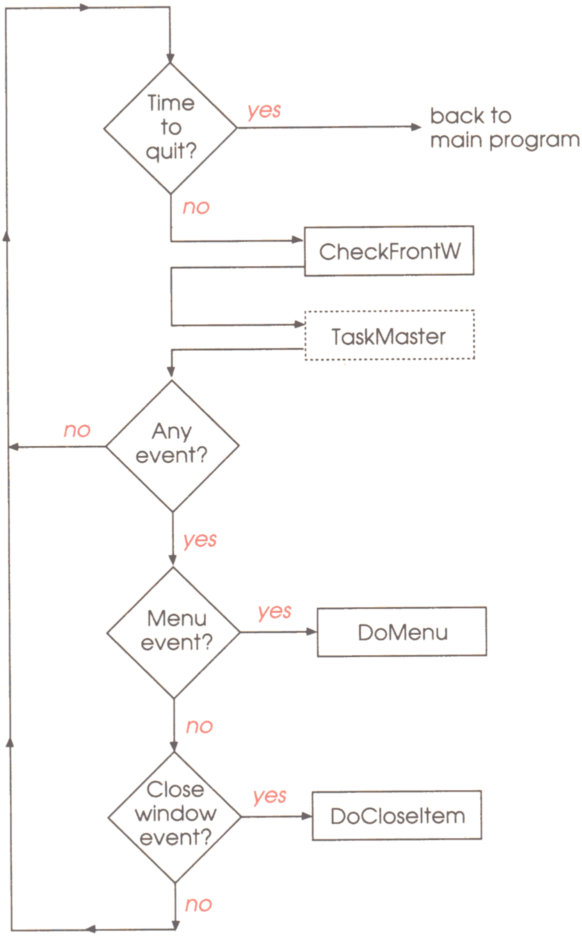


Figure 2-5
HodgePodge's main event loop

The loop

MainEvent is in the source file EVENT.PAS.

Here is the code for HodgePodge's main event loop. Compare it with Figure 2-5. Depending on its features, your application may have an identical event loop, or it may respond to a different set of events.

```
procedure MainEvent;                        {begin MainEvent...}

var    code:  Integer;                      {the task code (or event code)
                                             returned by TaskMaster}

begin
   Event.wmTaskMask := $00001FFF;           {pass all events to TaskMaster}
   done := FALSE;                           {initialize the Quit flag}

   repeat
     CheckFrontW;                           {adjust menu items if necessary}
     code := TaskMaster($FFFF, Event);      {Call TaskMaster: let it handle
                                             all events; record name=Event; it
                                             returns the task code}
     case code of                           {If the task code represents...}
       wInGoAway:                           {If a window close box selected...}
         DoCloseItem;                       {...go to DoCloseItem}
       wInSpecial,                          {If an Edit-menu item or a...}
       wInMenuBar:                          {...regular menu item selected...}
         DoMenu                             {...go to DoMenu}
     end;                                   {end of Case statement}

   until done;                              {Stop when Done=TRUE}
end;                                        {End of MainEvent}
```

The different events are specified by toolbox-defined constants (such as `wInMenuBar`) that define Event Manager and TaskMaster event codes. See Chapter 3.

The main event loop here is much shorter than it would be if TaskMaster were not used. Without TaskMaster, there might have been as many as 16 separate items in the above case statement, each with its own subroutine call.

❖ *Check front window:* Each time through the loop, before checking for events, HodgePodge determines which window (if any) is the frontmost, and adjusts menu items accordingly. For example, if the front window is a font window, the Save item on the File menu should be disabled because HodgePodge does not save font-window contents to disk. If the front window is a desk accessory window, the Edit menu should be enabled.

  The routine that does this menu manipulation is `CheckFrontW`. It is in the source file `EVENT.PAS`. See Appendix G.

Step 3. Handle specific events

It may already seem that the organization of this program is a little different from what you expected. So far, we've seen no major divisions of the code into "Picture Window Stuff" and "Font Window Stuff," as you might expect in a program whose principal tasks are the manipulation of picture windows and font windows.

Event-driven programs have the equivalents to such modules, but they are chopped up and arranged in different ways. Elements of them are distributed throughout the flow of events in the program.

Therefore let's continue along the path of execution, seeing where we go when we leave the main event loop to handle the events that HodgePodge responds to. We'll mention each of the types of events and point you to where in the book to look for the specific routine that handles that event type.

TaskMaster-handled events

In HodgePodge, TaskMaster automatically handles all moving, resizing, scrolling, activating, updating, and redrawing of windows. It handles nearly all window events automatically. This is a great convenience (as you can imagine if you are a Macintosh programmer) and it means that, apart from closing a window, there is little for HodgePodge to do in terms of window manipulation.

Window closing is described under "Window-Related Events," later in this section.

In general, there is one thing that TaskMaster cannot do for an application, and that is draw the contents of a window. TaskMaster cannot know what purpose the application created the window for. But, if a window's contents can always be described by a routine, an application can provide TaskMaster with a way to call that routine whenever a window is drawn. That routine, although part of your program, acts as a sort of extension to TaskMaster, and it can do the redrawing of the window's contents. Such routines are called window-content *definition procedures.*

Window-content definition procedures are discussed under "Creating Windows" in Chapter 4.

HodgePodge uses this trick for both picture windows and font windows. Figure 2-6 is an extension to part of the event-loop diagram of Figure 2-5, and shows the window-drawing routines that are called from within TaskMaster.

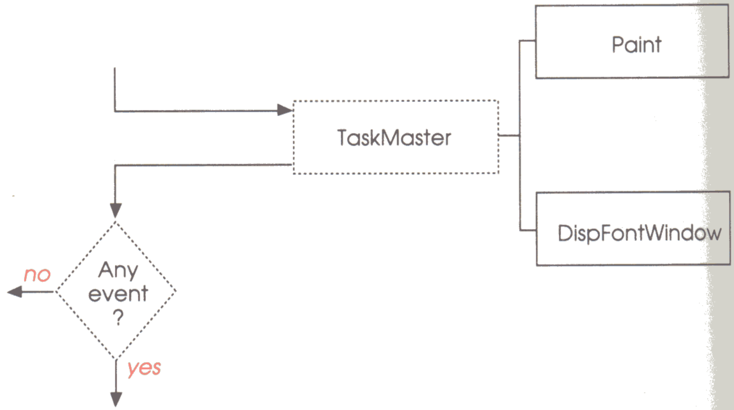


Figure 2-6
HodgePodge routines called by TaskMaster

❖ *Note:* Don't get the impression from Figure 2-6 that drawing window contents is *all* that TaskMaster does. TaskMaster does many more things, as already discussed, but `Paint` and `DispFontWindow` are the only *HodgePodge* routines that TaskMaster calls.

Picture window contents

Paint is in the source file PAINT.PAS.

When a picture window's contents need to be drawn or redrawn, TaskMaster calls the definition procedure `Paint`, which sets up the proper parameters and then calls the routine `PaintIt` to do the actual drawing. `PaintIt` is described under "Drawing to the Screen (and elsewhere)" in Chapter 3. `Paint` looks like this:

```
procedure Paint;                                   {begin Paint…}

var     tmpPort     : GrafPortPtr;                 {pointer to a grafPort}
        myDataHandle: WindDataH;                   {handle to a window-data record
                                                    --defined in GLOBALS.PAS}
begin
   tmpPort      := GetPort;                        {get a pointer to current port}
   myDataHandle := WindDataH(                      {Get a handle to the window-data…}
                        GetWRefCon(tmpPort));      {…record for the current port}
                                                   {Using the picture pointer in the…}
   PaintIt(myDataHandle^^.pict);                   {…record, call the routine that
                                                    draws picture-window contents}
end;                                               {end of Paint}
```

Note that Paint (and PaintIt too, as you will see) is completely unconcerned about where on the screen the window to be drawn appears, what other windows may or may not be in front of it, and even how big the window is or what part of the picture is being displayed. All these details are taken care of by the toolbox!

Font window contents

DispFontWindow is in the source file FONT.PAS.

When a font window's contents need to be drawn or redrawn, TaskMaster calls the definition procedure DispFontWindow, which sets up the proper parameters and then calls the routine ShowFont to do the actual drawing. ShowFont is described in Chapter 3, under "Drawing to the Screen." DispFontWindow looks like this:

```
procedure DispFontWindow;                      {begin DispFontWindow...}

var    tmpPort      : GrafPortPtr;             {pointer to a GrafPort}
       myDataHandle: WindDataH;                {handle to a window-data record
                                                --defined in GLOBALS.PAS}
begin
   tmpPort       := GetPort;                   {Get pointer to current port}
   myDataHandle := WindDataH(                  {Get a handle to the window-data...}
                         GetWRefCon(tmpPort)); {...record for the current port}

   with myDataHandle^^ do                      {Using font info from the record...}
     ShowFont(theFont,isMono);                 {...call the routine that draws
                                                font-window contents}
end;                                           {End of DispFontWindow}
```

Just as in the case of picture windows, DispFontWindow and ShowFont are completely unconcerned about where on the screen the window to be drawn appears, what other windows may or may not be in front of it, and even how big the window is or what part of the font display is to be drawn. The toolbox does it all.

Menu-related events

Each of the subroutines listed in this section is called as the result of a menu selection made by the user. Thus there is one subheading for each HodgePodge menu entry. Figure 2-7 is an extension to part of Figure 2-5; it shows which routines can be called when the main event loop sends a menu-related event to the routine DoMenu.

DoMenu is in the source file MENU.PAS.

When a menu item is selected (either with the mouse or with a keyboard-equivalent), TaskMaster returns 17 (= `wInMenuBar`—see "Handling Events" in Chapter 3) as the value of `myEvent`, which causes execution to pass to the subroutine `DoMenu`. TaskMaster also sets the `taskData` field of the extended task event record equal to the menu ID and the ID of the item selected, and then passes control back to HodgePodge so it may perform the specific task. `DoMenu` looks like this:

```
procedure DoMenu;                                  {begin DoMenu...}

var    menuNum:       Integer;
       itemNum:       Integer;

begin
   menuNum := HiWord(Event.wmTaskData);           {get number of menu and item}
   itemNum := LoWord(Event.wmTaskData);

   case itemNum of
     AboutItem:       DoAboutItem;                 {bring up "About HodgePodge" dialog}
     OpenItem:        DoOpenItem;                  {open a picture window}
     CloseItem:       DoCloseItem;                 {close a window}
     SaveAsItem:      DoSaveItem;                  {save a picture file}
     ChoosePItem:     DoChooserItem;               {choose a printer}
     PageSetItem:     DoSetupItem;                 {do page-setup}
     PrintItem:       DoPrintItem;                 {print contents of a window}
     QuitItem:        DoQuitItem;                  {quit HodgePodge}
     UndoItem:        ;
     CutItem:         ;
     CopyItem:        ;                            {ignore special menu items}
     PasteItem:       ;
     ClearItem:       ;
     FontItem:        DoOpenItem;                  {open a font window}
     MonoItem:        DoSetMono;                   {set font spacing}
   otherwise
     DoWindow(itemNum);                            {bring the chosen window to front}
   end;                                            {of Case statement}

   HiliteMenu(FALSE,menuNum);                      {unHighlight menu title}

end;                                               {End of DoMenu}
```

The menu ID variables (`CloseItem`, `AboutItem`, and so forth) are defined in the source file `GLOBALS.PAS`.

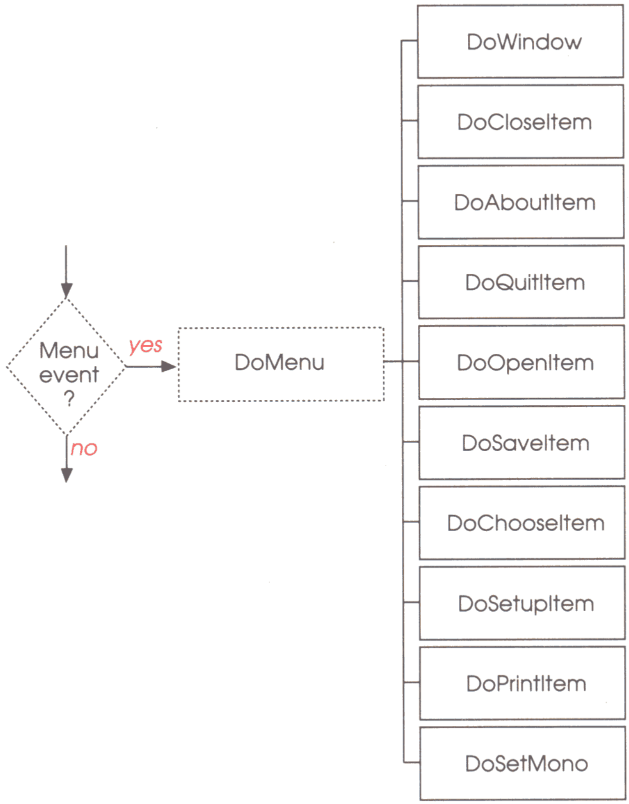


Figure 2-7
HodgePodge routines that handle menu-related events

The various routines called by `DoMenu` are listed either elsewhere in this book or in Appendix G. In brief, this is what each does:

- **DoAboutItem:** Brings up the "About HodgePodge" dialog box. `DoAboutItem` is listed under "Constructing Dialog Boxes and Alerts" in Chapter 4.
- **DoOpenItem:** Opens a font or picture window. `DoOpenItem` calls `OpenWindow` to open the window, then calls `AddToMenu` to add the window's name to the Windows menu. `DoOpenItem` is listed under "Opening a Window: An Example" in Chapter 4.

- **DoCloseItem:** Closes a font or picture window, releases its allocated memory, and adjusts the Windows menu. `DoCloseItem` is listed under "Window-Related Events," later in this section.
- **DoSaveItem:** Saves the contents of a picture window as a disk file. `DoSaveItem` is listed under "Communicating With Files and Devices" in Chapter 5.
- **DoChooserItem:** Brings up a dialog box permitting the user to choose a printing device. `DoChooserItem` is listed under "Communicating With Files and Devices" in Chapter 5.
- **DoSetupItem:** Brings up a dialog box permitting the user to set page-setup parameters. `DoSetupItem` is listed under "Communicating With Files and Devices" in Chapter 5.
- **DoPrintItem:** Prints the contents of the frontmost window. `DoPrintItem` is listed under "Communicating With Files and Devices" in Chapter 5.
- **DoQuitItem:** Assigns the value TRUE to the boolean variable `done`. That causes temination of the main event loop. `DoQuitItem` is in the source file `MENU.PAS`. See Appendix G.
- **DoSetMono:** Toggles a flag that controls whether fonts are displayed as monospaced or proportional, and updates the Fonts menu accordingly. `DoSetMono` is in the source file `FONT.PAS`. See Appendix G.
- **DoWindow:** Brings the selected window (chosen from the Windows menu) to the front. `DoWindow` is in the source file `MENU.PAS`. See Appendix G.

Window-related events

Closing is the only window-related event that HodgePodge must respond to explicitly. Figure 2-8 is an extension to part of Figure 2-5; it shows the routines that can be called when the main event loop encounters a window-related event.

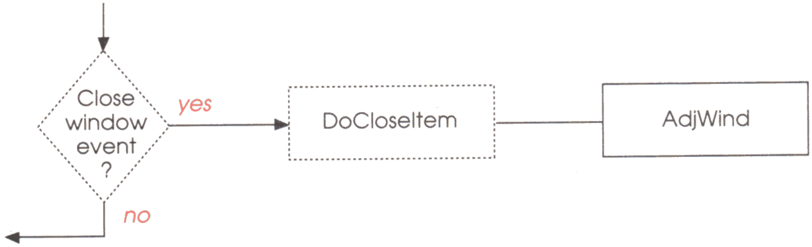


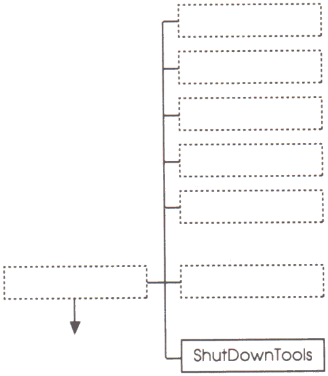
Figure 2-8
HodgePodge routines that handle window-related events

DoCloseItem is in the source file WINDOW.PAS.

Closing is a window event, but it is also a menu event . When the user clicks in an active window's close box, or selects Close from the File menu, TaskMaster returns that information to HodgePodge, which in turn calls `DoCloseItem`. `DoCloseItem` is also called at program shutdown, to close all windows. Its source code looks like this:

```
procedure DoCloseItem;                          {begin DoCloseItem...}

var    theWindow    : GrafPortPtr;              {ptr to window to be closed}
       myDataHandle: WindDataH;                 {window-data-record handle}

begin
  theWindow := FrontWindow;                     {Get a pointer to the front window}

  CloseNDAbyWinPtr(theWindow);                  {Assume that it's a desk acc. window}
  if isToolError then                           {If it wasn't an NDA window...}
    begin
     AdjWind(theWindow);                        {Call AdjWind to update menu}
     myDataHandle := WindDataH(                 {Get a handle to window's...}
                        GetWRefCon(theWindow)); {...window-data record}
      DisposeHandle(Handle(myDataHandle));      {Get rid of the window-data record}
      CloseWindow(theWindow);                   {Get rid of the window completely}
     Dec(wIndex);                               {decrease number of open windows}
    end;                                        {end of IF wasn't an NDA}
end;                                            {end of DoCloseItem}
```

❖ *AdjWind:* `DoCloseItem` calls the HodgePodge routine `AdjWind`, which removes the name of the just-closed window from the Windows menu. `AdjWind` is described under "Making and Modifying Menus" in Chapter 5.



Step 4. Shut down the program

When it's time for your application to quit, the following steps ensure a graceful exit:

1. Shut down all tool sets in reverse order from the way you started them up.
2. Release any memory your application requested from the Memory Manager.
3. Shut down the Memory Manager (with your application's User ID as input).
4. Shut down the Tool Locator.
5. In assembly language, use the ProDOS 16 QUIT call to leave the application. (In C and Pascal, this is taken care of for you).

HodgePodge terminates when the user selects Quit from the File menu. The routine `DoQuitItem` executes, setting the variable done to `TRUE`, which causes the main event loop to stop. Execution passes to the main program, which calls `ShutDownTools` and ends.

ShutDownTools is in the source file HP.PAS.

`ShutDownTools` shuts down all tool sets, in reverse order from startup. You may be able to use this code verbatim in your programs. It looks like this:

```
procedure ShutDownTools;                        {begin ShutDownTools…}
begin
   DeskShutDown;                                {shut down Desk Manager}
   if WindStatus <> 0 then                      {make sure Window Mgr. active…}
     HideAllWindows;                            {close all windows--this may take
                                                 some time if many open windows!}
   ListShutDown;                                {shut down List Manager}
   FMShutDown;                                  {shut down Font Manager}
   ScrapShutDown;                               {shut down Scrap Manager}
   PMShutDown;                                  {shut down Print Manager}
   QDAuxShutDown;                               {shut down Quick Draw Aux}
   SFShutDown;                                  {shut down Standard File}
   MenuShutDown;                                {shut down Menu Manager}
   DialogShutDown;                              {shut down Dialog Manager}
   LEShutDown;                                  {shut down Line Edit}
   CtlShutDown;                                 {shut down Control Manager}
   WindShutDown;                                {shut down Window Manager}
   EMShutDown;                                  {shut down Event Manager}
   QDShutDown;                                  {shut down QuickDraw II}
   MTShutDown;                                  {shut down Misc. Tool Set}
```

```
if MMStatus <> 0 then                          {If Memory Mgr. active...}
  begin                                        {delete the direct-page memory...}
    DisposeHandle(toolsZeroPage);              {...allocated at startup}
    MMShutDown(myMemoryID);                    {shut down Memory Manager}
  end;
 TLShutDown;                                   {shut down Tool Locator}
end;                                           {End of ShutDownTools}
```

❖ *HideAllWindows:* Note that ShutDownTools calls HideAllWindows, which simply closes all windows and releases their associated memory. HideAllWindows is in the source file WINDOW.PAS. See Appendix G.

Conclusion

This completes our overview of the organization of HodgePodge. You can see that it has a structure almost independent of the tasks it was written to perform. That, of course, is the intention—if all event-driven programs execute in a similar manner, they can present a uniform interface to the user. In addition, they can be extended easily to add new features, and they can remain compatible with future revisions of system software.

The rest of the book gives more details on how HodgePodge actually performs its individual tasks, and gives some of the concepts behind the tool calls that HodgePodge, like any event-driven program, needs to make. Most discussions are general, but HodgePodge listings are included where appropriate. See Table 2-1.

Table 2-1
HodgePodge routines described in this book

Routine	See chapter and section...
AddToMenu	Chap. 5: "Making and Modifying Menus"
AdjWind	Chap. 5: "Making and Modifying Menus"
AskUser	Chap. 5: "Communicating With Files..."
CheckToolError	App. D: "Error Handling"
CheckDiskError	App. D: "Error handling"
DispFontWindow	Chap. 2: "Handle Specific Events"
DoAboutItem	Chap. 4: "Constructing Dialog Boxes..."

Table 2-1 (continued)
HodgePodge routines described in this book

Routine	See chapter and section...
`DoChooseFont`	Chap. 3: "Drawing to the Screen"
`DoChooserItem`	Chap. 5: "Communicating With Files..."
`DoCloseItem`	Chap. 2: "Handle Specific Events"
`DoMenu`	Chap. 2: "Handle Specific Events"
`DoPrintItem`	Chap. 5: "Communicating With Files..."
`DoSaveItem`	Chap. 5: "Communicating With Files..."
`DoSetUpItem`	Chap. 5: "Communicating With Files..."
`DoTheOpen`	Chap. 4: "Creating Windows"
`DrawTopWindow`	Chap. 5: "Communicating With Files..."
`HodgePodge`	Chap. 2: "HodgePodge at a Glance"
`InitGlobals`	Chap. 2: "Start the Program"
`StartUptools`	Chap. 2: "Start the Program"
`LoadOne`	Chap. 6: "The ProDOS File System"
`MainEventLoop`	Chap. 2: "Cycle Through the MainEvent"
`MountBootDisk`	App. D: "Error Handling"
`OpenWindow`	Chap. 4: "Creating Windows"
`Paint`	Chap. 2: "Handle Specific Events"
`PaintIt`	Chap. 3: "Drawing to the Screen"
`SaveOne`	Chap. 6: "The ProDOS File System"
`SetUpDefault`	Chap. 2: "Start the Program"
`SetUpMenus`	Chap. 2: "Start the Program"
`SetUpWindows`	Chap. 2: "Start the Program"
`ShowFont`	Chap. 3: "Drawing to the Screen"
`ShutDownTools`	Chap. 2: "Shut Down the Program"

Chapter 3

Using the Toolbox (I)

In Chapter 2, the sample program HodgePodge showed an example of toolbox use in action. Now let's examine some of the concepts behind the toolbox calls HodgePodge makes. Even though an introductory book like this can only get you started with each tool set, the overall view of what the tools can do for you and the example of how HodgePodge integrates them should take you a long way toward understanding and exploiting their power.

The Apple IIGS Toolbox is made up of about 30 *tool sets.* Each tool set is made up of many *routines.* In all, there are more than 800 toolbox routines in ROM and RAM, covering a wide variety of tasks from managing memory to drawing to the screen to giving you the time of day. And don't worry—*you needn't memorize them all to write an Apple IIGS application.* Just the few you learn from this book will get you started.

The complete reference for all toolbox calls is the *Apple IIGS Toolbox Reference,* in two volumes.

You can think of the toolbox as a very large library of prewritten subroutines, optimized and integrated to relieve you of a large part of your programming burden. They exist to free you to concentrate on the fundamental, creative aspects of the program you want to write.

In this chapter we discuss events and how to handle them, and the basic process of drawing to the Apple IIGS screen by using QuickDraw II. Chapters 4 and 5 describe the remaining tool sets. We'll include actual examples from HodgePodge where appropriate, but otherwise the details of individual calls and their parameters are left for other books.

Starting up and calling the tools

Tool sets must be both *loaded* and *started up* before you can call any of their routines. Also, some tool sets call others, so you must start them up in the proper order.

Required tool sets

There are three tool sets required for any application using the Apple IIGS Toolbox. Start them first, and start them in this order:

1. The Tool Locator

2. The Memory Manager

3. The Miscellaneous Tool Set

Beyond these three, there are two other tool sets that, while not absolutely required for the Apple IIGS to function, are nevertheless used in nearly every application. Start them up in this order:

4. QuickDraw II

5. The Event Manager

Refer to "Set the Stage" in Chapter 2 to see how closely HodgePodge follows this sequence.

Other tool sets

After the required tool sets are in place, you should load and start up all other tool sets your application might use.

Loading

To simplify things, and to ensure that the correct versions of tool sets are available, it's best to load all of your needed tool sets at once, with the Tool Locator's LoadTools call. LoadTools does two things: it loads RAM-based tools into the computer (remember, some tools are not in ROM), and it checks the version numbers of all the specified tool sets, whether in ROM or RAM. That version check is important because some tool sets will not function without the proper minimum versions of other tool sets.

When you make the LoadTools call, you pass it a pointer to a **tool table,** which lists the total number of tool sets to load, and the number and minimum acceptable version of each tool set.

HodgePodge's tool table is initialized in the routine StartUpTools.

Important Make sure that all the RAM-based tools your program needs are in the TOOLS subdirectory of the SYSTEM directory on the system disk. See Appendix C.

For a list of all current tool sets and their numbers, see "User Tool Sets" in Chapter 8.

Starting up

After you have loaded the remaining tool sets, you must then start up each one. Each tool set has its own startup call; some calls require or return parameters, others have no inputs or outputs. Because some tool sets require the presence of other tool sets in order to function, tool sets must be started in proper order. Table 3-1 gives the suggested startup order.

Table 3-1
Tool set startup order

Hex.	Dec.	Name
$01	1	Tool Locator
$02	2	Memory Manager
$03	3	Miscellaneous Tool Set
$04	4	QuickDraw II
$06	6	Event Manager
$0E	14	Window Manager
$10	16	Control Manager
$0F	15	Menu Manager
$14	20	LineEdit Tool Set
$15	21	Dialog Manager
$05	5	Desk Manager
$17	23	Standard File Operations Tool Set
$16	22	Scrap Manager
$1C	28	List Manager
$13	19	Print Manager
$1B	27	Font Manager

You can assume that tool sets not on this list are either started up already, or can be started in any order.

❖ *HodgePodge:* You may have noticed that HodgePodge doesn't follow this sequence exactly when it starts its tool sets. Specifically, it starts up the Menu Manager *after* starting the LineEdit Tool Set and the Dialog Manager. So in some instances it may be possible to alter startup order slightly, but it is safest just to follow the order given in Table 3-1.

In addition to the dependencies reflected in the startup order for tool sets, there are additional, complex dependencies among tool sets because a routine in one tool set may call routines in other tool sets (which may call routines in still other tool sets, and so on). These dependencies are beyond the scope of this book; see the individual tool set and routine descriptions in the *Apple IIGS Toolbox Reference.*

Calling an individual routine

You can access toolbox routines easily from either assembly language or high-level languages. The initial languages offered with the Apple IIGS Programmer's Workshop (APW, described in Chapter 7) are 65816 assembly language and C; macro libraries and interface libraries are available for these languages. Any other languages with similar interface libraries (such as the TML Pascal used to write the Pascal version of HodgePodge) allow similar tool-calling procedures.

The Tool Locator

Every time you make a tool call, your request goes through the Tool Locator, the first tool set started up at the beginning of your program. The Tool Locator (in ROM) keeps tables in RAM that point to the individual routines (which may be in either ROM or RAM). The pointer tables are kept in RAM so that they may be easily modified when tool sets are updated, moved to ROM from RAM, or otherwise changed. Your application needn't know or care where a routine is—it just tells the Tool Locator to get it.

The Tool Locator is documented fully under "The Tool Locator" in the *Apple IIGS Toolbox Reference*.

Calling from assembly language

The simplest way to make tool calls from assembly language is to use macros. The macros provided with APW relieve you of having to remember the tool set number and routine number for each call. Assembly-language HodgePodge makes all its calls with macros.

Make a tool call as follows:

1. If the function has any output, push the correct amount of space for it on the stack.
2. If the function has any inputs, push them on the stack in the specified order.
3. Invoke the appropriate macro by name. A macro name is the same as the routine it calls, except that, by convention, it has a leading underline character.
4. Check for errors, as described under "Machine State on Return from the Call," later in this section.
5. Pull the output, if any, from the stack.

The input and output parameters for each assembly-language tool call are described in the *Apple IIGS Toolbox Reference*.

You can make an assembly-language tool call without macros, of course. The method is almost identical to that just described, except that instead of calling the routine by name, you jump to the Tool Locator's entry point ($E1 0000) with a JSL instruction, with the tool set number and routine number as the high-order and low-order bytes in the X register. All tool set and routine numbers are documented in the *Apple IIGS Toolbox Reference.* Nevertheless, it is probably best to use macros because names are easier to remember and read.

Calling from a high-level language

The interface libraries that allow C programmers to access the Apple IIGS Toolbox are included in APW C. Those libraries contain the function definitions for the tools. The steps to call a routine are as follows. Other high-level languages will have similar libraries, appropriate to the languages' structures.

1. Make the routine accessible by using an `#include` statement that includes the appropriate file (for example, `QuickDraw.h` for QuickDraw II calls). The included file will provide the function declarations and the necessary constants and data structures.

The names of the parameters for each tool routine in the C language are described in the *Apple IIGS Toolbox Reference.*

2. Invoke the call by entering its name and supplying the correct parameters.
3. Examine the global error variable (`_toolErr` in C, `ToolErrorNum` in Pascal) for errors, if necessary. If the variable is equal to zero, no errors occurred; otherwise, it contains the number of the error.

Machine state on return from the call

When it completes a call, a toolbox routine returns control directly to the application that called it. The accumulator contains zero and the **c flag** (carry bit) is cleared to zero if the call was completed successfully. Other flags and registers have values dependent on the specific routine called.

For a complete description of register and flag states after a toolbox call, see "Using the Apple IIGS Tool Sets" in the *Apple IIGS Toolbox Reference.*

If an error occurred during the call, the carry bit is set (=1) and the accumulator contains an error code in this format:

high-order byte = tool set number
low-order byte = error number

All toolbox error codes are summarized in an appendix to Volume II of the *Apple IIGS Toolbox Reference*.

❖ *Error passing:* With this method, an error can be properly identified even if it occurs during a call to one tool set, but doesn't actually show up until a call returns from another tool set. For example, using this method, a QuickDraw II call can pass on an error message from the Memory Manager.

Handling events

The central part of any event-driven program is its main event loop. As Figure 2-5 shows in the case of HodgePodge, the program continually cycles through the event loop, waiting for an event to act upon. The application decides what to do from moment to moment by looking at each event and responding to it appropriately.

What constitutes an event? An event is a notification to the program that something has occurred, something that the program may wish to respond to. It may be a signal from outside the program, such as a keystroke. Or it may be something internal, such as the need to redraw part of a window when an overlapping window has been moved.

❖ *Interrupts:* An event is different from an *interrupt* in that it is generated in software, and that it does not *force* action by the application. An application can ignore any event it does not need to act upon.

The Event Manager is the Apple IIGS tool set that notes these occurrences and records them as events (by creating **event records**). For example, whenever the user presses or releases the mouse button, the Event Manager records the action in an event record. The Event Manager collects events from a variety of sources and reports them to the application on demand, one at a time. The Event Manager doesn't necessarily report the events in the exact order they occur, because some have higher priority than others.

The Event Manager is also used by other parts of the toolbox. For instance, when HodgePodge calls TaskMaster in its main event loop, TaskMaster in turn calls the Event Manager. See "Using TaskMaster," later in this section.

The event queue

Most events are placed in an **event queue,** which is an ordered list of event records. As events occur, they are placed at one end of the queue; as the application cycles through its event loop, it pulls events off the other end, one at a time, by making a call such as GetNextEvent.

❖ *TaskMaster:* Rather than call GetNextEvent directly, we suggest that your application call TaskMaster instead. The end result, however is the same—Taskmaster calls GetNextEvent, which pulls events off the event queue as usual. See "Using TaskMaster," later in this section.

Figure 3-1 shows a simplified view of how events are presented to an application. All event types, except switch events and the window-related events *activate* and *update,* pass through the event queue. The various types of events are ordered by priority before the application sees them. Also, the application can filter out, or **mask,** types of events that don't apply to a particular situation.

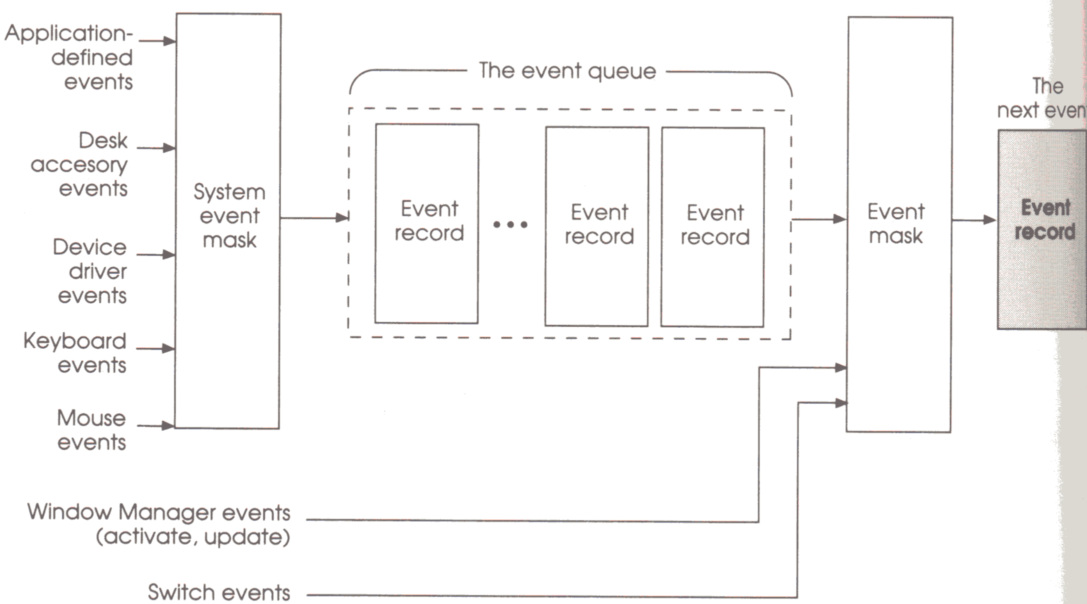


Figure 3-1
Events and the event queue

Event types

Events are of various types. Some report actions by the user; others are generated by the Window Manager, device drivers, or the application itself for its own purposes. The system handles some events before the application ever sees them, and it leaves others for the application to handle.

Each event's type is described by an **event code,** a numeric value that the Event Manager returns to the application getting the event. For programming convenience, there is also a set of predefined constants for these codes. Table 3-2 lists the codes and constants. The first half of the `TaskTable` in the assembly-language version of HodgePodge's main event loop (in the file `EVENT.ASM`) is a code equivalent to Table 3-2; C and Pascal HodgePodge, on the other hand, use the predefined constants to describe event codes.

Table 3-2
Event Manager event codes

Value	Constant	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	`driverEvt`	device-driver event
12	`app1Evt`	application-defined event
13	`app2Evt`	application-defined event
14	`app3Evt`	application-defined event
15	`app4Evt`	application-defined event

From Table 3-2 you can see that 16 is the maximum number of cases your main event loop has to consider if your application calls GetNextEvent. If it calls TaskMaster instead, many of these events are handled automatically; however, there is an additional set of codes, called *task codes,* returned by TaskMaster. See "Using TaskMaster," later in this section.

Event records and masks

Every event is represented by an event record containing all pertinent information about that event. The event record includes the following information:

- **What:** the event code, such as *mouse-down*
- **When:** the time the event was posted (the *tick count*)
- **Where:** the location of the mouse at the time the event was posted, in global coordinates (see "Global and Local Coordinate Systems," in this chapter)
- **Modifiers:** the state of the mouse buttons and modifier keys at the time the event was posted, such as *Option key down*
- **Message:** any additional, event-specific information, such as which key the user pressed or which window is being activated

Some of the Event Manager routines can be restricted to operate on a specific event type or group of types; in other words, some event types are enabled while all others are disabled. For instance, instead of just requesting the next available event, the application can specifically ask for the next keyboard event. It does so by supplying an **event mask** as a parameter. The mask disables any unwanted event types.

There's also a global *system event mask* that controls which event types, the Event Manager posts into the event queue in the first place. When the system starts up, the system event mask is set to post all events.

Responding to events

Here are some typical application responses to commonly occurring events.

❖ *TaskMaster:* These responses apply to a program that uses GetNextEvent in its event loop. If you are using TaskMaster instead, see "Using TaskMaster," later in this section.

Mouse events

Mouse-down and **mouse-up** events occur when the mouse button is pressed or released. Mouse movements cause the cursor position to be updated, but do not create events.

On receiving a mouse-down event, an application should first call the Window Manager to find out where the cursor was on the screen when the mouse button was pressed, and then respond in whatever way is appropriate. Depending on where the cursor was when the button was pressed, the application may have to call toolbox routines in the Menu Manager, the Desk Manager, the Window Manager, or the Control Manager.

If the application attaches special significance to the user pressing a modifier key or keys along with the mouse button, it can discover the state of the modifier keys by examining the appropriate flags in the modifiers field of the event record.

If you want your application to respond to mouse double-clicks, it must detect them itself. It can do so by comparing the time and location of a mouse-up event with the time and location of the mouse-down event immediately following the mouse-up event.

Mouse-up events can be significant in other ways; for example, they can signal that the user has stopped dragging the mouse after selecting a group of objects. Most applications, however, can ignore mouse-up events, and handle dragging with other calls such as TrackControl.

❖ *HodgePodge:* HodgePodge does not need to respond to mouse events directly. See "Using TaskMaster," later in this section.

❖ *Alternative pointing devices:* All applications that use the Event Manager work with alternative devices just as they do with the mouse. When a device such as a graphics tablet is being used, its X-Y location and button status appear in the event records in place of the mouse information. Mouse-up and mouse-down events are posted when the alternative device's buttons change state.

Keyboard events

Key-down events occur when character keys are pressed. Modifier keys (Shift, Caps Lock, Control, Option, and Apple) generate no keyboard events of their own—whenever an event is posted, the state of the modifier keys is reported in a field of the event record. The character keys also generate **auto-key events** when the user holds them down.

For a key-down event, the application should first check the modifiers field to see whether the character was typed with the Apple key held down; if so, the user may have been choosing a menu item by typing its keyboard equivalent.

If the key-down event is not a menu command, the application should respond to the event in whatever way is appropriate. For example, if one of the windows is active, the application could insert the typed character into the active document; if none of the windows is active, it might choose to ignore the event.

Most applications can handle auto-key events the same way they handle key-down events. However, you may want your application to ignore auto-key events that invoke commands you don't want continually repeated.

❖ *HodgePodge:* The only key events in HodgePodge are the keyboard equivalents to menu commands. TaskMaster handles those events and returns the menu-selection information to HodgePodge, so HodgePodge itself needn't respond to key events at all.

Window events

See "Creating Windows" in Chapter 4 for a discussion of window features.

To coordinate the display of windows on the screen, the Window Manager generates **activate events** and **update events.** Activate events occur whenever an inactive window becomes active or an active window becomes inactive. Update events occur when all or part of a window's contents need to be drawn or redrawn, usually as a result of the user's opening, closing, activating, or moving a window.

When the application receives an activate event for one of its own windows, the Window Manager will already have done all of the normal housekeeping associated with the event, such as highlighting or unhighlighting the window. The application can then take any further necessary action, like showing or hiding a scroll bar, or highlighting or unhighlighting a selection.

On receiving an update event for one of its own windows, the application is responsible for updating (redrawing) the contents of the window.

❖ *HodgePodge:* Activate and update events in HodgePodge are handled automatically through TaskMaster.

Other events

Device-driver events are generated by device drivers in certain situations; for example, an application might set up a driver to report an event when its transmission of data is interrupted.

A **desk accessory event** occurs whenever the user enters the special keystoke (Control–Apple–Escape) to invoke a classic desk accessory. See "Supporting Other Desktop Features" in Chapter 5.

An application can define as many as four **application-defined events** of its own and use them for any purpose.

Switch events are reserved for future use.

The Event Manager returns a **null event** if it has no other events to report. Most applications ignore null events and continue through the event loop.

Using TaskMaster

For a full discussion of TaskMaster, see "Window Manager" in the *Apple IIGS Toolbox Reference.*

TaskMaster is a routine that can handle many standard events. Technically, it is part of the Window Manager, and it handles window-related events such as drawing, scrolling, activating, and updating windows. It is discussed here because it replaces the GetNextEvent call for an application, and it also does preliminary event-handling for mouse-down and key-down events.

When your program calls TaskMaster instead of GetNextEvent, the following happens:

1. TaskMaster calls GetNextEvent.
2. If no event is ready to be handled, TaskMaster returns zero.
3. If an event is ready, TaskMaster looks at it and tries to handle it.
4. If Taskmaster can't handle the event, it returns the Event Manager event code to your application. The application can handle the event as if the event were coming from GetNextEvent.
5. If TaskMaster can handle the event, it calls standard toolbox functions to carry out the task. For example, if the user presses the mouse button in an active window's zoom region, TaskMaster detects it and calls TrackZoom; it then calls ZoomWindow if the user actually selects the zoom region; and finally it returns no event.

Event codes are listed in Table 3-2.

Window regions are discussed under "Creating Windows" in Chapter 4.

When calling TaskMaster, you pass a pointer to a TaskMaster record, the **extended task event record.** The beginning of the record is the same as an event record, as described under "Event Records and Masks," earlier in this section. When TaskMaster calls GetNextEvent, it passes the provided pointer, so the event record part of that record is set by GetNextEvent. The record also includes a **task mask,** similar to the event mask; it tells TaskMaster which types of events to handle.

Sometimes TaskMaster can handle an event only up to a point. If the user presses the mouse in the active window's content region, TaskMaster detects it, but won't be able to go any further, so it tells the application that a mouse-down event occurred in the active window's content region, and lets the application decide what to do next.

Because it only partially handles some events, TaskMaster generates its own set of "events" that a program's main event loop needs to respond to. Each type of TaskMaster event has a **task code,** a numeric value that TaskMaster returns to the application. Just as for the Event Manager events described earlier in this section, there is a set of predefined constants for these codes. Table 3-3 lists the codes and constants. The second half of `TaskTable` in the assembly-language version of HodgePodge's main event loop (in the file `EVENT.ASM`) is a code representation of Table 3-3; C and Pascal HodgePodge, on the other hand, use the predefined constants.

❖ *Note:* Many of these task codes are just the results returned by the call FindWindow, which TaskMaster makes after calling GetNextEvent.

Table 3-3
TaskMaster task codes

Value	Constant	Meaning
16	`wInDesk`	in the desktop area
17	`wInMenuBar`	in the system menu bar
18		(undefined)
19	`wInContent`	in a window's content region
20	`wInDrag`	in a window's drag (title bar) region
21	`wInGrow`	in a window's grow (size box) region
22	`wInGoAway`	in a window's go-away (close box) region
23	`wInZoom`	in a window's zoom (zoom box) region

24	`wInInfo`	in a window's information bar
25	`wInSpecial`	in the special menu item bar (predefined items in the Edit menu)
26	`wInDeskItem`	in a desk accessory menu item on the Apple menu
27	`wInFrame`	in a window, but not in any of the above parts of it
28	`wInactMenu`	in an inactive menu item
$8xxx	`wInSysWindow`	in a system (desk-accessory) window

Together, Table 3-2 and Table 3-3 show that TaskMaster *can* return to your application up to 25 or so events that your main event loop may have to deal with. In most situations, though, TaskMaster handles most of them automatically. HodgePodge, as we saw in Chapter 2, responds only to task codes 17, 22, and 25 (`wInMenuBar`, `wInGoAway`, and `wInSpecial`).

You should use TaskMaster for at least two reasons:

- It can help you get an application running as quickly as possible, still taking advantage of the standard user interface. TaskMaster represents one of the steps taken to remove the most tedious user-interface chores from the application.
- TaskMaster will help assure upward compatiblity. New, as yet unknown, features may be added to the Apple IIGS system in the future. It may be possible to incorporate those features by modifying TaskMaster without adversely affecting past applications. In other words, your application may be able to use new features without any modification on your part.

Drawing to the screen (and elsewhere)

Any time your desktop application needs to draw something, it uses the Apple IIGS tool set QuickDraw II (and its extension, QuickDraw II Auxiliary). QuickDraw II is an adaptation and extension of the Macintosh toolbox component *QuickDraw*—it performs similar operations but has been enhanced to support Apple IIGS color.

QuickDraw II allows you to perform graphic operations easily and quickly. QuickDraw draws text in different fonts with styling variations such as italics and boldface. It draws lines and shapes of various sizes and patterns. It can draw items in a variety of colors or in gray scales.

The Print Manager is described under "Communicating With Files and Devices," in Chapter 5.

QuickDraw II can draw to the screen or to other parts of Apple IIGS memory. In fact, printing a document with the Print Manager involves using QuickDraw to "draw" your document into a memory buffer used by the Print Manager.

❖ *Note:* For brevity, we'll use the terms *QuickDraw* and *QuickDraw II* synonymously here. Unless otherwise explicitly stated, *QuickDraw* means the Apple IIGS tool sets QuickDraw II and QuickDraw II Auxiliary, not the Macintosh version.

To get our bearings, we'll first consider *where* QuickDraw II draws. Then we'll briefy discuss *how* it draws, and finally look at *what* it draws. The chapter ends with two examples that tie together several of the key ideas.

Where QuickDraw II draws

The question of *where* QuickDraw II draws involves consideration of Apple IIGS memory (including screen memory) as well as QuickDraw's own internal representation of its drawing universe. These are the main concepts:

- Drawings are stored in Apple IIGS memory as *pixel images,* ordered collections of bytes that represent rectangular arrays of pixels. Screen memory contains a special pixel image—its contents are displayed on the computer's monitor.
- QuickDraw II draws its text and graphic objects on an abstract two-dimensional mathematical surface called the *coordinate plane.* Points on a plane are much easier to visualize and manipulate than addresses in memory. Locations on the QuickDraw II coordinate plane are related to pixel-image memory locations by specific *location information* supplied to QuickDraw.
- Quickdraw draws most objects within the context of *graphic ports.* A port is a complete drawing environment and defines, among other things, a specific part of memory and a specific rectangular area on the coordinate plane where drawing can occur. There can be many open ports at a time—some for drawing to the screen, some for drawing to other parts of memory. Different ports' drawing spaces may be separate from each other or they may overlap.

- QuickDraw II can be made to *clip*, or constrain its drawing, to within limits of arbitrary size, shape, and location.
- By manipulating two independent sets of coordinates (*global coordinates* and *local coordinates*), an application can easily control both what gets drawn inside a port's drawing space and where, on the screen or other pixel image, that drawing space appears.

The coordinate plane

QuickDraw locates every action it takes in terms of coordinates on a two-dimensional grid (Figure 3-2). The grid is QuickDraw's **coordinate plane;** coordinates on the plane are integers ranging from –16K to +16K in both the X- and Y-directions. The point (0,0), therefore, is in the middle of the grid. Note also that grid values increase to the right and *downward* on the plane; this is different from what you might be used to, but it is the same direction and order in which video scan lines are drawn.

Distances on the grid are measured in *pixels*. Thus a 10 x 10 "square" on the coordinate plane is equivalent to a rectangle 10 pixels by 10 pixels on the display screen (which would not be a square, of course, because Apple IIGS pixels are not square). Only a very small portion of the coordinate plane can be displayed on the screen at any one time—the plane is 32,000 pixels on a side, whereas the screen can show a maximum of 640 pixels by 200 pixels at a time. Figure 3-2 shows the approximate size of the screen (and user) compared to the coordinate plane.

Important QuickDraw must not be asked to draw outside the coordinate plane. Commands to draw outside this space will produce unpredictable results. They won't generate errors.

❖ *Macintosh programmers:* This conceptual drawing space is not the same size as that used by QuickDraw on the Macintosh. On the Macintosh, the drawing space is 64K by 64K pixels centered around 0,0, thus making the boundary coordinates -32K,-32k and 32K,32K.

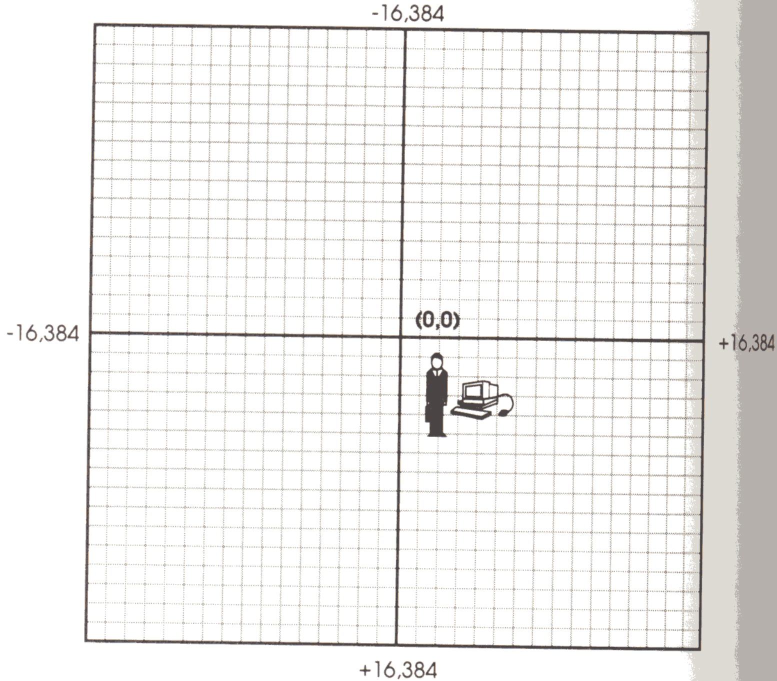


Figure 3-2
The QuickDraw II coordinate plane

To understand how QuickDraw does its drawing, we need to consider how it represents some basic graphic elements. On the coordinate plane, grid lines are considered to be infinitely thin. A point is defined as the intersection of two grid lines, so it also has no dimensions. Pixels, on the other hand, have a definite size; they are thought of as falling *between* the lines of the grid. The smallest element that QuickDraw can draw is a pixel, so if it were to draw a point at the location (3,3) on the coordinate plane, it must draw a single pixel. But which one? Four pixels touch the point. QuickDraw defines the pixel corresponding to each point on the plane as the pixel immediately *below and to the right* of the point. See Figure 3-3.

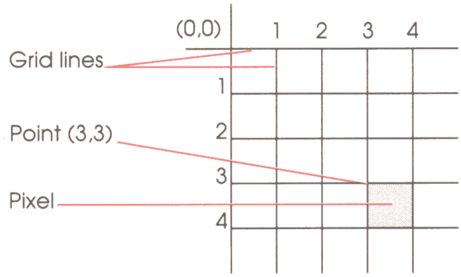


Figure 3-3
Grid lines, points, and pixels on the coordinate plane

Pixel images and the coordinate plane

A **pixel image** is an area of memory that contains a graphic image. The image is organized as a rectangular grid of pixels occupying contiguous memory locations. Each pixel has a value that determines what color in the graphic image is associated with that pixel.

❖ *Macintosh programmers:* QuickDraw II's pixel images are similar to Macintosh QuickDraw's *bit images*. The major difference is that a pixel is described by more than a single bit.

As described above, QuickDraw II draws to the coordinate plane. However, the coordinate plane is really just an abstract concept. Inside the Apple IIGS, drawing actually occurs by modifying pixel images—that is, by modifying the contents of certain memory locations. In particular, drawing something visible on the screen involves modifying the contents of screen memory.

The data structure that ties the coordinate plane to memory is the **LocInfo** (for *location information*) record. The LocInfo record tells QuickDraw where in memory to draw, how the pixel image in that part of memory is arranged, and what its position on the coordinate plane is. In Pascal, the LocInfo record definition looks like this:

```
LocInfo = Record
   portSCB       : Word
   ptrToPixImage : Ptr
   width         : Integer
   boundsRect    : Rect
end
```

The scan-line control byte and the differences between 640 mode and 320 mode are discussed further under "Drawing in Color," later in this section.

The record consists of four fields:

- **portSCB** (a replica of the *scan-line control byte*) tells QuickDraw how many bits per pixel there are in this image—two for 640 mode, four for 320 mode.
- **ptrToPixImage** (or *image pointer*) is the memory address of the image. It points to the first byte of the pixel image, which contains the first (upper-leftmost) pixel.
- **width** (or *image width*) specifies the width (in *bytes*, not pixels) of each line in the pixel image. QuickDraw needs to know this so it can tell where each new row in the image starts. (The image width must be an even multiple of 8 bytes.)
- **boundsRect** (for *boundary rectangle*) is a rectangle that maps the pixel image onto the coordinate plane. The upper-left point in the rectangle corresponds to the first pixel in the image. The lower-right corner of the rectangle describes the extent of the pixel image (as far as QuickDraw is concerned). See Figure 3-4.

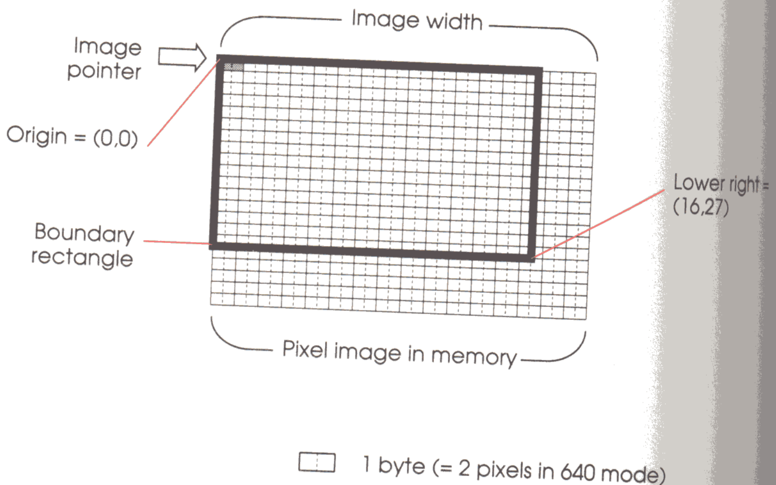


Figure 3-4
Pixel image and boundary rectangle

❖ *Note*: Remember, what separates one pixel image from another is where *in memory* it is stored, not where on the QuickDraw coordinate plane its boundary rectangle happens to be. You can think of each pixel image as having its own private copy of the entire coordinate plane to play with, so that even if two pixel images have overlapping coordinate plane locations, there won't be any conflict between them if they occupy completely different parts of computer memory.

GrafPort, port rectangle, and clipping

Windows are described further under "Creating Windows" in Chapter 4.

Most drawing takes place in conjunction with a data structure called a **GrafPort** (for **graphic port**). Each GrafPort contains a complete specification of a drawing environment, including the location information (LocInfo record) described above. In addition to the location information, a GrafPort contains three other fields that restrict where drawing in a pixel image can take place: the *port rectangle, clipping region,* and *visible region.*

The **port rectangle** (or portRect) is a rectangle on the coordinate plane. Any drawing in a GrafPort occurs only inside its portRect. When you look at a *window* on the screen in a desktop application, its interior (everything but its frame) corresponds to a port rectangle.

The port rectangle can coincide with the boundary rectangle or it can be different. You can think of it as a movable opening, allowing access to all or part of the pixel image. As Figure 3-5 shows, QuickDraw can draw only where the boundary rectangle and port rectangle overlap.

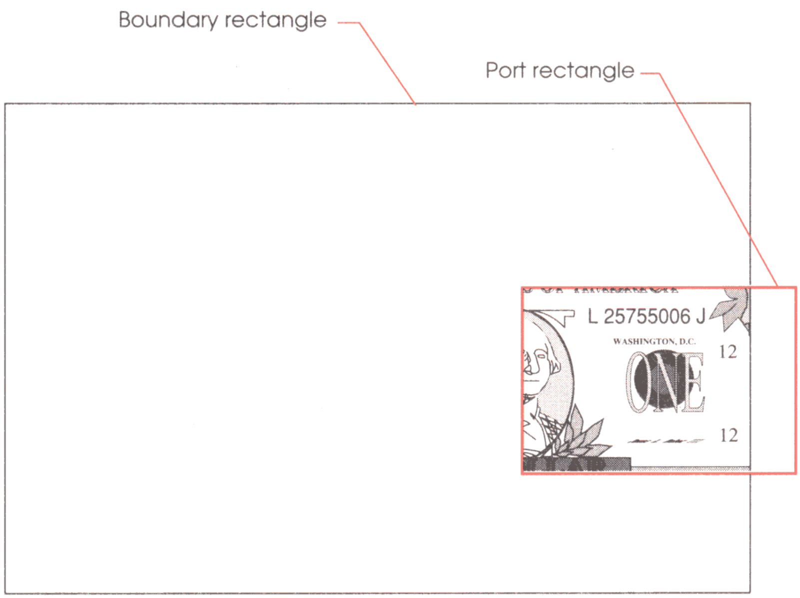


Figure 3-5
Boundary rectangle/port rectangle intersection

The **clipping region** (or clipRgn) is provided for an application to use. When a GrafPort is opened or initialized, the clipping region is set to the entire coordinate plane (effectively preventing any clipping from occuring). The program can use the clipRgn in any way it wants. Any drawing to a pixel image through a GrafPort occurs only inside the clipping region.

The **visible region** (or visRgn) is normally maintained by the Window Manager. An application can have multiple windows on the screen, each one associated with a GrafPort. Windows can overlap, and each port's visible region represents the parts of the window that are visible.

In summary, drawing occurs in a pixel image only in the *intersection* of the boundary rectangle, port rectangle, clipping region, and visible region.

Global and local coordinate systems

Everything is positioned in QuickDraw's universe in terms of coordinates on the plane. However, if you think of multiple open windows on the screen, you can see that there are at least two different ways in which you might want to locate objects:

- □ You may want to specify where windows appear on the screen (for example, when they are moved).
- □ You may want to specify where objects appear within windows (for example, when scrolling), independently of where on the screen the windows may be.

❖ *HodgePodge:* Because TaskMaster takes care of all window events related to tasks such as moving and scrolling, HodgePodge itself doesn't worry about coordinates at all when it draws a window.

The **origin** of a rectangle, in QuickDraw II, is its upper-left corner.

The toolbox needs **global coordinates** whenever more than one GrafPort share the same pixel map; the global coordinates tell QuickDraw exactly where every port rectangle is compared to every other one. The global coordinate system for each GrafPort is that in which the boundary rectangle for its pixel map has its **origin** at (0,0) on the coordinate plane. For drawing to the screen, you can think of global coordinates as *screen coordinates,* where the upper-left corner of the screen is the point (0,0).

However, each port also has its own **local coordinate** system. For example, when drawing into a port it might be more convenient to think in terms of distance from the port rectangle's origin rather than the boundary rectangle's origin. By defining the port rectangle as starting at (0,0), you can base all your drawing commands on distance in from the left edge and down from the top of the portRect.

That's convenient for drawing in a window, but local coordinates are more of a convenience than that. They aren't constrained to a value of (0,0) for the port rectangle origin—you can set them to any coordinate-plane value. Why would you want to? Because of the way drawing commands work.

Suppose you are using a window to display portions of a document that is larger than the port rectangle in size—a fairly common occurrence. You are using drawing commands that draw the entire document, and you know that's no problem because the drawing will be automatically clipped to the port rectangle. But how do you control *which part* of the document shows in your window? You do it by adjusting local coordinates.

All QuickDraw's drawing commands are based on the current port's *local* coordinate system. So if location (0,0) in your GrafPort's local coordinates corresponds to the port rectangle's upper-left corner, any time you draw your document into that port, its upper-left corner will be displayed. If you define your local coordinates differently, different parts of your document will appear in the window. Thus you can think of local coordinates as *document coordinates*—the upper-left corner of the document being viewed in the port has the value (0,0) in local coordinates. See Figure 3-6.

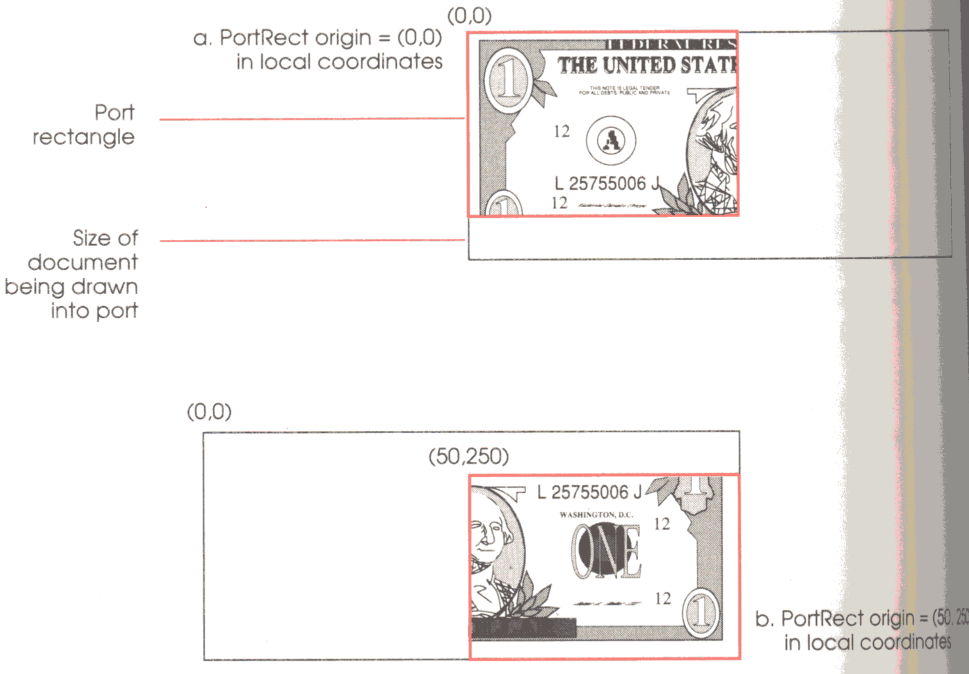


Figure 3-6
Drawing different parts of a document by changing local coordinates

❖ *Note:* When the local coordinates of a GrafPort are changed, the coordinates of the GrafPort's *boundary rectangle* and *visible region* are similarly recalculated, so (as noted) the port will not change its relative position on the screen or in relation to other open ports on the screen.

Pen location and other pen characteristcs are described next, under "How QuickDraw II Draws."

However, when the local coordinates are changed the GrafPort's *clipping region* and *pen location* are not changed—that is, they appear to shift right along with the image that is being viewed *in* the port. It makes sense to have the pen, which is used to modify the image being viewed, and the clipping region, which is used to mask off parts of the image being viewed, "stick" to it.

How QuickDraw II draws

How QuickDraw II draws any of its objects depends on the drawing environment specified in the current GrafPort. Each GrafPort record includes location and clipping information (described above), information about the graphics pen (described next), information about any text that will be drawn (described under "...And Text Too," later in this section), and other information such as pen patterns to draw with.

The drawing pen

Each open port has its own drawing **pen**. By means of several characteristics modifiable by the application, the pen controls where and how drawing (of both text and graphics) occurs.

Pen location: The pen has a coordinate-plane location (in local coordinates). The pen location is used for drawing lines and text only—other shapes are drawn independently of pen location.

Pen size: The pen is a rectangle that can have almost any width or height. Its default size is 1 x 1 (pixels). If either the width or height is set to 0, the pen will not draw.

Pen pattern: The pen pattern is a repeating array (8 pixels by 8 pixels) that is used like ink in the pen. Wherever the pen draws, the pen pattern is drawn in the image. The pattern is always aligned with the coordinate plane so that adjacent areas of the same pattern drawn at different times will blend in a continuous manner.

Background pattern: The background pattern is an array similar to the pen pattern. **Erasing** is the process of drawing with the background pattern.

Drawing mask: The drawing mask is an 8-bit by 8-bit pattern that is used to mask, or screen off, parts of the pattern as it is drawn. Only those pixels in the pattern aligned with an *on* (=1) bit in the mask are drawn. Figure 3-7 shows how a mask affects drawing with a pattern.

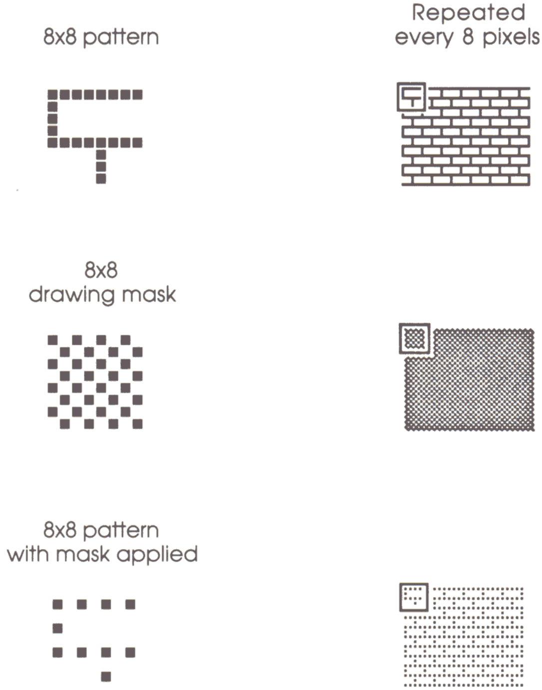


Figure 3-7
Drawing with pattern and mask

Note that drawing with a mask in which every bit has the value 1 is like drawing with no mask at all—all pen pixels are passed through to the image. Likewise, drawing with a mask that is all zeros is like not drawing at all—all pen pixels are blocked.

All eight pen modes (also called *transfer modes*) are described and diagrammed under "QuickDraw II" in the *Apple IIGS Toolbox Reference.*

Pen mode: The pen mode specifies one of eight Boolean operations (COPY, notCOPY, OR, notOR, XOR, notXOR, BIC and notBIC) that determine how the pen pattern is to affect an existing image. When the pen draws, QuickDraw II compares pixels in the existing image with their corresponding pixels in the pattern, and then uses the pen mode to determine the value of the resulting pixels. For example, with a pen mode of COPY, the existing pixels' values are ignored—a solid black line is black regardless of the image already on the plane. With a pen mode of notXOR, the bits in each pen pixel are inverted and then combined in an exclusive-OR operation with the bits in each corresponding existing pixel. Figure 3-8 shows a rectangle drawn over an existing circle, in both COPY and notXOR mode.

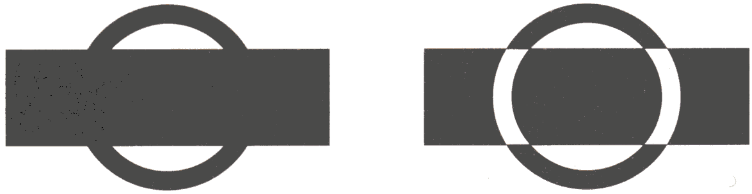


Figure 3-8
How pen mode affects drawing

Basic drawing functions

QuickDraw draws *lines* with the current pen size, pen pattern, drawing mask, and pen mode.

QuickDraw's shapes are described next, under "What QuickDraw II Draws."

QuickDraw draws other shapes (*rectangles, rounded-corner rectangles, ovals, arcs, polygons,* and *regions*) in five different ways:

- **Frame:** QuickDraw draws an outline of the shape, using the current pen size, pen pattern, drawing mask, and pen mode.
- **Paint:** QuickDraw fills the shape, using the current pen pattern, drawing mask, and pen mode.
- **Erase:** QuickDraw fills the shape, using the current background pattern and drawing mask.
- **Invert:** QuickDraw inverts the pixels in the shape, using the drawing mask.
- **Fill:** QuickDraw fills the shape, with a specified pattern and using the drawing mask.

QuickDraw draws *text* as described under "...And Text Too," later in this section.

What QuickDraw II draws

QuickDraw II can draw a number of graphic objects into a pixel image. It draws text characters in a variety of monospaced and proportional fonts, with styling variations that include italics, boldfacing, underlining, outlining, and shadowing. It draws straight lines of any length, width, and pattern. It draws hollow or pattern-filled rectangles, circles, and polygons. It draws elliptical arcs and filled wedges, irregular shapes and collections of shapes. It also draws *pictures*—combinations of these simple shapes. Figure 3-9 summarizes them.

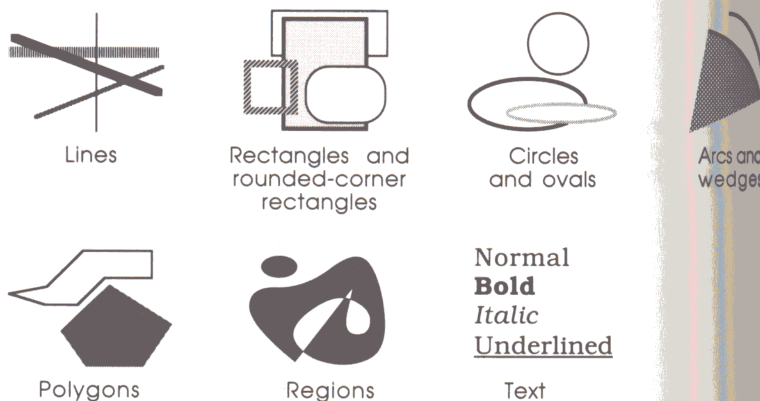


Figure 3-9
What QuickDraw II draws

Points and lines

A **point** is represented mathematically by its Y- and X-coordinates—two integers. A **line** is represented by its ends—two points, or four integers. Like a point, a line is infinitely thin. When drawing a line, QuickDraw II moves the upper-left corner of the pen along the straight-line trajectory from the current pen location to the destination location. The pen hangs below and to the right of the trajectory, as illustrated in Figure 3-10.

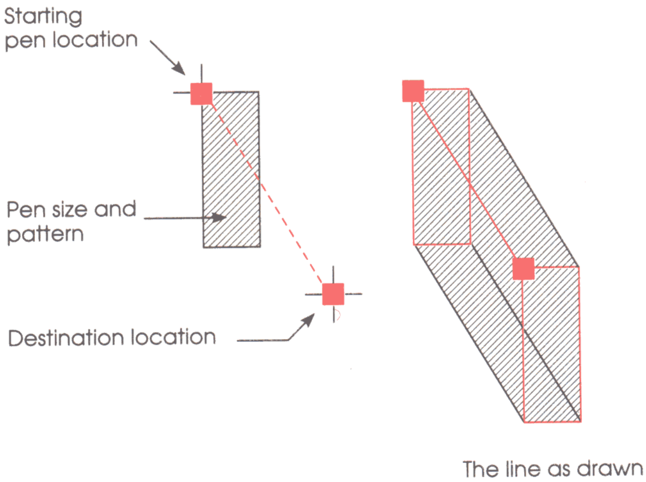


Figure 3-10
Drawing lines

Before drawing a line, you can use QuickDraw calls to set the current pen location and other characteristics such as pen size, mode, and pattern.

Important QuickDraw's data structure that defines a point has the vertical coordinate first: (y,x) rather than (x,y).

Rectangles

A **rectangle** (Figure 3-11) is also represented by two points: its upper-left and lower-right corners. The borders of a rectangle are infinitely thin. Rectangles are fundamental to QuickDraw; there are many functions for moving, sizing, and otherwise manipulating rectangles.

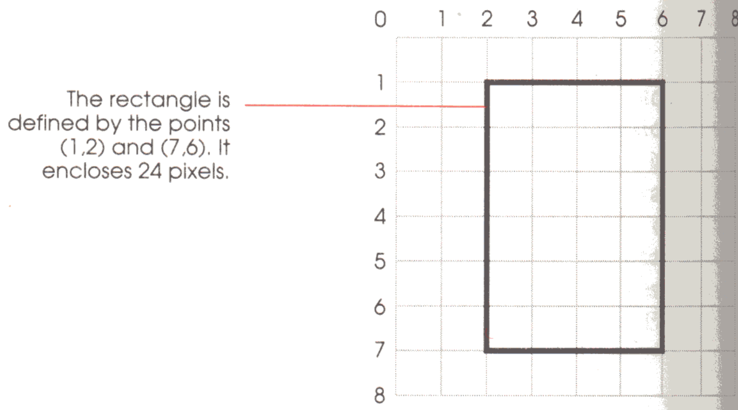
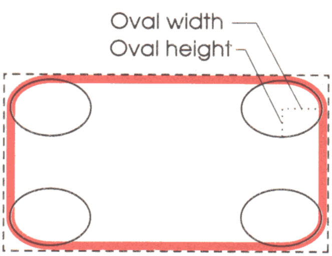


Figure 3-11
A rectangle

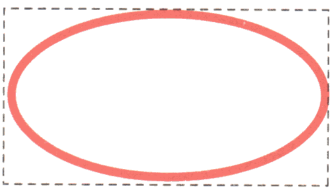


The pixels associated with a rectangle are only those *within* the rectangle's bounding lines. Thus the pixels immediately below and to the right of the bottom and right-hand lines of the rectangle are not part of it.

Rectangles may have square or rounded corners. The corners of rounded-corner rectangles are sections of *ovals* (described next); they are specified by an *oval height* and *oval width*.

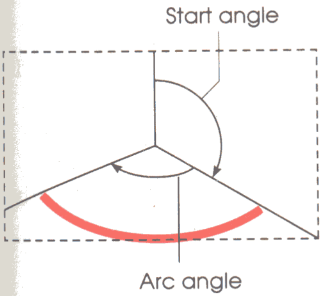
Important The QuickDraw data structure that defines a rectangle has coordinates in the following order: top, left, bottom, right. Thus the defining coordinates for the rectangle in Figure 3-11 are (1,2,7,6). This may seem strange, but it is consistent with the (y,x) ordering of points.

Circles, ovals, arcs, and wedges



Ellipses and portions of ellipses form another class of shapes drawn by QuickDraw II. An **oval** is an ellipse, and it is defined just like a rectangle—the only difference is that QuickDraw is told to draw the ellipse inscribed within the rectangle rather than the rectangle itself. If the enclosing rectangle is a square, the resulting oval is a circle.

❖ *Pixel shape:* Remember, Apple IIGS pixels are not square. A true circle on the screen, or a true square, will have unequal horizontal and vertical dimensions in terms of pixels.

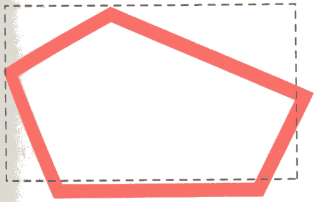


An **arc** is a portion of an oval, defined by the oval's enclosing rectangle and by two angles (the starting angle and the arc angle), measured clockwise from vertical.

If an arc is painted, filled, inverted, or erased, it becomes a **wedge;** its fill pattern extends to the center of the enclosing rectangle, within the area defined by the lines bounding the arc angle.

Polygons

A **polygon** is any sequence of connected lines. You define a polygon by moving to the starting point of the polygon and drawing lines from there to the next point, from that point to the next, and so on.



Polygons are not treated in exactly the same manner as other closed shapes such as rectangles. For example, when QuickDraw II draws (*frames*) a polygon, it draws outside the actual boundary of the polygon, because the line-drawing routines draw below and to the right of the pen locations. When it paints, fills, inverts, or erases a polygon, however, the fill pattern stays within the boundary of the polygon. If the polygon's ending point isn't the same as its starting point, QuickDraw adds a line between them to complete the shape.

Regions

A region is another fundamental element of QuickDraw, one that can be considerably more complex than a line or a rectangle. A **region** can be thought of as a collection of shapes or lines (or other regions), whose outline is one or more closed loops. Your application can draw, erase, move, or manipulate regions just like any other QuickDraw structures.

You can define regions by drawing lines, framing shapes, manipulating existing regions, and equating regions to rectangles or other regions.

Regions are particularly important to the Window Manager, which must keep track of often irregularly shaped, noncontiguous portions of windows in order to know when to activate the windows or what parts of them to update.

Pictures

Pictures are used for transferring data between applications, via the Clipboard. See "Scrap Manager" in the *Apple IIGS Toolbox Reference.*

A **picture** is a collection of any QuickDraw drawing commands. Its data structure consists of little more than the stored commands. QuickDraw plays the commands back when the picture is reconstructed with a DrawPicture call. A complex mechanical drawing produced from an Apple IIGS drafting program might be saved as a single QuickDraw II picture.

...And text too

QuickDraw II doesn't draw graphic images only—it also does all text drawing for desktop applications. As an application programmer, you can easily control the placement, size, style, font, and color of display text with QuickDraw calls.

Your program can provide QuickDraw II with text in a number of formats:

- **character:** a single ASCII character at a time
- **Pascal string:** a length byte followed by a sequence of ASCII characters
- **C string:** a sequence of ASCII characters terminated by a zero byte
- **text block:** an arbitrary number of ASCII characters in a buffer

Text mode is similar to *pen mode,* discussed earlier in this section.

However it receives the text, QuickDraw II draws it in the same way. It draws each character at the current *pen location,* with the current *font,* using the current *text mode,* with the current character *style,* and using the current *foreground* and *background* colors. After drawing each character, QuickDraw updates the pen location for drawing the next one.

Providing QuickDraw with various fonts and character styles is the job of the Font Manager. The Font Manager is a tool set that supports QuickDraw's character-drawing ability by providing an application with different fonts and styled variations of fonts. If you want to allow the user to choose from all of the fonts available when the application is run, or if you're developing an application that requires a specific font, the Font Manager can help you.

Characters

To help understand just where text appears and how much space it takes up, let's define a few terms. Refer to Figure 3-12.

Text fonts are made up of individual **characters.** A character is represented in memory as a rectangular array of bits, called a **character image,** representing rows and columns of pixels. The *on* (=1) bits are the **foreground pixels;** the *off* (=0) bits are the **background pixels.**

Every character in a font has a base line. The **base line** is a horizontal line, in the same position for every character in the font. Any foreground pixels of a character image that lie below the base line constitute the character's *descender* (characters like *p* and *q* have descenders). The *ascent line* is the horizontal line just above the top row of a character (including any blanks); the distance from the base line to the ascent line is the font's **ascent,** and is equal to the height of the tallest character in the font. The *descent line* is the line just below the bottom row of the character (including any blanks); the distance from the base line to the descent line is the font's **descent,** and is equal in size to the largest descender in the font.

Each character's **origin** is a point on the base line that is used to position the character for drawing. This point need not touch any foreground pixels of the character image. When the character is drawn, it is placed in the destination location so that its character origin *coincides with the current pen location.* For many letters, the character origin is located on the left edge of the character image; then, when the character is drawn, its leftmost foreground pixels fall just to the right of the pen location.

The **font height** is the sum of the ascent and descent heights, and it is the same for all characters in a font. The **character width** is the number of pixels the pen position is to be advanced after the character is drawn. It includes the width of the character itself and any needed space between it and the next character to be drawn.

Font height, ascent, descent, character width, and **leading** (the space between lines of text) are needed for calculating string lengths and line spacings when you display text on the screen.

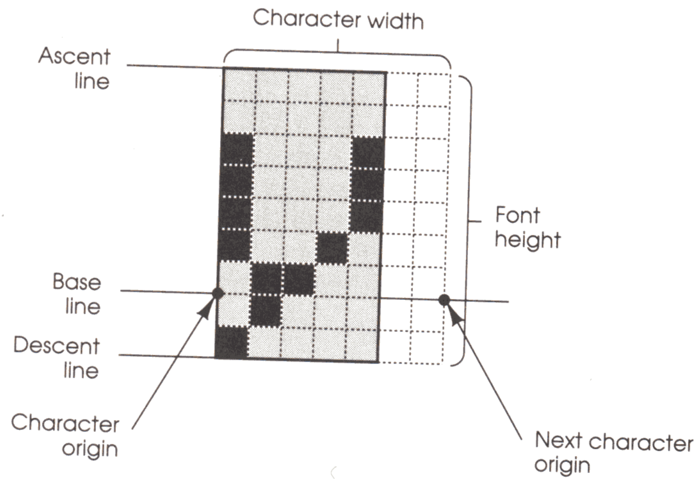


Figure 3-12
A character image

The basic commands necessary to draw characters on the screen are quite simple. Recall from Chapter 2 how HodgePodge puts up the "One moment please..." message when the program loads tools:

```
MoveTo(20,20);                        {move pen to upper left of screen}
SetBackColor(0)                       {background color = black}
SetForeColor(15);                     {foreground color = white}
DrawString('One Moment Please...');   {write the message}
```

Once the foreground and background colors are set, all that's needed to display a character string is to move the pen to the desired location, and call the QuickDraw routine DrawString.

Fonts

Each collection of related characters is called a **font.** With the font manipulation capabilities of the Font Manager, your Apple IIGS applications can show sophisticated text display in a variety of fonts, sizes, and styles.

The font strike: All the character images making up a font are stored in memory as a **font strike.** A font strike is a long, rectangular array of bits consisting of the character images of every defined character in the font, placed sequentially in order of increasing ASCII code. The character images in the font strike abut each other; no blank columns are left between them.

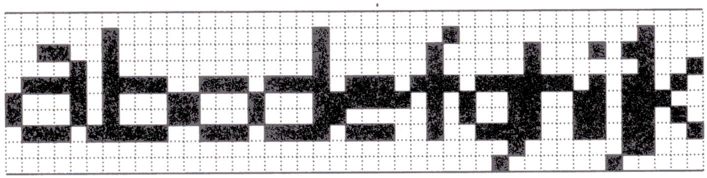


Figure 3-13
Part of a font strike

A given font strike need not contain a character image for every possible ASCII code. The font may leave some characters undefined; these are called *missing characters*. Immediately following the last defined character in the font strike is a character known as the **missing symbol,** which is to be used in place of any missing character. In many fonts the missing symbol is a hollow rectangle; in the Apple IIGS system font, it's a white-on-black question mark. Whenever the QuickDraw II text-handling routines encounter a missing character, they substitute the missing symbol for the character.

Choosing a font: Fonts for the Apple IIGS are grouped into **font families.** Individual fonts within families can have various characteristics, as noted in the following list. When your application requests a font, the Font Manager searches all available fonts and chooses the one that most closely matches the request, in these categories:

The family name of the Apple IIGS system font (in ROM) is *Shaston.*

- **Name:** Every font family has a name. The name refers to both **plain-styled** characters of all sizes, and any **styled variations,** such as bold or italics.
- **Number:** Every font family has a number, also independent of point size or style modifications. Every family number is unique, and corresponds to a single family name. $0000 represents the system font.

  Whenever an application requests a font whose family number is not available, the Font Manager substitutes the system font.
- **Size:** An individual font has a size, described in points. A **point** is a typesetting measure equal to about 1/72nd of an inch.

The Font Manager can provide both real and scaled fonts. A real font is one that actually exists on disk at a particular point size. Conversely, a **scaled font** is one that was enlarged or reduced by calculation from a font of a different size. The Font Manager may scale a font from an existing size if the requested size is not available. Real fonts generally have a better screen appearance than scaled fonts.

- **Style:** An individual font also has a style (or combination of styles). The presently defined styles are
  - Plain
  - **Bold**
  - *Italic*
  - Underline
  - Outline
  - Shadow

  There are two different ways to obtain styled variations of fonts. First, the Font Manager will provide a styled font if one is available—one whose characters are designed with (for example) bold or italic styling. Second, QuickDraw II can *style* a font—that is, it can produce a bold or italicized version of a plain-styled font. In fact, it can produce any combination of the defined styles.

❖ *Note:* Fonts that are already styled will not be further styled (in the same manner) by QuickDraw II, regardless of the text styling selected. For example, an italic font is not further italicized if that option is selected on a style menu. However, it could be underlined.

❖ *Underlining:* Text cannot be underlined unless the font's characters have a descent value (distance between the base line and descent line) of at least 2 pixels. The Apple IIGS system font (Shaston 8) has a descent value of 1, and therfore cannot be underlined.

Important The Font Manager looks for fonts in the subdirectory called FONTS/ in the SYSTEM/ subdirectory on the system disk. This subdirectory must contain all fonts (except the system font) that are to be available to applications. See Appendix C.

DoChooseFont is in the source file FONT.PAS.

Your application can allow the user to select a font by calling the Font Manager routine ChooseFont. That's what HodgePodge does in its `DoChooseFont` subroutine, called from the routine `DoMenu`:

```
function DoChooseFont: Boolean;                    {begin DoChooseFont…}

var    theFont    : FontID;
       dummy      : Integer;
       tmpPort    : GrafPortPtr;
       tmpPortRec: GrafPort;
       famName    : Str255;

begin

  tmpPort := GetPort;                              {Save current port…}
  OpenPort(@tmpPortRec);                           {…and open new one, so that
                                                    the current port is not affected}
  theFont := ChooseFont(desiredFont,0);            {Bring up a dialog prompting
                                                    the user to select a font}
  if LongInt(theFont) = 0 then                     {If the user cancels…}
    DoChooseFont := FALSE                          {DoChooseFont unsuccessful}
  else
    begin
     desiredFont := theFont;                       {Update global variable DesiredFont}
     dummy := GetFamInfo(dDesiredFont.famNum,
                          famName);                {Get the font name from its number…}
     myReply.filename :=
             concat(famName,
             ' ',                                  {…and put the font name and size in…}
             IntToString
                  (desiredFont.fontsize));         {…global variable myReply.filename}
     DoChooseFont := TRUE;                         {DoChooseFont completed successfully}
    end;                                           {end of IF user doesn't cancel}

   ClosePort(@tmpPortRec);                         {Close the temporary port…}
   SetPort(tmpPort);                               {…and restore the current port}
end;                                               {End of DoChooseFont}
```

ShowFont is listed under "Displaying Documents in Ports: Two Examples," later in this section.

The `ShowFont` subroutine in HodgePodge is an example of how to draw text strings in a specific font with QuickDraw. It is called when a font window is first opened and also whenever it needs to be redrawn.

Drawing in color

The video display hardware of the Apple IIGS includes advanced color capabilities. Although tool calls make it unneccessary for you to manipulate the hardware directly, knowledge of a few background concepts will help you understand the way QuickDraw II manipulates the colors on the screen.

The Apple IIGS offers two Super Hi-Res graphics modes. Both modes have 200 scan lines, but the scan lines differ in horizontal resolution—one mode has 320 pixels (the color of each specified by 4 bits), and the other has 640 pixels (the color of each specified by 2 bits). In changing from 320 mode to 640 mode, the horizontal resolution is doubled at the expense of dividing the color resolution by four.

Both modes use a **chunky pixel** organization (in which the bits for a given pixel are contained in adjacent bits within one byte), as opposed to **bit planes** (in which adjacent bits in memory affect adjacent pixels on the screen). Therefore the 4 bits of a pixel in 320 mode are in the same memory locations as the 4 bits of a pair of adjacent 2-bit pixels in 640 mode.

Colors on the Apple IIGS are determined from **master color values**, which are mathematical combinations of the primary red, blue, and green hues available on a color monitor. A master color value is a 2-byte number. The low-order nibble of the low-order byte, controls the intensity of the color blue. The high-order nibble of the low-order byte, controls the intensity of the color green. The low-order nibble of the high-order byte, controls the intensity of the color red. The high-order nibble of the high-order byte is not used. Figure 3-14 illustrates the format of a master color value.

	Byte 1								Byte 0							
Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value:	(not used)				red				green				blue			

Figure 3-14
Master color value format

A 3-digit hexadecimal number can describe each master color, with one digit ($0–$F) for each primary color. Thus a master color value of $000 denotes black, $FFF is white, $00F is the brightest possible blue, $080 is a medium-dark green, and so on. Because each primary color has 16 possible values, a total of 4096 colors are possible.

At any one time, the Apple IIGS can display only a small subset of all possible colors. An application specifies its colors by constructing one or more *color tables,* short lists of the available colors for any one pixel.

Color tables and palettes

Applications cannot specify pixel colors directly by using master color values. Pixels contain only 2 or 4 bits, and it takes 12 bits to specify a master color value. That's why color tables are necessary. A **color table** is a table of 16 2-byte entries. Each entry in the table is a master color value; any of the 4096 possible color values may appear in any position in the color table.

An application determines the color of a given pixel by specifying an *offset* into the color table. The number of bits used to describe a pixel limits how far into the table it can reach. The colors available to the application, as specified in its color tables, constitute its **palette.** See Figure 3-15.

Pixels in 320 mode are represented in memory by 4-bit integers. For each pixel, that 4-bit value is used as an offset in a color table. With 4 bits, there are 16 possible pixel values, so the palette in 320 mode is 16 colors—the entire color table.

Pixels in 640 mode are represented in memory by 2-bit integers. With 2 bits, there are 4 possible pixel values to offset into the color table, so the palette in 640 mode consists of only 4 colors. That would seem to leave three-quarters of the color table unused in 640 mode, and severely restrict the use of color, but it's not really so.

In the first place, each 4 adjacent pixels in 640 mode use 4 different parts of the *same* color table; a color table, then, consists of four *mini-palettes,* which needn't have the same sets of master colors. Therefore, although each individual pixel in 640 mode can have one of only four colors, groups of four pixels can have a total of 16 colors from which to choose. How to use this ability to create a large variety of colors is described under "Dithered Colors in 640 Mode," later in this section.

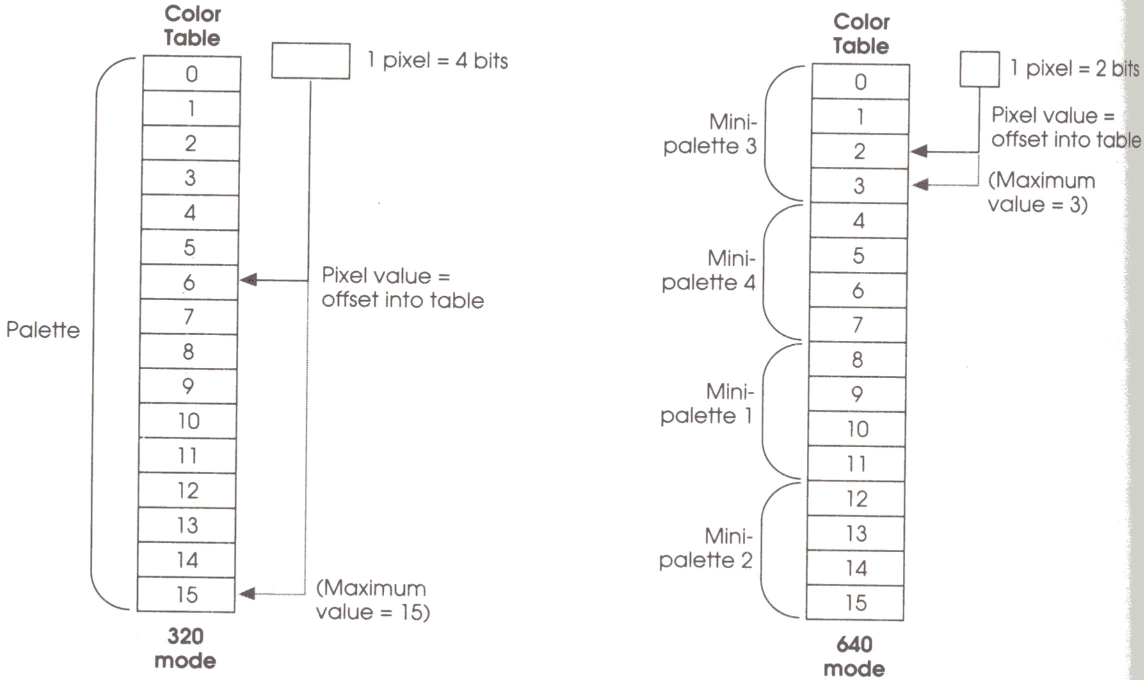


Figure 3-15
Accessing the color table in 320- and 640 mode

The **scan line control byte** is discussed under "The Video Displays" in the *Apple IIGS Hardware Reference* and under "QuickDraw II" in the *Apple IIGS Toolbox Reference.*

An application may construct as many as 16 different color tables to choose from. Each of the 200 scan lines in Super Hi-Res graphics can use any one of the 16 tables. For each scan line, a **scan line control byte (SCB)** decides which color table is active. The SCB also controls screen display mode (320 or 640), interrupt mode (whether or not to generate an interrupt during **horizontal blanking**), and **fill mode** (whether or not pixel values of zero can be used to fill areas of color in 320 mode).

Standard color palette (320 mode)

The standard palette (the default color table) for 320 mode is shown in Table 3-4. In the table, *offset* means positon in the color table, and *value* means master color value, the hexadecimal value controlling the fundamental red-green-blue intensities.

Table 3-4
Standard palette—320 mode

Offset	Color	Value
0	Black	0 0 0
1	Dark Gray	7 7 7
2	Brown	8 4 1
3	Purple	7 2 C
4	Blue	0 0 F
5	Dark Green	0 8 0
6	Orange	F 7 0
7	Red	D 0 0
8	Beige	F A 9
9	Yellow	F F 0
10	Green	0 E 0
11	Light Blue	4 D F
12	Lilac	D A F
13	Periwinkle Blue	7 8 F
14	Light Gray	C C C
15	White	F F F

The standard palette was selected because of its flexibility and appearance; we recommend that you use it unless you have a specific need to change it.

Dithered colors in 640 mode

As explained above, only four colors are available for each pixel in 640 mode. But when small pixels of different colors are next to each other on the screen, their colors blend. For example, a black pixel next to a white pixel appears to the eye as a larger gray pixel. By cleverly choosing the entries in the color table we can make more colors appear on the screen. This process is called **dithering.**

At the same time, in order to preserve the maximum resolution for displaying text, both black and white must be available for each pixel. This leaves only two remaining colors per pixel to choose from, which seems like a severe restriction. But with dithering, you can have 640-mode resolution for text and still display 16 or more colors, if you are willing to resort to a few simple tricks.

Consider the following byte with four pixels in it:

Bit value	0	1	0	1	0	1	0	1
Pixel number	1		2		3		4	

Each pixel has the value 1, which is an index into the *second* place in each of the color table's minipalettes (as shown in Figure 3-15). So pixel 1's color is determined by entry 1 in minipalette 1, pixel 2's color is determined by entry 1 in minipalette 2, and so on. If we use the standard 640-mode color table (shown in Table 3-5) then pixels 1 and 3 will appear blue ($00F), and pixels 2 and 4 will appear red ($D00). The eye will average these colors and see violet.

There are 16 different combinations of values that a pair of pixels can assume in 640 mode, meaning that you can obtain 16 colors by this dithering method. To implement it, just make sure that the pattern you use for drawing or filling consists of a repeating array of 4-bit (= 2-pixel) values.

Table 3-5
Standard palette—640 mode

Offset	Color	Value	(minipalette offset)
0	Black	0 0 0	0
1	Blue	0 0 F	1
2	Yellow	F F 0	2
3	White	F F F	3
4	Black	0 0 0	0
5	Red	D 0 0	1
6	Green	0 E 0	2
7	White	F F F	3
8	Black	0 0 0	0
9	Blue	0 0 F	1
10	Yellow	F F 0	2
11	White	F F F	3
12	Black	0 0 0	0
13	Red	D 0 0	1
14	Green	0 E 0	2
15	White	F F F	3

❖ *Black and white:* Note that the entries in the minipalettes for the standard 640-mode color table are set up so that black and white appear in the same positions in each palette. This arrangement provides pure black and white at full 640 resolution, allowing crisper text display.

Displaying documents in ports: two examples

Commonly you may want to have an application open up a port and display, within its port rectangle, a portion of a *previously created* drawing or text document. You might even want to allow the user to scroll around within that document, showing different parts of it in the port.

This is not as complicated as it sounds. We'll just give you a brief idea here of two simple ways to approach it—two of the methods used in HodgePodge. Please consult *the Apple IIGS Toolbox Reference* for more details. See also "Creating Windows" in Chapter 4 for more information on scrolling.

❖ *Note:* Ultimately, you are likely to want to let the user *alter* a part of a document while viewing it in a port, and be sure that the changes made are reflected in updates to the document itself. HodgePodge does not have that capability; you may want to add it as a programming exercise.

Pixel images

The keys to displaying a portion of a pixel image in a GrafPort are the QuickDraw (and QuickDraw Auxiliary) routines that copy pixels from one region to another. One is CopyPixels; the one we use here is PPToPort (for *Paint-Pixels-to-Port*). PPToPort transfers pixels from a given source pixel image to the current GrafPort's pixel image (the destination pixel image). To use this method to view a document you might try the following:

1. Define a LocInfo record that describes the offscreen pixel image you wish to display.
2. Open an onscreen GrafPort; its boundary rectangle is the screen image boundary rectangle. Make its port rectangle any size you wish, up to full screen size. Now anything you draw into that port will be visible on screen.
3. Set your port's *local coordinates,* to control which portion of the image you want to display first. To show the upper-left corner of the image, set the port rectangle's origin to (0,0).

4. Call PPtoPort to copy the offscreen image onto the onscreen rectangle. The call asks QuickDraw to draw the *entire* image, but because drawing is automatically clipped to the port rectangle, you only get the part you want.

PaintIt is in the source file PAINT.PAS.

HodgePodge uses PPtoPort to draw the contents of its picture windows. Here is the routine that does the drawing (`PaintIt`, called by the routine `Paint`):

```
procedure PaintIt (pict: Handle);                  {begin PaintIt...}

var    srcLoc : LocInfo;                            {a LocInfo record}
       srcRect: Rect;                               {a rectangle}

begin
   HLock(pict);                                     {Lock the image's memory block}

   with srcLoc do                                   {Define the LocInfo fields:}
     begin
      portSCB       := $0080;                       {set 640 mode}
      ptrToPixImage := pict^;                       {pointer to the image}
      width         := 160;                         {row-width of image in bytes}
      SetRect(boundsRect,0,0,640,200);              {boundary-rectangle coordinates}
     end;

   SetRect(srcRect,0,0,640,200);                    {rectangle to copy FROM}

   PPToPort(srcLoc,                                 {Copy pixels from this LocInfo...}
            srcRect,                                {...and this source rectangle...}
            0,                                      {...to location (0,0) in the current...}
            0,                                      {...GrafPort's local coordinates...}
            srcCopy);                               {...with a pen mode of COPY}

   HUnLock(pict);                                   {Unlock the image now that we're done}
end;                                                {End of PaintIt}
```

Text documents

Many documents, such as text files, have no explicit pixel image in memory that represents the contents of the file. If you want your application to permit displaying or scrolling through such a document in a port, you wouldn't transfer pixels from one image to another—you would draw directly into the port you opened.

The pixel-based image of a document as seen in a window is commonly called the *data area*. See "Creating Windows" in Chapter 4.

Nevertheless, the concept of local coordinates is still important and is used identically. Instead of using an actual pixel image somewhere in memory, you need to calculate a "virtual image" of the document. You need to know its exact size, *in pixels*, and be able to place it properly in relation to the port rectangle so that the part you want displayed is within the rectangle. Then you can set local coordinates accordingly and draw the document to the port, knowing that it will be clipped appropriately.

❖ *Note:* Pascal HodgePodge calculates document height when it creates a text window, but it simply assumes a particular width. Assembly-language and C versions of HodgePodge, on the other hand, calculate document width also, in the routine `FindMaxWidth`. See Appendixes E and F.

ShowFont is in the source file FONT.PAS.

HodgePodge calculates line height in pixels and locates all drawing in relation to the document origin—point (0,0)— when it displays the contents of a font window using the routine `ShowFont`. The routine first installs a font (loads it into memory), then calculates line height, and then uses that information to draw the text. `ShowFont` is called from the routine `DispFontWindow`.

```
procedure ShowFont(TheFontID: FontID;
                   IsMono: Boolean);              {Begin ShowFont…}

var    fontInfo   : FontInfoRec;                  {a record to hold font information}
       currHeight: Integer;                       {line height of selected font}
       i,j        : Integer;
       theCh      : Integer;
       currPt     : Point;
       fontStr    : Str255;                       {string variable to hold characters}

begin
   InstallFont(TheFontID,0);                      {Load the font into memory…}
   GetFontInfo(fontInfo);                         {…and store its data in FontInfo}
   currHeight := fontInfo.ascent +
                 fontInfo.descent +
                 fontInfo.leading;                {Calculate the font's line height}

   i := GetFamInfo(TheFontID.famNum,fontStr);     {Get the font's name…}
   fontStr := concat(fontStr,' ',
                 IntToString
                  (TheFontID.fontsize));          {…make a title string with
                                                  font's name and size}
    i := GetFontFlags;                            {Get current mono/prop. setting}
    if IsMono then                                {If menu selection says "mono"…}
     i := BitOr(i,$0001)                          {…set bottom bit = fixed width}
    else                                          (Otherwise:}
     i := BitAnd(i,$0000);                        {Clear bottom bit = proportional}
    SetFontFlags(i);                              {Store result in font flags}
```

```
                                                   {Now draw the lines of text:}
    MoveTo(5,currHeight);                          {Move pen to start of first line}
    DrawString(fontStr);                           {Draw the title string}

    MoveTo(5,currHeight*3);                        {Skip a line, move to start of next}
    DrawString(                                    {Draw second line}
        'The quick brown fox jumps over the lazy dog.');
    MoveTo(5,currHeight*4);
    DrawString(                                    {Draw third line}
        'She sells sea shells down by the sea shore.');

    MoveTo(5,currHeight*5);                        {Now draw all characters in the font:}
    for i := 0 to 7 do                             {For each of 7 lines...}
      begin
        GetPen(currPt);                            {starting at current pen location...}
        MoveTo(5,currPt.v + currHeight);           {...drop to start of next line}
        theCh := i * 32;                           {...calculate starting character...}
        for j := 1 to 32 do                        {For each of 32 chars. in the line...}
          begin
            fontStr[j] := chr(theCh);              {put the character into fontStr}
            Inc(theCh);                            {go to the next character}
          end;                                     {end filling fontStr for this line}

        fontStr[0] := chr(32);                     {Now set the length byte to 32...}
        DrawString(fontStr);                       {...and draw the character string}

      end;                                         {end of drawing the line}
end;                                               {End of ShowFont}
```

Chapter 4

Using the Toolbox (II)

This chapter continues the discussion of the Apple IIGS Toolbox. Starting up, handling events, and basic drawing to the screen were covered in Chapter 3; here we look at tool sets that help you create windows, dialog boxes, and alerts. Chapter 5 presents the remaining tools.

Creating windows

The concepts of *GrafPort* and *port rectangle* are covered in Chapter 3.

A **window** is basically a *port rectangle* with a frame; when you create a window you create a *GrafPort,* along with some additional information that makes a window. When you draw into a window, you are drawing into the port rectangle of the GrafPort associated with that window. The **Window Manager** is the Apple IIGS tool set that creates these "ports with frames," keeps track of their characteristics, and makes it easy for you to deal with the fact that there may be multiple, movable, scrollable, overlapping windows on the screen at any one time.

Window basics

To begin to understand windows, let's look at some basic concepts, specifically:

- how windows relate to GrafPorts
- what data structures define a window's features
- what types of window frames and controls are available
- what window regions are, and how the *content region* of a window relates to its *data area*

Windows and GrafPorts

It's easy to use windows—a window is a port that your application can draw into conveniently with QuickDraw II routines. When you first create a window, the pixel image and boundary rectangle for its GrafPort correspond to the entire screen (QuickDraw II's default assignment), and the pen pattern and other characteristics are also the default values for a GrafPort. You can accept these default characteristics unchanged, or you can easily change them with QuickDraw II routines.

But there is more to a window than the port in which the application draws. The other part of a window is called the **window frame.** It usually surrounds the window, and is not part of the window's GrafPort. You don't draw into the window's frame area directly—the Window Manager takes care of that.

❖ *Note:* For drawing window frames, the Window Manager uses a GrafPort that has the entire screen as its port rectangle; this GrafPort is called the *Window Manager port.*

Window records and templates

The Window Manager keeps all the information it requires for its operations on a particular window in a **window record.** The record consists of the window's GrafPort record plus other information the Window Manager needs to manage windows. Application access to record information is restricted to calls through the Window Manager and directly to the GrafPort part of the window record. As in the case of any toolbox records, accessing their fields through calls instead of reading them directly, helps to guarantee that your application will remain compatibile with future toolbox revisions.

When you create a new window with the NewWindow call, you pass the Window Manager a **NewWindow parameter list**, a template that defines the details of the window to be created, including its size and location and what controls it will have. The Window Manager uses this information to construct the window's record. Three fields in the NewWindow parameter list are worth specific mention:

For examples of the use of these fields in HodgePodge, see the listings of Paint or DispFontWindow under "Handle Specific Events" in Chapter 2. See also the listing of the routine DoTheOpen, under "Opening a Window: An Example," later in this section.

- □ `wFrame`, a set of bit flags that controls, among other things, whether the window is to have frame scroll bars. Simply by setting bits in this field, HodgePodge specifies that its windows are to have both horizontal and vertical scroll bars.
- □ `wRefCon`, which can have any contents an application wants. HodgePodge uses this field to store a pointer to information about the type of window (font or picture) being created.
- □ `wContDefProc`, which, if nonzero, contains a pointer to a routine (**definition procedure,** or *defProc*) that draws the contents of the window. HodgePodge stores pointers to the routines `Paint` or `DispFontWindow` in this field.

Window frames and controls

There are two kinds of predefined window frames, **document** and **alert.** Figure 4-1 illustrates them.

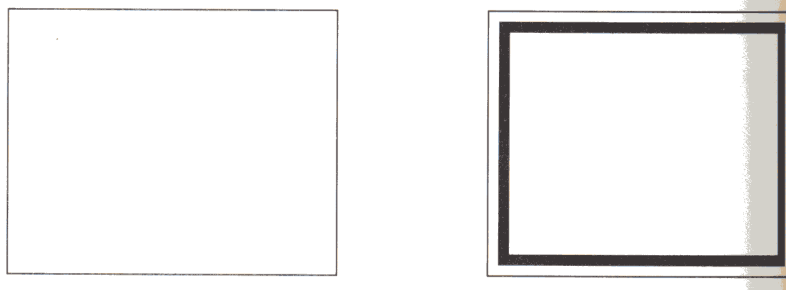


Figure 4-1
Window frames

The alert window is used by the Dialog Manager; see "Constructing Dialog Boxes and Alerts," later in this chapter. The document window is what an application typically uses. Inside a document window can be **standard window parts,** which include the following:

Some of the standard window parts are controls. Controls are described in more detail in the following section, "Putting Controls in Windows."

- **Title bar,** a rectangle at the top of the window that displays the window's title, may hold the close and zoom boxes, and may be a drag region for moving the window.
- **Close box (go-away box),** a small square in the title bar that the user **clicks** on to remove the window from the screen.

Clicking refers to pressing and releasing the mouse button while the mouse pointer is stationary on the screen.

- **Zoom box,** a small square in the title bar that the user clicks on to alternately make the window its maximum size or return it to its previous size and position.
- **Right scroll bar,** a rectangle on the right side of the window that the user manipulates to scroll vertically through the data shown in the window.
- **Bottom scroll bar,** a rectangle at the bottom of the window that the user manipulates to scroll horizontally through the data shown in the window.
- **Size box,** a small square at the lower-right corner of the window that the user **drags** to change the size of the window.

Dragging refers to pressing the mouse button to select something and holding the button down while moving the mouse.

- **Information bar,** a rectangular area where the application can display information that won't be affected by the scroll bars.

These standard parts may be used in document windows only; they may not be added to alert windows. They are illustrated in Figure 4-2.

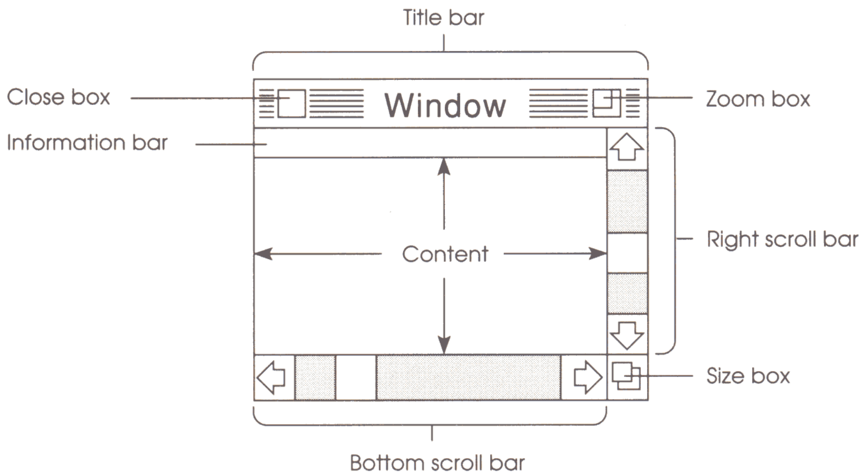


Figure 4-2
Standard window controls

A document window may have any or all of the standard window parts. The only restrictions are that if there is a close or zoom box, there must also be a title bar, and if there is a size box, there must also be a vertical scroll bar. Common sense suggests that there be a zoom box if there is a size box, but this is not a requirement.

Color in an application should be designed carefully. Please refer to *Human Interface Guidelines: The Apple Desktop Interface* for suggestions.

❖ *Color:* You can specify the colors of the frame and controls of a window you create. Colors are selected from a color table. See "Window Manager" in the *Apple IIGS Toolbox Reference* for details.

It's possible to define your own type of window, such as round or hexagonal. See the *Apple IIGS Toolbox Reference* for more information.

You can use the standard window types, or you can create your own window types. Some windows may be created indirectly for you when you use other parts of the toolbox—for example, the Dialog Manager creates a window to display an alert. Windows created either directly or indirectly by an application are collectively called **application windows.** There's also a class of windows called **system windows;** those are the windows in which desk accessories are displayed.

Content region and data area

A *region* is a graphic object defined by QuickDraw II. See "Drawing to the Screen" in Chapter 3.

A window is composed of *regions*. The window as a whole (the **structure region**) is made up of the *content region* and the *frame region*:

- □ The **content region** is bounded by the rectangle you specify when you create the window (that is, the port rectangle of the window's GrafPort). The content region is where your application presents information to the user.
- □ The **frame region** is the rest of the window. It may include several subregions that correspond to the locations of the standard window parts described earlier. When the user manipulates a certain control, the Window Manager sees it as an event occurring in a certain subregion. See "Handling Window-Related Events," later in this section.

The content region of a window is what the user "sees" within the window. It commonly represents a larger area, containing more information than the screen can display at one time. The window is then like a microfiche machine—what is seen at any one time in its content region, like what is seen in a microfiche viewer, might be only a small portion of the window's entire **data area,** equivalent to the microfiche sheet.

The routines PaintIt and ShowFont in Chapter 3 show how HodgePodge represents the data areas of its windows.

The data area is a pixel-based "picture" of whatever document is being displayed in the window. For a pixel image (such as a HodgePodge picture file), the data area is the pixel image itself. For a text document (such as a HodgePodge font window display), the data area is a conceptual representation of what the document *would look like* if it were a pixel image. The document doesn't exist in that form anywhere; the appropriate parts of it are calculated and drawn in the window's GrafPort each time the window is drawn or updated.

How scrolling is accomplished is described later in this section under "Handling Window-Related Events."

Scroll bars are the controls used to scroll the data area through the content region of the window. The size box and zoom box are used to display more, or less, of the data area at one time. When the window as a whole is moved to another location on the screen, the data area is moved with it, so the view in the content region remains the same.

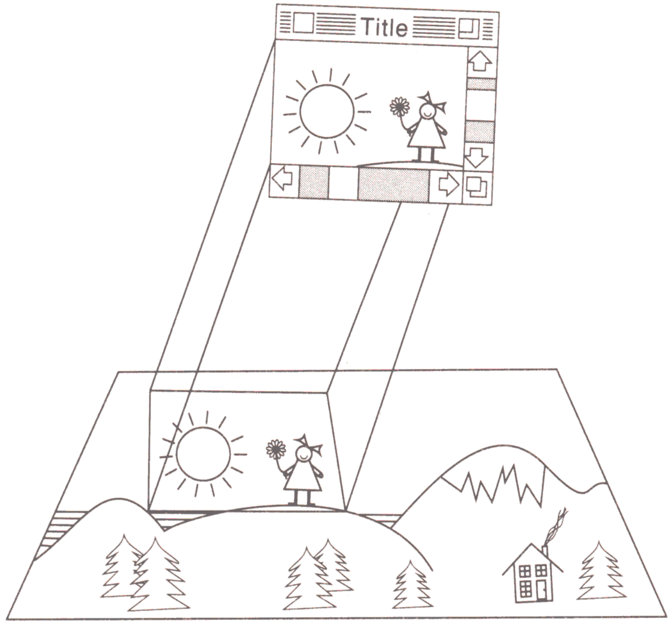


Figure 4-3
A window displays part of its data area

Handling window-related events

A window's **plane** is its front-to-back position on the screen, in relation to other windows.

The Window Manager's principal function is to keep track of overlapping windows. Your application can draw in any window without running over onto windows in front of it. You can move windows to different places on the screen, change their **plane,** or change their size, all without concern for how the various windows overlap. The Window Manager keeps track of any newly exposed areas and provides a convenient mechanism for you to ensure that they are properly redrawn.

There are two ways to handle user input in relation to windows. You can poll the user with the Event Manager routine GetNextEvent, or with the Window Manager routine TaskMaster, which handles most events dealing with standard user interfaces. See "Using TaskMaster" in Chapter 3.

Compare these results from FindWindow with the TaskMaster task codes in Table 3-3.

If you are using GetNextEvent, you should call FindWindow every time a mouse-down event occurs, to see if the mouse button was pressed inside a window. The FindWindow call determines which region is affected, and returns the information to you. The following are the various subregions recognized by FindWindow, and the standard actions to take in each case.

- The content region has already been described. If the mouse button is pressed in a window's content region, call SelectWindow if the window is not the active window. Otherwise, handle the event according to your application.
- The **drag area** corresponds to the window's title bar (except for the close and zoom boxes, if present). Dragging in this subregion pulls an outline of the window across the screen, moves the window to a new location, and makes it the active window (if it isn't already).

  If the mouse button is pressed in a window's drag region, call DragWindow.
- The **go-away area** corresponds to the close box in the window's title bar. Clicking in this subregion closes the window. Depending on your application, the window may disappear permanently or simply become hidden.

  If the mouse button is pressed in the active window's close box, call TrackGoAway. If TrackGoAway returns TRUE, call CloseWindow, or HideWindow, perhaps after saving whatever the user was working on inside the window. You may also want to close any disk file associated with the closed window.
- The **zoom area** corresponds to the zoom box in the window's title bar. Clicking in this subregion toggles between the current position and size, to a maximum size and back again.

  If the mouse button is pressed in the active window's zoom area, call TrackZoom. If TrackZoom returns TRUE, call ZoomWindow.
- The **grow area** corresponds to the size box in the window's lower-right corner. Dragging in this region pulls the lower-right corner of an outline of the window across the screen with the window's origin fixed, and then resizes the window when the mouse button is released.

  It the mouse button is pressed in the active window's grow area, call GrowWindow. When the button is released, call SizeWindow.

- □ The **menu bar** is not a window subregion, but a result returned by FindWindow that means "not on the desktop."

  If the mouse button is pressed somewhere outside the desktop, it is most likely in the system menu bar. Call MenuSelect.

❖ *Inactive window:* Clicking in any region (other than the drag region) of an inactive window should have no effect other than making it the active window. It is brought to the front and highlighted to indicate that it is active.

❖ *TaskMaster:* If you are using TaskMaster, it calls FindWindow for you. It also calls MenuSelect, DragWindow, TrackGoAway, or other appropriate calls depending on the results of FindWindow. In general, you needn't handle any window-related mouse events, except possibly in the content region of an active window. TaskMaster may not know what you want drawn in an active window.

Drawing or redrawing a window

When a window is drawn or redrawn, the window frame is drawn first, followed by the window contents. The Window Manager handles all frame drawing.

When a window's contents need to be redrawn, the Window Manager generates an *update event* that includes a pointer to the affected window in the message field of the event record. Your application should respond to update events as follows:

1. Call BeginUpdate. This procedure temporarily replaces the *visible region* of the window's GrafPort with the intersection of the visible region and the **update region.** It then clears (resets to zero size) the update region for that window.
2. Draw the window contents. Because of step 1, the redrawing is automatically clipped, or limited, to the part of the visible region that needs updating.
3. Call EndUpdate to restore the actual visible region.

❖ *TaskMaster:* If you use TaskMaster, this procedure is done for you, as long as you provide TaskMaster with a routine that draws your window's contents (equivalent to step 2, above).

The *visible region* is the portion of a window that is not offscreen or hidden by other windows. It is one of the fields in a GrafPort that clips drawing commands. See "Drawing to the Screen" in Chapter 3.

The **update region** is the portion of a window that needs to be redrawn. It may be a part exposed by moving or closing another window, or it may be a new part of the data area exposed during scrolling.

❖ *HodgePodge:* Although it uses TaskMaster and doesn't really need an update routine, HodgePodge has a short example of an update routine in the code that creates one of its dialog boxes. See the listing of `ShowPleaseWait`, under "Constructing Dialogs and Alerts," later in this chapter.

Making a window active

A number of Window Manager routines change the state of a window from inactive to active or from active to inactive. For each such change, the Window Manager generates an activate event. When the Event Manager finds out from the Window Manager that an activate event has been generated, it passes the event on to the application through GetNextEvent.

Activate events for dialog and alert windows are handled by the Dialog Manager, so your application doesn't have to bother with them. In response to activate events for windows created directly by your application, you might take actions such as the following:

- Inactivate controls in inactive windows, and activate controls in active windows.
- In a window that contains text being edited, remove the highlighting or blinking cursor from the text when the window becomes inactive, and restore it when the window becomes active.
- Enable or disable a menu or certain menu items as appropriate to match what the user can do when windows become active or inactive.

❖ *TaskMaster:* If you use TaskMaster, highlighting of standard windows and controls is handled for you. Enabling and disabling of menu items is not.

❖ *HodgePodge:* To keep menu highlighting in agreement with activate events, HodgePodge calls its subroutine `CheckFrontW` each time through the event loop.

Scrolling

Scrolling is the process by which the user can bring different parts of a document (data area) into view in a window. To accomplish scrolling, the user manipulates **scroll bars,** standard window controls managed by the Control Manager. An application (or TaskMaster) responds to user manipulation of scroll bars by:

1. Updating the appearance of the scroll bars to reflect the change in position of the data area. This step is described under "Putting Controls in Windows," later in this chapter.
2. Showing a new part of the document in the window. The application (or TaskMaster) does this by shifting the image in the window, then changing the window's *local coordinates* and redrawing the parts of the data area brought into view. This step is described below.

Local coordinates are discussed under "Drawing to the Screen" in Chapter 3.

❖ *TaskMaster:* If your application uses TaskMaster, it can have TaskMaster-controlled scroll bars (*frame scroll bars*) in its windows. In that case the application need have no scrolling routines at all. The following applies only if your application creates and manipulates its own scroll brars.

❖ *HodgePodge:* Because it calls TaskMaster, HodgePodge has no scrolling procedure.

Consider a pixel image, part of which is displayed in a window, such as the dollar bill in Figures 3-5 and 3-6. Let's say that the window presently shows George Washington's face (Figure 4-4), and the user wants to scroll the image to bring into view the circular Federal Reserve seal to the left of Washington. With the mouse, the user activates the left-facing arrow on the bottom scroll bar. When your application determines that there has been a mouse-down event in that part of the scroll bar, it should respond as follows:

1. Call the QuickDraw routine ScrollRect and tell it to move all the pixels in the content area of the window a certain number of pixels to the right. George shifts a bit to the right. The way ScrollRect works, any pixels moved off the right edge of the window are lost, and extra pixels added to the left edge of the image are blank (colored with the background pattern).
2. Your onscreen image has been shifted, but QuickDraw hasn't automatically filled in the new part of the image that has come into view. However, ScrollRect returns information to you that tells you exactly what part of your window needs redrawing. Call InvalRgn to add that newly exposed area to the window's update region.

Your application determines how many pixels are needed to shift the image. See "Putting Controls in Windows," later in this chapter.

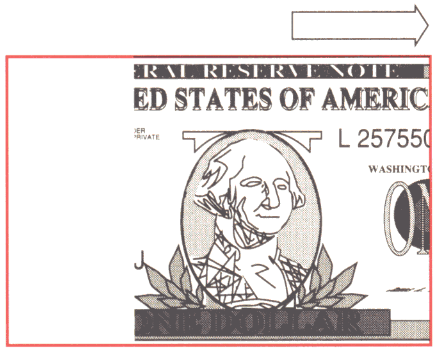
3. Call BeginUpdate.
4. Call your routine that draws window contents. What that routine should do is:
   a. Set the local origin to its scrolled value: what it was the last time the window's contents were drawn PLUS the (negative value of the) number of pixels that ScrollRect shifted the image.
   b. Draw the window's contents (perhaps by calling PPToPort). The image is properly shifted and clipped so that just the needed part is drawn. There's the seal!
   c. Set the local origin back to (0,0).
5. Call EndUpdate.

See the toolbox reference for instructions on installing a control action procedure.

If you put the above steps into a control action procedure, they will be called repeatedly as long as the user holds the mouse button down with the pointer in the scroll bar. The image will scroll continuously.

Alternatively, if *continuous* scrolling is unnecessary, you can ignore steps 3 through 5. The InvalRgn call causes the Window Manager to generate an update event for exactly that part of the window, and the next time through the event loop, your regular update routine redraws the window. The redrawing won't happen, though, until the user releases the mouse button.

a. Part of a document displayed in a GrafPort



b. Application scrolls image to the right. Pixels moving off right edge are lost; new area filled with background pixels.

c. Application updates the new area scrolled into view by shifting coordinates and redrawing.

Figure 4-4
Scrolling a pixel image in a window

Important

When a window is created, the Window Manager assigns its port rectangle origin a value of (0,0) in local coordinates. Whenever it redraws the window frame, the Window Manager requires the origin to have that same value.

Therefore, every time you draw your window's contents you should (1) set the origin to whatever is appropriate, (2) draw the contents, and then (3) restore the origin to (0,0).

Opening a window: an example

The sequence of subroutine calls described in this section is diagrammed in Appendix D.

The following example from HodgePodge shows the steps involved in allocating the memory for, creating, and drawing the initial contents of a window. Remember that in HodgePodge there are two types of window: one type displays picture files and the other displays lines of text using a particular font.

DoOpenItem is in the source file MENU.PAS.

The sequence starts when the user chooses Open from the File menu or Show Font from the Font menu. In either case execution passes from `DoMenu` to the routine `DoOpenItem`. `DoOpenItem` is very short:

```
procedure DoOpenItem;                 {begin DoOpenItem…}

begin
  if wIndex < LastWind then           {If there's room for another window…}
    if OpenWindow then                {call OpenWindow. If it opens OK…}
      AddtoMenu                       {…add its name to the Windows menu}
    else
  else                                {If 16 windows already open…}
    ManyWindDialog;                   {put up a dialog and disallow open}
  end;                                {End of DoOpenItem}
```

Note that `DoOpenItem` calls both `OpenWindow` (to open the window) and `AddToMenu` (to add the window's name to the Windows menu). `AddToMenu` is described under "Making and Modifying Menus" in Chapter 5. If 16 windows are already open, HodgePodge does not allow another to be opened.

OpenWindow is in the source file WINDOW.PAS.

`OpenWindow` determines which type of window is to be opened, and prompts the user for the necessary information (picture filename or font characteristics). It then calls the routine `DoTheOpen`, which actually opens the window. `OpenWindow` looks like this:

```
function  OpenWindow: Boolean;                        {begin OpenWindow...}

begin
  OpenWindow := FALSE;                                {initial value of function = FALSE}
  if (LoWord(Event.wmTaskData =FontItem) then         {if it is a font window...}
    begin
     if DoChooseFont then                             {...and if the user doesn't cancel...}
       if DoTheOpen then                              {...and if the window opens OK...}
         OpenWindow := TRUE                           {OpenWindow completes successfully}
    end
  else                                                {if it is a picture window...}
    begin
     if AskUser then                                  {...and if the user doesn't cancel...}
       if DoTheOpen then                              {...and if the window opens OK...}
         OpenWindow := TRUE                           {OpenWindow completes successfully}
    end;

end;                                                  {End of OpenWindow}
```

`DoChooseFont` was described earlier in this chapter. `AskUser` is described in Chapter 5, under "Communicating With Files and Devices."

DoTheOpen is in the source file WINDOW.PAS.

Once it has all the information it needs, `OpenWindow` calls `DoTheOpen` to open the window. `DoTheOpen` looks like a long and complex routine, but that's partly because it is two routines in one; it handles two types of windows. It also does a lot of assignment and initialization that your programs may not need at this point. We'll break its description into chunks to make it easier to follow.

The complete format for the NewWindow parameter list is given under "Window Manager" in the *Apple IIGS Toolbox Reference.*

`DoTheOpen` starts by allocating memory for the window data record (a structure defined by HodgePodge), and putting some initial values into the *NewWindow parameter list,* a toolbox-defined structure that is a required input to the NewWindow call.

```
function DoTheOpen : Boolean;                         {begin DoTheOpen...}

var    theWindow    : GrafPortPtr;                    {a pointer to our window}
       myDataHandle: WindDataH;                       {a handle to our own window-data
                                                       record--defined in GLOBALS.PAS}
       theMenuStr   : Str255;                         {window's title for menu display}
       ourFontInfo  : FontInfoRec;                    {to hold font information}
```

```
begin
  DoTheOpen := FALSE;                          {initial value of function = FALSE}

  myDataHandle := WindDataH(                   {get a handle to our record by...}
            NewHandle(sizeof(WindDataRec),     {...requesting memory with these...}
                   myMemoryID,                 {attributes:  size, User ID,}
                   attrLocked+attrFixed,       {locked and fixed,}
                   Ptr(0)));                   {anywhere}
  if isToolError then
    Exit;                                      {terminate if memory unavailable}
                                               {myWind is a window parameter block,}
  with myWind do                               {...required input to NewWindow call}
    begin                                      {Initialize the window's features:}
     paramLength  := sizeof(ParamList);        {total size of list}
     wFrameBits   := $DDA0;                    {this specifies scroll bars, etc.}
     wRefCon      := LongInt(myDataHandle);    {handle to our window data record}
     SetRect(wZoom,0,26,520,190);              {window size & postion when zoomed}
     wColor       := NIL;                      {no colors for this window}
     wYOrigin     := 0;                        {y-coord. of port rect origin}
     wXOrigin     := 0;                        {x-coord. of port rect origin}
     wDataH       := 188;                      {document height}
     wDataW       := 640;                      {document width}
     wMaxH        := 200;                      {max. window height to grow}
     wMaxW        := 640;                      {max. window width to grow}
     wScrollVer   := 4;                        {amt. to scroll if v. arrow clicked}
     wScrollHor   := 16;                       {amt. to scroll if h. arrow clicked}
     wPageVer     := 40;                       {amt. to scroll if v. page clicked}
     wPageHor     := 160;                      {amt. to scroll if h. page clicked}
     wInfoRefCon  := 0;                        {no info. bar for this window}
     wInfoHeight  := 0;                        {no info bar for this window}
     wFrameDefProc:= NIL;                      {no special frame-drawing routine}
     wInfoDefProc := NIL;                      {no special info-bar content routine}
     wPlane       := -1;                       {make this window frontmost}
     wStorage     := NIL;                      {let Window Mgr allocate the memory}
    end;

  theMenuStr := concat('==',                   {Make a title for the...}
                  myReply.filename,
                  '\N',                        {...window to appear in...}
                  IntToString(
                  FirstWindItem+wIndex),
                  '\0.');                      {...theWindows menu.}

  with myDataHandle^^ do                       {In the window-data record...}
    begin
     name     := myReply.filename;             {...fill in the name field}
     menuStr  := theMenuStr;                   {...fill in the menu title field}
     menuID   := FirstWindItem+wIndex;
    end;
```

After this initial allocation, DoTheOpen sets up the window to display either text or a picture. It inserts into the NewWindow parameter list a pointer to the procedure that draws the window's interior, and sets the remaining fields in the window-data record.

```
if LoWord(Event.wmTaskData) = FontItem then    {if it is a font window...}
 begin
  myWind.wContDefProc := @DispFontWindow;      {DispFontWindow will draw contents}
  with myDataHandle^^ do
   begin
    flag     := 1;                              {1 means it's a font window}
    theFont  := DesiredFont;                    {store present font ID in theFont}
    isMono   := isMonoFont;                     {store present setting of isMonoFont}
   end;
  InstallFont(desiredFont,0);                   {load the desired font into memory}
  GetFontInfo(ourFontInfo);                     {...get its characteristics...}
  myWind.wDataH := 15*(ourFontInfo.ascent+      {...and calculate document height--}
                  ourFontInfo.descent);         {15 = 2 + no. of lines in document}
 end                                            {end of IF it's a font window}

else                                            {But if it's a picture window...}
 begin
  myWind.wContDefProc := @Paint;                {Paint will draw contents}
  with myDataHandle^^ do
   begin
    flag := 0;                                  {0 means it's a picture window}
    pict := picHndl;                            {store handle to desired picture...}
   end;                                         {...(determined by AskUser)}
 end;                                           {end of IF it's a picture window}
```

Now DoTheOpen determines where on the screen the window is to appear. Each newly-opened window is offset down and to the right from the previously opened window. Recall from SetUpWindows (Chapter 2) that ISizPos is the initial position and size of a window.

```
with myWind do                                  {In the window-data record...}
 begin
  wTitle := @myDataHandle^^.name;               {set window title to name field}
  SetRect(wPosition,
          wXoffset + iSizPos.h1,
          wYoffset + iSizPos.v1,                {Add the window dimensions to the...}
          wXoffset + iSizPos.h2,                {...current X- and Y- offsets}
          wYoffset + iSizPos.v2);
 end;                                           {end of setting record fields}

wXoffset := wXoffset + 20;                      {Then increment offsets...}
wYoffset := wYoffset + 12;                      {...to set position of next window}
if wYoffset > 120 then
  wYoffset := 12;                               {(after 10 windows make another row)}
```

Finally, now that everything is all set up, DoTheOpen creates the window itself. It uses the NewWindow call, passing to NewWindow the parameter list that DoTheOpen just filled in. You can see from the following code that opening a window is quite simple and short. It is the preparation and initialization that makes the routine seem long and complicated.

```
    theWindow := NewWindow(myWind);       {Open the window--NewWindow
                                          returns a pointer to it}
    SetPort(theWindow);                   {Make the window the active port}
    SetOriginMask($FFFE,theWindow);       {Adjust window origins to make
                                          dithered colors come out right}
    InitCursor;                           {Go back to the arrow cursor}
    DoTheOpen := TRUE;                    {DoTheOpen completes successfully}
end;                                      {End of DoTheOpen}
```

Putting controls in windows

A **control** is an object on the IIGS screen with which the user, using the mouse, can cause instant action with graphic results or change settings to modify a future action. Controls are fundamental to the concepts behind the Human Interface Guidelines; they provide a simple, intuitive interface, permitting the user to affect the course of an application. If well-designed, they reinforce the feelings of user control, friendliness, and consistency that mark a good desktop application.

The Control Manager is the part of the Apple IIGS Toolbox that helps you create and manipulate controls. The Control Manager carries out the actual operations, but it's up to you to decide when, where, and how.

Controls may be of various types, each with its own characteristic appearance on the screen and responses to the mouse. Each individual control has its own specific properties—such as its location, size, and setting—but controls of the same type behave in the same general way.

Types of controls

Certain standard types of controls are predefined for you. Your application can easily use these standard types, or define its own custom controls. Predefined controls perform a number of functions:

- **Buttons** cause an immediate or continuous action when clicked or pressed with the mouse. They typically appear on the screen as rounded-corner rectangles with a title centered inside.
- **Check boxes** retain and display a setting, either checked (on) or unchecked (off); clicking with the mouse reverses the setting. On the screen, a check box appears as a small square with a title to the right of it; the box is either filled in with an X (checked) or empty (unchecked). Check boxes are frequently used to control or modify some future action, instead of causing an immediate action of their own. More than one box may be checked at any one time.
- **Radio buttons** also retain and display an on-or-off setting. They're organized into families; only one button in a family should be on at a time. Clicking any button on should turn off all the others in the family, like the buttons on a car radio. The radio button that's on is filled with a small black circle.
- **Dials** display a quantitative setting or value, typically in some pseudo-analog form such as the position of a sliding switch, the reading on a thermometer scale, or the angle of a needle on a gauge. The setting may be displayed numerically as well.

  The control's moving part that displays the current setting is called the indicator. The user may be able to change a dial's setting by dragging its indicator with the mouse, or the dial may simply display a value not under the user's direct control (such as the amount of free space remaining on a disk).

The standard controls and a few other typical controls are illustrated in Figure 4-5.

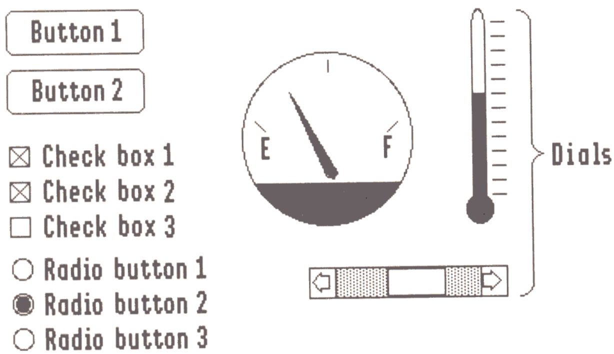


Figure 4-5
Standard and typical controls

Scroll bars

Scroll bars are predefined dials. Selecting the arrows in a scroll bar scrolls data a *line* at a time (or an analogous number of pixels in the horizontal direction); selecting the paging regions scrolls data a *page* at a time; and dragging the thumb to any position within the scrolling area locates the window equivalently within the data area. Although each of these components may seem to behave like individual controls, they are all parts of a single control, the scroll-bar type of dial. You can define other dials of any shape or complexity if your application needs them.

❖ *Note:* For scrolling, what constitutes a *page* and what constitutes a *line* are definable by your application.

Figure 4-6 shows the parts of the vertical and horizontal scroll bars.

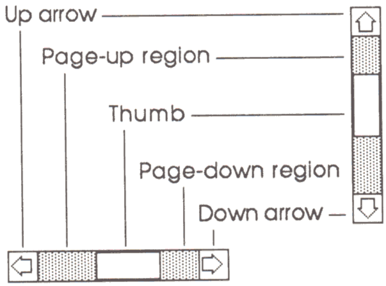


Figure 4-6
Parts of the scroll bars

Scroll bars are proportional—that is, they show the relationship between the total amount of data and the amount viewed (and where the view is in the data). As Figure 4-7 shows, the thumb is the same ratio to the scrolling area (the total distance between the arrows) as the content region is to the data area.

When the user clicks in a scroll bar, the Control Manager returns to your application a *part code,* telling it what part of the scroll bar the event occurred in. Depending on whether it is in an arrow, paging region, or thumb, your application probably should scroll the document by a different amount. Once you know how much the view should be scrolled, you can recalculate the scroll bar values to keep the proportions as illustrated in Figure 4-7. Then, call SetCtlValue to redraw the scroll bar with the thumb in the proper new position.

❖ *Note:* Part codes are returned for all types of controls, but they are most significant for complex controls such as scroll bars.

With the SetCtlParams call, you can store in a scroll bar's record the current sizes of your document's data area and content region (window size). Then you can easily calculate their proportions for setting scroll bar values after scrolling or resizing.

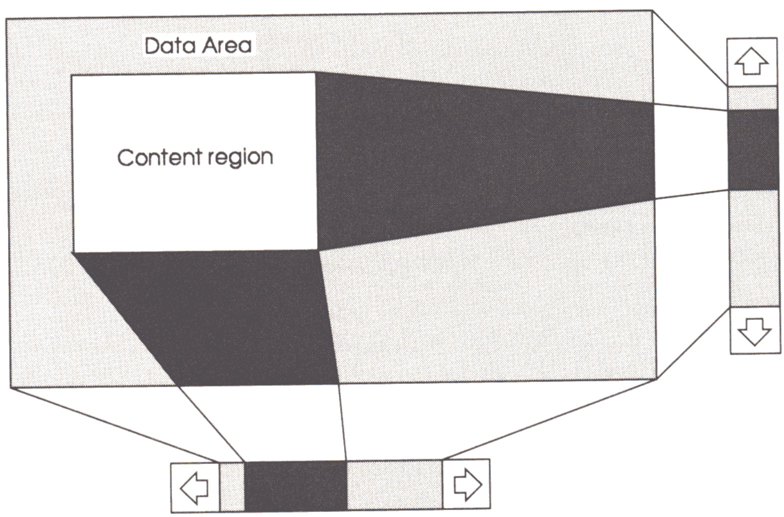


Figure 4-7
Relation of scroll bars to data area

Highlighting can mean different things in different instances, but it often consists of inverting an object—that is, changing all its black pixels to white, and vice versa .

Active controls and highlighting

If the user presses the mouse button when the cursor is over a control, the control is usually **highlighted;** see Figure 4-8. It's also possible for just part of a control to be highlighted: for example, if the user presses the mouse button when the pointer is inside an arrow in a scroll bar, the arrow, not the whole scroll bar, becomes highlighted.

A control may be **active** or **inactive.** Active controls respond to the user's mouse actions; inactive controls don't. A control should be made inactive when it has no meaning or effect in the current context, such as an *Open* button when no document has been selected to open, or a scroll bar when there's currently nothing to scroll to. An inactive control is shown in some special way, depending on its control type. Figure 4-8 illustrates some active and inactive controls.

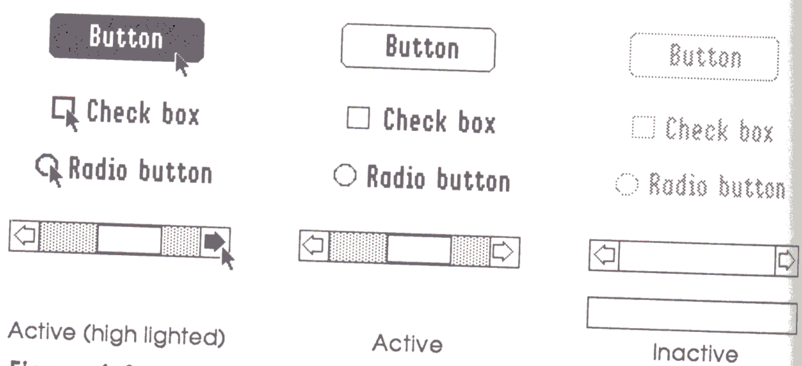


Figure 4-8
Active controls and inactive controls

The title and outline of a button, check box, or radio button are dimmed automatically when the control is made inactive. Figure 4-8 shows two different appearances that an inactive scroll bar can take.

You can make a control inactive by setting its value to a particular number. You can also render a control inactive by making it invisible. Invisible controls are inactive in the sense that they can't be selected.

Using controls

Controls and windows

Every control belongs to a window: when the control is displayed, it appears within that window's content region; when manipulated with the mouse, it acts on that window. All coordinates pertaining to the control (such as those describing its location) are given in the window's local coordinate system. Even the state of the control can be tied to the state of the window. A bit in the window's record can be set so the controls in the window will be considered inactive if the the window is inactive. See "Window Manager" in the *Apple IIGS Toolbox Reference*.

❖ *Frame scroll bars:* Frame scroll bars (manipulated by TaskMaster) work the same as other controls, but are part of a window's frame region rather than its content region.

Controls and events

When GetNextEvent reports that an update event has occurred for a window, your application should call DrawControls to redraw the window's controls as part of the process of updating the window.

❖ *TaskMaster:* If you're using TaskMaster, you needn't redraw controls that are part of the window frame—TaskMaster takes care of it for you.

When GetNextEvent reports a mouse-down event for a window that contains controls, do this:

1. Call FindWindow to determine which part of which window the cursor was in when the user pressed the mouse button. If it was in the content region of the active window, continue with step 2.
2. Call FindControl to find out where the event occurred.
3. If FindControl indicates that the event occurred in an active control, call TrackControl to handle user interaction with the control. TrackControl handles the highlighting of the control and determines whether the mouse is still in the control when the mouse button is released. The routine also handles the dragging of the thumb in a scroll bar and responds to presses or clicks in the other parts of a scroll bar.

4. If TrackControl confirms that a valid control was selected, do whatever is appropriate as a response. (If no control was selected, then of course no action is necessary).

The application's exact response to mouse activity in a control that retains a setting depends upon the current setting of the control. For example, when a check box or radio button is clicked you'll make a Control Manager call to change the setting and draw or clear the mark inside the control.

❖ *TaskMaster:* If your application calls TaskMaster, the above procedure is handled automatically for frame scroll bars in standard windows. Only if you have other controls will you need a control-drawing routine.

❖ *HodgePodge:* Because it uses only window-frame controls, and because it calls TaskMaster, HodgePodge has no specific routine to manipulate or draw controls.

Defining your own controls

In addition to predefined controls, you can also define your own custom controls. Perhaps you need a three-way selector switch, a memory-space indicator that looks like a thermometer, a thruster control for a spacecraft simulator, or some other special type of control. Controls and their indicators may occupy regions of any shape.

To define your own type of control, you place a control definition procedure in your application. The Control Manager stores the address of the procedure in the *ctlProc* field of the control record when you create the control with a NewControl routine. Later, when the Control Manager needs to perform a type-dependent action on the control, it calls the control definition procedure. See the *Apple IIGS Toolbox Reference* for details.

Manipulating lists of selectable items

If your program displays a list of available fonts, files, telephone numbers, icons, or other items in a window, it may put them in *lists,* as defined by the Apple IIGS Toolbox. A **list** is a vertical arrangement of similar items on the screen, with a scroll bar to the right. Each item in the list is *selectable,* meaning it can be highlighted individually, with a mouse click or other action.

The List Manager is the Apple IIGS tool set that creates, manipulates and supports lists. It relieves you (the programmer) of much of the housekeeping involved with building and maintaining complicated lists of items the user may select from. Lists created by the List Manager are custom controls, called **list controls;** that's why we mention the List Manager here, under the Control Manager.

You create a list as a *list record,* with a specific format. You may use the List Manager to sort the list, if desired, and then to create the list control. Once the list control is drawn on the screen, the user can select individual items or a range of items from the list. How your application handles those selections is, of course, up to you.

Constructing dialog boxes and alerts

Two of the most useful and versatile means of commmunicating with your application's user are provided by *dialog boxes* and *alerts*. The Dialog Manager provides these capabilities in a way consistent with the Apple Human Interface Guidelines.

The **Dialog Manager** is a sophisticated window- and control-manipulation tool set. It automatically performs many functions your application would otherwise have to manage explicitly through Event Manager, QuickDraw II, Window Manager, LineEdit, and Control Manager calls.

What are dialog boxes?

Your application typically puts a dialog box on the screen when it needs more information from the user in order to carry out a command. As shown in Figure 4-9, a **dialog box** resembles a form on which the user checks boxes and fills in blanks.

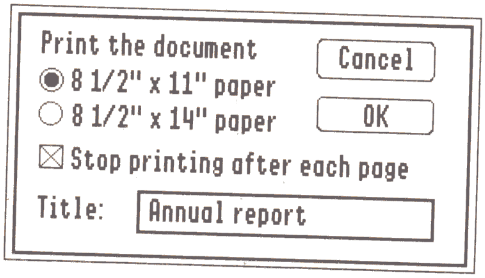


Figure 4-9
A modal dialog box

By convention, a dialog box appears slightly below the menu bar, is somewhat narrower than the screen, and is centered between the left and right edges of the screen. It may contain any of the following:

- informative or instructional text
- rectangles in which text may be entered (initially blank or containing default text that can be edited)
- controls of any kind
- graphics (icons or QuickDraw II pictures)
- anything else your application wants

The user provides the necessary information in the dialog box, for example by entering text or clicking a check box. There's usually a button labeled *OK* to tell the application to accept the information provided and perform the command, and a button labeled *Cancel* to cancel the command as though it had never been given (retracting all actions since its invocation). Some dialog boxes may use a more descriptive word than *OK;* for simplicity, we'll refer to the button as the *OK button.* There may even be more than one button that will perform the command, each in a different way.

Modal or modeless

Most dialog boxes require the user to respond before doing anything else. Clicking a button to perform or cancel the command makes the box go away; clicking outside the dialog box only causes a beep from the speaker. This type of box is called a **modal dialog box** because it puts the user in the state (or *mode*) of being able to work only inside the dialog box. Figure 4-9 is an example of how a modal dialog box might look; note that it has no close box.

One of the buttons in a modal dialog box may be boldly outlined; it is called the OK button (whatever text it may contain). Pressing the Return key has the same effect as clicking the OK button; it should initiate the preferred (or safest) action in the current situation. If there's no boldly outlined button, pressing the Return key has no effect. A Cancel button, if present, closes the dialog box and cancels the effects of all work done while the box was open.

Other dialog boxes do not require the user to respond before doing anything else. They are called **modeless dialog boxes.** The user can work in a document window on the desktop between clicking buttons in a modeless dialog box. Modeless dialog boxes can be set up to respond to the standard editing commands in the Edit menu. Clicking a button in a modeless dialog box does not make the box go away; it stays on the desktop so that the user can perform the command again.

As shown in Figure 4-10, a modeless dialog box typically looks like a document window. It can be moved, made inactive and active again, or closed like any document window. When you're finished with the command and want the box to go away, you can click its close box, or you can choose Close from the File menu if the dialog box is the active window.

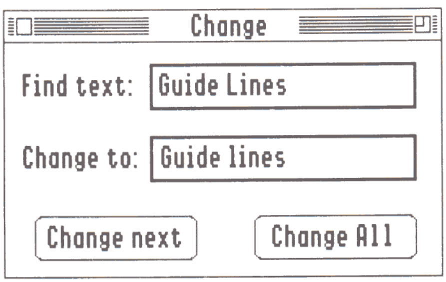


Figure 4-10
A modeless dialog box

Update routines are described under "Creating Windows," earlier in this chapter

ShowPleaseWait and HidePleaseWait are in the source file DIALOG.PAS.

Some dialog boxes may in fact require no response at all. For example, while an application is performing a time-consuming process, it can display a dialog box that contains only a message telling what it's doing; then, when the process is complete, the application can simply remove the dialog box. HodgePodge does this with the `ShowPleaseWait` and `HidePleaseWait` routines, called up during tool initialization. `ShowPleaseWait` also demonstrates how to bring up a dialog box and show it immediately, without even waiting for an update event to trigger its display. It does this by having its own little update routine:

```
procedure ShowPleaseWait;                          {begin ShowPleaseWait...}

var     r          : Rect;                         {rectangle to display dialog in}
        origPort   : GrafPortPtr;                  {common variable with HidePleaseWait}
        msgWindPtr: GrafPortPtr;                   {common variable with HidePleaseWait}

begin
   origPort   := GetPort;                          {Save the current GrafPort}
   msgWindPtr :=                                   {Open the dialog, with the template...}
       GetNewModalDialog(@PlsWtTemp);              {...created in InitGlobals}
   SetRect(r,70,19,640,200);                       {Define rectangle dimensions}
   NewDItem (                                      {Create an item for the dialog:}
       msgWindPtr,1502,r,15,                       {...with these parameters...}
       @'Please wait while we set things up.',     {...displaying this string...}
       0,0,Pointer(0));                            {...and with these other parameters}
   BeginUpdate (msgWindPtr);                       {Treat this like an update...}
   DrawDialog  (msgWindPtr);                       {...manually draw the dialog...}
   EndUpdate   (msgWindPtr);                       {...and end the update-handling}
end;                                               {End of ShowPleaseWait}


procedure HidePleaseWait;                          {begin HidePleaseWait...}

begin
   CloseDialog (msgWindPtr);                       {Remove dialog from the screen}
   SetPort     (origPort);                         {Restore the original GrafPort}
end;                                               {End of HidePleaseWait}
```

Figure 4-11 shows what the dialog box created by `ShowPleaseWait` looks like. It is a *message* dialog box because it requires no response from the user, and disappears on its own when no longer needed.

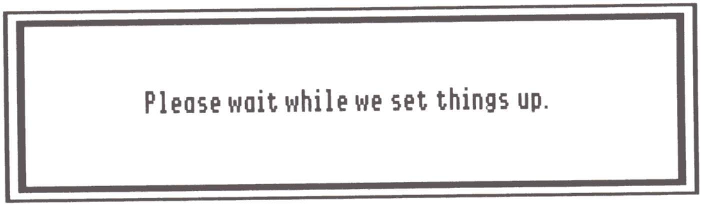


Figure 4-11
HodgePodge message dialog box

Alerts

With alerts, your applications have a standardized way to report errors or give warnings. An **alert box** is similar to a modal dialog box, but appears only when something has gone wrong or must be brought to the user's attention. The alert box is usually placed slightly farther below the menu bar than a dialog box. To help the user who isn't sure how to proceed when an alert box appears, the preferred button to use in the current situation is doubly outlined so that it stands out from the other buttons in the alert box. The outlined button is the alert box's default button; if the user presses the Return key, the effect is the same as clicking this button.

There are three standard kinds of alerts—Stop, Note, and Caution—each indicated by a particular icon in the upper-left corner of the alert box. Figure 4-12 illustrates a Stop alert. You can put anything you like in the upper-left corner of an alert, including blank space.

The alert mechanism also provides another type of signal: sound from the speaker. The application can base its response on the number of consecutive times an alert occurs; the first time, it might simply beep, and thereafter it may present an alert box. The sound isn't limited to a single beep but may be any sequence of tones, and may occur either alone or along with an alert box. As an error is repeated, there can also be a change in which button is the default button (perhaps from OK to Cancel). You can specify different responses for up to four occurrences (*stages*) of the same alert.

If you want to write a custom alert sound procedure, see "Miscellaneous Tool Set" and "Sound Tool Set" in the *Apple IIGS Toolbox Reference*.

HodgePodge's main error handler, `CheckDiskError`, is an example of a routine that puts up a Stop alert (Figure 4-12). The exact message displayed depends on the particular error that occurred. `CheckDiskError` is listed and described under "Error Handling" in Appendix D. Some of its features are described under "Item Lists," later in this section.

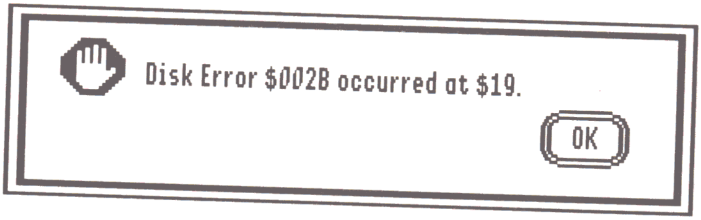


Figure 4-12
HodgePodge Stop alert

Dialog and alert windows

A dialog box appears in a dialog window. When you call a Dialog Manager routine to create a dialog, you supply the same kind of information as when you create a window with a Window Manager routine. You can manipulate a modeless dialog window with Window Manager or QuickDraw routines, just like any other window—showing it, hiding it, moving it, or changing its size and plane, for example. If you want clipping to occur, you can set the dialog box GrafPort's clipping region with QuickDraw calls.

An alert box appears in an alert window. You don't have the same flexibility in defining and manipulating an alert window, however. The Dialog Manager chooses the window definition procedure, 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. You don't specify the alert window's plane; it always comes up in front of all other windows. Because an alert box requires the user to respond before doing anything else, and the response makes the box go away, the application doesn't manipulate the alert window.

Dialog records

To create a dialog, you pass information to the Dialog Manager, with which it creates a *dialog record*. The dialog record contains the window record for the dialog window, a handle to the dialog's item list, and some additional fields. The Dialog Manager creates the dialog window by calling the Window Manager.

The Dialog Manager passes to your application a pointer to the dialog port, which you use thereafter to refer to the dialog in Dialog Manager routines or even in Window Manager or QuickDraw II routines. The dialog pointer is equivalent to the window pointer for the dialog box. It is not a pointer to the dialog record or even to the window record. It is a pointer to the GrafPort record only.

You can do all the necessary operations on a dialog without accessing the fields of the dialog record directly. To get or change information about an item in a dialog, you pass the dialog pointer and the *item ID* to a Dialog Manager routine. You'll rarely access information directly through the handle to the item.

Item ID's are discussed in this section, under "Item Lists."

Items

A dialog box or alert is a window with *items*. To create a dialog box or an alert, the Dialog Manager needs to know what items the window contains. It also needs to know the following information for each item:

- The *item type*. This includes not only whether the item is a standard control, editable text, or other type, but also whether it is **enabled.**
- A *display rectangle*, which determines the location of the item within the dialog or alert box.
- An *item ID* number uniquely identifying the item in the dialog. All subsequent Dialog Manager calls referring to that item will need its ID number.
- Other information specific to certain types of items, such as the item's title, its initial value, its colors, its orientation, and whether it is visible or invisible.

If an item is **enabled**, the Dialog Manager notifies the application whenever the user selects the item.

Item type

Only a few types of items normally appear in dialog boxes and alerts; Figure 4-13 shows most of them. Item types are specified by predefined constants or combinations of constants. See "Dialog Manager" in the *Apple IIGS Toolbox Reference* for more details.

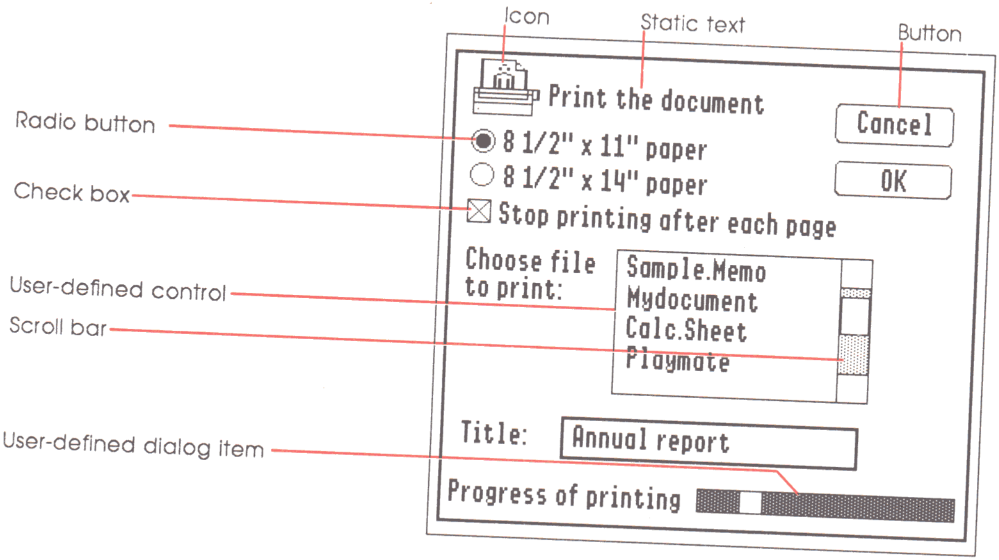


Figure 4-13
Dialog item types

An editable text item (predefined constant = `editLine`) initially may be empty or it may have default text. Text entry and editing is handled by the LineEdit Tool Set, described later in this section.

If the predefined constant `itemDisable` is specified for an item, the Dialog Manager ignores events involving that item. For example, if you want to prevent the user temporarily from manipulating an item, you can disable it.

Important Some dialog items are also controls. Disabling an item is not quite the same as making a control *inactive* with the Control Manager procedure HiliteControl. An inactive control is dimmed; a disabled item is unchanged in appearance.

You can make an item invisible if you want. This technique can be useful, for example, if your application needs to display a number of similar dialog boxes with one item missing or different in some of them.

Display rectangle

Each item in the item list is displayed within its *display rectangle:*

- For standard controls, scroll bars and user controls, the display rectangle becomes the control's enclosing rectangle.
- For an editable text item, it becomes LineEdit's *view rectangle.* The text is clipped (not drawn) wherever it extends beyond the rectangle. In addition, the Dialog Manager uses QuickDraw II to draw a bordering rectangle outside the display rectangle.
- Static text items are displayed in generally the same way as editable text items, except that a rectangle isn't drawn outside the display rectangle. Also, there are three different formats for static text.
- Icons are aligned with the display rectangle's origin.

The view rectangle and other aspects of LineEdit are described under "LineEdit Tool Set" in the *Apple IIGS Toolbox Reference.*

❖ *Note:* Clicking anywhere within the display rectangle is considered a click in that item. If display rectangles overlap, a click in the overlapping area is considered a click in whichever item comes first in the item list.

Item ID

Each item in an item list is identified by an **item ID,** a unique number within the list. By convention, the OK button in an alert's item list should have an ID of 1 and the Cancel button should have an ID of 2. The Dialog Manager provides predefined constants equal to the item ID for OK and Cancel, as follows:

```
ok     =  1
cancel =  2
```

In a modal dialog's item list, the item whose ID is 1 is assumed to be the dialog's default button (unless specified otherwise); 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 the Apple Human Interface Guidelines, the Dialog Manager automatically outlines the default button in bold, unless there is no default button (that is, no button item with ID 1).

❖ *Note:* If you don't want a default button, do not create any item with an ID of 1.

Example

`MakeATemplate` is a routine called by `CheckDiskError` (described earlier and listed in Appendix D) in order to fill in the dialog record and the item list for the HodgePodge stop alert shown in Figure 4-12. `MakeATemplate` describes the basic alert box, including what is to happen at each stage, and defines two items: an OK button for the user to click, and a static text item that contains the error message.

MakeATemplate is in the source file DIALOG.PAS.

```
procedure MakeATemplate ( TheTemplate:
                           AlertTempPtr;
                           TheStr: StringPtr);     {begin MakeATemplate...}

var    currentItem1: ItemTemplate;                 {toolbox-defined structure}
       currentItem2: ItemTemplate;

begin
   with TheTemplate^ do                            {First define alert box:}
     begin
      SetRect (atBoundsRect,120,30,520,80);        {bounding rectangle for alert}
      atAlertID := 1500;
      atStage1  := $80;
      atStage2  := $80;                            {at each stage, make alert...}
      atStage3  := $80;                            {...visible but silent}
      atStage4  := $80;
      atItem1   := @currentItem1;                  {ptr to first item's template}
      atItem2   := @currentItem2;                  {ptr to 2nd item's template}
      atItem3   := NIL;                            {terminates item list}
     end;                                          {end of defining box template}

   with currentItem1 do                            {Now define item 1:}
     begin
      itemID     := 1;                             {item #1 = default item}
      SetRect (itemRect,320,25,0,0);               {display rectangle}
      itemType   := 10;                            {it's a buttton item}
      itemDescr  := @'OK';                         {text in button}
      itemValue  := 0;                             {initial value = 0}
      itemFlag   := 0;                             {=default style}
      itemColor  := NIL;                           {no color}
     end;                                          {end of item 1}

   with currentItem2 do                            {Now define item 2:}
     begin
      itemID     := 2;
      SetRect (itemRect,72,11,639,199);            {display rectangle}
      itemType   := 15 + $8000;                    {disabled static text}
      itemDescr  := Pointer (TheStr);              {the string passed to this routine}
      itemValue  := 0;                             {no initial value}
      itemFlag   := 0;                             {default style}
      itemColor  := NIL;                           {no color}
     end;                                          {end of item 2}
   end;                                            {End of MakeATemplate}
```

Using dialogs

In most cases, you probably won't have to make any changes to the dialogs from the way they're defined at their creation. However, there are calls to modify items, move controls, or change text. If you want the font in your dialog and alert windows to be something other than the system font, call SetDAFont to change the font.

To handle events in a modal dialog, call the routine ModalDialog after putting up the dialog box. If your application includes any modeless dialog boxes, they're a bit more complex to handle; part of your event-handling will include determining whether events need to be handled as part of the dialog box. You can support the use of the standard cut, copy, paste, and delete editing commands in a modeless dialog box.

You can substitute text in static text items with text that you specify in the ParamText routine. This means, for example, that a document name supplied by the user can appear in an error message.

Editing text with LineEdit

To provide simple text-editing capabilities needed for dialog boxes and other general purposes, the Apple IIGS Toolbox includes the LineEdit Tool Set. The routines in LineEdit provide basic text-editing capabilities that follow the Apple Human Interface Guidelines. These capabilities include

- inserting new text
- deleting characters that are backspaced over
- translating mouse activity or arrow keys into text selection
- deleting selected text and possibly inserting it elsewhere
- copying selected text without deleting it

The desk scrap is described under "Supporting Other Desktop Features" in Chapter 5.

LineEdit uses inverse highlighting to show the current text selection, or a blinking vertical bar to show the insertion point. LineEdit places cut or copied text into the LineEdit scrap—different from the *desk scrap*.

LineEdit is not a complete text editor. It does not support

- more than 256 characters per line (except when using LETextBox or LETextBox2)
- fully justified text; that is, text aligned with both the left and right margins (except when using LETextBox2)
- automatic word wrap (except when using LETextBox2)
- scrolling
- fonts that kern characters
- more than one font or stylistic variation per line (except when using LETextBox2)
- "intelligent" cut and paste (adjusting spaces between words during cutting and pasting)
- tabs

The Dialog Manager automatically handles editing of text in dialog boxes by making calls to LineEdit. If you wish to use LineEdit yourself in other situations, see "LineEdit Tool Set" in the *Apple IIGS Toolbox Reference.*

Dialog summary: HodgePodge's "About..." box

DoAboutItem is in the source file DIALOG.PAS.

`DoAboutItem` is the subroutine that displays the "About HodgePodge" dialog box; it's called when the user selects the first entry in the Apple menu. This subroutine shows how to create a dialog box in an atypical way: in-line in your code, rather than by calling up templates. When you create a dialog box in-line, you invoke it with the call *NewModalDialog,* rather than the more common call *GetNewModalDialog,* used by the HodgePodge routine `ShowPleaseWait` (described earlier).

The routine starts out by accessing and allocating space for the Apple icon we want to display in the box. It then defines an OK button for the user to click. Finally, it draws the text items in the box.

```
procedure DoAboutItem;                              {begin DoAboutItem…}

var     aboutDlog : GrafPortPtr;                    {pointer to this dialog}
        r         : Rect;
        itemHit   : Integer;                        {item selected by user}
        appleIconP: Ptr;                            {pointer and handle to the Apple…}
        appleIconH: Handle;                         {…icon created in InitGlobals}

begin
   SetRect (r,146,20,495,192);                      {= rectangle the dialog appears in}
   aboutDlog := NewModalDialog(r,TRUE,0);           {Open the dialog: in rectangle r,
                                                     visible, no reference value}
   SetRect (r,270,153,0,0);                         {Define a display rectangle and…}
   NewDItem(aboutDlog,1,r,ButtonItem,               {…make a dialog item for it:…}
            @'OK',0,0,NIL);                         {…the OK button}

   SetRect (r,20,135,0,0);                          {Define another display rectangle…}
   appleIconP := @AppleIcon;
   appleIconH := @AppleIconP;                       {…get a handle to the Apple icon…}
   NewDItem(aboutDlog,3,r,
            iconItem+itemDisable,
            AppleIconH,0,0,NIL);                    {make it a disabled icon item}

                                                    {For the rest of the text, simply
                                                     write it directly in the port,
                                                     rather than creating dialog items}

   SetPort (aboutDlog);                             {make sure this is the active port}
   SetForeColor (0);                                {foreground color = black}
   SetBackColor (15);                               {background color = white}

   MoveTo (40,17);                                  {move the pen to starting position…}
   SetTextFace(8);                                  {(change to outline text)}
   DrawString
      ('                 HodgePodge');              {Draw the first line…}
   SetTextFace (0);                                 {(go back to plain text)}
   MoveTo (40,27);                                  {Move to next line and continue…}
   DrawString
      ('      A potpourri of routines that');
   MoveTo (40,37);
   DrawString
      ('      demonstrate many features of');
   MoveTo (40,47);
   DrawString
      ('            the Apple IIGS Tools.');
   MoveTo (40,67);
   DrawString
      ('      By the Apple IIGS Development Team');
   MoveTo (36,77);
   DrawString
      ('Translated to TML Pascal by TML Systems');
```

```
    MoveTo (40,87);
    DrawString
        ('      Copyright Apple Computer, Inc.');
    MoveTo (40,117);
    DrawString
        ('      1986-87, All rights reserved');
    MoveTo (40,127);
    DrawString
        ('                v4.0',  23-Sep-87');

    itemHit := ModalDialog(NIL);        {call ModalDialog; it returns when
                                         any enabled item is selected}
    CloseDialog(aboutDlog);             {Close the Dialog when OK clicked}

end;                                    {End of DoAboutItem}
```

Figure 4-14 shows what the dialog box constructed by this routine looks like (the assembly-language and C versions have slightly different text from the Pascal example).

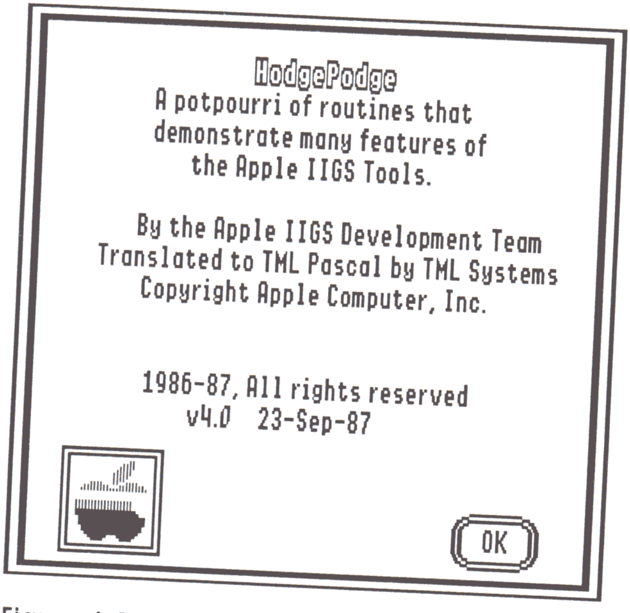


Figure 4-14
The "About HodgePodge..." dialog box

Chapter 5

Using the Toolbox (III)

This chapter concludes our brief discussion of the Apple IIGS Toolbox. The tool sets described here can help you accomplish these tasks:

- creating menus
- supporting other desktop features such as desk accessories and cut-and-paste
- accessing external devices and files
- generating and playing sounds
- performing mathematical computations
- controlling parts of the Apple IIGS operating environment

Making and modifying menus

Pull-down menus are an important part of the desktop environment. Menus allow users to examine all choices available to them at any time without being forced to choose one of them, and without having to remember command words or special keys.

The Menu Manager is the Apple IIGS tool set that supports menus of the style recommended by the Apple Human Interface Guidelines. The user displays a menu by positioning the cursor in the *menu bar* and pressing the mouse button over a *menu title*. The Menu Manager highlights the selected title (by redrawing it in inverted colors) and "pulls down" the menu below it. As long as the mouse button is held down, the menu is displayed. Dragging through the menu causes each of its *menu items* (commands) to be highlighted in turn. If the mouse button is released over an item, that item is considered chosen. The item blinks briefly to confirm the choice, and the menu disappears.

When the user chooses an item, the Menu Manager tells the application which item was chosen, and the application performs the corresponding action. When the application completes the action, it removes the highlighting from the menu title, indicating to the user that the operation is complete.

If the user moves the cursor out of the menu with the mouse button held down, the menu remains visible, though no menu items are highlighted. If the mouse button is released outside the menu, no choice is made; the menu just disappears and the application takes no action. *The user can always look at a menu without causing any changes in the document or on the screen.*

Menu bars

A **menu bar** is an outlined rectangle that holds the titles of all the menus associated with the bar. A menu in the bar may be enabled or temporarily disabled. A **disabled menu** can still be pulled down, but its title and all the items in it are dimmed and not selectable.

The principal menu bar is the **system menu bar;** see Figure 5-1. There can only be one system menu bar on the screen at one time. The system menu bar always appears at the top of the Apple IIGS screen; nothing but the cursor ever appears in front of it. In applications that support desk accessories, the first (leftmost) menu should be the desk accessory menu (also called *Apple menu,* the menu whose title is a colored apple symbol). The desk accessory menu contains the names of all available desk accessories, and usually the name of a dialog box that gives brief information about the application itself. When the user chooses a desk accessory from the menu, the title of the menu belonging to the desk accessory may appear in the menu bar for as long as the accessory is active, or the entire menu bar may be replaced by menus belonging to the desk accessory.

All applications should support desk accessories. See "Supporting Other Desktop Features," later in this chapter.

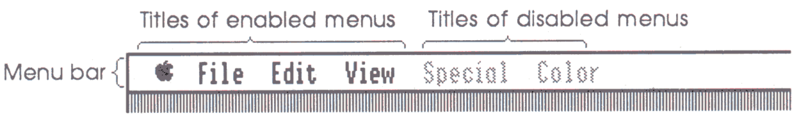


Figure 5-1
The system menu bar

In addition to the system menu bar, your application can have various **window menu bars.** These can appear anywhere on the screen and in windows. Window menu bars are provided to give you more menu space, particularly because of the limited resolution in 320 mode. Window menu bars should be used moderately, if at all.

Window menu bars are described under "Menu Manager" in the *Apple IIGS Toolbox Reference.*

A shadowed rectangle is one that appears to have a thin shadow just below and to the right of it, making it appear to stand out slightly from the desktop.

Menu appearance

A standard menu consists of a number of menu items listed vertically inside a *shadowed rectangle*. Items on a menu may be the text of a command, a solid color, or just a line dividing groups of choices. Menus always appear in front of everything except the cursor. Figure 5-2 shows a menu with six items, including two dividing lines.

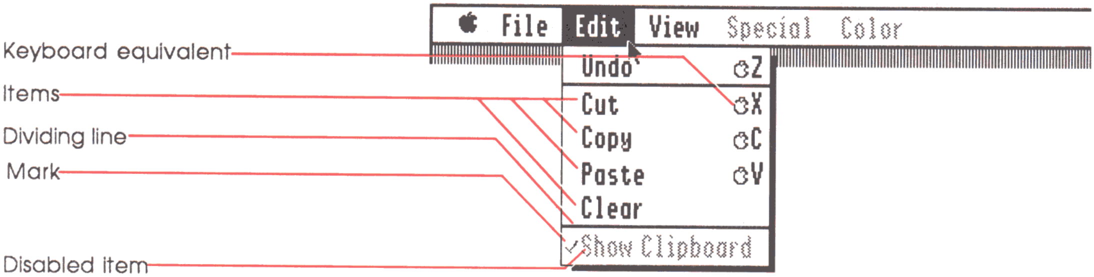


Figure 5-2
A standard menu

Figure 5-2 shows some of the typical variations in an item's appearance:

- A mark may appear on the left side of the item, to denote the status of the item or of the mode it controls.
- An Apple logo followed by a capital letter may appear to the right of the item, to show that the item may be invoked from the keyboard (that is, it has a *keyboard equivalent*). If the user presses the letter key while holding down the Apple key, the menu item is invoked just as if it had been chosen from the menu.
- Each item's text may have its own text style.
- An item can be dimmed to indicate that it is disabled and can't be chosen.
- A dividing line is a separate menu item. Dividing lines should always be disabled.

See "Menu Manager" in the *Apple IIGS Toolbox Reference* for information on how to create custom menus.

If a standard menu doesn't suit your needs—for example, if you want more graphics, or perhaps a nonlinear text arrangement—you can write a custom menu definition procedure. The Menu Manager will call that procedure when it draws the menu. The custom menu can be visibly very different, and yet respond to your application's Menu Manager calls just like a standard menu. The items in the menu can have any appearance.

Keyboard equivalents

Your program can set up a **keyboard equivalent** for any of its menu commands in order to allow the user to invoke the command from the keyboard. The character you specify for a keyboard equivalent should be a letter that the user can type in either uppercase or lowercase. For example, typing either "G" or "g" while holding down the Apple key invokes the command whose equivalent is "Ć G."

❖ *Note:* For consistency among applications, you should specify the letter in uppercase in the menu.

Constructing menus

It's simple to construct your application's menus. All you need to do is define the text of the menu titles and items, and assign ID numbers to each menu title and item.

❖ *Note:* The menu bar does not allow for a large number of menus or menus with lengthy titles. If you're having trouble fitting your menus into the menu bar, you should review their organization and titles. Furthermore, if your program is likely to be translated into other languages, remember that translated menu titles may take up more space.

Menu lines and item lines

You create menus by constructing a list of menu and item lines, and passing a pointer to that list to the NewMenu routine. NewMenu parses the menu and item lines, allocates enough memory for necessary records, and initializes those records. The menu and item lines must remain in memory as long as the menu exists.

The list must follow a specific syntax; here is an example:

```
>>Title 1\N1
--Item string 1\N256
--Item string 2\N257
--Item string 3\N258
.
```

This is a simple list of one menu line and three item lines. The first character on the first line is the *title character;* it denotes the start of a menu. The first character on any line other than a title line is the *item character;* it denotes an item in the menu. The second character in each line can be anything (it is changed by the Menu Manager)—here it just repeats the first character. Each line is terminated by a return (decimal 13) or a null byte (0). Finally, a termination character, different from the menu and item character, denotes the end of the list.

In the example above, ">" is the title character, "-" is the item character, and a period is the termination character. But you may use any characters, as long as the title and item characters are different, and the termination character is different from the item character. (Thus, the title and termination character may be the same.)

For a complete discussion of menu- and item-line syntax, including a description of all special characters, see "Menu Manager" in the *Apple IIGS Toolbox Reference.*

Before the terminating character of each line, "N" followed by a number specifies the menu and menu item ID number.

For an example of menu and item lines using multiple special characters and different title, item, and terminating characters, see the HodgePodge source code listing of `InitGlobals`, under "Start the Program" in Chapter 2. In `InitGlobals` the title character is ">", the item character is "=", and the termination character is a period. The second character in each line repeats the first. You can see from the listing that, depending on how you want your menus to appear, the syntax can be quite complex.

Using just the "@" symbol in a title provides the Apple logo. The @ must follow the character denoting a menu title, and then be followed by an end-of-line mark (carriage return). Do not place a space before or after the @, as you must with other menu titles. See the `InitGlobals` example.

Menu and item ID numbers

ID numbers are assigned in the menu/item line list. The ID numbers must be allocated as shown in Table 5-1.

Important A Menu ID must be unique for each menu; that is, no two menus can have the same ID. Similarly, no two items, whether in the same or separate menus, can have the same Item ID.

Table 5-1
Menu ID number assignment

Hexadecimal	Decimal	Meaning
Menu ID numbers		
$0000	0	Internal use, generally means front, or first menu in bar.
$0001-$FFFE	1-65534	Reserved for application use.
$FFFF	65535	Internal use, generally means end, or last menu in bar.
Item ID numbers		
$0000	0	Internal use, generally means front, or first item in menu.
$0001 - $00F9	1-249	Reserved for desk accessory items.
$00FA	250	Undo edit item.
$00FB	251	Cut edit item.
$00FC	252	Copy edit item.
$00FD	253	Paste edit item.
$00FE	254	Clear edit item.
$00FF	255	Close command item.
$0100 - $FFFE	256-65534	Reserved for application use.
$FFFF	65535	Internal use, generally means end, or last item in menu.

HodgePodge uses symbolic constants for menu ID numbers in its menu- and item-line definitions. It assigns menu ID's to those constants in the file GLOBALS.PAS, as follows:

```
AppleMenuID        = 300;
  AboutItem        = 301;
FileMenuID         = 400;
  OpenItem         = 401;
  CloseItem        = 255;          {reserved ID number}
  SaveAsItem       = 403;
  ChoosePItem      = 405;
  PageSetItem      = 406;
  PrintItem        = 407;
  QuitItem         = 409;
EditMenuID         = 500;
  UndoItem         = 250;          {reserved ID number}
  CutItem          = 251;          {reserved ID number}
  CopyItem         = 252;          {reserved ID number}
  PasteItem        = 253;          {reserved ID number}
  ClearItem        = 254;          {reserved ID number}
WindowsMenuID      = 600;
  NoWindowsItem    = 601;
  FirstWindItem = 2000;            {window menu ID's are allocated
                                    dynamically starting at 2000}
FontsMenuID        = 700;
  FontItem         = 701;
  MonoItem         = 702;
```

How HodgePodge sets up the menu bar when the program executes is demonstrated in Chapter 2.

Accepting user input

How your application responds to menu selections made by the user depends largely on whether or not the application calls TaskMaster.

Without TaskMaster, an application typically calls GetNextEvent each time through the event loop. If the user selects a menu item with the mouse, a mouse-down event occurs and the application responds as follows:

1. It calls FindWindow, which (in this case) returns to the application the information that the mouse button was pressed in the menu bar.

Tracking means following changes in cursor position betwen the time a mouse button is pressed and when it is released. That way a user's selection is not finalized until the mouse button is released.

2. It then calls MenuSelect, which **tracks** the mouse, opening menus and highlighting selections until the mouse button is released. If it is released in a menu selection, MenuSelect returns to the application the number of the menu and the number of the item in the menu that was selected. It also highlights the menu's title.
3. The application then branches to the subroutine that handles the menu item selected.
4. When the task is completed, the application unhighlights the menu title and continues in the main event loop.

❖ *Keyboard equivalent:* If the menu item was selected with its equivalent keystroke combination rather than with the mouse, a key-down event occurs. The application must look at the modifiers field of the event record to know that the Apple key was pressed at the same time, meaning a menu selection has been made. The application then highlights the menu title and proceeds from step 3 (above).

On the other hand, if your application calls TaskMaster instead of GetNextEvent each time through the event loop, most of the above procedure is handled automatically. For both mouse-down and key-down events, TaskMaster takes care of finding out whether they represent menu-selection actions. If the user selects a menu item with the mouse or with a keyboard-equivalent, TaskMaster highlights the menu and returns a task code of `wInMenuBar` (meaning a menu selection was made). Your application can examine the *taskData* field of the extended task event record to see which item in which menu was selected. Then it can branch directly to the appropriate subroutine.

DoMenu is listed and described under "Handle Specific Events" in Chapter 2.

❖ *HodgePodge:* HodgePodge uses TaskMaster. After receiving a `wInMenuBar` task code from TaskMaster, HodgePodge jumps to its menu-event dispatcher, `DoMenu`. `DoMenu` gets the individual menu item's ID number from the `Event.taskData` field of the extended event record, and jumps to the proper subroutine.

Modifying menus during execution

AddToMenu is in the source file MENU.PAS.

If your menu bar, or items in a menu, are going to change while on the screen, you can use Menu Manager calls to rearrange the menus and items. In the routine `AddToMenu` (called from the routine `DoMenuItem`), HodgePodge adds a new item to the Windows menu when a new window is opened on the desktop. `AddToMenu` does this principally by calling InsertMItem and DeleteMItem. AddToMenu also adjusts the *window list*—a list of pointers to all open windows.

```
procedure AddToMenu;                                   {begin AddToMenu...}

var    theWindow   : GrafPortPtr;
       myDataHandle: WindDataH;                        {window-data-record handle}

begin
   theWindow := FrontWindow;                           {Get a pointer to the front window...}
   windowList[wIndex] := theWindow;                    {add the pointer to the window list}
   myDataHandle := WindDataH(                          {...then get a handle to its...}
                    GetWRefCon(theWindow));            {...window-data record, to get name}

   InsertMItem(@myDataHandle^^.menuStr[1],             {Insert window's name at the end...}
                    $FFFF,WindowsMenuID);              {...of the Windows menu}

   if wIndex = 0 then                                  {If this is the first open window...}
     begin
       DeleteMItem(NoWindowsItem);                     {...remove "No Windows Allocated" item}
       SetMenuFlag($FF7F,WindowsMenuID);               {...enable the Windows menu...}
       DrawMenuBar;                                    {...and draw it}
     end;

   CalcMenuSize(0,0,WindowsMenuID);                    {Recalculate the size of the menu}
   Inc(wIndex);                                        {Increment the number of open windows}
end;                                                   {End of AddToMenu}
```

The above example shows how HodgePodge adds items to a menu. On the other hand, when windows are removed from the desktop, HodgePodge *deletes* the corresponding menu item with code in the routine `AdjWind`. `AdjWind` is called from `DoCloseItem` when the user selects Close from the File menu or when the user clicks the close box of the frontmost window.

AdjWind is in the source file WINDOW.PAS.

`AdjWind` makes the menu-related calls InsertMItem, DeleteMItem and CalcMenuSize. It also adjusts the window list to reflect the fact that a window has been removed.

```
procedure AdjWind (TheWindow: GrafPortPtr);         {begin AdjWind...}

var    i      : Integer;
       theOne : Integer;

begin
  i := FirstWind;                                   {start with menu ID of 1st window}
  while windowList[i] <> TheWindow do               {...and run through the window list}
    Inc (i);
  theOne:=i;                                        {...to get this window's position.}

  if wIndex = 1 then                                {If we're closing the LAST window...}
    begin
     InsertMItem(@noWindStr[1],                     {...reinsert "No Windows Allocated"...}
                 FirstWindItem+theOne,              {...after this item...}
                 WindowsMenuID);                    {...in the Windows menu.}
     SetMenuFlag($0080,WindowsMenuID);              {...disable the Windows menu...}
     DrawMenuBar;                                   {...redraw the menu bar...}
     wXoffset := 20;                                {...and reinitialize the position...}
     wYoffset := 12;                                {...of the next-opened window}
    end;                                            {end of IF its the last window}

  DeleteMItem(firstWindItem+theOne);                {Delete item on the Windows menu...}
  CalcMenuSize(0,0,WindowsMenuID);                  {...and recalculate size of the menu}

  Inc (i);                                          {Now go to the next window on list}
  while i < lastWind do                             {...and for each window in turn...}
    begin                                           {move it down one position...}
     windowList[i-1] :=windowList[i];               {...in the window list}
     Inc (i);
    end;

  for i := theOne to lastWind do                    {now renumber items in Windows menu:}
    SetMItemID (firstWindItem+i-1,                  {its new ID number}
                firstWindItem+i);                   {its old ID number}

end;                                                {End of AdjWind}
```

❖ *Note:* `AdjWind` performs some rather complex manipulations of pointer lists and menu IDs. Your program can easily remove menu items without going through such acrobatics if menu item IDs are not going to change and if menu changes do not require adjustment of other lists in memory.

Supporting other desktop features

Two other important desktop-programming features have tool sets that support them. The *Desk Manager* controls desk accessories (called from the Apple menu) and the *Scrap Manager* handles cutting, copying, and pasting from the Edit menu.

Desk accessories

Any application you write should support desk accessories. **Desk accessories** are short programs such as clock displays, note pads, and calculators that a user might want to access without having to leave your program. Desk accessory support is a convenience for the user, it enhances the multitasking feel of the desktop, and it is consistent with the aims of the Human Interface Guidelines. Furthermore, it's easy to include in your programs.

The Desk Manager is the tool set that enables your application to support desk accessories. There are two types of desk accessories in the Apple IIGS environment: classic desk accessories and new desk accessories.

- **Classic desk accessories** (CDA's) are desk accessories designed to function in a non-desktop, non-event-driven environment. Unlike new desk accessories, a CDA gets full control of the computer during what is basically an interrupt state (generated by a keypress). The desk accessory is responsible for saving and restoring any of the application's memory that it uses, as well as handling all I/O. The Control Panel is a classic desk accessory.
- **New desk accessories** (NDA's) are desk accessories designed to execute in a desktop, event-driven environment. NDA's run in a window and get control when that window is the frontmost window.

Just what kind of control an NDA exercises is described under "Desk Manager" in the *Apple IIGS Toolbox Reference*.

❖ *Macintosh Programmers:* New desk accessories are the style of desk accessories available on the Macintosh.

Supporting classic desk accessories

A user activates a classic desk accessory from the CDA menu. The CDA menu is displayed by pressing Apple-Control-Escape at any time during program execution. Two CDA's are built into the system:

- Control Panel
- Alternate Display Mode

Any others (up to eleven) are loaded from disk. From the CDA menu, a user can select any of the CDA's currently in the system. The desk accessory selected is activated and retains control until it shuts down. When it shuts down, the Desk Manager redisplays the CDA menu. Only when the user selects Quit from the CDA menu does the original application resume operation.

See "Controlling the Apple IIGS Operating Environment" in this chapter, and "The Scheduler" in the *Apple IIGS Toolbox Reference* for more information on the Busy flag.

When can the CDA menu be displayed? The Desk Manager gets control in two ways. If the Event Manager is active, the Desk Manager is called in conjunction with GetNextEvent. If the Event Manager is not active, the Desk Manager gets control whenever the user presses Apple-Control-Escape, which generates an interrupt. Before the Desk Manager displays the CDA menu, it checks the system **Busy flag.** If something in the system is busy, the Desk Manager waits until the Busy flag is cleared, then "wakes up" and displays the CDA menu. This guarantees that CDA's have all system resources available to them when they are called.

If you want to *write* a classic desk accessory (CDA), see Chapter 8 of this book.

The only thing your application needs to do to support classic desk accessories is make sure that interrupts are not disabled for long periods.

Supporting new desk accessories

New desk accessories are loaded automatically by the operating system at boot time. An application that wants to make NDA's available to the user does not have to do a lot of work, particularly if the application is using the Window Manager routine TaskMaster. By convention, however, desk accessories can assume that the following tool sets are already available for them to use, so the application must make sure that they are loaded and started up:

- Tool Locator
- Memory Manager
- Miscellaneous Tool Set
- QuickDraw II
- Event Manager

- Window Manager
- Control Manager
- Menu Manager
- LineEdit Tool Set
- Dialog Manager
- Scrap Manager

With TaskMaster: If the Application uses TaskMaster, it only needs to make three calls to support new desk accessories after it has loaded and started up the proper tool sets:

- DeskStartup—to initialize the Desk Manager
- FixAppleMenu—to add to the list of NDA's in the Apple menu
- DeskShutdown—to shut down the Desk Manager before the other tool sets are shut down

After the FixAppleMenu call has been made, TaskMaster automatically handles opening NDA's in response to menu selections, calling SystemTask and SystemClick when appropriate. If the application sets up the menu items correctly, TaskMaster can even call SystemEdit when the user selects an item from the Edit menu, or close a desk accessory in response to the user's selecting Close from the File menu.

❖ *HodgePodge:* The three calls listed above are in the routines `StartUpTools`, `SetUpMenus`, and `ShutDownTools`.

Without TaskMaster: Applications that do not use TaskMaster must take the following steps to support new desk accessories.

1. Call DeskStartup to start up the Desk Manager.
2. Call FixAppleMenu to add to the list of NDA's in the Apple menu.
3. Call OpenNDA when the user selects an NDA from the Apple menu.
4. Call SystemTask frequently (at least every time through the event loop).
5. Call SystemClick when a mouse-down event occurs in a system window.
6. Call SystemEdit when a desk accessory is active and the user selects Undo, Cut, Copy, Paste, or Clear from the Edit menu.
7. Close the desk accessory when the user selects Close from the File menu. You can use CloseNDA or CloseNDAbyWinPtr to do this.
8. Call DeskShutdown to shut down the Desk Manager.

If you want to *write* a new desk accessory (NDA), see Chapter 8 of this book.

Cutting and pasting

An important part of the convenience provided by desktop applications is the ability they give the user to transfer and copy fragments of text or graphics within a document, or from one document to another.

The Scrap Manager is the tool set that lets an application handle cutting and pasting of the **desk scrap.** From the user's point of view, all data that's cut or copied resides in the Clipboard. The Cut command deletes data from a document and places it in the Clipboard; the Copy command copies data into the Clipboard without deleting it from the document. The Paste command—whether applied to the same document or another, in the same application or another—inserts the current contents of the Clipboard at a specified place. See Figure 5-3.

An application that supports cutting and pasting may also provide a Clipboard window for displaying the current contents of the scrap; it may show the Clipboard window at all times or only when requested via the toggled command Show (or Hide) Clipboard.

❖ *Note:* The Scrap Manager is designed to transfer small amounts of data; attempts to transfer very large amounts of data may fail from lack of memory.

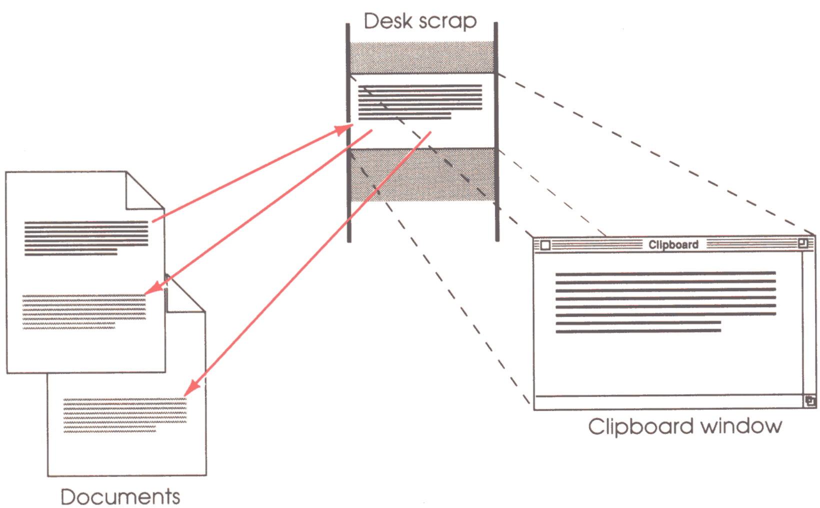


Figure 5-3
The Clipboard and the desk scrap

The desk scrap is usually stored in memory, but can be stored on disk (in the file CLIPBOARD in the SYSTEM subdirectory of the boot volume) if there's not enough room for it in memory. The Desk Manager keeps track of whether the scrap is in memory or on the disk, so you don't have to worry about loading it first.

The nature of the data to be transferred varies according to the application: a word processor handles formatted text; a graphics application handles pictures. The Scrap Manager allows for a variety of data types, and provides a mechanism whereby applications have some control over how much information is retained when data is transferred.

Desk scrap data types

From the user's point of view there can be only one thing in the Clipboard at a time, but the application may store more than one version of the information in the scrap, each representing the same Clipboard contents in a different form. For example, text cut from a word processor document may be stored in the desk scrap both as text and as a QuickDraw II picture.

Why would an application want to do this? Applications like to keep information in their own internal format, but they also want to be able to communicate via the Clipboard with other applications. When a user cuts or copies something to the Clipboard, the application can put it there two different ways:

- The internal way so that a subsequent paste (within the same application) can be easily handled. Precisely the information needed by the application can be transferred.
- The public way so that if the user tries to paste it into another application or desk accessory, the other application can at least deal with it, even if some information might be lost.

There are two defined public scrap types: text and picture. Applications must write at least one of these standard types of data to the desk scrap, and must be able to read both types.

Using the Scrap Manager

If your application supports display of the Clipboard, you should call the Desk Manager each time through your main event loop to see if the Clipboard window needs to be updated.

When a Cut or Copy command is given, use the appropriate Desk Manager calls to write the cut or copied data to the desk scrap.

When a Paste command is given, use other Desk Manager calls to access the particular type of data in the desk scrap that you want, and to get information about the data.

❖ *Edit menu:* The user accesses the desk scrap through the Edit menu. Whether or not your application supports cutting and pasting, it must include an Edit menu. Desk accessories may need it.

❖ *HodgePodge:* HodgePodge does not support cutting and pasting. It has an Edit menu, but all items are initially dimmed (disabled).

Setting up a private scrap

If your application defines its own private type of data, or if very large amounts of data might be cut and pasted, you may want to set up a **private scrap** for this purpose. A private scrap can have any format, because no other application will use it. Your application must, however, be able to convert data between the format of its private scrap and the format of the desk scrap.

If you use a private scrap, be sure that the right data is always pasted when the user gives a Paste command. The right data is whatever was most recently cut or copied from *any* application or desk accessory. Make sure also that the Clipboard, if visible, always shows the current data. You should copy the contents of the desk scrap to your private scrap at application startup and whenever a desk accessory (NDA) is deactivated. When your application quits or when a desk accessory is activated, you should copy the contents of your private scrap to the desk scrap.

❖ *LineEdit:* The LineEdit scrap is a private scrap for applications that use LineEdit. LineEdit provides routines for accessing this scrap; you'll need to transfer data between the LineEdit scrap and the desk scrap so that the Clipboard will always be current.

❖ *Scrap too large:* If your application has problems copying one scrap to another, it should alert the user. If the desk scrap is too large to copy to the private scrap, the user may want to leave the application or proceed with an empty Clipboard; if the private scrap is too large to copy to the desk scrap, the user may want to stay in the application and cut or copy something smaller.

Communicating with files and devices

The Apple IIGS Toolbox includes several tool sets that handle input/output functions. They include

- □ Standard File Operations Tool Set
- □ Print Manager
- □ Apple Desktop Bus Tool Set
- □ Text Tool Set

Using these tool sets makes your application easier to write and ensures a uniform user interface. Almost every application needs the Standard File Operations Tool Set and the Print Manager; fewer programs need the Apple Desktop Bus Tool Set or the Text Tool Set.

Accessing files

The Standard File Operations Tool Set provides the standard user interface for selecting a file to be opened or saved. The tool set displays dialog boxes that allow the user to open and save a file on a disk in any drive, and change disks in a drive. The user is completely freed from having to know how the operating system handles those tasks.

Before you make calls to the Standard File Operations Tool Set, it must be loaded and started up. If you think it may not be needed every time the program is run, you can choose to load the tool set only when you need to present the dialog boxes.

Opening a file

When the user makes a request to open a file, your application calls the SFGetFile routine to present the standard Open File dialog box (Figure 5-4) and retrieve the filename. SFGetFile allows you to specify where the standard dialog box will be placed on the screen, to specify the prompt at the top of the box, and to select, or filter, the types of files to be displayed in the box. The routine does not allow you to modify the appearance of the box; if you wish to construct your own custom dialog box, another routine is available.

The HodgePodge routine OpenFilter, listed under "The ProDOS File System" in Chapter 6, is an example of how an application can filter file types in its Open File dialog box

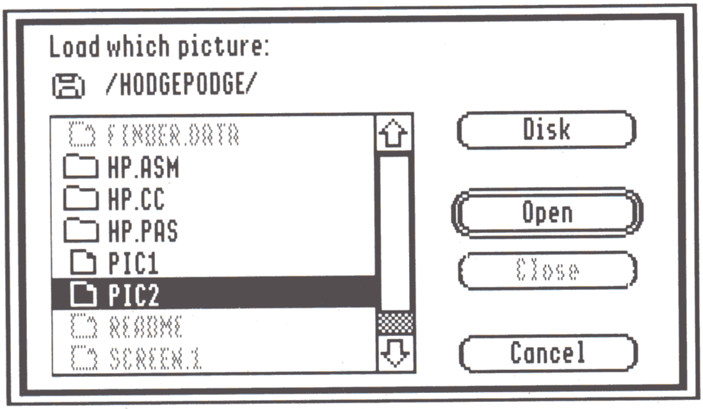


Figure 5-4
The Open File dialog box

In HodgePodge, the opening of a file is initiated when the user chooses Open from the File menu. That menu choice causes the execution of the routine `DoOpenItem`, which calls `OpenWindow`, described in Chapter 4. When opening a picture file rather than a font window, `OpenWindow` calls `AskUser`, the routine that uses Standard File Operations to select which file to open. `AskUser` looks like this:

AskUser is in the source file PAINT.PAS.

```
function  AskUser: Boolean;                {begin AskUser…}

var    ourTypeList: TypeListPtr;           {a record that lists file types:
                                            defined by Std. File Operations}
begin

   SFGetFile(                              {Call up the dialog box…}
            20,20,                         {upper-left corner = 20,20}
            'Load wich picture:',          {= message to user}
            @OpenFilter,                   {OpenFilter screens file types}
            TypeListRecPtr(0),             {NIL ptr--show all file types}
            myReply);                      {place the results here}

   AskUser := FALSE;                       {initialize this function}
   if myReply.good then                    {if SFGetFile not cancelled…}
     if LoadOne then                       {…and if the file opens OK…}
       AskUser := TRUE;                    {AskUser completes successfully}
end;                                       {End of AskUser}
```

The complete sequence of routines that execute when a file is opened is diagrammed in Appendix D.

`AskUser` calls `LoadOne`, which allocates the memory for and actually opens the requested file by making Memory Manager and ProDOS 16 calls. SFGetFile calls `OpenFilter`, a routine that controls which types of files are displayed in the dialog box and how they are highlighted. `LoadOne` and `OpenFilter` are described in Chapter 6, under "The ProDOS File System."

Saving a file

When the user makes a request to save a file, use the SFPutFile routine to present the standard Save File dialog box (Figure 5-5). SFPutFile allows you to specify where the standard dialog box will be placed on the screen, to specify the prompt at the top of the box, and to specify the maximum number of characters the user may type. If you wish to construct your own custom dialog box, you use another routine.

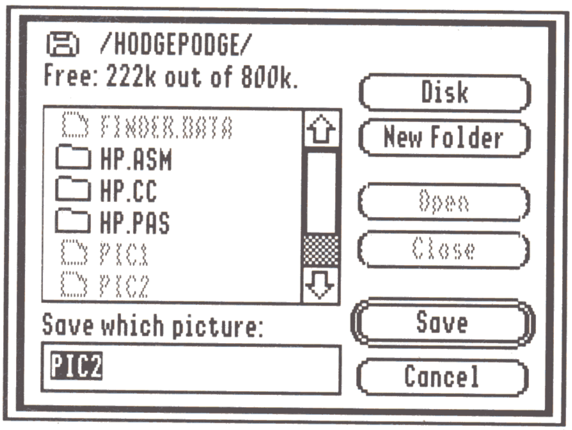


Figure 5-5
The Save File dialog box

DoSaveItem is in the source file PAINT.PAS.

In HodgePodge, `DoSaveItem` is executed when the user selects Save As from the File menu. (`CheckFrontW` makes sure that Save As is enabled only when a picture window is in front, because only picture windows can be saved.) `DoSaveItem` first calls SFPutFile to bring up the standard SaveFile dialog box, and then calls `SaveOne`, which saves the contents of the specified window to disk.

```
procedure DoSaveItem;                                {begin DoSaveItem…}

var    theWindow    : GrafPortPtr;                   {pointer to a window}
       myDataHandle: WindDataH;                      {handle to our window-data record}
       i            : Integer;
begin
  theWindow     := FrontWindow;                      {Get a pointer to the front window}
  myDataHandle := WindDataH(                         {Get a handle to the window-data…}
                       GetWRefCon(theWindow));       {…record for the window}
  SFPutFile(                                         {Bring up the Save File dialog…}
            20,20,                                   {…at location (20,20)…}
            'Save which picture:',                   (…with this prompt string…}
            myDataHandle^^.name,                     {…default = current filename…}
            15,                                      {…allow 15 characters in name…}
            myReply);                                {…put answers in Reply record--
                                                      format specified by Std. File}
  if myReply.good then
    begin                                            {If user doesn't cancel…}
      WaitCursor;                                    {Put up the watch cursor and…}
     SaveOne(myDataHandle^^.pict);                   {…save the file to disk.}

     with myDataHandle^^ do                          {Update our window-data record:}
       begin
        name   := myReply.fileName;                  {Update the window name}
        menuStr:= Concat('=',                        {Make a new menu string…}
                      myReply.fileName,
                      '\N',
                      IntToString(menuID),
                      '\0.');
        for i := firstWind to lastWind do            {Go through the window-pointer list:}
          if WindowList[i] = theWindow then          {If this window is the one…}
            Leave;                                   {…exit from this loop}
        SetMItem(MenuStr,
                      FirstWindItem+i);              {Change menu name to new window}
      end;                                           {end updating window-data record}

    SetWTitle(myDataHandle^^.name,theWindow);        {Update window name to filename}
    CalcMenuSize(0,0,WindowsMenuID);                 {Resize menu for new window name}
    InitCursor;                                      {go back to arrow cursor}
   end;                                              {end of IF myReply.good=TRUE}

end;                                                 {End of DoSaveItem}
```

The disk writing is done by the routine `SaveOne`. `SaveOne` is described under "The ProDOS File System" in Chapter 6.

Don't forget to shut down the Standard File Operations Tool Set after you have finished using it—either right afterward, or with the other tool sets at application shutdown. If you wish, you can also unload the tool set from memory and thus save space.

❖ *Note:* If you choose to unload the Standard File Operations Tool Set, be sure to reload it before making its startup call again.

Printing

The Print Manager is an Apple IIGS tool set that allows you to use standard QuickDraw II routines to print text or graphics. The Print Manager calls a printer driver to do the specific printing tasks, so your application doesn't need to know what kind of printer is connected to the computer. However, the Print Manager also includes low-level calls to the printer drivers so that you can implement alternate, low-level printing routines.

An application that supports printing must have three items in its File menu: Choose Printer, Page Setup, and Print. Choosing these items brings up dialog boxes that allow the user to specify how a document will be printed.

Choosing a printer

When the user selects the Choose Printer item, the **Choose Printer** dialog box is displayed (Figure 5-6). It lets the user select a destination device from among the printer drivers on the system disk. The Choose Printer dialog box also lets the user pick which port or slot the device is connected to, from among the port drivers on the system disk.

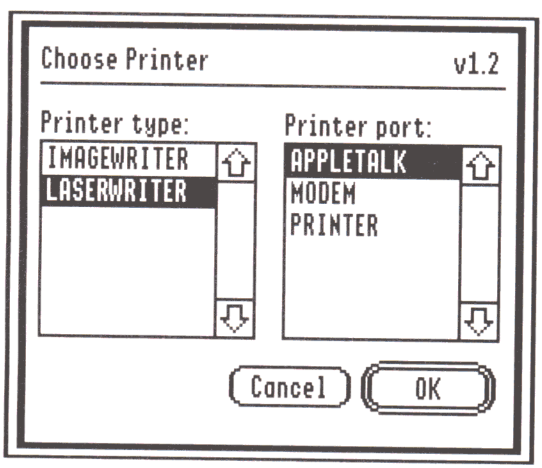


Figure 5-6
The Choose Printer dialog box

If the AppleTalk network is installed and the AppleTalk selection is made in the dialog box, the network is scanned for the names of all connected printers. If one or more printers of the chosen type are available on the network, an additional dialog box appears so that the user can select one.

❖ *Macintosh programmers:* On the Apple IIGS, the Choose Printer function is part of the Print Manager, rather than part of the Chooser desk accessory as on the Macintosh.

DoChooserItem is in the source file PRINT.PAS.

The HodgePodge routine that brings up the Choose Printer dialog box is called `DoChooserItem`. It is called from `DoMenu`, when the user selects Choose Printer from the File menu.

```
procedure DoChooserItem;                    {begin DoChooserItem...}

var     dummy: Boolean;                     {returned value is unimportant here}

begin
   dummy := PrChooser;                      {Bring up dialog box--that's it!}
end;                                        {End of DoChooserItem}
```

Making page settings

When the user selects the Page Setup item, a **Style dialog box** is displayed (Figure 5-7). It allows the user to specify formatting information, such as the page size and printing orientation. This information is not changed frequently and is usually saved with the document. The LaserWriter offers two style options unavailable for the ImageWriter: smoothing of bitmapped fonts, and font substitution.

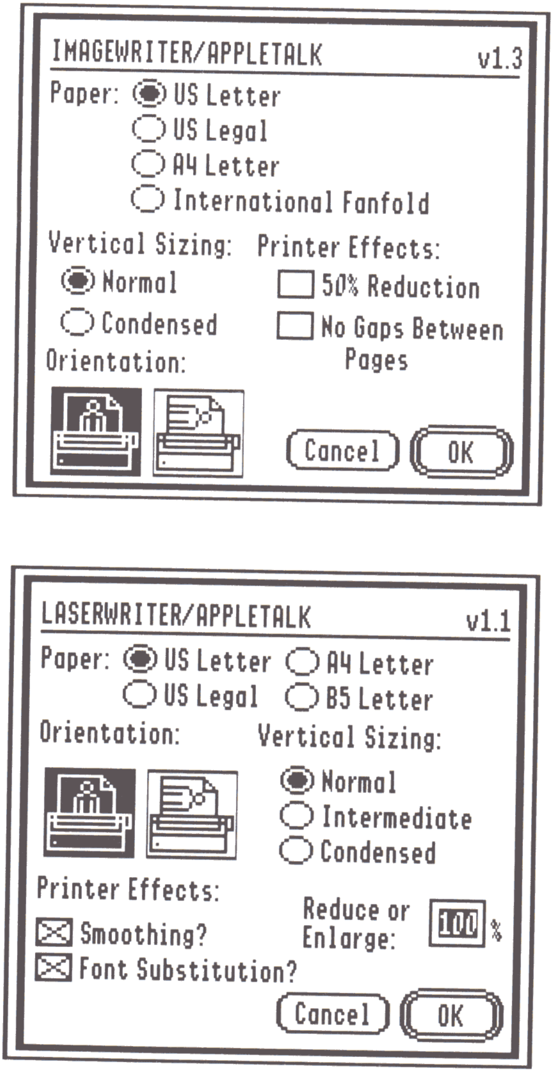


Figure 5-7
Style dialog boxes

DoSetupItem is in the source file PRINT.PAS.

Page setup in HodgePodge is handled by the routine `DoSetupItem`, called from `DoMenu` when the user selects Page Setup from the File menu. `DoSetupItem` calls the Print Manager routine PrStlDialog, passing it a handle to a print record. The print record has been allocated and initialized by the routine `SetUpDefault`, called at startup.

```
procedure DoSetupItem;                        {begin DoSetUpItem…}

var     dummy: Boolean;                       {function result unimportant here}

begin
   dummy := PrStlDialog(printHndl);           {Call up the dialog, pass it the
                                              handle to our print record}
end;                                          {End of DoSetupItem}
```

Printing

When the user chooses to print a document, usually by making a selection on the File menu, the **Job dialog box** is displayed (Figure 5-8). The Job dialog box lets the user select print quality, page range, number of copies, and other printer-specific information.

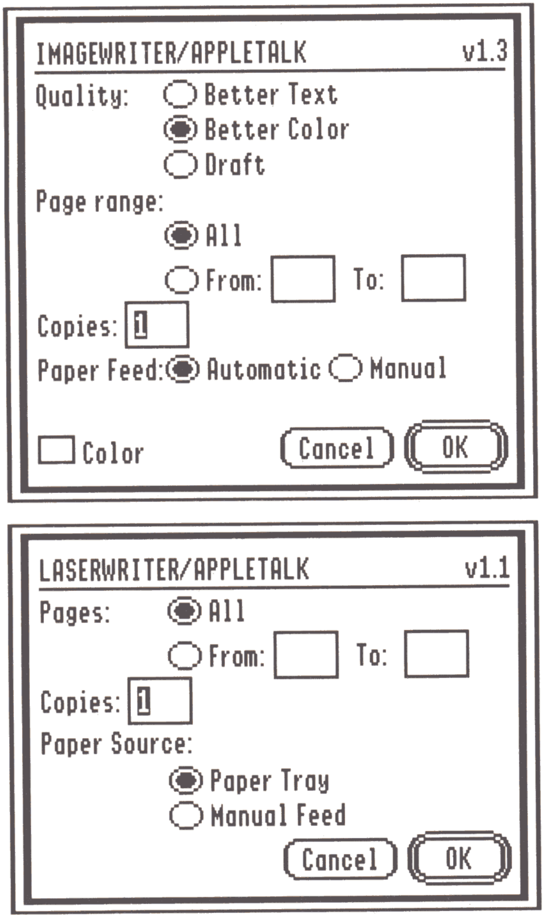


Figure 5-8
Job dialog boxes

The Print Manager automatically gives you a QuickDraw II GrafPort when you open a document for printing. You then print text and graphics by drawing into this port with QuickDraw II calls, just as if you were drawing on the screen. The Print Manager installs its own versions of QuickDraw II's low-level drawing routines in this GrafPort, causing your higher-level QuickDraw II calls to drive the printer instead of drawing on the screen.

DoPrintItem is in the source file PRINT.PAS.

The HodgePodge routine that prints files is `DoPrintItem`, called from `DoMenu` when the user selects Print from the File menu. `DoPrintItem` calls the routine PrJobDialog to bring up the Job dialog box, and then calls `DrawTopWindow` to print the file:

```
procedure DoPrintItem;                          {begin DoPrintItem...}

var     prPort    : GrafPortPtr;                {pointer to a printing GrafPort}
        theWindow: GrafPortPtr;                 {window pointer}

begin
   theWindow := FrontWindow;                    {Get a pointer to the front window}
   if theWindow <> NIL then                     {If there IS a window open...}
      if PrJobDialog(printHndl) then            {...bring up the dialog box; if...}
       begin                                    {...the user doesn't cancel...}
         WaitCursor;                            {put up the watch cursor...}
         prPort := PrOpenDoc(printHndl,NIL);    {open a printing GrafPort...}

         PrOpenPage(prPort,NIL);                {begin a new (& only) page...}
         DrawTopWindow(theWindow);              {draw the contents of the page...}
         PrClosePage(prPort);                   {...close the page}

         PrCloseDoc(prPort);                    {...close the GrafPort}
         PrPicFile(printHndl,NIL,NIL);          {...print the spooled file}
         InitCursor;                            {...and restore the regular cursor}
       end;                                     {end of printing}
                                                {end of IF a window is open}
end;                                            {End of DoPrintItem}
```

See "Using the Print Manager," later in this section, for explanations of some of the Print Manager calls that `DoPrintItem` makes.

DrawTopWindow is in the source file PRINT.PAS.

`DoPrintItem` calls the subroutine `DrawTopWindow`, which does the actual drawing in the printer GrafPort. `DrawTopWindow` acts no differently than if it were drawing to the screen; it calls either `ShowFont` or `PaintIt`, depending on what type of window is to be printed:

```
procedure DrawTopWindow(TheWindow:WindowPtr);      {begin DrawTopWindow...}

var    myDataHandle: WindDataH;                    {handle to window-data record}

begin
  myDataHandle := WindDataH(                       {Get a handle to the window's...
                    GetWRefCon(TheWindow));        {...window-data record}

  with myDataHandle^^ do
    if Flag = 0 then                               {If it's a picture window...}
      PaintIt(pict)                                {paint the picture w/this handle}
    else                                           {But if it's a font window...}
      ShowFont(theFont,isMono);                    {draw text w/this font & style}
end;                                               {End of DrawTopWindow}
```

Using the Print Manager

The structure of a print record is shown in the *Apple IIGS Toolbox Reference*.

Print records: Before you can print a document, you need a valid print record. The print record describes information such as page dimensions and resolution. You can either use an existing print record (for instance, one saved with a document) or create one through Print Manager calls. HodgePodge uses the same print record for all documents. That record can be modified by the user through the Style and Job dialog boxes.

❖ *Note:* Whenever your application saves a document, it should save an appropriate print record with the document. This sets up the printing parameters for the document so that they can be used the next time the document is printed.

Important In most instances your application should not directly change the data in the print record—it should use the standard dialog routines for changing this information. Attempting to set certain values directly in the print record can produce unexpected results.

Draft and spool printing: There are two basic methods of printing documents: draft and spool.

In **draft printing,** your QuickDraw II calls are converted directly into command codes the printer understands, which are then immediately used to drive the printer. The LaserWriter always uses draft printing, because the QuickDraw II calls are translated immediately into PostScript commands. The ImageWriter and other nonintelligent dot matrix printers are written to in draft mode for text only. High-quality pixel images are produced by spool printing.

In **spool printing** the Print Manager processes your printing requests in two steps. First it writes out (*spools*) a representation of your document's printed image to a disk file or to memory. Second, this information is converted into a bit image and printed. This method is used to print graphics on the ImageWriter.

The printing loop: To print a document, you call the following routines:

Compare this sequence of calls with the listing of the HodgePodge routine DrawTopWindow, earlier in this section.

1. PrOpenDoc, which returns a pointer to the GrafPort to be used for printing
2. PrOpenPage, which starts each new page (reinitializing the GrafPort)
3. QuickDraw routines, for drawing the page into the port created by PrOpenDoc
4. PrClosePage, which terminates the page
5. PrCloseDoc, at the end of the entire document, to close the GrafPort being used for printing
6. PrPicFile, to print the spooled document

Steps 2 through 4 are the printing loop itself; they are repeated for as many pages as are printed. Each page is either printed immediately (draft printing) or written to disk or to memory (spool printing). Your application may inspect the print record to see whether spooling was done, but it doesn't have to. The proper method is always selected automatically, and PrPicFile is executed only if needed.

You should check for errors after each Print Manager call. If an error occurs and you cancel printing (or if the user aborts printing), be sure to exit properly from the printing loop so that all files are closed correctly—that is, be sure that every PrOpenPage is matched by a PrClosePage, PrOpenDoc is matched by PrCloseDoc, and PrPicFile is still called.

❖ *Note:* The maximum number of pages in a spool file is 16,382. If, for some strange reason, you need to print more than 16,382 pages at one time, just repeat the printing loop.

QuickDraw II consequences and limitations: Even though you print by making QuickDraw calls, remember that printing to paper is not really the same as drawing to the screen. Clipping regions and character spacings don't translate exactly. Erasing, of course, can't be done on a printer. Some transfer modes and some drawing routines don't work on a LaserWriter. For more information about optimizing your printing code, see the *Apple IIGS Toolbox Reference* and the *LaserWriter Reference*.

Transfer modes are discussed under "Drawing to the Screen," in Chapter 3.

Background procedure: An optional **background procedure** runs whenever the Print Manager has directed output to the printer and is waiting for the printer to finish. It is typically a dialog box that informs the user that a print job is in progress, and allows the user the option of canceling it.

If you don't designate a background procedure, the Print Manager uses a default procedure for canceling printing: the default procedure just polls the keyboard and sets a Print Manager error code if the user presses Apple-Period. If you use this option, you should display a dialog box during printing to inform the user that the Apple-Period option is available.

Sending text to Apple II character devices

If you are writing a native-mode Apple IIGS application but don't want to use QuickDraw II and the graphic desktop interface, you may need the Text Tool Set. It provides an interface between Apple II character device drivers, which must be executed in emulation mode, and new applications running in native mode. It also provides a means of redirecting I/O through RAM-based drivers. The Text Tool Set makes it possible to deal with the text screen without switching 65816 processor modes and moving to bank zero. Dispatches to RAM-based drivers still occur in full native mode.

The multiple-segment sample program listed under "Creating Segmented Code: Three Examples" in Chapter 7 includes calls to the Text Tool Set.

The Text Tool Set has global routines that are used to set or read the current global parameters used by RAM and the Pascal and BASIC text drivers. The tool set also has I/O directing routines that direct I/O from the tool set to a specific type of character device driver, or request information about the directing of a specific I/O driver. Finally, the tool set has text routines that interface with any BASIC, Pascal 1.1, or RAM-based character device driver. See "Text Tool Set" in the *Apple IIGS Toolbox Reference* for more details.

The Pascal and BASIC character device drivers are discussed in the *Apple IIGS Firmware Reference*.

Communicating with Apple Desktop Bus devices

The Apple Desktop Bus (ADB) is a hardware channel and a protocol for connecting input devices, such as keyboards and mouse devices, with personal computers. The personal computer is considered to be the host during the communication, and controls the communication on the bus by issuing ADB commands to the devices.

The Apple Desktop Bus Tool Set sends commands and data between the Apple Desktop Bus microcontroller and the rest of the system. Typically, the tool set is used to control ADB activity, but other commands, which are used by diagnostic routines and the Control Panel, are available.

Most applications have no need to use the ADB Tool Set. However, if your program needs to modify the system's interface with the mouse, keyboard, or other ADB device, the ADB Tool Set is indispensable.

More details about the bus can be found in the *Apple IIGS Firmware Reference* and the *Apple IIGS Hardware Reference.* The tool set is described under "Apple Desktop Bus Tool Set" in the *Apple IIGS Toolbox Reference.*

Making sounds

The Apple IIGS has a very advanced sound-generation system, capable of creating and reproducing complex music, sound effects, and speech. Sound tools at several levels give you access to the sound hardware and make music generation easy.

The sound hardware

The Apple IIGS sound hardware supports two sound-generation methods. In the first method, which replicates the Apple IIe sound capabilities, an application toggles a soft switch which in turn generates clicks in a speaker. The application can control the rate of clicking and the volume of the speaker.

The second method uses a digital oscillator chip (DOC) and the rest of the sound hardware, as diagrammed in Figure 5-9: 64K of dedicated RAM, the Sound GLU (general logic unit), the analog section, and the sound connector.

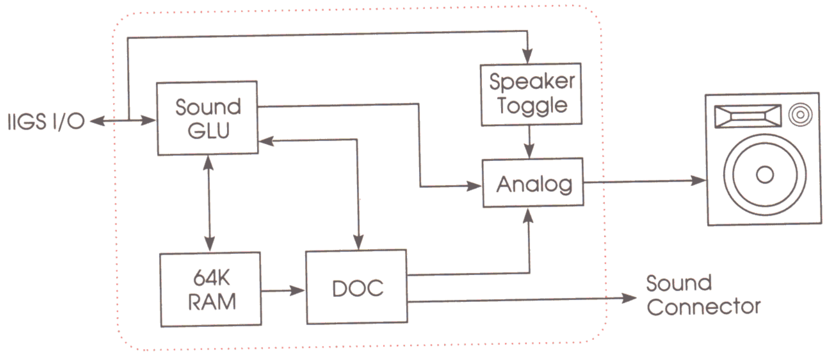


Figure 5-9
Sound hardware block diagram

The sound GLU acts as the I/O interface between the Apple IIGS system hardware and the sound hardware. The dedicated RAM stores the waveforms used for sound generation. From them the DOC can create sounds of practically any pitch and duration.

For further information on the DOC, see the *Apple IIGS Hardware Reference*.

The analog section contains all the circuitry needed to amplify and filter the signal coming from the Sound GLU or the DOC. From there the signal is sent to the speaker.

The sound connector provides the connection to interface cards that can take the tones generated by the DOC and modify them further. Three examples of possible sound cards are programmable filter cards, stereo interface cards, and sound sampling cards.

Oscillators and generators

An oscillator is the basic sound-generating unit in the DOC. The DOC contains 32 oscillators, each of which can function independently from all the other oscillators.

One of the modes of the DOC is called swap mode. The Sound Tool Set (described next) uses this mode to generate sounds. In swap mode, a pair (*swap pair*) of oscillators forms a functional unit called a *generator*. There are 15 generators defined in the Apple IIGS sound system. The oscillators in a generator take turns making sound, each signaling the end of its sound by generating an interrupt.

The System Failure Manager is described under "Miscellaneous Tool Set" in the *Apple IIGS Toolbox Reference*.

Oscillators 30 and 31 are reserved for system use and should not be be used by applications. If an interrupt is generated by oscillator 30 or 31 it is a fatal error—a sound interrupt is reported to the System Failure Manager, which halts execution.

The Sound Tool Set

The Sound Tool Set gives you the ability to access the sound hardware without having to know specific hardware I/O addresses. Sound Tool Set calls can be divided into two groups: high-level and low-level.

High-level calls constitute the *free-form synthesizer*. Calls to the free-form synthesizer are made through the normal tool call mechanism, with parameters being passed to and from the called routines on the stack. With high-level calls you can

- □ write multibyte sound data to and read it from DOC RAM
- □ get or set the volume of individual generators
- □ start and stop sound on an individual generator

See "Sound Tool Set" in the *Apple IIGS Toolbox Reference* for details on both high-level (free-form synthesizer) and low-level calls.

Low-level routines read from and write to the DOC hardware registers and individual DOC RAM locations. Unlike the other Sound Tool Set routines, which use the stack to pass parameters in the normal tool call fashion, these routines use registers to pass parameters and are entered through a jump table. The low-level routines can move information faster than the higher-level calls to the free-form synthesizer, but they do none of the error checking and housekeeping of the higher-level routines. Furthermore, if you use the low-level routines, you will have to install your own interrupt handler to service sound interrupts.

The Note Synthesizer

The **Note Synthesizer** gives your application a convenient way to play musical notes. You use the Note Synthesizer by making tool calls to start and stop individual notes. The general sequence of calls is something like this:

1. Allocate an individual generator.
2. Start a note, with the NoteOn call. The call's parameters specify the generator to play the note on, the note's volume and pitch, and what instrument to use. An **instrument** is a data structure that specifies such parameters as the amplitude envelope (attack and decay shapes), pitchbend and vibrato characteristics, and the specific waveforms that characterize the sound to be played.
3. Stop the note with the NoteOff call. When the note stops playing, the generator is automatically deallocated.

The Note Synthesizer provides for priority in allocation of individual generators. If all generators are in use, generators producing low-priority sound (such as notes trailing off) can be "stolen" to produce higher-priority sounds (such as notes starting up). Priority assignment can assure that there will always be a generator available when a note needs to be played.

❖ *Enable interrupts:* Interrupts must be enabled in order for the Note Synthesizer to function. Anything that disables interrupts (such as accessing a disk drive) will disrupt the sound being played.

The Note Sequencer

The **Note Sequencer** is the tool set that makes it easy for you to include music in your programs. In particular, it allows music to be played asynchronously, in the background.

The Note Sequencer builds upon the Note Synthesizer, in that it strings together individual notes created by the sythesizer.

You can think of the Note Sequencer as a cross between a player piano and a language interpreter. A **sequence** is a series of commands that tell the computer which notes to play and when. The Note Sequencer plays back that sequence to generate muscial sound.

Sequences are built up from simpler components. Individual basic commands to the Sequencer are called **items.** Items typically turn a note on or off, or control some aspect of the note's sound, such as vibrato. Items are assigned to one or more **tracks,** to facilitate the concept of multi-instrument music and chords. A **pattern** is a series of items; the pattern groups those items in terms of their mutual timing relationships.

A **phrase** is a set of pointers to patterns or to other phrases. Phrases make it easy to build repetitive, complex passages out of simple patterns. A sequence is a top-level phrase, one which points to patterns or other phrases but is not pointed to by any other phrases.

You play music with the Sequencer by making a StartSeq call. It plays a specified sequence. In *interrupt mode,* the sequence is played automatically until it finishes. In *step mode,* your application can play the sequence item-by-item. Step mode is useful if you need to synchronize the sequence with other events in your program.

❖ *Enable interrupts:* Interrupts must be enabled in order for the Sequencer to function. Anything that disables interrupts (such as accessing a disk drive) will disrupt the sound being played.

MIDI stands for *Musical Instrument Digital Interface,* an international standard for electronic transfer of musical data.

❖ *MIDI:* The Sequencer is not directly compatible with the MIDI protocol. If you wish to communicate with a MIDI synthesizer on your Apple IIGS, you will need to install a MIDI interface card or a MIDI serial adapter (manufactured for the Macintosh Plus). At the time of this writing, there are no software tools to allow the Note Synthesizer or Sequencer to manipulate MIDI data.

Computing

If your applications require mathematical computations on either integers or floating-point numbers, there are Apple IIGS tool sets that provide you with fast, consistent, and accurate algorithms.

Integer Math

The Integer Math Tool Set supports multiplication and division of several types of numbers, and also converts numbers from one type to another. The types of numbers dealt with are these:

integer	16-bit signed or unsigned value
longint	32-bit signed or unsigned value
fixed	32-bit signed value with 16 bits of fraction
frac	32-bit signed value with 30 bits of fraction
extended	80-bit signed value with 64 bits of fraction

❖ *Note:* The extended type really serves as a pathway to the Standard Apple Numeric Environment. See the next section in this chapter, "High-Precision Floating-Point Math (SANE)."

The Integer Math Tool Set also manipulates *Integer Math strings,* which are ASCII-string representations of numbers. An Integer Math string consists of only digits (hexadecimal or decimal) and blanks and has no length byte within it.

Within the tool set, there are *math routines* and *Integer Math string routines.* Math routines support multiplication and division of integer, long integer, fixed, and frac numbers, perform simple trigonometric calculations, and convert from one type of value to another. Integer Math string routines convert between a binary value and an ASCII character string representing that value. The binary value can be either an integer or a long integer. The character string can be in either hexadecimal or decimal format.

Your application can make use of the Integer Math routines at any time; the tool set is always active. Furthermore, the Integer Math Tool Set does not rely upon the presence of any other tool sets.

High-precision floating-point math (SANE)

For high-precision calculations on floating-point numbers, your application should use the **Standard Apple Numerics Environment (SANE).** SANE is a collection of routines that perform extended-precision IEEE arithmetic, with elementary functions. SANE scrupulously conforms to IEEE standard 754 for binary floating-point arithmetic and to the proposed IEEE standard 854, which is a radix-independent and word-length-independent standard for floating-point arithmetic.

SANE provides sufficient numeric support for most applications includes

- IEEE types single (32-bit), double (64-bit), and extended (80-bit)
- a 64-bit type for large-integer computations, as in accounting
- fundamental floating-point operations (+ – * / √ rem)
- comparisions
- binary-to-decimal and floating-point-to-integer conversions
- scanning and formatting for ASCII numeric strings
- logarithmics, trigonometrics, and exponentials
- compound and annuity functions for financial computations
- a random number generator
- functions for management of the floating-point environment
- other functions required or recommended by the IEEE Standard

The Apple IIGS SANE tool set matches the functions of the Macintosh SANE packages, and the 6502 assembly-language SANE software from which it is derived.

Additional information on SANE routines is found under "SANE Tool Set" in the *Apple IIGS Toolbox reference*

The functions of SANE are completely documented in the *Apple Numerics Manual,* which you will need if you are going to use the routines in your application.

Controlling the operating environment

Many components make up the Apple IIGS **operating environment,** the overall hardware and software setting within which application programs run. Several tool sets' principal functions are to control and modify that environment. You might call them low-level tool sets, in contrast to the higher-level, desktop interface tools.

The Event Manager, described earlier, and the Memory Manager and System Loader, described in the next chapter, are three of the most important tool sets in this group. Two others are the Miscellaneous Tool Set and the Scheduler, described here.

The Miscellaneous Tool Set

The Miscellaneous Tool Set is a collection of several small tool sets. Most of them set or return information about various low-level functions of the Apple IIGS. Several other managers and tool sets make calls to the Miscellaneous Tool Set.

Many of the routines in this tool set retrieve the address or return the value of a given parameter so that your program need not rely on fixed addresses. Please use these calls instead of directly accessing memory locations; there is no guarantee that an address being used for something in one version of system software will be used the same way in subsequent versions.

Groups of routines

- You can use *Battery RAM* routines to write and read data to and from **parameter RAM.** Any data written to parameter RAM will affect the default system configuration, which will be used the next time the system is booted.

Parameter RAM, also known as *Battery RAM,* retains clock-calendar and Control Panel information when the computer power is off.

- The *clock routines* provide you with a way to read the current time either in hex or ASCII format, or set the current time using hex format. The *GetTick* routine reads the current **tick count.**

Tick count is (approximately) the number of 60th-second intervals elapsed since system startup.

- *Vector routines* set or return the vector address for a specified interrupt manager or handler. *Interrupt control* routines allow your application to enable or disable certain interrupt sources and get the current status of those interrupts.

For more information about interrupt sources and handlers, see the *Apple IIGS Firmware Reference* and the *AppleIIGS ProDOS 16 Reference.*

- *Address and entry routines* return the addresses and native-mode entry points of some important firmware parameters and routines.
- The *HeartBeat routines* allow you to install or delete tasks from the **HeartBeat Interrupt Task queue.** Such tasks might include controlling cursor movement, or posting a disk-insert event, or checking the stack. They are called at some multiple of every "heartbeat" (vertical blanking interval), 60 times a second.
- The *System Failure Manager* routine allows you to customize the system failure message. Thus, if the user causes your application to crash, you can have the System Failure Manager display a message that gives the user an idea of what happened.

- The *User ID Manager* routines create and delete the numbers by which the ownership of all allocated memory blocks is specified. Every program on the Apple IIGS has a User ID, assigned by the User ID Manager; each block that the Memory Manager allocates for that program is given the program's User ID.
- The *mouse routines* allow your application to directly set or ge the mouse location. However, the Event Manager calls these routines automatically, so most applications don't need to make the calls. If you're not using the Event Manager or TaskMaster, you may need to use the mouse routines.
- The *PackBytes* routine packs data to make a file smaller. This can be useful for such things as graphic images, which would ordinarily take up too much space on disk. *UnPackBytes* unpacks the data from the PackBytes format.
- The *Munger* routine allows your application to manipulate strings easily.
- The *SysBeep* routine causes the system speaker to beep.

"The Miscellaneous Tool Set" in the *Apple IIGS Toolbox Reference* describes in detail all of the above groups of routines.

The Scheduler

The Scheduler is a tool set that delays the activation of a desk accessory or other task until the resources that the desk accessory or task needs become available. Much of the system code is not **reentrant;** that is, the code cannot be called again while it is executing. Because of that, activating a desk accessory within non-reentrant code almost always causes the system to fail. Thus, the Apple IIGS provides a *Busy flag* that the Scheduler can check to discover if a needed resource is busy or available.

To write a typical application, you won't need to use the Scheduler. Even if you are writing a classic desk acccessory you won't need to call the Scheduler—the Desk Manager does it for you. Perhaps the only time you need to use it is when you are writing interrupt handlers that access ProDOS 16 or the toolbox routines. For example, an application that performs background printing might need to access the Scheduler.

The Scheduler is completely documented under "The Scheduler" in the *Apple IIGS Toolbox Reference*.

Scheduler maintains a queue of tasks waiting to execute, and consults the Busy flag before dispatching them. When a non-reentrant module is entered, your interrupt handler should set the Busy flag; when exiting from the module, the application should clear the Busy flag, permitting the Scheduler to execute any tasks that have been placed in its queue.

Your interrupt handler should therefore check the state of the Busy flag before it calls any system software. If the word is nonzero, the necessary system resources are not currently available, and you should add your task to the Scheduler's queue.

Chapter 6

Memory, Segments, and Files

In Chapters 2 through 5 we showed you the event-driven program HodgePodge, and demonstrated how it implements the Apple Desktop Environment by making calls to the Apple IIGS Toolbox. In this chapter we concentrate on the Apple IIGS operating system, and how to write programs that take advantages of lower-level system software. In particular, we discuss

- how to work with the Memory Manager to request and release blocks of memory
- what segmented load files are, and how the Memory Manager and System loader work together to place them in memory
- how to use the System Loader to launch other programs from your program, load other files, and load individual segments
- how to use the ProDOS 16 QUIT call to pass execution to another program, and then bring your program back to execute again
- what direct-page/stack space is and how to set it up for your program
- how to access disk files

You do not need detailed knowledge of all of these topics in order to write an application. But if you use the toolbox you should know what direct page/stack space is; if you work with disk files you need to understand ProDOS 16; and if you want to write large, complicated programs you must be familiar with segments and the System Loader. Most important of all, *whatever kind of program you write,* you should use and respect the Memory Manager.

The Memory Manager is in charge!

As a programmer, especially if you are an Apple II programmer, you may be used to analyzing a computer's memory map and deciding just where to place all your program and data segments, file buffers, and miscellaneous work areas.

The large amount of available memory on the Apple IIGS makes it impractical to think in such terms any more. System software now relieves you of the burden of having to assign explicit locations; in fact, you are strongly discouraged from doing so, because it may interfere with the efficient use of memory and the functioning of your own or other programs. Instead, you should rely on the *Memory Manager.*

What the Memory Manager does

For a complete description of Memory Manager functions, see "Memory Manager" in the *Apple IIGS Toolbox Reference*.

The **Memory Manager** is a ROM-resident Apple IIGS tool set that controls the allocation, deallocation, and repositioning of memory blocks in the Apple IIGS. The Memory Manager works closely with ProDOS 16 and the System Loader to provide the needed memory spaces for loading programs and data and for providing I/O buffers. All Apple IIGS software, including the System Loader and ProDOS 16, must obtain needed memory space by making calls to the Memory Manager.

The Memory Manager keeps track of how much memory is free and what parts are allocated to whom. Memory is allocated in **blocks** of arbitrary length; each block possesses several attributes that describe how the Memory Manager may modify it (such as moving it or deleting it), and how it must be aligned in memory (for example, on a page boundary). Table 6-1 describes the memory block attributes and lists the predefined constants with which each can be specified.

Table 6-1
Memory block attributes

Attribute	Constant*	Explanation
Fixed (yes/no)	`attrFixed`	Must the block remain at the same location in memory?
Fixed address (yes/no)	`attrAddr`	Must it be at a specific address?
Fixed bank (yes/no)	`attrBank`	Must it be in a particular memory bank?
Bank-boundary limited (yes/no)	`attrNoCross`	Is it prohibited from extending across a bank boundary?
Special memory not usable (yes/no)	`attrNoSpec`	Is it prohibited from residing in banks \$00, \$01, and parts of banks \$E0, \$E1?
Page-aligned (yes/no)	`attrPage`	Must it be aligned to a page boundary?
Purge level (0 to 3)	`attrPurge`	Can it be purged (made available for other use)? If so, with what priority?
Locked (yes/no)	`attrLocked`	Is the block locked (temporarily fixed and unpurgeable)?

* Equivalent to "yes" if present

❖ *HodgePodge:* For an example of the use of predefined constants (column 2 of Table 6-1) in specifying memory-block attributes, see any of HodgePodge's NewHandle calls—such as in the routine `StartUpTools` (Chapter 2). See also "How Your Application Obtains Memory," later in this section.

The System Loader is described under "Loading Programs and Segments," later in this chapter.

Memory-block attributes are specified in an **attributes word.** When you request a block of memory, you supply the attributes word for that block. Later, you can modify the value of the attributes word to change the block's characteristics.

In addition to creating and deleting memory blocks, the Memory Manager moves blocks when necessary to consolidate free memory and relieve **memory fragmentation.** When it compacts memory in this way (Figure 6-1), the Memory Manager can move only those blocks that needn't be fixed in location. Therefore as many memory blocks as possible should be movable (not fixed), if the Memory Manager is to be efficient in compaction. Data segments and segments containing **position-independent** code can generally be placed in movable blocks.

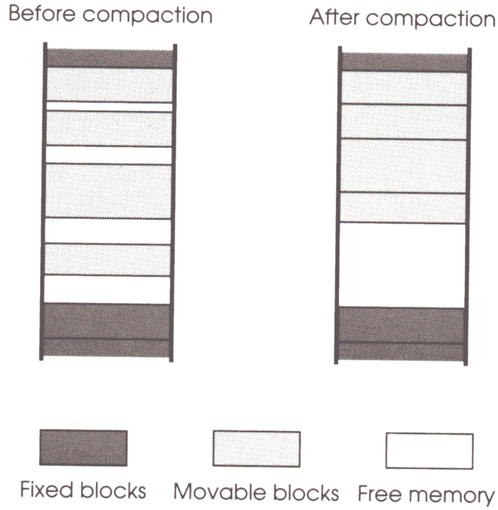


Figure 6-1
Memory fragmentation and compaction

Pointers and handles to memory blocks

To access an entry point in a movable block, an application cannot use a simple pointer, because the Memory Manager may move the block and change the entry point's address. Instead, each time the Memory Manager allocates a memory block, it returns to the requesting application a **handle** referencing that block.

A handle is a pointer to a pointer: it is the address of a fixed (nonmovable) location that contains the address of the block. If the Memory Manager changes the location of the block, it updates the address in the fixed location; the value of the handle itself is not changed. Thus the application may continue to access the block by using the handle, no matter how often the block itself is moved in memory.

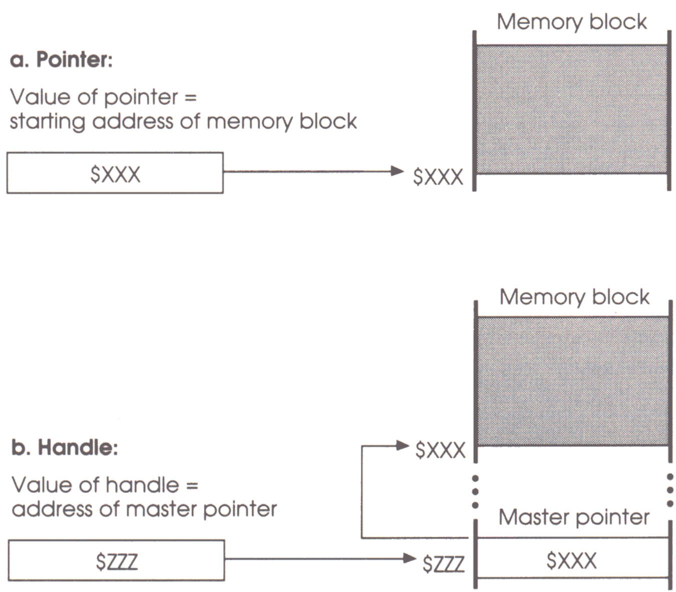


Figure 6-2
Pointer and handle

If a block will always be in the same place in memory (that is either **locked** or **fixed**), it may be referenced by a pointer instead of by its handle. To obtain a pointer to a particular block or location, an application can **dereference** the block's handle. The application reads the address stored in the location pointed to by the handle—that address is the pointer to the block. Of course, if the block is ever moved that pointer is no longer valid.

In most high-level languages, dereferencing is a simple, single-statement task. For example, in C the statement

```
z=*y
```

dereferences the memory handle `y`. The variable `z` now contains a pointer to the memory block whose handle is `y`. In assembly-language it takes a few more statements; the HodgePodge routine `Deref` (in the file `GLOBALS.ASM`) looks like this:

```
Deref       START
            sta 0           ; store low word of handle at zero-page address 0
            stx 2           ; store high word of handle at zero-page address 2
            ldy #4          ; put the value "4" in Y register
            lda [0],y       ; set the…
            ora #$8000      ; …attributes bit that…
            sta [0],y       ; …locks the block
            dey             ; now Y=3
            dey             ; now Y=2
            lda [0],y       ; put high word of pointer into acumulator
            tax             ; put high word of pointer in X register
            lda[0]          ; put low word of pointer in accumulator
            rts             ; return to caller
            END
```

When a memory block is **purged,** the memory that its handle pointed to becomes available for other use but the handle itself remains in memory. A purged memory handle points to the address $00 0000, but retains its User ID and all its attributes as listed in Table 6-1, so that the memory block can be quickly and easily reallocated if necessary.

A memory handle that points to a value of zero is called **NIL.**

When all the attributes of a memory handle as well as the memory it points to are discarded, the handle is said to be **disposed.** A disposed memory handle is no longer associated with a particular program. Your application can get rid of memory it no longer needs by making a DisposeHandle call.

Pointers and handles must be at least 3 bytes long to access the full range of Apple IIGS memory. However, pointers and handles passed as parameters are always 4 bytes long, because they are then easier to manipulate in the 16-bit registers of the 65C816 microprocessor.

Important Do not use the high-order byte of a 4-byte pointer or handle to store data. The unused byte is reserved for system use—your application should always fill it with zeros.

How your application obtains memory

When an application makes a call to the operating system or other system software that requires allocation of memory (such as opening a file or writing from a file to a memory location), the system software first obtains any needed memory blocks from the Memory Manager and then performs its tasks. When an application informs the operating system that it no longer needs that memory, the information is passed on to the Memory Manager which in turn frees that application's allocated memory. In these cases the memory allocation and deallocation is completely automatic, as far as the application is concerned.

Requesting memory

Any other memory that an application needs for its own purposes must be requested directly from the Memory Manager. The shaded areas in Figure 6-3 represent those parts of the Apple IIGS memory that can be allocated through requests to the Memory Manager. Apple IIGS applications should avoid requesting absolute (fixed-address) blocks—it defeats the Memory Manager's ability to allocate memory as efficiently as possible, and increases the probability that the program will not be able to load or run.

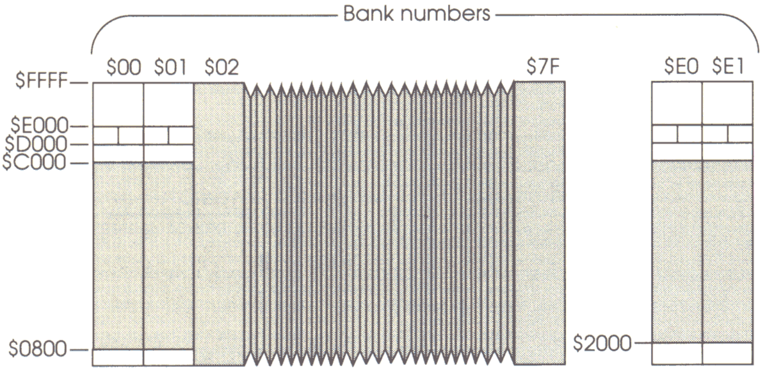


Figure 6-3
Memory allocatable through the Memory Manager

Your application requests memory with the Memory Manager's NewHandle call. Here is an example from HodgePodge:

```
toolsZeroPage := NewHandle(TotalDP,
                           myMemoryID,
                           attrBank+attrFixed+attrLocked+attrPage,
                           Ptr(0));
```

Direct-page space is described in more detail later in this chapter.

In this example HodgePodge is requesting direct-page space for tool set use. `ToolsZeroPage` is a handle to the requested space. Inputs to the call are: size (`TotalDP`), User ID (`myMemoryID`), predefined constants specifying attributes (as described in Table 6-1), and a pointer to where the block is to begin (bank $00 in this case).

User IDs

The User ID Manager is described under "Miscellaneous Tool Set" in the *Apple IIGS Toolbox Reference*.

Many Memory Manager calls use the block's **User ID**, a code number that shows what program owns the memory block. User ID's are assigned by the User ID Manager.

When your application starts up the Memory Manager, the operating system has already assigned a **master User ID** for that execution of the application. The operating system gives the master User ID number to the Memory Manager, which in turn passes that ID to your application in the MMStartUp call. You must save that ID for use when you shut down your application.

	Byte 1								Byte 0							
Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value:	typeID				auxID				mainID							

Figure 6-4
User ID format

As Figure 6-4 shows, User IDs are made up of three fields—the typeID, auxID, and mainID fields—contained in a word-length parameter. The value in the **mainID** field is assigned by the User ID Manager. The **typeID** field contains a number that describes the general kind of program segment that will occupy the block—such as application, desk accessory, or tool set. The **auxID** field is entirely definable by the program requesting the memory. Its initial value is 0; your application can store any 4-bit value there.

Using the auxID field, your application can create up to 15 new and distinct User IDs from the single master User ID returned by the Memory Manager at startup. You can use each new User ID to allocate as many additional, private memory blocks as needed; when finished with the memory allocated under a particular ID, discard it all at once by calling DisposeAll with that ID. An example of this technique is shown in the following assembly-language code fragment.

```
pushword #0    ; space for master User ID
_MMStartUp
pla            ; retrieve master User ID
sta MasterID   ; store master User ID
ora #$0100     ; create User ID with AUX ID = 1
sta MyID       ; store ID for use w/ private memory
...
...            ; (your code here)
...
...            ; (ready to exit program)
pushword MyID
_DisposeAll    ; discard all of my private memory
...            ; (continue with termination
...            ; processing)
```

Important Do not specify an auxID of 0. The Memory Manager routines PurgeAll and DisposeAll treat an auxID field with 0 in it as a wildcard that matches all values.

The main advantage of this method is that you can dispose of all allocated blocks quickly and easily, with a DisposeAll call, instead of making sure to keep track of all allocated blocks and deallocating them individually.

You don't *have* to use this method. You could simply use the master User ID, unchanged, to obtain new private memory. However, your application could not then use the DisposeAll call to discard everything—it would be disposing of itself too. Another method is to obtain an entirely new User ID for private memory. This method allows you to discard all private memory at once, but leaves open the possibility of allocated blocks remaining in memory after your application quits.

❖ *HodgePodge:* HodgePodge makes very few memory-allocation requests. It uses an unmodified master User ID when it does so, and it makes sure to dispose of its requested memory blocks individually.

Locking and unlocking, purging and disposing

If you need to access a movable memory block directly—that is, if you need to dereference its handle—you must first lock it so that it won't move *while you are using it*. When you no longer need it to be locked, make sure to **unlock** it so the Memory Manager can move it during compaction. Don't lock blocks that you are not currently accessing.

If you are *temporarily* through using a block, and don't mind if its contents must be reconstructed the next time they are needed, you can set the block's purge level to make it **purgeable.** Then the Memory Manager can purge it if more space is needed. If the Memory Manager does purge a block, you can quickly restore it with the same attributes, User ID, and size.

Important When the Memory Manager purges a block, all data in it is lost. Your application is responsible for saving and restoring the data appropriately.

When your application is *completely* finished with its own private memory, it should dispose of it—for example, by calling the DisposeAll routine and specifying the User ID with a modified auxID field, as described earlier. If your application doesn't dispose of all memory that it has acquired, the memory management system can become clogged.

For more information on memory management, see the *Apple IIGS Toolbox Reference* and the *Apple IIGS ProDOS 16 Reference.*

Important Do not call DisposeAll with the unmodified master User ID for your own program (the one in which auxID = 0).

Load segments and memory blocks

In Chapter 1 we introduced the idea of segmented programs. The executable versions of program files are called load files, and they consist of one or more **load segments.** Load segments are placed in memory by the System Loader. The System Loader must work closely with the Memory Manager because different types of segments require memory blocks with different attributes.

The System Loader is described in the next section of this chapter.

When the System Loader loads a program segment, it calls the Memory Manager to allocate a memory block for the segment. The attributes assigned to that memory block are closely tied to the attributes of the segment that will inhabit the block.

If the program segment is static, and therefore must not be unloaded or moved, its memory block is marked as *unpurgeable* and *fixed*. That means that the Memory Manager cannot change that segment's position or contents as long as the program is running. If the program segment is dynamic, its memory handle is initially marked as *purgeable* but *locked* (temporarily unpurgeable and fixed; subject to change at the request of the application). If the dynamic segment is *position-independent*, its memory handle is marked as movable; otherwise, it is fixed.

In summary, a typical load segment will be placed in a memory block that is

- locked
- fixed
- purge level = 0 (that is, unpurgeable) if the segment is static
- purge level = 1 if the segment is dynamic

Depending on other requirements the segment may have, such as alignment in memory, the load segment-memory block relationship may be more complex. Consult the *Apple IIGS ProDOS 16 Reference* for details.

Loading programs and segments

The System Loader loads all programs and segments of programs. It is called by ProDOS 16 when an application starts or quits, it is called automatically to load dynamic segments during program execution, and it can be called by your application to load and unload other programs or program segments. This section describes both the automatic operation of the loader and the ways in which your program can call it directly.

❖ *Note:* If you are writing a typical application, you don't have to call the System Loader at all. All its operations are automatic for most programs, even those with dynamic segments. If you are not interested in System Loader details, skip ahead to "Quitting and Launching Under ProDOS 16."

❖ *HodgePodge:* HodgePodge makes no loader calls.

How the System Loader works

The System Loader, although a tool set, is documented in the *Apple IIGS ProDOS 16 Reference.*

The System Loader is the Apple IIGS tool set that manages the loading of program segments into the Apple IIGS. It works very closely with the Memory Manager and with the ProDOS 16 operating system.

The System Loader is a program that processes load files—it is not concerned with source files or object files. Each load file consists of *load segments* that the loader treats differently, depending upon their attributes:

- **Static segments** are loaded into memory at application startup. They stay in memory until the program quits.
- **Dynamic segments** are placed in memory only as needed during program execution. They may be removed when no longer needed.
- **Absolute segments** are loaded at specified, fixed locations in memory.
- **Relocatable segments** are placed wherever the System Loader can find sufficient memory space. Once they are loaded, their memory blocks are locked so they can't move.
- **Position-independent segments** are placed wherever the System Loader can find sufficient memory space. Their memory blocks are initially locked, but once unlocked they can be moved from one location to another between executions.

The Jump Table, Pathname Table, and other System Loader tables are discussed in detail in the *Apple IIGS ProDOS 16 Reference.*

Some load segments consist of typical program code or data; others are more specialized. The *Jump Table segment,* when loaded into memory, becomes the **Jump Table;** it provides a mechanism by which segments in memory can trigger the loading of other needed segments. The *Pathname segment* becomes the **Pathname Table,** a cross-reference between pathnames on disk and load segments in memory. An **initialization segment** contains any code that has to be executed first, before the rest of the segments are loaded.

When the System Loader is called to load a program, it loads all static load segments and constructs the tables necessary to allow automatic loading of dynamic segments.

To **unload** a segment, the System Loader calls the Memory Manager to make the corresponding memory block purgeable. If the segment is dynamic, the loader also alters the Jump Table to reflect the fact that the segment may no longer be in memory.

Controlling programs are discussed under "Loading Applications," later in this section.

To unload *all* segments associated with a particular application (for example, at shutdown), a *controlling program* such as a shell calls the System Loader's User Shutdown function, which in turn calls the Memory Manager to make purgeable, purge, or dispose of the application's memory blocks (depending whether the application is *restartable* or not—see "Shutting Down and Restarting Programs in Memory," later in this section).

Loading a relocatable segment

When a relocatable segment is loaded into memory, its code is placed at the location assigned to it by the Memory Manager. The loader then performs **relocation** on the code—it patches address operands that refer to locations both within and external to the segment.

1. Local references are coded in the load segment as offsets from the beginning of the segment. The loader adds the starting address of the segment to each offset, so that the correct memory address is referenced.
2. External references may be to routines in static or dynamic segments. If the reference is to a *static* segment, the loader finds the memory location of the routine in that static segment and patches the reference with its address. If the reference is to a *dynamic* segment, the loader patches the reference to point to a Jump Table entry. The Jump Table entry contains the information necessary to transfer control to the external segment when it is loaded.

You can see that most Apple IIGS code cannot be moved once it is in memory: relocation happens only when the segment is loaded, so if the segment is ever moved its address operands will no longer be correct. Only position-independent code, which needs no relocation, can be moved around in memory. And position-independent code is difficult to write—therefore, most Apple IIGS code is relocatable, but not position-independent.

Loading applications

Object module format is the file format produced by Apple IIGS development systems such as the Apple IIGS Programmer's Workshop. See Chapter 7.

The functioning of the System Loader is completely transparent to most applications. Any program that is in proper **object module format** (with any combination of static and dynamic segments) will be automatically loaded, relocated, and executed whenever it is called. Unless you want your program to load dynamic segments manually, or load and execute other progams, you need not know how to use the System Loader.

However, you can indirectly affect the functioning of the System Loader by the method in which you segment your programs. If your program is divided into static and dynamic segments, you may experiment with several configurations of a single program after it has been assembled to see how loading of dynamic segments affects performance. See Chapter 7 for further program design considerations involving static and dynamic segments.

Application control of segment loading

Most applications do not need to make loader calls directly, but for programs with specialized requirements the System Loader offers this capability.

One advantage of manually loading a dynamic segment is that the segment can be referenced in a more direct manner than an automatically loaded dynamic segment. Automatically loaded dynamic segments can be referenced only through a JSL to the Jump Table; however, if the segment consists of data such as a table of values, you would want to simply access those values rather than passing execution to the segment. By manually loading the segment into a locked memory block, and dereferencing its memory handle (obtaining a pointer to the start of the segment), you can then reference any location in the table directly. Of course, because the loader does not resolve any symbolic references in the manually loaded segment, the application must know the segment's exact structure.

Your program is responsible for managing the segments it loads. That is, it must unload them with System Loader calls when they are no longer needed.

Loading by controlling programs (shells)

The ProDOS 16 QUIT call is explained under "Quitting and Launching Under ProDOS 16," later in this chapter.

A program may cause the loading of another program in one of two ways:

- The program can make a ProDOS 16 QUIT call. ProDOS 16 and the System Loader remove the quitting program from memory, then load and execute the specified new program.
- The program can call the System Loader directly. The loader loads the specified new program without unloading the original program, then hands control back to the original program.

Most applications use the first method. Even if you want your application to launch another specific program, and even if you want control to return to your application after the succeeding program quits, the ProDOS 16 QUIT call is all that is needed. For example, a finder or program launcher, which always regains control between execution of applications, uses the QUIT call to launch the applications.

Programs that use the second method are called **controlling programs.** Certain types of finders, switchers, and shells may be controlling programs. ProDOS 16 is a controlling program; the Apple IIGS Programmer's Workshop Shell is a controlling program. An application needs to be a controlling program only if it must *remain* in memory after it calls another program, usually because it has functions or sets up an environment needed by the programs it executes.

More detailed requirements for controlling programs and their subprograms (called *shell applications*) are listed in Chapter 8.

The controlling program is completely responsible for the subprogram's ultimate disposition. When the subprogram is finished, the controlling program must remove it from memory and release all resources associated with its User ID. The best way to do this is to call the System Loader's User Shutdown function.

Shutting down and restarting programs in memory

By using System Loader calls, a controlling program can rapidly switch execution among several applications. For switching to be efficient, the loader must be able to shut a program down without removing it from memory, and the program must be able to re-execute itself without having to be reloaded from disk.

The User Shutdown function can put an application into such a **dormant** state. It does this by purging an application's *dynamic* segments, and making all its *static* segments purgeable. This process frees space but keeps the dormant application's essential segments in memory. As long as all the static segments are still in memory, the Restart function brings the application back rapidly because disk access is not necessary. However, if for any reason the Memory Manager purges one of those static segments, the application can no longer be restarted—the next time it is needed, it must be loaded from its disk file.

Restartable software reinitializes its variables every time it gains control; it also makes no assumptions about the state of the machine it will find when it starts up.

Only software that is **restartable** can be executed in this way. In general, if your program has a code routine that defines and initializes all variables, and if that routine is called every time the program runs, and if the code in that routine is not modified during execution, the program is probably restartable.

When an application quits with a ProDOS 16 QUIT call (described next), it tells its controlling program whether it (the application) is restartable or not. (The controlling program simply takes the application's word for this, by the way.) If the application says it wants to be restarted and claims to be restartable, the controlling program makes it dormant. If the application says it is not restartable, the controlling program removes all of its segments from memory.

❖ *Note:* It is difficult to make some programs in some languages restartable; they require initialization information to be loaded from disk every time they execute. To help in such cases, the System Loader supports RELOAD segments. If all initialization information is put into a RELOAD segment, a program that could not otherwise be restarted can make itself restartable. When a program is restarted from a dormant state, only its RELOAD segments (plus any initialization segments) are read from disk.

Quitting and launching under ProDOS 16

ProDOS 16 and the System Loader provide a sophisticated method for passing control among different applications. Through the ProDOS 16 QUIT call, an application can do one of three things:

- □ Quit permanently.

- □ Quit permanently, but tell ProDOS 16 to launch another specified application.
- □ Quit to a specified application *temporarily,* telling ProDOS 16 it wants to be re-executed after the specified application quits.

When it launches another application or quits temporarily through the QUIT call, an application is *not* functioning as a controlling program. It is not maintained in memory (except, possibly, in a dormant state) while the other program executes. A finder or program launcher, for example, is an application that quits temporarily each time an application is launched, returning after the application quits. It is not a shell.

❖ *Note:* If you are writing a typical application in a high-level language, you may not need any of the information here—your compiler determines the manner in which your program quits. If you are writing a typical application in asssembly language, be sure to read the "HodgePodge" note at the end of this section.

Quitting, launching, and returning

The system file level is described later in this chapter, under "The ProDOS File System."

Calling QUIT terminates the present application. It also closes all open files, sets the current *system file level* to zero, and deallocates any installed interrupt handlers. ProDOS 16 can then

- □ launch a file specified by the quitting program
- □ automatically launch a program specified in the quit return stack

The **quit return stack** is a table of User ID's maintained in memory by ProDOS 16. It provides a convenient means for a program to function like a shell—the program can pass execution to subsidiary programs (even other shell-like applications), while ensuring that control eventually returns to it.

For example, a program selector may push its User ID onto the quit return stack whenever it launches an application (by making a QUIT call). That program may or may not specify yet another program when it quits, and it may or may not push its own User ID onto the quit return stack. Eventually, however, when no more programs have been specified and no others are waiting for control to return to them, the program selector's User ID will be pulled from the stack and it will be executed once again.

When your application makes a QUIT call, it specifies these two parameters:

1. Pathname pointer—if specified, it indicates the program to be loaded and executed. If no pathname is specified, ProDOS 16 pulls a User ID from the quit return stack and executes the program with that User ID.

The exact format of the flag word, and the rest of the ProDOS 16 QUIT call, is given in the *Apple IIGS ProDOS 16 Reference.*

2. Flag word—it contains two boolean values: a *return flag* and a *restart-from-memory flag*. The return flag tells ProDOS 16 whether the program making the QUIT call wants to return; if so, its User ID is pushed onto the quit return stack. The restart-from-memory flag tells ProDOS 16 whether the quitting program is restartable. If it is not, the program must be reloaded from disk the next time it is run. The information from this flag is saved on the quit return stack along with the User ID.

Using the ProDOS 8 QUIT call on the Apple IIGS is discussed in the *Apple IIGS ProDOS 16 Reference.*

❖ *ProDOS 8:* This automatic return mechanism is specific to the ProDOS 16 QUIT call, and therefore is not available to ProDOS 8 programs on the Apple IIGS. When a ProDOS 8 application quits, it can pass control to another program but it cannot put its own ID on the quit return stack.

How a particular application quits is language-specific. For example, C programs terminate with a left-facing bracket, and Pascal programs end with an `END.` statement. In either case there is no way to make an explicit QUIT statement. The actual quit statement is inserted when the program is compiled. Assembly-language programs, however, make explicit QUIT calls.

❖ *HodgePodge:* The assembly-language version of HodgePodge has the followingProDOS 16 (macro) QUIT statement:

```
_Quit   QuitParams
```

where the `_Quit` macro translates directly into a ProDOS 16 QUIT call, and the `QuitParams` parameter list consists of four null bytes (corresponding to a null pathname pointer), followed by a word-length flag value of $4000 (meaning that HodgePodge is restartable from memory).

Setting up direct-page/stack space

For assembly-language programmers, the 65C816 processor provides the convenience of a *direct page*. Accessing and indexing from direct-page addresses are efficient because address operands are a single byte, rather than the three bytes required for a full address on the 65C816.

For all programmers, direct page is of interest because several Apple IIGS tool sets require that the application provide direct-page space for them.

The size and location of the *stack* may also be of particular interest to you if you are writing heavily recursive routines that require large stack space.

How direct page and stack are organized

In the Apple IIGS, the 65C816 microprocessor's **stack pointer** register is 16 bits wide; that means that the hardware stack may be located anywhere in bank $00 of memory. Also, the stack may be as much as 64K deep. In theory, then, the stack may occupy any unused space of any size in bank $00.

Standard-Apple II stack and zero page are discussed in the *Apple IIc Technical Reference Manual* and the *Apple IIe Technical Reference Manual*

The **direct page** is the Apple IIGS equivalent to the **zero page** on a standard Apple II computer. The difference is that it need not be page zero in memory. Like the stack, the direct page may be placed in any unused area of bank $00. The microprocessor's **direct register** (D register) is 16 bits wide, and all zero-page (direct-page) addresses are added as offsets to the contents of that register. Because the direct page can be located anywhere in bank $00, you can allocate more than 256 bytes (that is, more than one page) as *direct-page space* for your program. Then, by changing the value of the D register while the program is running, you can use direct addressing to access any portion of the direct page space.

In principle, the entire 64K of bank $00 could be used for the combined direct-page/stack space. In practice, however, less space is available. First, only the lower 48K of bank $00 can be allocated; the rest is reserved for I/O and system software. Also, because more than one program can be in memory at a time, there may be more than one stack and more than one direct page in bank $00. Furthermore, many applications may have parts of their code as well as their stacks and direct pages in bank $00.

Your program should therefore be as efficient as possible in its use of direct-page/stack space. The total size of both should probably not exceed about 4K in most cases. Still, with a space that size you can write programs that require stacks and direct-page space much larger than the 512 bytes available on standard Apple II computers.

❖ *Note:* By convention, the direct page and stack occupy a single memory block in bank $00. Direct-page addresses are positive offsets from the base of the allocated space, and the stack grows downward from the top of the space.

Creating a direct-page/stack segment

Only you can determine how much stack and direct-page space your program will need when it is running. The best time to make that determination is during program development, when you create your source files. There are three ways to allocate the direct-page/stack space you need:

- Define it as a program segment.
- Use the ProDOS 16 default.
- Create it at run time.

Define it as a program segment

You can specify the size and contents of your program's stack and direct-page space by creating a direct-page/stack segment when you assemble (or compile) and link your program. The size of the segment is the total amount of stack and direct-page space allocated to your program, and the contents of the segment are whatever initial contents you want the direct-page/stack space to have.

Each time a program is started, the System Loader looks for a **direct-page/stack segment.** If it finds one, it loads the segment and passes its base address and size to ProDOS 16, along with the program's User ID and starting address. ProDOS 16 sets the A (accumulator), D (direct), and S (stack) registers as shown below, then passes control to the program.

Register	Contents
A	User ID assigned to the program
D	address of the first (lowest) byte in the direct-page/stack space
S	address of the last (highest) byte in the direct-page/stack space

LinkEd is described in the *Apple IIGS Programmer's Workshop Reference*. An example of a LinkEd file is shown in "Creating Segmented Code: Three Examples" in Chapter 7.

KIND is a segment-description field. See "Object Module Format" in the *Apple IIGS Programmer's Workshop Reference* or "System Loader Technical Data" in the *Apple IIGS ProDOS 16 Reference*.

To specify the direct-page/stack space for your program, use the following procedure (in APW assembly language, using LinkEd). See also Figure 6-5.

1. Create a data segment in your source file with the size and contents you want for your initial direct page and stack.
2. Assemble the program.
3. Use a LinkEd file to link the program. Make the direct-page/stack segment a load segment by itself, with KIND=$0812 (meaning it is a static, absolute-bank, direct-page/stack segment).

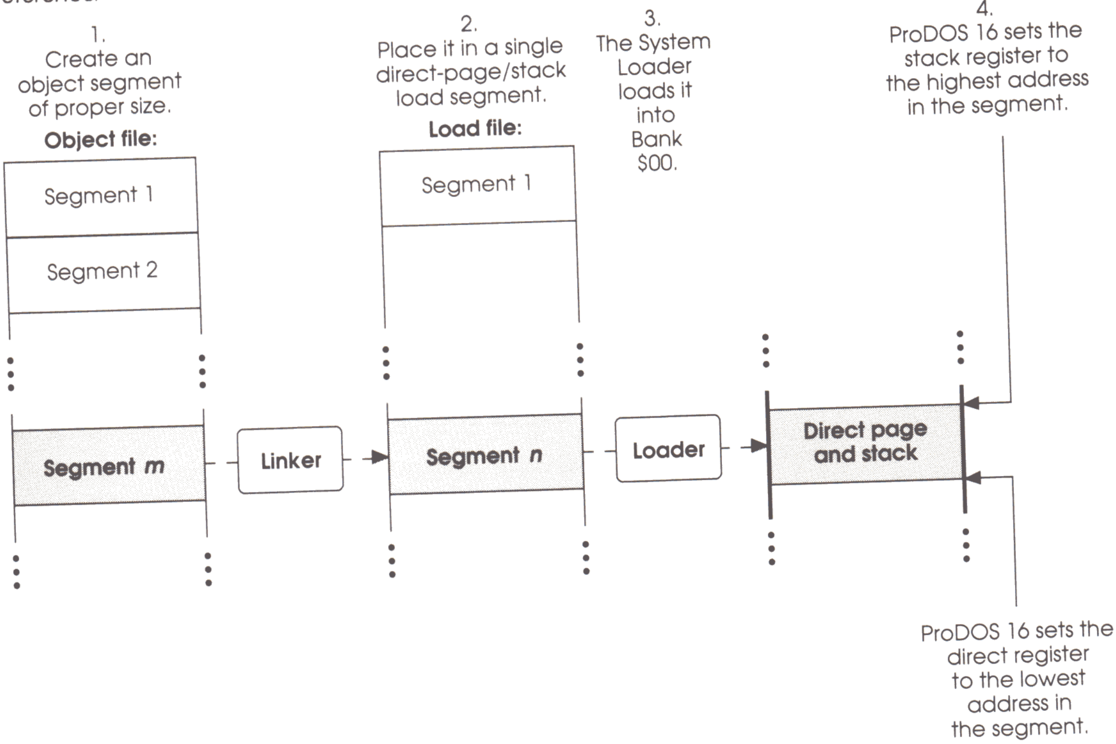


Figure 6-5
Loading a direct-page/stack segment

Use the ProDOS 16 default

If the loader finds no direct-page/stack segment in a file at load time, ProDOS 16 itself calls the Memory Manager to allocate a default direct-page/stack segment, in a memory block with these attributes:

See the *Apple IIGS Toolbox Reference* for a general description of memory block attributes assigned by the Memory Manager.

Size	1,024 bytes
Owner	program with the User ID returned by the loader
Fixed/movable	fixed
Locked/unlocked	locked
Purge level	1
May cross bank boundary?	no
May use special memory?	yes
Alignment	page-aligned
Absolute starting address?	no
Fixed bank?	yes—bank $00

Once allocated, the default direct-page/stack space is treated just as it would be if it had been specified by the program: ProDOS 16 sets the A, D, and S registers before handing control to the program, and at shutdown the System Loader makes the segment purgeable.

For many assembly-language applications, the 1K default stack and direct page space allocated by ProDOS 16 are sufficient. Individual high-level language systems may have the same or different default sizes; check your language reference manual.

HodgePodge's direct-page allocation for tool sets is demonstrated under "Start the Program" in Chapter 2.

❖ *HodgePodge:* HodgePodge accepts the default direct-page/stack space set up for it by ProDOS 16. In addition, it manually creates a direct-page space for tool sets, by a method similar to that described next, under "Create It at Run Time."

Create it at run time

If the ProDOS 16 default space is the wrong size for your application, and if for some reason you do not want to specify the size of your direct-page/stack space at link time, you can include ProDOS 16 and Memory Manager calls in your program that allocate a direct-page/stack space during program execution. In that case, when ProDOS 16 transfers control to your program, save the User ID value left in the accumulator (or use the User ID returned by the Memory Manager startup call) before doing the following:

1. Using the starting or ending address left in the D or S register by ProDOS 16, make a FindHandle call to the Memory Manager to get the memory handle of the automatically provided direct-page/stack space. Then, using that handle, get rid of the space with a DisposeHandle call.
2. You can now allocate your own direct-page/stack space through the Memory Manager NewHandle call. Make sure that the allocated block is *purgeable*, *unmovable*, and *locked*.
3. Place the appropriate values (beginning and ending addresses of the segment) in the D and S registers.

Cautions

When your program terminates with a QUIT call, the System Loader makes the direct-page/stack segment purgeable, along with the program's other static segments. Bank $00 is heavily used, and if the direct-page/stack segment is purged, your entire program will have to be reloaded from disk when it reexecutes.

If your direct-page/stack load segment contains initialization data, you need to make it a RELOAD segment if you want your program to be restartable.

There is no provision for extending or moving the direct-page/stack space after its initial allocation. Because bank $00 is so heavily used, the space you request may be unavailable—the memory adjoining your stack is likely to be occupied by a locked memory block. Make sure that the amount of space you specify at link time fills all your program's needs.

Important The Apple IIGS provides no mechanism for detecting stack underflow or overflow (collision of the stack with the direct page). Your program must be carefully designed and tested to make sure this cannot occur.

The ProDOS file system

The term *ProDOS* (as in *ProDOS file system*) refers to features common to both ProDOS 8 and ProDOS 16. The term *ProDOS 16* (as in *ProDOS 16 prefixes*) is used to describe features that ProDOS 8 does not have.

You use the Apple IIGS disk operating system, ProDOS 16, to open, close, create, delete, and otherwise manipulate files on disk. This section describes the filename and prefix conventions used by ProDOS 16 and introduces some of the ProDOS 16 functions that your program may call.

Filenames and pathnames

A ProDOS **filename** or **volume name** is up to 15 characters lo
It may contain uppercase letters (A-Z), digits (0-9), and period
and it must begin with a letter. Lowercase letters are automatic
converted to uppercase. A filename must be unique within its
directory.

A ProDOS **pathname** is a series of filenames, each preceded b
slash (/). The first filename in a pathname is the name of a
volume directory; it, too, is preceded by a slash. Successive
filenames indicate the path, from the volume directory to the f
that ProDOS must follow to find a particular file. The maximum
length for a pathname is 64 characters, including slashes.

Pathname prefixes

All calls that require you to name a file, accept either a full
pathname or a **partial pathname.** A partial pathname is a portio
of a pathname; it doesn't begin with a slash and doesn't include
volume name. The maximum length for a partial pathname is 64
characters, including slashes.

ProDOS constructs a complete pathname from a partial
pathname by adding a pathname *prefix* to it. A **prefix** is a part of
a pathname that starts with a volume name; it may be the volume
name alone, or it may be the volume name folowed by one or
more names of subdirectories. ProDOS 16 allows you to define
more than one prefix, and refer to each prefix by its *prefix
number*. When you specify no particular prefix number with a
partial pathname, ProDOS 16 adds the *default prefix*.

ProDOS 16 supports 9 prefixes, referred to by the prefix numbers
0/, 1/, 2/,...,7/, and */. A **prefix number** appears at the beginning
of a partial pathname, and includes a terminating slash to
separate it from the partial pathname. When ProDOS 16
processes the pathname, it replaces the prefix number with the
actual prefix it represents.

One of the prefix numbers (*/) has a fixed value, and the others
have default values assigned by ProDOS 16. The predefined
prefixes are as follows:

*/	**Boot prefix:** the name of the volume from which the presently running ProDOS 16 was booted.
0/	**Default prefix:** (automatically attached to any partial pathname that has no prefix number)—it has a value dependent on how the current program was launched. In most cases the default prefix is equal to the boot prefix.
1/	**Application prefix:** the pathname of the subdirectory that contains the currently running application.
2/	**System library prefix:** the pathname of the subdirectory (on the boot volume) that contains the library files used by applications.
3/—7/	**Null strings:** (unless previously defined by an application).

Your application may change the values of all prefixes except prefix */.

Prefix 0/, the default prefix, is similar to the ProDOS 8 *system prefix* in that ProDOS 16 automatically attaches prefix 0/ to any partial pathname for which you specify no prefix. However, its initial value is not always equivalent to the ProDOS 8 system prefix's initial value.

See the *ProDOS 8 Technical Reference Manual* for more information on ProDOS 8 prefix conventions.

The maximum length for a prefix is 64 characters. The minimum length for a prefix is zero characters; a prefix of zero length is known as a **null prefix.** You set and read prefixes using the calls SET_PREFIX and GET_PREFIX. The 64-character limits for the prefix and partial pathname combine to create a maximum effective pathname length of 128 characters.

Table 6-2 shows some examples of prefix use. The pathname provided by the caller is compared with the full pathname constructed by ProDOS 16. The examples assume that prefix 0/ is /VOLUME1/ and prefix 5/ is /VOLUME1/TEXT.FILES/

Table 6-2
Examples of prefix use

Case illustrated	Pathname provided	Pathname as expanded
Full pathname	/VOLUME1/TEXT.FILES/CHAP.3	/VOLUME1/TEXT.FILES/CHAP.3
Implicit use of prefix 0/	TEXT.FILES/CHAP.3	/VOLUME1/TEXT.FILES/CHAP.3
Explicit use of prefix 0/	0/TEXT.FILES/CHAP.3	/VOLUME1/TEXT.FILES/CHAP.3
Use of prefix 5/	5/CHAP.3	/VOLUME1/TEXT.FILES/CHAP.3

Note: These examples assume that prefix 0/ is set to /VOLUME1/ and that prefix 5/ is set to /VOLUME1/TEXT.FILES/.

Important When your application is launched, all nine prefix numbers are assigned to specific pathnames (some are meaningful pathnames, and others may be null strings). However, prefixes 0/ and 2/ may *not* have the expected ProDOS 16 default values—they may reflect changes made by the previous application. Beware of assuming any particular initial value for any particular prefix.

Creating and destroying files

All of these file attributes are fully explained in Appendix A of the *Apple IIGS ProDOS 16 Reference*.

A file is placed on a disk by the ProDOS 16 CREATE call. When you create a file, you assign it several properties, including

- pathname
- access attributes (deletable, renamable, writeable, readable, backup-required)
- file type
- auxiliary type
- creation date and creation time

Once a file has been created, it remains on the disk until it is deleted (by using the DESTROY call).

Opening, closing, and flushing files

Before you can read information from or write information to a file, you must use the ProDOS 16 OPEN call to open the file for access. The OPEN call returns a reference number for the file. All subsequent references to the open file must use its reference number. The file remains open until you use the CLOSE call on it.

When you finish reading from or writing to a file, you must use the CLOSE call to close the file. CLOSE writes any unwritten data from the file's I/O buffer to the file, and it updates the file's size in the directory if necessary. To access the file again, you have to reopen it.

FLUSH, like CLOSE, writes any unwritten data from the file's I/O buffer to the file, and updates the file's size in the directory. However, FLUSH keeps the file open.

File levels

When a file is opened, it is assigned a file level according to the **system file level.** You can determine the current system file level with a GET_LEVEL call, and can change the level with a SET_LEVEL call. When you specify 0 as the reference number in the CLOSE and FLUSH calls, all files having a file level greater than or equal to the current system file level are closed or flushed.

This feature allows controlling programs to quickly close all files associated with their subprograms. For example, when a shell program takes control of the Apple IIGS, it can execute a GET_LEVEL call to determine the current system file level, then execute a SET_LEVEL call to set the system file level to a higher level. Each file opened by the shell and by the programs that run under the shell is then assigned the new file level by ProDOS 16.

When the shell is ready to quit, it can execute a CLOSE call with a reference number of 0, and all files opened under the shell (that is, those with a file level equal to or greater than the current system file level) are closed. The shell can then execute a SET_LEVEL call to return the system file level to its previous value, and finally execute a QUIT call.

Reading and writing files

READ and WRITE calls to ProDOS 16 transfer data between memory and a file. For both calls, the application must specify the location in memory of a buffer that contains, or is to contain, the transferred data. When the request has been carried out, ProDOS 16 passes back to the application the number of bytes that it actually transferred.

A read or write request starts at a specific position in the file, and continues until the requested number of bytes has been transferred (or, on a read, until the end-of-file has been reached). Read requests can also terminate when a specified character (the newline character set by the NEWLINE call) is read.

LoadOne is in the source file PAINT.PAS.

The HodgePodge routine that reads files is `LoadOne`, called from the routine `AskUser`, which itself is called from `DoTheOpen` when the user wants to open a picture window. `LoadOne` makes the ProDOS 16 calls OPEN, READ, and CLOSE:

```
function  LoadOne: Boolean;                    {begin LoadOne...}

var     openBlk : OpenRec;                      {ProDOS 16 parameter blocks...}
        readBlk : FileIORec;                    {...defined in ProDOS 16 interface}

begin
   LoadOne := FALSE                             {Initialize value of function}
   WaitCursor;                                  {put up watch cursor}
   pictHndl := NewHandle($8000,                 {request memory to hold the picture...}
                          myMemoryID,           {HodgePodge's User ID}
                          0,                    {not purgeable, no restrictions}
                          Ptr(0));              {anywhere}
   if isToolError then                          {If the memory is unavailable...}
     Exit;                                      {...leave this subroutine}

   HLock(pictHndl);                             {Lock handle so picture won't move}
                                                {Now fill in parameter block:...}
   openBlk.openPathname :=
                    @myReply.fullPathname;      {pathname from Std. File results...}
   openBlk.ioBuffer := NIL;                     {zero this parameter}

   OPEN(openBlk);                               {make a ProDOS 16 OPEN call}
   if CheckDiskError(27) then                   {If it fails for some reason...}
     Exit;                                      {...display error and exit}
                                                {Fill in parameter block for READ:...}
   readBlk.databuffer   := pictHndl^;           {pointer to where to put data}
   readBlk.requestCount := $8000;               {requested no. of bytes to read}
   readBlk.fileRefNum   := openBlk.openRefNum;  {file's reference number}

   READ(readBlk);                               {make a ProDOS 16 READ call}
   if CheckDiskError(28) then                   {If it fails for some reason...}
     Exit;                                      {...display error and exit}
                                                {Open file no longer necessary:...}
   CLOSE(readBlk);                              {Make a ProDOS 16 CLOSE call}
   HUnLock(pictHndl);                           {Unlock the handle until we...}
                                                {...need the picture again}
   LoadOne := TRUE;                             {function successfully completed}

end;                                            {end of LoadOne}
```

SaveOne is in the source file PAINT.PAS.

The HodgePodge routine that creates files and saves them to disk is `SaveOne`. It is called from the routine `DoSaveItem` (which saves the contents of a picture file to disk), described under "Communicating With Files and Devices" in Chapter 5. `SaveOne` makes the ProDOS 16 calls CREATE, DESTROY, OPEN, WRITE, and CLOSE:

```
procedure SaveOne(pict: Handle);                    {begn SaveOne...}

var   destroyBlk : PathnameRec;                     {a ProDOS 16 parameter block}
      createBlk  : FileRec;                         {a ProDOS 16 parameter block}
      openBlk    : OpenRec;                         {a ProDOS 16 parameter block}
      writeBlk   : FileIORec;                       {a ProDOS 16 parameter block}

begin
  destroyBlk.pathname :=                            {Put pathname from DoSaveItem...}
                 @myReply.fullPathname;             {...reply record into param. block}

  DESTROY(destroyBlk);                              {Delete any existing file with
                                                     that pathname}

  createBlk.pathname    :=
                 @myReply.fullPathname;             {Put the pathname in the block...}
  createBlk.fAccess     := $C3;                     {...give it this access value...}
  createBlk.fileType    := $C1;                     {...assign a file type (unpacked)...}
  createBlk.auxType     := 0;                       {...aux. type = 0...}
  createBlk.storageType:= 1;                        {...make it a seedling file...}
  createlk.createDate   := 0;                       {...let ProDOS 16 assign...}
  createBlk.createTime  := 0;                       {...creation date and time.}

  CREATE(createBlk);                                {Create the new file}
  if CheckDiskError(25) then                        {If the file can't be created...}
    Exit;                                           {...display error and exit}

  openBlk.openPathname :=
                 @myReply.fullPathname;             {Put the pathname into the block}
  openBlk.ioBuffer   := NIL;                        {(this field must be zero)}

  OPEN(openBlk);                                    {Open the file we've just created}
                                                    {Make a pointer to the buffer...}
  writeBlk.dataBuffer   := pict^;                   {...from the handle to the picture}
  writeBlk.requestCount := $8000;                   {Transfer the entire 32K-byte file}
  writeBlk.fileRefNum   := openBlk.fileRefNum;      {supply file's ref_num}
  WRITE(writeBlk);                                  {Write the data to the file}
  if CheckDiskError(26) then                        {If the file can't be written to...}
    Exit;                                           {...display error and exit}

  CLOSE(writeBlk);                                  {Close the file}

end;                                                {End of SaveOne}
```

Brief explanations of certain ProDOS 16 parameters, such as access and file type, are found elsewhere in this section.

The parameter lists for the ProDOS 16 calls used in `LoadOne` and `SaveOne` are all combined into the single record `P16Blk`, defined in the Pascal interface library to ProDOS 16. Complete documentation of required parameters for all ProDOS 16 calls is in the *Apple IIGS ProDOS 16 Reference*.

The EOF and Mark

To aid reading from and writing to files, each open file has one number indicating the end of the file (the **EOF**), and another defining the current position in the file (the **Mark**). ProDOS 16 moves (increments or decrements) both the EOF and the Mark automatically when necessary, but an application program can also manipulate them independently of ProDOS 16.

The EOF is the number of readable bytes in the file. The Mark cannot exceed the EOF. If during a write operation the Mark meets the EOF, both the Mark and the EOF are moved forward one position for every additional byte written to the file.

To move the EOF and Mark, use the SET_EOF and SET_MARK calls. To determine the current values of the EOF and the Mark, use the GET_EOF and GET_MARK calls.

❖ *HodgePodge:* HodgePodge doesn't pay much attention to EOF and Mark in its file access, because it reads and writes only entire files at a time.

File attributes

The directory entry for each file contains information that may be useful to your program. This section describes the following fields in directory entries and headers:

- creation and last-modification dates
- access attributes
- file type
- auxiliary type

For more information on all file attributes, see Appendix A of the *Apple IIGS ProDOS 16 Reference*.

If you want to know the properties of a given file, use the GET_FILE_INFO call. If you want to change the file's name, use the CHANGE_PATH call. To alter the other properties, use the SET_FILE_INFO call.

Creation and last-modification date and time

The date and time of creation of a file are stored in the file's directory entry. When your program creates a new file, ProDOS 16 automatically gives the file the current system date and time. When your program modifies a preexisting file, ProDOS 16 automatically sets the last-modification date and time to the current date and time. In general, your program should not have to change these attributes.

Access

The access attribute field, or access byte, determines whether the file can be read from, written to, deleted, or renamed. It also contains a bit that can be used to indicate whether a backup copy of the file has been made since the file's last modification.

ProDOS 16 sets the **backup bit** whenever the file is changed (that is, after a CREATE, RENAME, CLOSE after WRITE, or SET_FILE_INFO operation). This bit should be reset by a backup utility (using CLEAR_BACKUP_BIT) whenever it makes a backup copy of the file. No other program should ever reset the backup bit.

❖ *HodgePodge:* When HodgePodge creates its picture files, it assigns them the access value of $C3, meaning that they may be destroyed, renamed, read from, and written to.

File type

The `file_type` field in a directory entry identifies the type of file described by that entry. This field should be used by applications to guarantee file compatibility from one application to the next. The currently defined hexadecimal values of this byte are listed in Table 6-3.

Table 6-3 also lists the 3-character mnemonic file-type codes that might appear in catalog listings. For any file type without a specified mnemonic code, most catalog programs substitute the hexadecimal file type number.

SOS is the operating system for the Apple III computer.

❖ *SOS:* SOS file types are included in Table 6-3 because SOS and ProDOS have identical file structures. Each may read the other's files.

❖ *HodgePodge:* When HodgePodge creates its picture files, it assigns them the file type $C1 (picture file, unpacked format).

Table 6-3
ProDOS file types

File type	Code	Description
$00		Uncategorized file (SOS and ProDOS)
$01	BAD	Bad block file
$02 *	PCD	Pascal code file
$03 *	PTX	Pascal text file
$04	TXT	ASCII text file (SOS and ProDOS)
$05 *	PDA	Pascal data file
$06	BIN	General binary file (SOS and ProDOS 8)
$07 *	FNT	Font file
$08	FOT	Graphics screen file
$09 *	BA3	Business BASIC program file
$0A *	DA3	Business BASIC data file
$0B *	WPF	Word processor file
$0C *	SOS	SOS system file
$0D–$0E *		SOS reserved
$0F	DIR	Directory file (SOS and ProDOS)
$10 *	RPD	RPS data file
$11 *	RPI	RPS index file
$12 *		AppleFile discard file
$13 *		AppleFile model file
$14 *		AppleFile report format file
$15 *		Screen library file
$16–$18 *		SOS reserved
$19	ADB	AppleWorks® Data Base file
$1A	AWP	AppleWorks Word Proc. 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	APW run-time library file
$B5	EXE	ProDOS 16 shell application file
$B6	PIF	ProDOS 16 permanent initialization file
$B7	TIF	ProDOS 16 temporary initialization file
$B8	NDA	New desk accessory
$B9	CDA	Classic desk accessory
$BA	TOL	Tool set file

Table 6-3 (continued)
ProDOS file types

File type	Code	Description
$BB		Driver file
$BC		General ProDOS 16 load file
$BD-$BF		Reserved for ProDOS 16
$C0		Apple IIGS picture file (packed formats)
$C1		Apple IIGS picture file (unpacked format)
$C2–$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	IVR	Integer BASIC variable file
$FC	BAS	Applesoft program file
$FD	VAR	Applesoft variables file
$FE	REL	Relocatable code file (EDASM)
$FF	SYS	ProDOS 8 system program file

*apply to Apple III (SOS) only

Auxiliary type

Some applications use another field in a file's directory entry, the auxiliary type field (`aux_type`), to store additional information not specified by the file type. Some catalog listings may display the contents of this field under the heading "Subtype."

For example, APW source files (file type $B0) include a language-type designation in the `aux_type` field. The starting address for ProDOS 8 executable binary files (file type $06) may be in the `aux_type` field. The record size for random-access text files (file type $04) may be specified in the auxiliary type field.

For most file types, ProDOS 16 and ProDOS 8 impose no restrictions (other than size) on the contents or format of the auxiliary type field. Individual applications may use those two bytes to store any useful information.

❖ *HodgePodge:* When HodgePodge creates its picture files, it assigns them an auxiliary type value of 0. It stores no information in the auxiliary type field.

Controlling user access to files

SFGetFile sends to OpenFilter only the file types specified in its typeList parameter—see "Communicating With Files and Devices" in Chapter 5.

OpenFilter is in the source file PAINT.PAS.

The picture files read and stored by HodgePodge are ProDOS file type $C1. Because HodgePodge cannot handle other file types, the user should not be permitted to select anything but $C1 files. On the other hand, it might be useful to let the user *see*, if not *select*, other files in a directory.

The HodgePodge routine `OpenFilter` is called by the Standard File Operations Tool Set to find out how to display files of various types in the Open dialog box. For each file entry it encounters, SFGetFile calls this routine. If the routine returns 0, the filename is not displayed; if the routine returns 1, the filename appears (dimmed) but the file is not selectable; if the routine returns 2, the filename is not dimmed and the file is selectable. `OpenFilter` dims all file types but $C1.

The variable `FileTypePtr` below is a pointer to the file-type field in the directory entry for the file under consideration. The file-type field is at an offset of 10 bytes into the file entry.

```
function OpenFilter(dirEntry:longint): Integer;      {begin OpenFilter...}

type    BytePtr     = ^byte;

var     fileTypePtr : BytePtr;

begin
  fileTypePtr := Pointer (dirEntry + $10);           {First, get a pointer to the file's...}
  if (BitAND(fileTypePtr^,$00FF) = $C1) then         {file type from its directory entry.}
    OpenFilter := 2                                  {If it's unpacked Picture File type...}
  else                                               {...make it black and selectable}
    OpenFilter := 1;                                 {If it's any other file type...}
  end;                                               {...it's dimmed and nonselectable}
                                                     {End of OpenFilter}
```

Chapter 7

Creating a Segmented Application

In this chapter we consider the mechanics of developing applications on the Apple IIGS computer. In particular, we show you how to create a segmented application.

Segmentation may be important to you only if you are writing a large program. The executable form of HodgePodge, for example, is a single load segment. But you'll also find general hints on program design and development in this chapter that may be useful no matter what size your program is.

The chapter begins with a brief description of the Apple IIGS Programmer's Workshop, the programming environment used for all the sample programs in this book. Next, we discuss the types of files used during program development—source files, object files, and load files—and relate these file types to the steps involved in developing a program. Then we discuss the segmentation of object files and of load files, explaining why you might want to segment your program, and describing how to go about it. We also discuss library files, which are special kinds of segmented files.

Near the end of the chapter, we present three sample programs that illustrate the use of segmentation. Finally, we give some hints that will help you debug your segmented programs.

Apple IIGS Programmer's Workshop

The Apple IIGS Programmer's Workshop (APW) is a complete development environment for the Apple IIGS computer that includes the following components:

- shell
- editor
- linker
- utility programs

These components of APW can be used by any of several programming languages, and are described in the *Apple IIGS Programmer's Workshop Reference*. Other current and future components of the development environment, described in separate manuals, include

- 65816 assembler
- C Compiler
- other compilers
- debuggers

Program descriptions

The programs included in the Apple IIGS Programmer's Workshop relate to each other as illustrated in Figure 7-1. The APW Shell's command interpreter serves as the interface between you and the rest of the Apple IIGS system. The shell allows you to call the other programs that constitute the Apple IIGS Programmer's Workshop, and serves as the link between APW and the Apple IIGS Toolbox and operating system. The toolbox and operating system (including ProDOS 16, the System Loader, and the Memory Manager) are the interface between APW and Apple IIGS hardware and firmware.

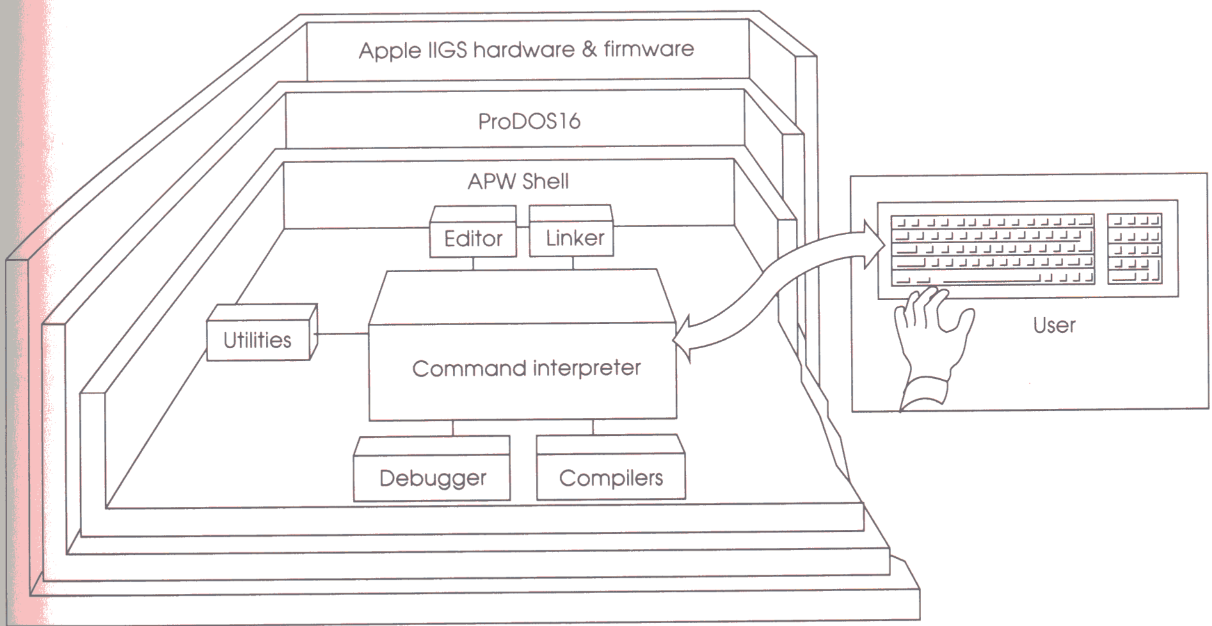


Figure 7-1
APW programs in the Apple IIGS system

Shell

The shell program is the interface that allows you to execute APW commands and programs. With it you can perform a variety of housekeeping functions, such as copying and deleting files or listing a directory. The shell supports input and output redirection and pipelining of APW programs.

The shell also acts as an interface and extension to ProDOS 16, providing several functions, called **shell calls,** that can be called by programs running under the shell. Shell calls can be used by utility programs, compilers, linkers, or assemblers to perform such functions as passing parameters and operations flags between the shell and APW programs. The format for making these calls is exactly like that used for making a ProDOS 16 call.

ProDOS 16 calls are described in the *Apple IIGS ProDOS 16 Reference*.

Editor

This full-screen text editor is designed for use with APW assemblers and compilers. It allows you to enter, copy, delete, and move text, and provides automatic search and search-and-replace functions.

Assembler

This full-featured assembler allows users to write 65816 assembly-language programs for the Apple IIGS computer, with complete support for the standard Apple IIGS file format and library files. The Apple IIGS Programmer'sWorkshop Assembler includes **macros** to facilitate assembly-language programming, and allows users to write their own macros and library files.

Macros are commands, each one of which replaces several assembly-language instructions or assembler directives. When a program is assembled, the assembler replaces macros with their equivalent instructions and directives.

The APW Assembler is specifically designed for writing relocatable code, because the APW Linker, System Loader, and Memory Manager are all designed to work most efficiently with relocatable code.

C Compiler

The Apple IIGS Programmer's Workshop C Compiler is a complete implementation of the C programming language. It consists of a C compiler, the Standard C Library, the Apple IIGS Interface Libraries, and the C SANE Library. The object files output by the C compiler are fully compatible with those output by the APW Assembler and consist of relocatable code.

Linker

The APW Linker takes the files (called *object files*) created by the APW Assembler or any of the APW compilers, and generates files that the System Loader can load into memory (*load files*). The linker resolves external references and creates relocation dictionaries, which allow the System Loader to relocate code at load time.

Although the APW Linker is a single program, conceptually there are two APW linkers:

- □ Normally, the linker is called directly by a shell command (such as the ASML command, which assembles and links a program). These commands provide a limited number of linker options; most linker options either are not available or are set to default values. In this manual, this aspect of the linker is referred to as the **standard linker.**
- □ Alternatively, all functions of the APW Linker can be controlled by compiling a file of linker commands. The linker command language, called *LinkEd,* allows you to do such things as place specific object-file segments in specific load-file segments, create dynamic load segments, set load addresses for nonrelocatable code, search libraries, and control the output printed by the linker. You can compile and execute LinkEd commands separately from your source code by using the ASSEMBLE, COMPILE, or ALINK commands of the APW Shell. In this manual, the aspect of the linker controlled by LinkEd files is referred to as the **advanced linker.**

The advanced linker is provided for programmers who require maximum flexibility from the system; for most purposes, the standard linker is completely adequate. When a statement in this book applies equally to the standard and advanced aspects of the APW Linker, the terms *APW Linker* or *linker* are used.

Because all Apple IIGS Programmer's Workshop assemblers and compilers create object code that conforms to the same format, the APW Linker can link together object files written in any combination of the development-environment languages.

Utility programs

The Apple IIGS Programmer's Workshop includes several programs, called APW Utilities, that perform functions not built into the shell. Utilities include the following.

- **Compact,** which makes load files more compact so they load faster and take up less space on disk.
- **Crunch,** which combines multiple object files created by partial assemblies or compiles into a single object file.
- **DumpOBJ,** which lists an object-module-format file to standard output (usually the screen).

- **Equal,** which compares two files or directories for equality of their contents, dates, and file types.
- **Files,** which lists the contents of a directory, including subdirectories. Files can also search for a file whose name contains a string you specify.
- **Init,** which initializes (formats) a disk.
- **MacGen,** which generates a custom macro file for a program.
- **MakeBin,** which creates a ProDOS 8 binary file from a ProDOS 16 load file.
- **MakeLib,** which creates a library file from object files or modifies an existing library file.
- **Search,** which searches a text or source file for a string that you specify.

ProDOS 8 binary files are executable standard-Apple II programs.

Apple IIGS Debugger

The Apple IIGS Debugger is documented in the *Apple IIGS Debugger Reference.*

To facilitate the debugging of programs, Apple provides a debugger that works with 65816 machine code. The Apple IIGS Debugger allows you to trace or step through a program one instruction at a time or to execute the program at full speed; in either case, you can insert breakpoints at which the debugger halts execution so that you can inspect the contents of the registers, memory, direct page, and stack.

The debugger can display a variety of types of information on the screen, including a disassembly of the code being traced, the contents of memory, the normal display of the program being tested, the contents of the program's direct page, the contents of Apple IIGS registers, and the contents of the program's stack.

Because the debugger can provide only an assembly-language listing of machine code, it is most useful for debugging programs written in assembly language. However, if you have a good understanding of how your high-level-language program is compiled into machine code, you can use the the debugger to help find the subroutine containing the problem.

❖ *Note:* The Apple IIGS Debugger is not part of APW. It is a separate product, available through APDA. See Chapter 9.

Language considerations

See Chapter 9 for more information on the Apple Programmer's and Developer's Association (APDA).

The APW package includes a powerful 65816 assembler. At the time of this printing, the other languages available for APW include C and Pascal. The APW environment is designed to support any number of programming languages; check with your Apple dealer and the Apple Programmer's and Developer's Association to find out what other languages are available. Before you purchase any language, make sure that it creates APW-compatible files and provides full and convenient toolbox support.

One of the advantages of working with APW is that the object files created by any APW assembler or compiler are compatible with those created by any other assembler or compiler. This means that you can link together routines written in any combination of APW languages to create a program.

For example, you can write an application in a high-level language such as C or Pascal, in order to make it portable to other computers and to speed up development time. Most programmers find it faster to write programs in high-level languages than in assembly language. Once the program is complete, you can determine which routines run most slowly and then write assembly-language versions of only those routines to enhance the performance of the program.

❖ *Parameter-passing:* The exact method by which parameters are passed is usually of no concern to your application as long as you work in a single language—your language's interface libraries and compiler take care of all parameter passing to and from the toolbox and among routines. However, if you are writing a segmented program where parameters are passed between routines written in *different* languages, you need to understand the parameter-passing details of your system. See your language reference for futher information.

Source files, object files, and load files

The Apple IIGS uses three fundamental types of program files: source files, object files, and load files.

- **Source files** are ASCII files consisting of code and data, and follow the conventions of a particular programming language.
- **Object files** are binary files created by assemblers and compilers; they represent an intermediate step in the program-development process. Object files cannot be read and modified like source files; neither can they be loaded by the System Loader. Object files (and their close relatives, library files) are used only as input to the linker.
- **Load files** are binary files created by the linker. Load files can be loaded by the System Loader. They cannot be used as input to the linker; if you want to link a new routine to a program you must go back to the object files to do so.

Object module format is defined in the *Apple IIGS Programmer's Workshop Reference.*

There is a single binary file format used by APW and the Apple IIGS operating system: the Apple IIGS **object module format** (OMF). Although OMF defines the structure and record types of both object files and load files, do not get the impression that object and load files are two versions of the same thing. They share some similarities of structure, but object files and load files serve different purposes and are read by different programs.

Symbolic references and relocatable code

A source file consists of programming-language instructions, directives, functions, and so forth, together with data needed by the program. In the source code, a specific instruction, subroutine, or block of data is often labeled with a name. You can refer to the name, for example, when you want to execute a subroutine. Such a label is called a **symbolic reference** (that is, a *symbol* that can be *referenced* or referred to).

In high-level programming languages, the use of symbolic references is often the only way to jump from one place in a program to another. In assembly language, on the other hand, it is possible to specify *absolute references,* the actual locations in the computer's memory to which you want the program to jump. Code whose location in memory is specified through absolute references is called *absolute.*

The code created by an APW compiler normally contains no absolute references, and so need not be loaded into a specific location in memory. It is referred to as *relocatable.* Note that this term is somewhat misleading: a relocatable program can be loaded into any location in memory, but it cannot necessarily be moved once it has been loaded.

The term *relocation* in this context means the process of inserting into the program in memory (or *patching*) the actual memory addresses to which jumps must be made. Relocation on the Apple IIGS is done during program load by the System Loader.

Absolute code, relocatable code, and the process of relocation by the System Loader are discussed in more detail in Chapter 6.

When source code is assembled or compiled, it is converted into object code containing machine-language instructions, data, and symbolic references. Before the program is actually run, the symbolic references must be *resolved*—they must be replaced with code that the loader can use to patch in the proper addresses at load time. The program that resolves the symbolic references is the APW Linker. (The linker gets its name from the fact that it can combine, or link together, several object files to create a single load file.)

Do not write absolute code

The advantages of using relocatable code for the Apple IIGS are considerable. Relocatable code can be placed in memory at whatever location the Memory Manager chooses. Because desk accessories, shell programs, RAM-based tool sets, and so on are placed in memory by the System Loader and Memory Manager, absolute code is likely to conflict with other code already in memory. It is very unlikely that your program will have sole control of the computer when it executes.

Object module format exists primarily as a specification for relocatable, segmented code. The Apple IIGS System Loader and the Memory Manager are designed to support relocatable code. The APW Assembler and compilers are all designed to generate relocatable code. It is easy to write relocatable code. *Do not write absolute code.*

Four steps to creating a program

The conversion of a source file into an executable program loaded in memory is done in four main steps, as follows (and shown in Figure 7-2):

1. You create one or more source files with a text editor. In this step you design the program, create its data structures, and write its subroutines. The source file(s) may be in one or more APW languages.
2. You assemble or compile each source file. Depending on the programming language used in the source file, the APW Assembler, C Compiler, or some other assembler or compiler processes the source file to create one or more object files. The object files contain 65816 machine-language instructions, data, and symbolic references to program routines.

   Object files, then, consist of machine-language instructions plus unresolved symbolic references.
3. You link the object files, using the APW Linker. The linker combines all of the object files into a single load file and resolves symbolic references. The linker verifies that every routine referenced is included in the load file; if there are any routines that the linker has not found when it has finished processing all of the object files, then it searches through any available library files for the missing routines. The linker replaces symbolic references with entries in special tables it creates, called **relocation dictionaries.**

   The load file, then, consists of blocks of machine-language code that can be loaded directly into memory (called **memory images**), plus relocation dictionaries that contain the information necessary to patch address references when the program is loaded into memory.
4. You execute the load file. It is loaded into memory by the System Loader. The loader calls the Apple IIGS Memory Manager to request blocks of memory for the load file, loads the memory images, and uses the relocation dictionaries to patch the actual memory addresses into the machine-language code in memory.

❖ *Segments:* The entire load file is not necessarily loaded into memory at one time; all OMF files are divided into **segments,** which can be processed independently. OMF file segmentation is a fundamental Apple IIGS concept—what segments are is discussed in Chapters 1 and 6; how to create them is considered next.

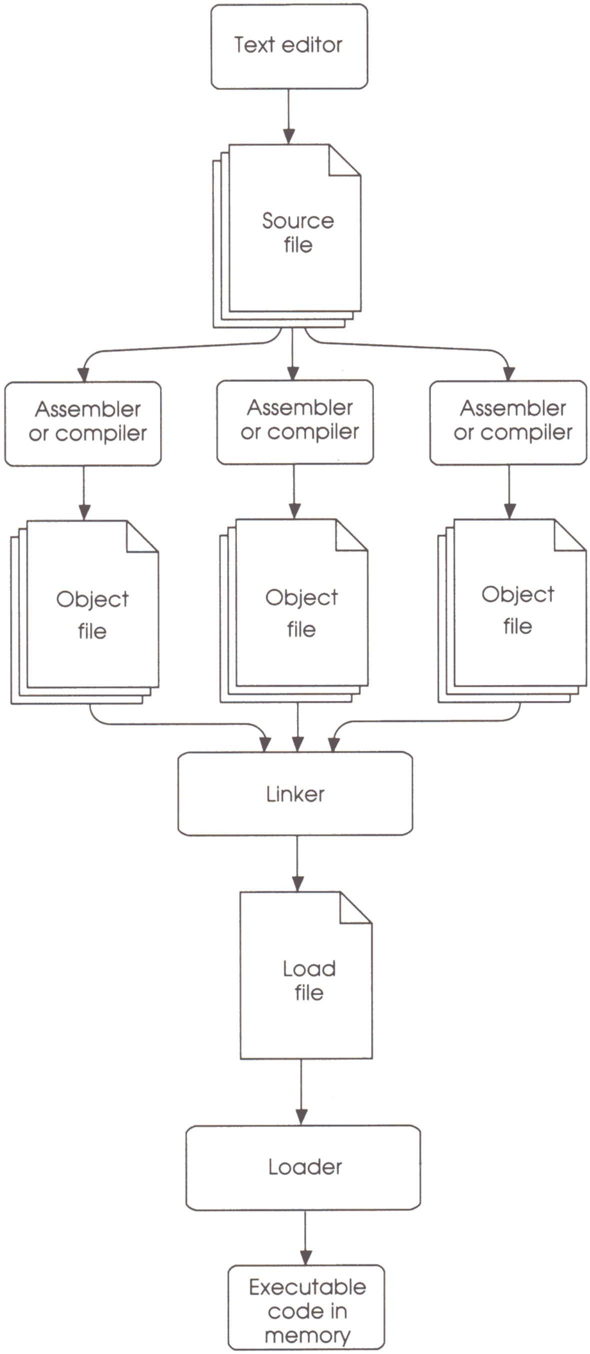


Figure 7-2
Creating an executable Apple IIGS program

Segments

When you write a program, it is generally considered good programming practice to divide the source code up into smaller units called *subroutines*. Subroutines make the program easier to write, read, and modify.

Similarly, it is easier to link a program if the object files are divided up into smaller units. In this case, we call the units *object segments*.

Load files, too, can be easier to load into memory if divided into smaller units. The subunits of load files are called *load segments*.

Important Although it is sometimes convenient to use the same or related divisions for subroutines, object segments, and load segments, it is important to keep in mind that they need not correspond. An object segment can contain one to many subroutines, and a load segment can contain one to many object segments.

The proper use of subroutines (source-code segments) is a subject for another book. How to create object segments and load segments by using APW is discussed in the following sections.

Defining object segments

Each APW language provides some means for specifying in your source file the subroutines that will go into each object segment, and the name of the object segment. In some languages, such as APW Assembly Language, you can specify the start and end for each object segment and can include any number of subroutines within the segment. In some languages, such as APW C, each subroutine becomes an object segment and the object segment name is the same as the subroutine name.

Figure 7-3 illustrates the conversion of source-file divisions into object segments.

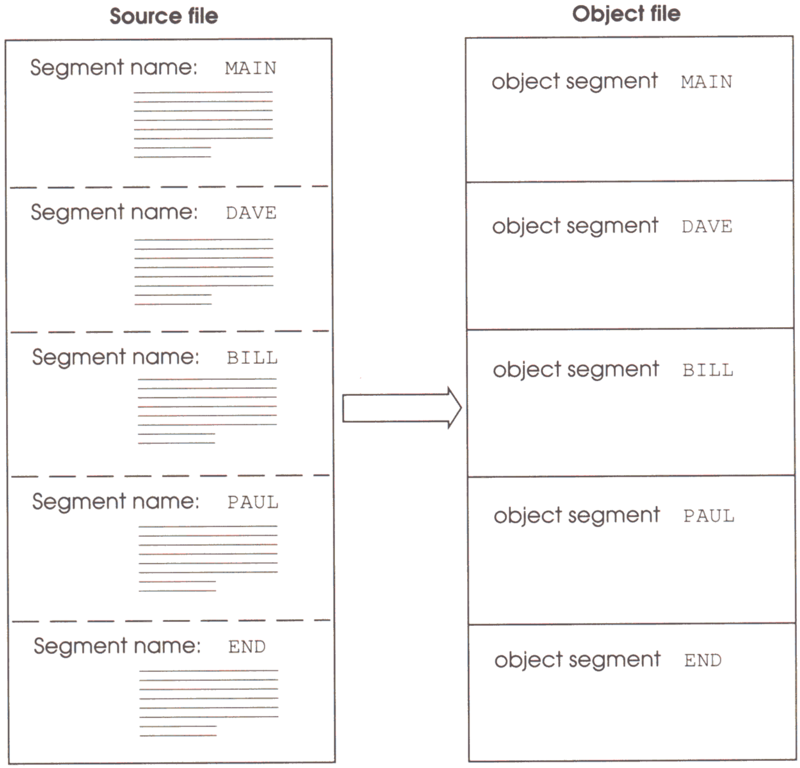


Figure 7-3
Assigning object segments in your source code

About load segments

The APW Linker creates load files from object files and library files. The linker cannot extract from an object file a portion of code smaller than an object segment. So, to the linker, the object segment is the fundamental unit of an object or library file. The load file consists of one or more load segments, each of which is loaded into memory separately. So, to the System Loader, the load segment is the fundamental unit of a load file.

Keep in mind that object segments and load segments are different entities. When you link a program, you tell the linker into which load segment you want each object segment to go. You can assign any number of object segments to the same load segment. You can assign each object segment to its own load segment, place the entire program into a single load segment, or anything in between.

How many load segments?

It is not generally necessary or desirable to divide a load file up into too many pieces, as the loader must handle each load segment independently. For small programs, in fact, you may want to have a single load segment.

On the other hand, it is often desirable to have more than one load segment. Because two consecutive load segments do not have to be loaded into contiguous memory locations, a segmented program may load into memory when a nonsegmented program won't fit. In fact, it is necessary to segment some programs, because the 65816 processor does not allow single blocks of program code larger than 64K to be loaded (there is no such restriction on blocks of data). Programs that consist of segmented load files can often be started up more quickly than unsegmented programs because not all the load segments have to be processed during the initial load. Some segments can be left on disk until they are needed (if ever).

What is the optimum number of load segments for a program? Only you can answer this question for your own program. If it is a small program, all of which must be in memory for the program to run, a single load segment might be fine. If the program is large enough that machines with smaller amounts of memory might have trouble loading it, several smaller segments might be better. Fortunately, you can segment your load file during the link stage of program development; if you are not sure how many load segments will be best, you do not have to make the decision while you are writing the source code.

Which segments should be dynamic?

When you specify load segments, you can designate some as dynamic. A **dynamic** segment is loaded automatically by the loader and Memory Manager when it is needed during program execution. A segment that is not dynamic is referred to as **static.** A static segment is loaded at program boot time and is never unloaded or moved during execution.

When the System Loader first loads a program, it loads all the program's static segments and then passes control to the program. When any part of the program references a routine in a dynamic segments, the loader finds the dynamic segment on disk and loads it. The dynamic segment then remains in memory for as long as the program is running, unless the program *unloads* the segment with a System Loader call. Unloading a segment makes its memory block purgeable, so the Memory Manager can remove the segment from memory if it needs space to load some other segment.

One segment of every program—the program's main routine—must be static. Any other segments may also be static, but (especially for large programs) the system will run more efficiently if infrequently used segments are dynamic. There are several advantages to designating a segment as dynamic. Because dynamic segments are not loaded until they are needed, for example, the initial load of a program is faster if some of the segments are dynamic. Also, if there is a possibility that the computer will run out of memory while your program is running, you can use dynamic segments to allow several parts of the program to share the same portion of memory.

Overlays are program segments that are alternately loaded at exactly the same memory address. No two overlay segments can be in memory at the same time, and no other program can use that memory range.

When dynamic segments share the same general area of memory, they are similar to **overlays.** However, dynamic segments are much more versatile than overlays, because dynamic segments (assuming they are also relocatable) can be loaded at any location in memory when needed. Furthermore, one segment need not be removed from memory to load the next. A dynamic segment that is not being used is removed (purged by the Memory Manager) only if the application permits it (with an unload call), *and* only if the memory is needed for something else. Otherwise, the segment remains in memory and need not be reloaded the next time it is called.

For large programs, you will probably want to see what difference it makes to designate a particular segment as dynamic. Sometimes, for example, it may be more desirable to accept a delay in the initial load of a program than to have the program pause while it loads a dynamic segment during execution.

To try out a segment as both static and dynamic, you can either change the source file and recompile/reassemble, or use the advanced linker when you link the file. Most APW languages let you specify in the source file that a particular load segment is to be made dynamic. On the other hand, if you use the advanced linker, you need not recompile the program to change the type of a single segment. Either way, you do this when you specify load segments, as described next.

Assigning load segments in your source code

LinkEd, the standard linker, and the advanced linker are discussed under "Apple IIGS Programmer's Workshop," earlier in this chapter.

You can assign object segments to load segments with source-code directives or with LinkEd commands. Even if you make source-code load-segment assignments, you can always override those assignments at link time by using a LinkEd file rather than the standard linker.

In APW Assembly Language, for example, the beginning of each object segment is indicated with a directive (such as `START` or `DATA`). The label of the directive is the object segment name. You can use these directives to specify the name of the load segment to which each object segment should be assigned, by putting the load segment name in the operand field of the directive.

In APW C, on the other hand, each function is an object segment and the function name is used as the object segment name. In this case, you use the `segment` directive to indicate the start of a load segment; all functions between that `segment` directive and the next `segment` directive are assigned to the same load segment.

Figure 7-4 illustrates the assembly-language method, using the same object file as that in Figure 7-3. Note also from Figure 7-4 that you don't *have* to specify a load segment name for every object segment—all object segments without load-segment names are put into a single unnamed load segment.

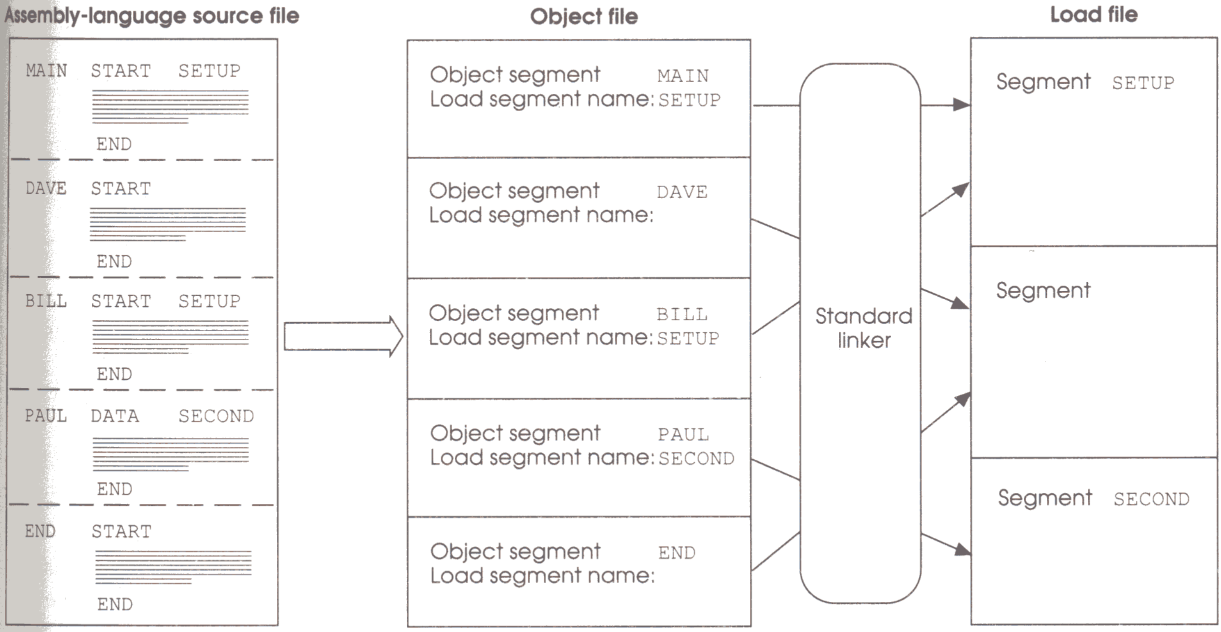


Figure 7-4
Assigning load segments in your source code

If you use the standard linker (that is, if you do not use a LinkEd file), the source-code load-segment assignments are used when you link the file. Object segments assigned to the same load segment need not be contiguous in the source file; in fact, they do not even have to be in the same source file. The linker places all of the object segments that have the same load segment name into the same load segment.

❖ *Order of segments:* The order in which the linker finds the load segment names in the source file is the order in which it places the load segments in the load file. If the order of the load segments in your load file is important, then you must either order your source code accordingly, or use a LinkEd file to link the program.

The advantage of the standard linker over the advanced linker is that the standard linker is quite automatic. You do not have to list either the object segments or the load segments in the link command. Library files in the APW library prefix are searched automatically, and you can specify any other library files you wish.

But there are some disadvantages to the standard linker:

Initialization segments and direct-page stack segments are discussed in Chapter 6.

- ☐ You must alter the source code to alter load-segment assignments.
- ☐ Some APW languages may not allow you to assign special load segment types (such as initialization segments or direct-page/stack segments) in the source code.
- ☐ All of the object segments in the source code are linked, whether you want to include them in the load file or not.

If any of these restrictions cause a problem for you, you can use the advanced linker to link the file, as described next.

Assigning load segments with a LinkEd file

The APW Linker can recognize both the names of the object segments in an object file and the names of the load segments (if any) to which those object segments are assigned. You can use a LinkEd file to take advantage of this fact.

For example, suppose you have written, compiled, and linked a program (using the standard linker), and you find that one load segment is larger than 64K. Because no single block of code larger than 64K can be loaded into memory, you must break this load segment into smaller pieces. Rather than changing the load segment assignments in the source code and recompiling the program, you can link the program with the advanced linker, using LinkEd commands to specify the names of load segments and the object segments that go into each load segment.

Figure 7-5 illustrates this process. Note that the object file is identical to the object file in Figures 7-3 and 7-4. Let's assume that the unnamed load segment is too large. By using a LinkEd file, you place object segments `DAVE` and `END` into separate load segments, named `NANCY` and `LAST`, respectively. The code in object segment `END` has been put at the end of the program. Therefore we have accomplished two things by using the advanced linker: we have split one large load segment into two smaller load segments, and we have changed the order in which the code appears in the load file.

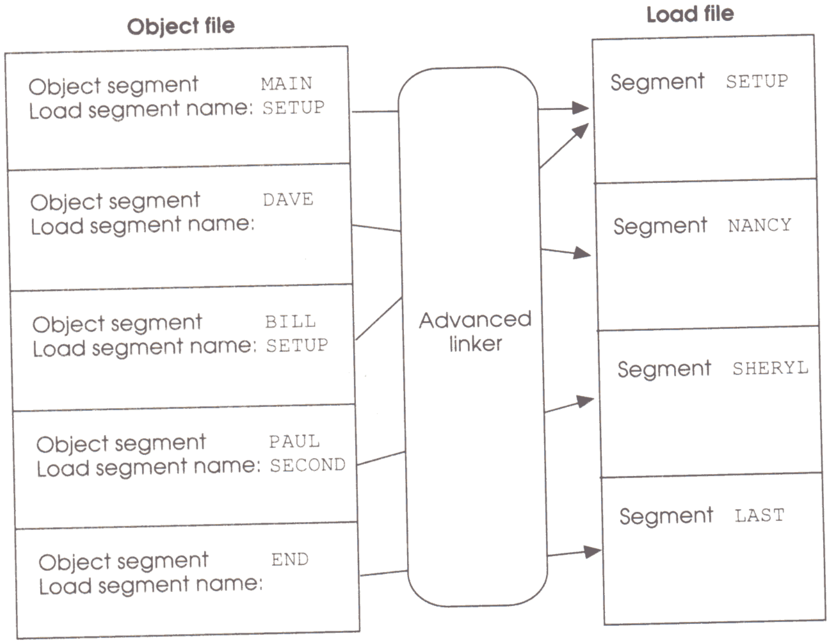


Figure 7-5
Assigning load segments with the advanced linker

The advanced linker gives you the freedom to ignore source-file load-segment assignments and to specify into which load segment each object segment should go. It also lets you specify special segment types for load segments, and the filename and file type of the output file. On the other hand, you must specify each object file to be included and each object segment to go into each load segment.

For small programs with only a few object segments or for larger programs with a simple load-file structure, the standard linker is easier to use. If you are developing a large program with many dynamic segments or with special segments such as a direct-page/stack segment or initialization segments, the advanced linker gives you much more flexibility. By changing the LinkEd file, you can change the number and sequence of load segments, the object files and object segments linked, and the segment types of load segments. For such a program, it is well worth the time and effort to learn how to use the LinkEd commands and to write a LinkEd file.

For more details on the standard and advanced linkers, see the *Apple IIGS Programmer's Workshop Reference*.

❖ *Note:* In using dynamic segments, it is important that the volumes containing all needed segments and libraries be on line at run time. If the System Loader cannot find a dynamic segment it needs to load, execution halts and the user is requested to mount the proper volume.

Library files

Library files are object files whose segments contain routines useful to many different programs. In APW, all library files are in object module format, regardless of the language of the source file. An Apple IIGS library file (ProDOS filetype $B2) can therefore be used by a program written in any source language. Some languages, such as APW C, come with a set of library files used by that language.

A **global symbol** is a label in one segment that can be referenced in another segment, as opposed to a **local symbol,** which can be used only within the segment in which it is defined.

A library file includes a special segment at the beginning of the file, called the **library dictionary segment.** The library dictionary segment is the first segment of a library file; it contains the names and locations of all the **global symbols** in the file. The linker uses the library dictionary segment to find the segments it needs.

When the linker processes one or more object files and cannot resolve a symbolic reference, it assumes that it is a reference to a segment in a library file. If you use the standard linker, it automatically searches all of the files in the APW library prefix (prefix number /2—usually *volume*/`APW/LIBRARIES/`, where *volume* is the volume name of your boot disk) as well as any library files you specify on the command line. If you use the advanced linker (that is, if you use a LinkEd command file), the linker searches only the library files that you specify. Unless you are using the advanced linker, you do not even need to know the names of the library files in order to use them; the standard linker automatically finds the files and extracts the segments it needs.

Creating library files

You can create your own library files from one or more object files by using the APW utility program MakeLib. Figure 7-6 illustrates the library-file creation process. You specify one or more object files to be included in the library file. MakeLib concatenates the files and creates the library dictionary segment.

The library dictionary segment makes it possible for the linker to search a library file for global symbols (the names of the subroutines it contains) much more rapidly than it can search an object file. Consequently, the linker will search a library dictionary segment several times if necessary to find segments referenced by other segments in the library file, and the sequential order of the segments in a library file is not important. But if you use several library files, the order in which the files occur is important because each is processed only once. It is for that reason that MakeLib allows you to include several object files in a single library file.

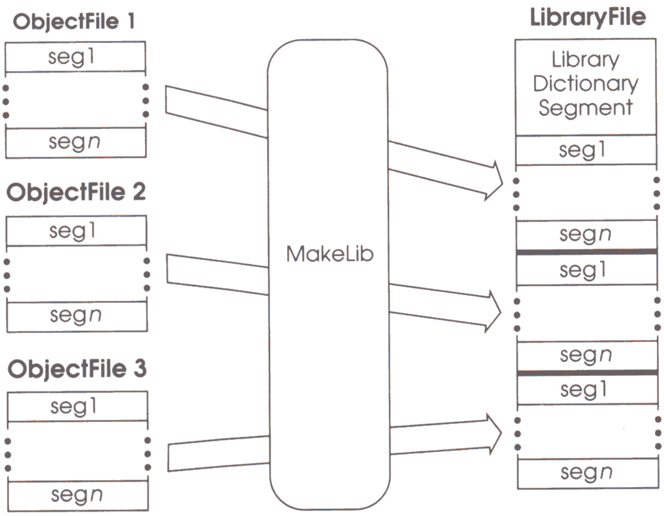


Figure 7-6
Creating a library file

MakeLib is described in detail in the *Apple IIGS Programmer's Workshop Reference*.

In addition to creating library files, the MakeLib utility allows you to modify existing library files, and even to recreate an object file that was a component of a library file.

Creating segmented code: three examples

This section presents examples of segmented programs. Three small program examples are provided: a program consisting of a single, static load segment; a program containing several static load segments; and a program using dynamic segments.

❖ *Note:* These examples are simple, text-based sample programs meant only to illustrate segmentation concepts. Your programs, whether segmented or not, should be event-driven, desktop-style applications.

A single, static load segment

The following is a typical sequence for writing, compiling, and linking a simple one-segment program. It has only one `START` directive, so only one segment is created. The segment is not explicitly made dynamic, so it is static.

1. Boot APW and set the system language to the language type of the source code you intend to write. We are going to write a simple assembly-language file for this example, so enter the following command:

   ```
   ASM65816
   ```

2. Set the default prefix to the subdirectory you want to use for your files. If your work disk is called `/MYPROGS`, for example, enter the following command:

   ```
   PREFIX /MYPROGS
   ```

3. Open a file for editing. We will call our source file `HW`. Enter the following command:

   ```
   EDIT HW
   ```

4. Write the source code for your program. For our example, type in the following program:

```
         KEEP       HELLO                 Output filename
         MCOPY      2/AINCLUDE/M16.UTIL   Macro file
MAIN     START                            Beginning of segment
         PHK                              Set data bank equal
         PLB                                to code bank
         WRITELN    #'Hello world!'       Macro that writes string
         LDA        #0                    Set error code to 0
         RTL                              Return to shell
         END                              End of segment
```

5. Press Apple-Q to quit the Editor. When the Quit menu appears, press S to save the file to disk, then E to return to the APW Shell command line.

6. To assemble, link, and execute the file HW, enter the following command:

```
RUN HW
```

The words `Hello world!` should appear on the screen, following the diagnostic output of the assembler and linker. If they do not, check your source file for errors and try again.

7. You now have a file on your work disk called HELLO. To execute this program, enter HELLO from the APW Shell command line.

❖ *Note:* This program cannot be executed from a finder or program launcher. It must run under APW.

Several static load segments

It is often desirable to write a program that consists of more than one load segment. For example, when there is no single contiguous block of memory large enough to load an entire program, the program may still be loadable if it is divided into several load segments. The program that follows is divided into three object segments, and each object segment is assigned to a different load segment.

See Chapter 8 for requirements for shell applications.

This program also illustrates a few of the basic functions that should be performed by any *shell application* before it begins to run: reading the User ID assigned to the program, reading the ID of the shell program that launched it, and checking the command line for parameters. This sample program merely prints this information to the screen; an actual application could do much more:

- ☐ It could use the User ID in calls to the Memory Manager and System Loader.
- ☐ It could use the Shell ID to determine whether it was launched by the shell program under which it was designed to run. For example, a compiler designed to run under APW might not be able to run under ProDOS or under another shell.
- ☐ It could use the parameters on the command line for whatever purpose the shell application was created to fulfill.

The program listed below has three segments: two that begin with a START directive, and one that begins with a DATA directive. The program assembles into the object segments MAIN, WRITE, and MSG, which are linked into the load segments MAIN, OUTPUT, and LABELS, respectively. To create the program, first use the following commands to set the current language to 65816 assembly language and to enter the editor:

```
ASM65816
EDIT SAMPLE.SRC
```

Then type in the following program:

```
            KEEP            SAMPLE
            MCOPY           SAMPLE.MACROS

MAIN        START           MAIN                Start segment
*       SET UP ENVIRONMENT
CLINE       GEQU            0                   Define CLINE as direct page
            PHK                                 Set data bank register equal to
            PLB                                   program bank register
            STA             USER_ID             Accumulator holds User ID
            STY             CLINE               X and Y registers contain
            STX             CLINE+2               pointers to command line
            PUSHWORD        USER_ID             Convert User ID to
            PUSHLONG        #USERID+1             hex number
            PUSHWORD        #4                    ASCII
            _INT2HEX                              string
            JSL             WRITE               Jump to next segment
            RTL
USER_ID     ENTRY
            DS              2                   Reserve space for User ID
USERID      ENTRY
            STR             '    '              Reserve space for User ID ASCII string
            END



WRITE       START           OUTPUT              Start second segment
*       WRITE USER ID TO SCREEN
            USING           MSG                 Use data in data segment
            PUSHLONG        #USRMSG             Pointer to output string
            _WRITECSTRING                       Writes 'User ID = '
            PUSHLONG        #USERID             Pointer to User ID ASCII string
            _WRITELINE                          Writes User ID, Carriage Ret
            LDA             CLINE               If pointer to
            ORA             CLINE+2               command line = 0,
```

```
        BNE                LB1                 no command line
        PUSHLONG           #NOLMSG            Pointer to output string
        _WRITECSTRING                         Writes 'No command line'
        JSR                LB5                If no command line, go to end
*   WRITE SHELL ID TO SCREEN
LB1     PUSHLONG           #IDMSG             Pointer to output string
        _WRITECSTRING                         Writes 'Shell ID = '
        LDY                #0                 Use Y for offset into Shell ID
        LDX                #8                 Shell ID is 8 chars, use X for counter
        PHX                                   Save X on stack
        PHY                                   Save Y on stack
LB2     PUSHWORD           [CLINE],Y          Push next letter of shell ID on stack
        _WRITECHAR                            Write one char of shell ID
        PLY                                   Pull Y from stack
        PLX                                   Pull X from stack
        INY                                   Increment Y
        DEX                                   Decrement X
        PHX                                   Save X on stack
        PHY                                   Save Y on stack
        CPX                #0                 Compare X to 0
        BNE                LB2                Return to LB2 if X not 0
        PLY                                   Pull Y from stack
        PLX                                   Pull X from stack
        PUSHWORD           #$0D               Write
        _WRITECHAR                              Carriage Return
*   WRITE COMMAND TO SCREEN
        PUSHLONG           #COMMSG            Pointer to output string
        _WRITECSTRING                         Writes 'Command is '
        LDY                #8                 Use Y for offset into command line
        PHY                                   Save Y on stack
LB3     LDA                [CLINE],Y          Load next character into accumulator
        AND                #$007F             Just look a t low 7 bits
        CMP                #' '               Test for Space character
        BEQ                LB4                Stop after first space
        PHA                                   Push next letter of command on stack
        _WRITECHAR                            Write one char of command string
        PLY                                   Pull Y from stack
        INY                                   Increment Y
        PHY                                   Save Y on stack
        BRA                LB3                Return to LB3
LB4     PUSHWORD           #$0D               Carriage Return
        _WRITECHAR
*   WRITE PARAMETERS TO SCREEN
        PUSHLONG           #PARMSG            Pointer to output string
        _WRITECSTRING                         Writes 'Parameters are'
```

```
             PLY                                   Pull Y from stack
             INY                                   Increment Y
LB6          LDA              [CLINE],Y            Load next character into accumulator
             AND              #$00FF               Test for Null
             BEQ              LB7                  Stop after Null
             PHY                                   Save Y on stack
             PHA                                   Push next letter of parameters on stack
             _WRITECHAR                            Write one char of parameters
             PLY                                   Pull Y from stack
             INY                                   Increment Y
             BRA              LB6                  Return to LB6
LB7          PUSHWORD         #$0D                 Carriage Return
             _WRITECHAR
LB5          LDA              #0                   Set return code to 0
             RTL                                   Return to segment Main to end routine
             END                                   End of segment


MSG          DATA             LABELS               Begin data segment
IDMSG        DC               C'Shell ID is:  ',H'00'
USRMSG       DC               C'User ID is:  ',H'00'
COMMSG       DC               H'0A',C'Command is:  ',H'00'
PARMSG       DC               H'0A',C'Parameters are:  ',H'00'
NOLMSG       DC               C'No Command Line.',H'00'
             END                                   End data segment
```

Press Apple-Q to quit the Editor. When the Quit menu appears, press S to save the file to disk, then E to return to the APW Shell command line.

The program uses macros in several places, including macros for calls to the Integer Math Tool Set and the Text Tool Set. Execute the following command to create a macro file for the program:

```
MACGEN SAMPLE.SRC SAMPLE.MACROS 2/AINCLUDE/M16.TEXTTOOL 2/AINCLUDE/M16.UTIL
2/AINCLUDE/M16.INTMATH
```

To assemble and link the program, use this command:

```
ASML SAMPLE
```

To run the program, enter this command:

```
SAMPLE ONE TWO BUCKLE MY SHOE
```

The output should look like this (the actual User ID will vary, as a new one is assigned each time the program is run):

```
User ID is:  1129
Shell ID is:  BYTEWRKS
Command is:  SAMPLE
Parameters are:  ONE TWO BUCKLE MY SHOE
```

Dynamic segments

As an example of a program with dynamic segments, we can take the same multisegment example we just created and make one segment dynamic. What's more, we won't have to rewrite a single line of the program's code or re-assemble it to do so.

To make the second segment of the program dynamic, you can link the program with a LinkEd file. First, if you have not done so already, assemble the program (without linking it) with the following command:

```
ASSEMBLE SAMPLE
```

Use the following commands to set the current language to the LinkEd language and to enter the editor:

```
LINKED
EDIT SAMPLE.LINK
```

Type in the following LinkEd program:

```
KEEP SAMPLE
SEGMENT MAIN
SELECT/SCAN SAMPLE.STD (MAIN)
SEGMENT/DYNAMIC OUTPUT
SELECT/SCAN SAMPLE.STD (WRITE)
SEGMENT LABELS
SELECT/SCAN SAMPLE.STD (MSG)
```

Press Apple-Q to quit the Editor. When the Quit menu appears, press S to save the file to disk, then E to return to the APW Shell command line.

Execute the following command to link the program:

```
ALINK SAMPLE.LINK
```

The APW Linker executes this LinkEd file. Each `SEGMENT` command starts a new load segment. Each `SELECT/SCAN` command scans through the files with the root filename `SAMPLE.STD` for the object segment named in parentheses. The load segment `OUTPUT` is dynamic. The load file, `SAMPLE`, contains four segments; the fourth load segment is the Segment Jump Table, created by the linker.

When you launch this version of `SAMPLE`, the first and third segments are loaded immediately, but the second segment is not loaded as part of the initial load. When the `JSL` to `WRITE` is executed, the loader loads the second segment.

❖ *Unloading:* Note that, once loaded, the dynamic segment remains in memory throughout execution of the program. To make this segment available for automatic unloading by the Memory Manager, you must include an Unload Segment call at the end of the segment.

Debugging

A variety of software instruments exist to help you locate and correct errors in your Apple IIGS programs. Some are sophisticated and some are simple. Although nothing can make debugging *easy,* the more experience you gain with these aids, the more efficiently you can find and solve problems.

Debugging with desk accessories

The fact that Apple IIGS code is typically relocatable can be a problem during debugging. You can't control where the loader puts your program, and once it is in memory, you have no obvious way to locate it. How can you debug something you can't even find?

Classic desk accessories are described under "Supporting Other Desktop Features" in Chapter 5.

Loader Dumper and Memory Mangler are two classic desk accessories (CDA's) provided with the Apple IIGS Debugger (described later in this section). They can give you very basic, and thus very important, information on exactly where in memory all the parts of your program are. Furthermore, because they are desk accessories, they are instantly available from within your program or debugger.

Loader Dumper

Loader Dumper is a classic desk accessory that permits you to dump (print out) the System Loader's data structures: the Memory Segment Table, Pathname Table, Jump Table, Loader global variables, load-segment information, and other information used by the System Loader.

A principal use of the Loader Dumper is to get your program's User ID from the Pathname Table. Then, using Memory Mangler (described next), you can find out where in memory all your program's segments are.

Memory Mangler

Memory Mangler is a classic desk accessory that can give you a listing of all allocated memory blocks with their associated User ID's, sizes, addresses, attributes, and other information.

Most commonly, you would use Memory Mangler to inspect all your programs' segments and buffers in memory. They are identified by User ID, which you might have obtained from running Loader Dumper. Once you have found a segment you want to look at more closely, you can go directly from Memory Mangler into the Apple IIGS Debugger or into the Monitor program (described next), to do detailed inspection and debugging.

❖ *Note:* Memory Mangler also allows you to execute Memory Manager calls, so you can use it as an exerciser program, to practice calls before writing them into your code. Even though this is not a direct debugging function, it is very useful because you need to understand the Memory Manager well. Memory-management errors are among the most common and most elusive bugs on the Apple IIGS.

Debugging with the Monitor program

The Apple IIGS Monitor program is a set of ROM-based routines that give the user direct access to program code in memory. Using the Monitor, you can perform these tasks:

- Inspect and modify the contents of any location in memory, in either hexadecimal or ASCII format.
- Move, compare, or fill ranges of memory.

- Search for specified patterns in memory.
- View and change the contents of various microprocessor and software registers and flags.
- Execute programs from within the Monitor.
- Disassemble code in memory.
- Use the mini-assembler to assemble small programs.

A special convenience of the Monitor is that you can call it from within the program you are debugging. To do so, however, you must make the call from bank $00, and the machine must be in emulation mode—with the Data Bank and Program Bank registers set to zero and with the direct page equal to the zero page. In other words, the machine must look exactly like a standard Apple II.

The Monitor program and how to call it are described in the Apple IIGS Firmware Reference.

A second method is to make a Miscellaneous Tool Set call to invoke the Monitor. In this case, the machine must be in full native mode and the Memory Manager must have been started up. The call can be made from anywhere in memory.

Debugging with the Apple IIGS Debugger

The Apple IIGS Debugger allows you to load your program into memory and to run through it under the debugger's control. As the program executes, you can examine the contents of the 65816's registers, of your program's direct page and stack, and of any locations in memory in which you are interested. You can interact with the program as required, returning to the debugger's display at breakpoints that you set, or when the program crashes.

The Apple IIGS Debugger can display an assembly-language disassembly of your program's machine code. It cannot execute your source code or recreate your source code from machine code. Therefore, the debugger is easiest to use with assembly-language programs. However, even if your program was written in a higher-level language and you have no knowledge of assembly language, you can use the debugger to determine in which load segment the problem lies. You can also gain a better understanding of the operation of your program by examining the contents of the stack, direct page, memory, and registers.

The debugger is described in detail in the Apple IIGS Debugger Reference.

We do not have the space here to examine in detail all of the abilities of the Apple IIGS Debugger, but we will give you some hints that should help get you started debugging your program.

Debugging segmented programs

In order to use the Apple IIGS Debugger to debug a segmented program, you must know where in memory each segment has been loaded. In the case of a dynamic segment, you must know whether it has been loaded and, if so, where. This information is available through the Loader Dumper desk accessory, described earlier in this section.

To load your program by using the debugger and to determine where in memory each segment is loaded, use the following procedure:

1. Start up the debugger.
2. Use it to load your program into memory.
3. Call the Loader Dumper from the desk accessories menu.
4. Use the Loader Dumper to get the User ID of your program.
5. With that User ID, use the Loader Dumper to get a listing of all your program's load segments and their memory addresses.

You now have several possible courses of action open to you. If you do not have any idea in which load segment your program is crashing, you can start by running the program until it crashes and then examining the debugger display to determine the location of the problem instruction. If you know in which segment the problem lies, you can go immediately to that segment, or you can set a breakpoint at the beginning of that segment and run the program until it stops automatically at that breakpoint.

Watching a running disassembly

If your program does not require any input from the keyboard, you can watch a disassembly on the debugger screen as the program executes to find the exact location at which it goes astray. This technique will probably be useful only for short programs or programs that crash almost immediately upon execution, because the program will execute very slowly while the debugger display is on the screen.

To run your program under control of the Apple IIGS Debugger, with a running disassembly appearing on the screen, use the following procedure:

1. Load your program with the debugger.

2. Put the debugger in single-step mode, starting at the first instruction of your program. Watch the contents of the registers and the stack (and any specific memory locations you have specified) as you execute individual commands.
3. You can leave single-step mode and execute commands automatically in quick succession by entering trace mode. Your program will begin executing under debugger control, one instruction at a time in rapid succession. Once in trace mode, you can stop execution at any time and then return to single-step mode

In trace mode, when your program executes a BRK instruction execution stops. The last instruction executed (the BRK instruction) is displayed on the screen, along with the previous several instructions executed. A BRK instruction is actually a null (a zero byte); because such an instruction is not a normal part of a program, the fact that your program executed one probably means that some previous instruction sent the program off to the wrong place in memory. With luck, the instruction that sent your program off into Never Never land will still be on the screen.

Using breakpoints

If you have to interact with your program in order for it to run, if you have some idea of which segment contains the bug, or if you just want to execute the program more quickly, you can set one or more breakpoints before running the program. A breakpoint is a location at which the debugger suspends execution of the program, giving you the opportunity to examine the disassembly and the state of the machine at that location.

To set breakpoints and run the program under debugger control, try the following procedure:

1. Load your program with the debugger.
2. As described above, use the Loader Dumper routine to determine the starting locations of the load segments of your program.
3. Back in the debugger, set breakpoints at the beginning of each load segment (if you do not know in which segment the bug lies) or at the beginning of any segment that you want to examine more closely.

4. Run your program under debugger control, with the debugger display turned off. When the debugger comes to a breakpoint, the program halts and the debugger's display appears on the screen, showing the location of the instruction at which the program stopped and other pertinent information. You can also view a disassembly of the program, starting at the breakpoint location.
5. While at a breakpoint you can switch to single step mode. Step through the segment one instruction at a time while watching the contents of the stack, the machine's registers, and up to 19 memory locations you specify. From single-step mode, you can return to executing the program automatically.

If at any time during execution of the program a dynamic segment is loaded, you can pause execution of your program and go back to Loader Dumper to find out where in memory it has been placed.

Breakpoints can be used for purposes other than finding a particular segment. Suppose, for example, that your program seems to run all right for awhile, then crashes after having lulled you into a false expectation of success. In this case, it is possible that some routine is failing, not the first time it is run, but only after going through several iterations. To handle such a situation without stopping the program every time the routine is executed, you can include a *trigger value* for a breakpoint. The debugger counts the number of times it encounters the breakpoint, and suspends execution only when the trigger value is reached.

If you must execute a routine at full speed in order for it to work correctly, you can insert *real breakpoints* into the code. When you do so, the debugger actually inserts BRK instructions into memory at the breakpoint locations. Trigger values work for real breakpoints that you have set; the debugger will still suspend execution any time it encounters a BRK instruction that you did *not* set as a breakpoint.

Using memory protection ranges

It may be that certain portions of your code must be executed at the full speed of the 65816 microprocessor. To cause this to happen automatically every time you trace through the program, you can set any areas of memory you choose as *code trace* ranges. When the program executes a jump to a location within a code trace range, the debugger relinquishes control to your program and the code is executed at full speed. The portion of memory used to run tool calls is automatically set as a code trace range when you load the debugger.

You can also set one or more portions of memory (the limits of your code as revealed by Loader Dumper, perhaps) as *code-window* ranges. If the program attempts to execute code outside the code-window ranges you have set, execution stops. You might want to set a code-window range, for example, if your program is executing a jump to some incorrect memory location and trashing memory before it stops, forcing you to reboot the machine every time you try to run the program with the debugger.

If your program loads a dynamic segment during execution and you want to pause as soon as control is transferred to the dynamic segment, you can set code window ranges to include all the static segments at the start of the program. Then when the dynamic segment is loaded and control is transferred to it, the program will be outside any code window range and execution will stop.

Important Once you have set any code-window range, no code will be executed that is not in a code-window range. Therefore, if you set a code-window range equal to the memory location of one of your program segments, you must set code-window ranges for all other segments that it calls.

Debugging multiple-language programs

One of the advantages of using the APW development environment is that it allows you to link together routines written in different programming languages. This facility can lead to unique problems, however, especially when information is passed between routines written in different languages.

Parameter passing may fail in your program for any of several reasons: you might have used a wrong variable type, for example, or a called routine might expect to receive parameters in a different order from the way they were passed by a calling routine.

To use the Apple IIGS Debugger to debug parameter-passing problems, use the following procedure:

1. Set breakpoints at the beginning of the calling segment and at the beginning of the called segment.
2. Run the program in trace or real-time mode until the first breakpoint is reached. Search this segment to find the JSL that calls the other segment.
3. Set a breakpoint just before the JSL that calls the second segment. You can remove the other two breakpoints now if you wish.
4. Run the program until the JSL breakpoint is reached. Parameters are normally passed either on the stack or in the A, X, and Y registers. The actual information passed may be a pointer to the data rather than the data itself. By examining the contents of the registers, the stack, and memory, determine the location of the parameter being passed, and see if it has the value you expect.
5. Execute the JSL. The return address should have been added to the stack.
6. Step through the segment in single-step mode. Is the called routine reading the parameters passed to it, in the proper form and order? By a careful study of the action of the called routine, you should be able to determine the source of the problem.
7. If all parameters are being passed correctly, perhaps the problem occurs when the results are passed back to the calling routine. Find the RTL, and study the stack and registers as before to determine whether the results are being passed correctly back to the calling routine.

The ProDOS 16 Exerciser

The ProDOS 16 Exerciser is on the disk that accompanies the *Apple IIGS ProDOS 16 Reference*.

The ProDOS 16 Exerciser is a program that allows you to practice making operating system calls in a controlled environment, before coding them into your applications.

The ProDOS 16 Exerciser is not really a debugging tool, but you can use it in several ways during the debugging process. For example:

- By practicing the ProDOS 16 calls you intend to use in your program, you can "debug" them in the sense that you can see exactly how they function, before writing them into your code.
- Because the Exerciser gives you direct access to file attributes, you can use it, for example, to change file types (such as from $B5 to $B3) without having to enter APW.
- The Exerciser allows you to inspect and modify the contents of any portion of memory, and any block on a disk—but see the warning below.
- The Exerciser allows you to enter the Monitor program directly. Once in the Monitor, you can use its debugging facilities, as described earlier.

Warning The ProDOS 16 Exerciser allows you unconstrained use of all ProDOS 16 calls, including those that modify disk directories and blocks. It also permits you to modify any portion of memory, including that occupied by system software or by the Exerciser itself. You can easily destroy the contents of a disk or cause a system crash. Be careful what you modify!

Chapter 8

What Type of Program to Write?

A list of all defined file types is given under "The ProDOS File System" in Chapter 6.

Under ProDOS 16 on the Apple IIGS computer, programs are classified by **file type.** Some rules for writing the following types of programs, most of which have unique file types, are given in this chapter:

- ☐ general applications (file type $B3)
- ☐ controlling programs, such as shells, switchers, and operating systems
- ☐ shell applications (file type $B5); that is, programs designed to run under a shell
- ☐ desk accessories (file types $B8 and $B9)
- ☐ initialization files (file types $B6 and $B7)
- ☐ interrupt handlers
- ☐ user tool sets (file type $BA)

For the most part, this book has been concerned with general applications (type $B3) only. However, if you are interested in writing some of the other, more special-purpose programs, the information in this chapter can help get you started.

General applications

In this book, **application** means a complete program, typically called by a user (rather than by another program), that can communicate directly with any other system software or firmware it needs. For example, word processors, spreadsheet programs, and language interpreters are applications. Data files and source-code files, as well as subroutines, libraries, device drivers, desk accessories, and utilities that must be called from other programs, are not applications.

To be a (stand-alone) application, an Apple IIGS program needs to meet certain requirements. It must

- consist of executable machine-language code
- be in Apple IIGS object module format
- have a file type of $B3
- use ProDOS 16 as its operating system
- observe the ProDOS 16 QUIT conventions
- get all needed memory from the Memory Manager

All other aspects of the program are up to you. But of course we strongly recommend that you design your progams to use the Apple desktop interface and follow the Apple Human Interface Guidelines.

❖ *ProDOS 8:* The above list refers specifically to Apple IIGS applications that run under ProDOS 16. Requirements for programs that run under ProDOS 8 are quite different; see the *ProDOS 8 Technical Reference Manual.*

Make it self-booting?

There are two ways to make your type $B3 application *self-booting,* so that it is automatically loaded and launched at system startup:

- Give it the filename extension `.SYS16`. By using this method, your program becomes a ProDOS 16 equivalent to a ProDOS 8 *system program* on a standard Apple II computer.
- Give it the filename `START` and place it in the `SYSTEM/` subdirectory. By using this method, your program substitutes for the finder or program launcher that normally executes first on the Apple IIGS.

ProDOS 8 system programs are described in the *ProDOS 8 Technical Reference Manual*

In either case, your progam must be the first (or only) program with the proper filename or filename extension that the boot loader program encounters on the boot disk. Figure 8-1 diagrams the program selection sequence at system startup.

For a more detailed description of system startup, see the *Apple IIGS ProDOS 16 Reference.*

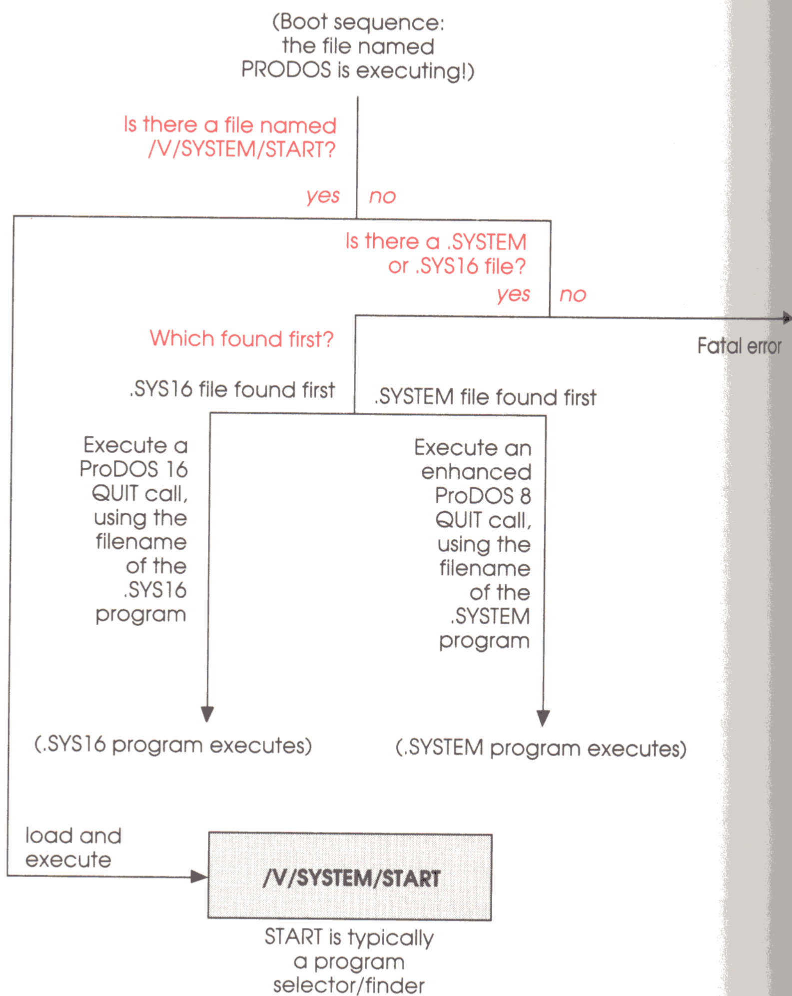


Figure 8-1
Startup program selection

❖ *Note:* Apple recommends that you do not name your application `START`—leave that name for a program launcher or finder. If you give your program a clearly identifiable name, the user can more easily launch it from any boot disk.

Make it restartable?

The concepts of dormant and restartable programs are discussed under "Loading Programs and Segments" in Chapter 6.

If you want your program to be able to be quickly relaunched from a *dormant* state in memory, it needs to be *restartable.* A restartable program reinitializes all its variables each time it gains control, without having to read in files or segments from disk. It also makes no assumptions about the state of the computer, such as register contents or flag settings, when it gains control. If all initialization information is in code statements in your program's initialization segments and static code segment(s), the program should be restartable.

For more information on the APW C language, see the *Apple IIGS Programmer's Workshop C Reference.*

It is difficult for programs in some languages to be restartable. In C, for example, all global variables are in segments named `~GLOBALS` and `~ARRAYS`, which must be initialized each time the program starts up. To get around this difficulty, the System Loader supports the concept of `RELOAD` segments. A `RELOAD` segment is a static data segment that is loaded from disk whenever an (otherwise) restartable program is launched from a dormant state. It contains whatever initialized data is needed by the program; the rest of the program's static segments (other than initialization segments) are not loaded at that time.

When your application quits, it passes a parameter to ProDOS 16 (and thence to the System Loader) stating whether the application is restartable or not. You must determine when you write the program what type it really is; neither ProDOS 16 nor the System Loader will check.

Controlling programs

A **controlling program** is a program that loads and executes other programs while itself remaining active in memory. An application needs to be a controlling program only if it must remain in memory after it calls another program. The APW Shell is a controlling program; ProDOS 16 is a controlling program.

Programs that run under shells are called **shell applications** and are file type $B5.

Writing a controlling program is far more involved than writing an application. This book does not show you how to write a controlling program; but if you do write one, please follow the guidelines below. They specify how your controlling program communicates with **shell applications.** See also the next section, "Shell Applications," for more information on how controlling programs and their subprograms interact.

- Your controlling program should use the System Loader's *Initial Load* function to load the subprogram. Initial Load returns the subprogram's starting address and User ID to your controlling program. When your controlling program passes execution to the subprogram, it should pass the subprogram's User ID in the accumulator.
- Your controlling program may also pass parameters and an identifier string to the subprogram, as described under "Shell Applications."
- Your controlling program is responsible for establishing the appropriate input and output vectors for its subprograms. For example, when ProDOS 16 launches a program, it initializes the Text Tool Set to use the Pascal I/O drivers for the keyboard and 80-column screen.
- Unless all its subprograms include direct-page/stack segments, your controlling program must provide a default direct-page/stack space for any subprogram that it launches. The controlling program should observe the ProDOS 16 conventions for register initialization and direct-page/stack allocation.

ProDOS 16 register and direct-page/stack conventions are discussed under "Setting Up Direct-Page/Stack Space" in Chapter 6, and fully described in the *Apple IIGS ProDOS 16 Reference*.

- Shell applications can terminate with either an RTL instruction or a ProDOS 16 QUIT call. If any of its subprograms might use QUIT, your controlling program must intercept all ProDOS 16 calls so that when the subprogram quits, the controlling program, rather than ProDOS 16, regains control.
- Your controlling program is totally responsible for disposing of the subprogram. When the subprogram is finished, the controlling program must remove it from memory and release all memory resources associated with its User ID. The best way to do this is to call the System Loader's User Shutdown function. If the program ends in a QUIT call, your controlling program is responsible for performing any other system tasks normally done by ProDOS 16 in response to a QUIT.

See "Quitting and Launching Under ProDOS 16," in Chapter 6.

Shell applications

See "Loading Programs and Segments" in Chapter 6 for a discussion of controlling programs and the System Loader's Initial Load function.

Shell applications (ProDOS 16 file type $B5) are executable load files that are run under a controlling program such as the APW Shell. The controlling program launches the shell application by calling the System Loader's Initial Load function, and transfers control to the shell application by means of a JSL instruction, rather than launching the program through the ProDOS 16 QUIT function. Therefore the shell does not shut down, and the program can use shell facilities during execution.

A shell application typically returns control to its shell with an RTL instruction. With a shell (such as the APW Shell) that intercepts ProDOS 16 calls, the shell application can end with a ProDOS 16 QUIT call.

Shell applications should use standard Text Tool Set calls for all nongraphics I/O; the controlling program is responsible for initializing the Text Tool Set routines.

❖ *Stand-alone:* A shell application can run alone under ProDOS 16 if it requires no support other than standard input from the keyboard and output to the screen. ProDOS 16 initializes the Text Tool Set to use the Pascal I/O drivers (discussed in the *Apple IIGS Toolbox Reference*) for the keyboard and 80-column screen. To be launched this way, a program must first be changed to file type $B3, and it must end with a ProDOS 16 QUIT call.

As soon as a shell application is launched, it should save the value of its User ID, passed in the accumulator from the controlling program. It should also check the X and Y registers for a pointer to the shell-identifier string and input line. The X register holds the high-order word and the Y register holds the low-order word of this pointer. The controlling program is responsible for loading this pointer into the index registers and for placing the following information in the area pointed to:

See an example of reading an identifier string in the second sample program under "Creating Segmented Code: Three Examples" in Chapter 7.

- □ An 8-byte ASCII string containing an identifier for the shell. The identifier for the APW Shell, for example, is `BYTEWRKS`. The shell application should check this identifier to make sure that it has been launched by the correct shell, so that the environment it needs is in place. If the shell identifier is not correct, the shell application should write an error message to standard error output (normally the screen), and exit with a ProDOS 16 QUIT call (if the controlling program supports it) or an RTL.

□ A null-terminated ASCII string containing the input line for the shell application. The shell can strip any I/O redirection or pipeline commands from the input line, because those commands are intended for the shell itself, but must pass on all input parameters intended for the shell application.

❖ *ProDOS 16:* ProDOS 16 does not support the identifier string or input line convention. When an application is launched by ProDOS 16, the X and Y registers contain zeros.

For more information on direct page and stack allocation, see "Setting up Direct-Page/Stack Space" in Chapter 6. See also the *Apple IIGS ProDOS 16 Reference*.

If the shell application does not have a direct-page/stack segment, it can expect the controlling program to provide a 1024-byte memory block in bank $00 for the shell application to use for its direct page and stack. Whether the shell application specifies a direct-page/stack segment or accepts the default, the address of the start of the direct-page/stack segment should be in the direct register (D), and the stack pointer (S register) should point to the last (highest-address) byte of the block containing the direct-page/stack segment.

Some shell applications may launch other programs; for example, a shell nested within another shell would be a controlling program as well as a shell application. Such an application must follow the conventions for both types of programs.

A shell application should use the following procedure to quit:

1. If the shell application has requested any memory buffers, it must release (dispose of) them.
2. The shell application should place an error code in the accumulator. If no error occurred, the code should be $0000. The code $FFFF can be used as a general (nonspecific) error code. Other error codes are up to the controlling program to define and handle.
3. The shell application should execute an RTL or, if the shell supports it, a ProDOS 16 QUIT call.

Desk accessories

A **desk accessory** is a small program that a user can run without closing down an already-running application. The Apple IIGS supports two different kinds of desk accessories:

- **Classic desk accessories (CDA'S)** are designed to execute in a non-desktop environment. The CDA interrupts the application and gets full control of the computer.

- **New desk accessories (NDA'S),** on the other hand, are designed to execute in a desktop environment. As such, they operate in a window and are subject to the same rules as an event-driven application. They are not stand-alone applications, however, because they rely upon another application to start up the Apple IIGS tools.

Neither type of desk accessory has a lot of extra programming overhead apart from the actual task the accessory performs. Both types depend heavily for support upon the Apple IIGS tool set called the Desk Manager.

For full details on the Desk Manager and its desk accessory suppport, see "Desk Manager" in the *Apple IIGS Toolbox Reference*.

Writing classic desk accessories

A classic desk accessory must start with a header consisting of a name string, a pointer to the start of the code, and a pointer to the shutdown routine.

```
StartofCDA      str'Name of DA'              ; DA name (this is an APW macro)
                dc i4'StartOfDACode'         ; Pointer to start of code
                dc i4'ShutDownRoutine'       ; Pointer to shutdown routine
```

When a CDA gets control from the Desk Manager, the processor is in full native mode. Because the Desk Manager has already saved the necessary parts of the old environment, the CDA can concern itself solely with its own work.

A CDA follows this basic procedure:

1. It initializes for the machine environment it needs. The Desk Manager has already saved the old state when the CDA gets control.
2. It does the actual work of the CDA. Like all Apple IIGS applications, a CDA should ask the Memory Manager for any space that it needs. In addition, the CDA must leave the stack as it was when the CDA got control.
3. It returns to the Desk Manager with an RTL. The Desk Manager then automatically restores the old state and returns to the desk accessory menu.

Every CDA must have a shutdown routine. The shutdown routine is called every time the Desk Manager shuts down.

Classic desk accessories have the ProDOS file type $B9. On disk, they must be in the `DESK.ACCS/` subdirectory of the `SYSTEM/` directory.

Writing new desk accessories

All new desk accessories are loaded from the disk at boot time. When an NDA gets control from the Desk Manager, the processor is in full native mode. By convention, the NDA can assume that the tool sets shown in Table 8-1 have already been loaded and started up. If the NDA needs any other tool sets, it must load and start them up itself.

Table 8-1
Tool sets loaded and available to new desk accessories

Tool set
Tool Locator
Memory Manager
Miscellaneous Tool Set
QuickDraw II
Event Manager
Window Manager
Control Manager
Menu Manager
LineEdit Tool Set
Dialog Manager
Scrap Manager

Note: The NDA may also assume that the Print Manager is available, although not necessarily loaded and started up

A new desk accessory has a structure fundamentally different from that of a desktop application. For one thing, it has no event loop—it relies on the application's event loop and the Desk Manager to open it, prod it into action, and close it. For another, it has only four nonprivate routines: init, open, action, and close:

- □ The Desk Manager calls the *init* routine to initialize the NDA when the Desk Manager starts up, and again when it shuts down.

- ☐ The Desk Manager calls the *open* routine when the NDA is selected by the user from the Apple menu. The open routine opens the desk accessory window and returns a pointer to it.
- ☐ The Desk Manager calls the *action* routine in response to an event within the NDA window, or when a specified time period has passed, or if a selection has been made from an NDA menu or the Edit menu, and in other special cases. The action routine performs whatever tasks the NDA was designed for. An *action code* passed in the accumulator tells the NDA why it was called.
- ☐ The Desk Manager calls the *close* routine to close the desk accessory window.

The processor is in full native mode on entry into all four routines. All four routines should end with an RTL instruction.

An NDA action routine follows this basic procedure:

1. It saves important global values, such as the application's current GrafPort.
2. Based upon the action code received, it takes appropriate action.
3. It restores the global values and returns to the Desk Manager with an RTL.

You must start the NDA with an identification section that specifies the pointers to the four routines, the NDA's *period* (how often it runs), its *event mask* (what events it wants), and its *menu line* (text defining its title on the Apple menu). For example, the identification section could look like this:

```
StartofNDA      dc i4'PtrToOpen'            ; Pointer to open routine
                dc i4'PtrToClose'           ; Pointer to close routine
                dc i4'PtrToAction'          ; Pointer to action routine
                dc i4'PtrToInit'            ; Pointer to init routine
                dc i2'Period'               ; How often to run
                dc i2'EventMask'            ; What events to retrieve
                dc c' MenuLine\H**'         ; Text for menu item
                dc i1'0'                    ; Terminator for the menu line
```

New desk accessories have the ProDOS file type $B8. On disk, they must be in the `DESK.ACCS/` subdirectory of the `SYSTEM/` directory.

Initialization files

Initialization files are files of types $B6 and $B7, in the `SYSTEM.SETUP` subdirectory. They are special-purpose programs that perform initialization at system startup, before any applications have been launched.

There are two types of initialization files—*temporary* and *permanent:*

- **Temporary initialization files** (type $B7) are loaded and executed just like applications ($B3), except that they must terminate with an RTL rather than a QUIT. They are removed from memory when finished.
- **Permanent initialization files** (type $B6) are loaded and executed just like applications ($B3), except for these conditions:
  - □ They must not be in special memory.
  - □ They cannot permanently allocate any direct-page/stack space.
  - □ They must terminate with an RTL rather than a QUIT.

  Permanent initialization files are *not* removed from memory when finished.

With initialization files, you can customize the operating environment before any applications are loaded. The `TOOL.SETUP` file is an example of an initialization file; it loads RAM patches to tool sets. `TOOL.SETUP` is a permanent initialization file because other system software needs it during program execution. If your initialization files need to execute only once, make them temporary.

❖ *Note:* Don't confuse these initialization *files* (programs executed at system startup) with initialization *segments* (pieces of an application executed when the application starts up). See "Loading Programs and Segments" in Chapter 6.

Interrupt handlers

An **interrupt** is a signal to a computer that stops the execution of an ongoing program while a higher-priority program is executed. It is usually an external, hardware-generated signal, but software interrupts are possible as well.

On the Apple IIGS, **interrupts** may be handled at either the firmware or the software level. The built-in interrupt handers are in firmware (discussed in the *Apple IIGS Firmware Reference*); user-installed interrupt handlers are software and may be called directly by the firmware or through ProDOS 16.

When the Interrupt Request (IRQ) line on the Apple IIGS microprocessor is activated, or when a software interrupt occurs, the microprocessor stops executing the current application and transfers control to the firmware interrupt-processing routines. The built-in interrupt handler processes the interrupt if the application has not provided its own interrupt handler.

The built-in interrupt handler

The Apple IIGS built-in interrupt handler is a firmware program that performs a sequence of steps to handle system interrupts. When a program is interrupted, the handler saves the current state of the system. The handler then processes the interrupt itself or passes execution to another handler, either internal or external. On completion of interrupt processing, the interrupted program regains control and can continue as though nothing had happened.

Figure 8-2 and the following explanation give a simplified picture of the steps taken by the built-in interrupt handler; they emphasize the course of execution when the interrupt is to be serviced by a user-installed handler.

1. When an interrupt signal occurs, execution jumps indirectly through the interrupt vector EIRQ if running in emulation mode when the interrupt occurred, or NIRQ if in native mode.
2. The system then tests to see whether the interrupt was the result of a software Break instruction. If it was, the system vectors to a break handler through a break handler vector in bank $E1. If no break handler is installed, execution passes through the user break vector at $3F0 in bank zero, which normally points to the Monitor program.

3. If the interrupt source was not a Break instruction, the interrupt handler saves the absolute minimum amount of information about the machine state—just that necessary to read an incoming serial character—and then tests for AppleTalk and serial port interrupts. This hasty action is necessary so that incoming characters in high-speed transmission will not be lost. If the interrupt is a serial interrupt, the firmware executes a JSL instruction to the serial port handler.

4. If the interrupt is not a serial interrupt, the interrupt handler saves the rest of the machine state and establishes a specific interrupt memory configuration, as described next under "Environment Handling for Interrupt Processing." It begins a polling loop, testing each of the possible interrupt sources in turn.

5. If no internal interrupt handler claims the interrupt, then (and only then) the firmware jumps through the user interrupt vector, to a user-installed routine that handles the interrupt. The address of the user interrupt routine is found in bank $00, addresses $3FE (low byte) and $3FF (high byte).

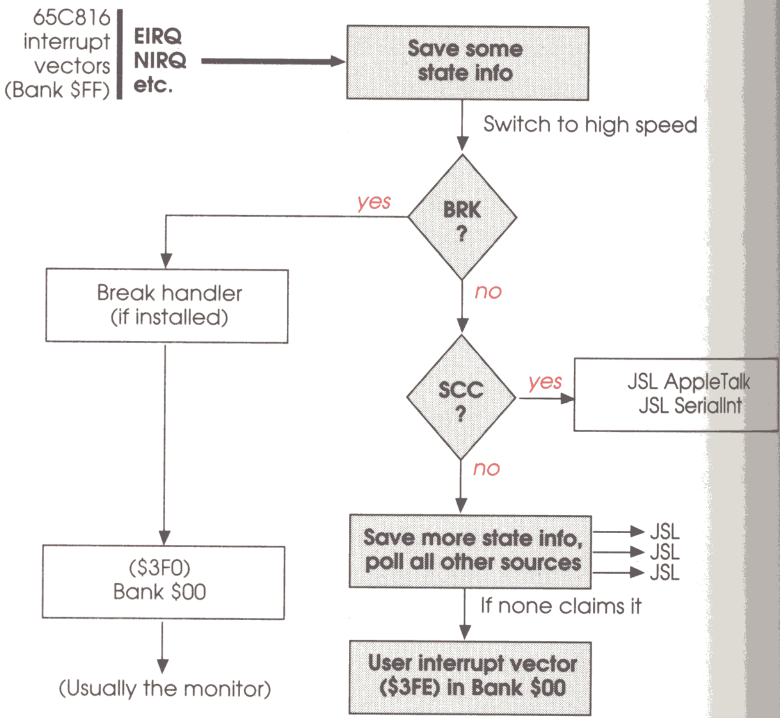


Figure 8-2
Built-in interrupt handler (simplified)

Environment handling for interrupt processing

For each type of interrupt, the processor can be in either emulation or native mode. The built-in interrupt handler must save the current environment in each case, set the *interrupt environment,* process the interrupt through the appropriate interrupt handler, and then restore the original environment before returning control to the interrupted program.

The **interrupt environment** is the machine state that your interrupt handler finds when it gains control. If your handler is called from the user interrupt vector, the environment includes these conditions:

- □ emulation mode
- □ slow speed (1MHz)
- □ text page 1 switched in (main screen holes available)
- □ main memory switched in (for reading and writing)
- □ $D000-$FFFF ROM mapped into bank $00
- □ main stack and zero page switched in
- □ main stack pointer active (auxiliary stack pointer saved)

If your handler is called through a JSL from the built-in handler *before* jumping to the user interrupt vector, the same state applies except that the machine is in 8-bit native mode and running at fast speed.

❖ *ProDOS 16:* If your interrupt handler is installed through PRoDOS 16, the machine state it finds is somewhat different. See "Interrupt Handling Under ProDOS 16," later in this section.

After the interrupt has been processed, the system interrupt handler restores the environment and registers to their preinterrupt state and executes an RTI (return from interrupt), returning to the executing program.

Writing and installing your own interrupt handler

Descriptions and locations of all interrupt vectors are listed in Appendix D of *Apple IIGS Firmware Reference.*

If you write your own interrupt processing routine, you can attach it to the system by modifying the interrupt vector locations, such as the user interrupt vector at $00 03FE. However, you must be careful to obey all of the conventions specified in Chapter 8 of the *Apple IIGS Firmware Reference* regarding interrupt processing, and make sure to restore the *interrupt environment* state that you found on entry to your handler. This allows the system in turn to restore the environment to its original state.

If you write a handler to be called from the $3FE interrupt vector, it must do the following tasks:

Interrupt soft switches are documented under "Soft Switches" in the *Apple IIGS Firmware Reference.*

1. Verify that the interrupt came from the expected source.
2. Handle the interrupt appropriately.
3. Clear the appropriate interrupt soft switch.
4. Restore everything to the state it was in when the interrupt routine was entered, if your routine has made any changes to the state of the machine.
5. Return to the built-in interrupt handler by executing an RTI instruction.

Here are some other factors to remember when you are dealing with programs that run in an interrupt environment:

- There is no guaranteed maximum response time for interrupts because the system may be performing a disk operation that lasts for several seconds when the interrupt occurs.
- Emulation-mode interrupts are supported in bank $00 only, whereas native-mode interrupts are supported everywhere in memory. Therefore, code running anywhere except in bank $00 must be native-mode code.
- Interrupt overhead will be greater if your interrupt handler is installed through an operating system's interrupt dispatcher, such as the ProDOS 16 interrupt handler described next. On the other hand, if your handler is installed through ProDOS 16 it needn't run in emulation mode.

Interrupt handling under ProDOS 16

You can write an interrupt handler and install it under ProDOS 16, if you wish. ProDOS 16 installs its own vector at location $00 03FE (page 3 in bank zero), so when an interrupt occurs, execution passes through that location. At that point the microprocessor is running in emulation mode, using the standard clock speed and 8-bit registers. The vector at $00 03FE points to another bank zero location, that in turn passes control to the ProDOS 16 interrupt dispatcher. The interrupt dispatcher switches the processor to full native mode (including higher clock speed) and then polls the user-installed interrupt handlers. When the interrupt has been serviced, ProDOS 16 returns to emulation mode and passes control back to the built-in interrupt handler.

Figure 8-3 is a simplified picture of what happens when a device generates an interrupt that is handled through a ProDOS 16 interrupt handler.

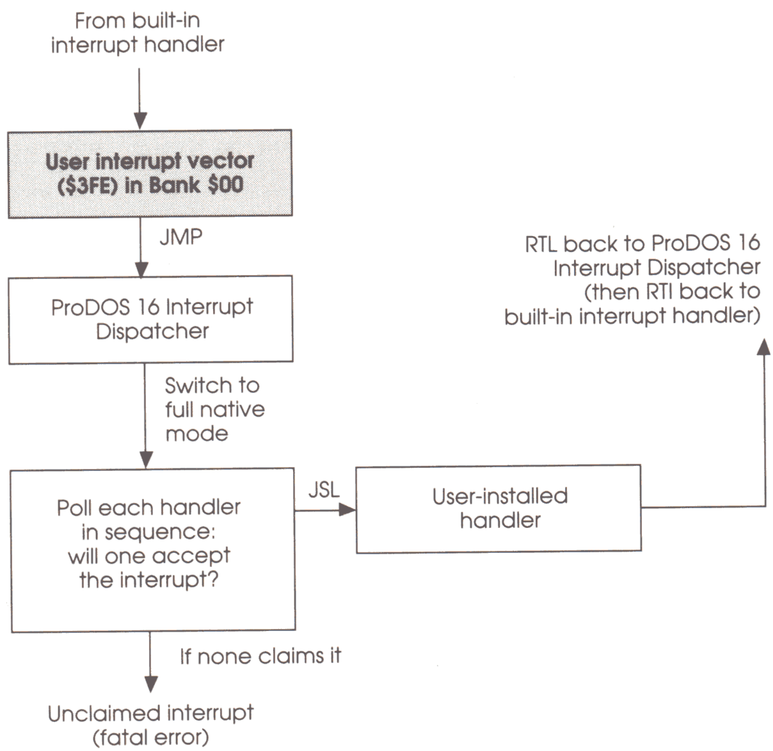


Figure 8-3
Interrupt handling through ProDOS 16

ProDOS 16 supports up to 16 user-installed interrupt handlers. When an interrupt occurs that is not handled by firmware, ProDOS 16 transfers control to each handler successively until one of them claims it. There is no grouping of interrupts into classes; their priority rankings are reflected only by the order in which they are polled.

If you write an interrupt handler to run under ProDOS 16, note these conventions:

- Your handler will gain control with the machine in full native mode (e, m, and x = 0), with a fast clock speed.
- Interrupts will be disabled. Do not re-enable interrupts from within your interrupt handler.
- The handler must exit with an RTL instruction. The machine should again be in full native mode, at fast speed. The carry flag must be cleared (= 0) if the interrupt was serviced, and set (= 1) if it was not—that is how ProDOS 16 knows whether or not your handler has claimed the interrupt.

To make your interrupt handler active, install it with the ProDOS 16 ALLOC_INTERRUPT call. To remove it, use the DEALLOC_INTERRUPT call. Be sure to enable the hardware generating the interrupt only *after* the routine to handle it is allocated; likewise, disable the hardware *before* the routine is deallocated.

User tool sets

The Apple IIGS Toolbox is quite extensive and provides a great deal of programming convenience; there are over 800 separate routines that you can call from your applications. Furthermore, because of the flexibility of the Tool Locator system, your application is not restricted to even this large number of tool calls. In addition to the *system tools* (provided by Apple), you can write and install your own tool sets, called *user tools*.

Writing and installing user tool sets is fully documented under "Writing Your Own Tool Set" in the *Apple IIGS Toolbox Reference*. We won't repeat that information here, beyond listing these few main points:

- The open-ended, flexible nature of the Tool Locator is what makes it possible to add your own tool sets. The Tool Locator requires no fixed ROM location and few fixed RAM locations, so it may easily modify its data structures to incorporate new tool sets.
- Each tool set is assigned a permanent *tool number.* System tools are assigned numbers by Apple; you can assign your own numbers to user tool sets that you create. Assignment starts at 1 and continues as successive integers (2, 3, 4, and so forth). Table 8-2 lists the presently defined system tool numbers.

Table 8-2
Tool set numbers

Hexadecimal	Decimal	Name
$01	1	Tool Locator
$02	2	Memory Manager
$03	3	Miscellaneous Tool Set
$04	4	QuickDraw II
$05	5	Desk Manager
$06	6	Event Manager
$07	7	Scheduler
$08	8	Sound Tool Set
$09	9	Apple Desktop Bus Tool Set
$0A	10	SANE
$0B	11	Integer Math Tool Set
$0C	12	Text Tool Set
$0E	14	Window Manager
$0F	15	Menu Manager
$10	16	Control Manager
$11	17	System Loader
$12	18	QuickDraw II Auxiliary
$13	19	Print Manager
$14	20	LineEdit Tool Set
$15	21	Dialog Manager
$16	22	Scrap Manager
$17	23	Standard File Operations
$19	25	Note Synthesizer
$1A	26	Note Sequencer
$1B	27	Font Manager
$1C	28	List Manager

- Each routine within a tool set is assigned a permanent *function number*. Function numbers start at 1 in each tool set and continue as successive integers. Certain standard calls must be present in every tool set, and so certain function numbers are reserved. Table 8-3 lists them. See the toolbox manual for explanations of what each function must do.
- There are some general rules and design considerations that tool sets must follow. For example, tool sets receive control in full native mode; they must obtain any needed work space from the Memory Manager; they must provide some sort of interrupt environment; and they must restore the caller's operating environment before returning control to the caller. See the *Apple IIGS Toolbox Reference* for details on these and other design requirements.

Table 8-3
Standard tool set routine numbers

FuncNum	Description
1	Boot initialization
2	Application startup
3	Application shutdown
4	Version information
5	Reset
6	Status
7	Reserved for future use
8	Reserved for future use

Chapter 9

Where to Go From Here

This is as far as we can take you in this book. For your next step, spread out your development-environment manuals, Apple IIGS technical manuals, and the *Apple IIGS ToolBox Reference*, and play with HodgePodge on the Apple IIGS or start your own applicaton. In parting, we'll give you a few hints—mostly summaries of the ideas presented throughout the book—and we'll mention two organizations that can help you become a successful Apple IIGS developer.

Modify HodgePodge

The easiest way to get started on your own desktop application may be to take HodgePodge and modify it incrementally. Recompile it and run it after each small change to see how your changes look (or even if they work).

You might begin by modifying text within dialog boxes, or changing the names of menu titles or items. As you become a little braver, try adding (or removing) menus or menu items, and adding (or removing) the subroutines called from those menu items. Remember that adding an item to a menu will require changing the routine `DoMenu` as well as changing the menu definitions themselves—not to mention writing a subroutine that *does* something when the menu item is selected.

Soon you'll become more ambitious, and you can branch in almost any direction. Add a routine that plays a song when called. Define a new window type, perhaps even one that permits the user to draw or type into it. Define a file type for that window type, and allow the user to save results to disk. Give HodgePodge the ability to cut and paste to and from the Clipboard, and display the Clipboard window. Play with menu- and window-frame colors.

Your imagination is your only real constraint. Have fun and challenge your limits; the Apple IIGS is a willing partner in this adventure.

Design your program carefully

We've discussed design considerations for Apple IIGS desktop applications throughout the book, but some in particular are worth repeating. As you work on your own programs, either by modifying HodgePodge or starting from scratch, keep these points in mind:

- **Follow the Human Interface Guidelines:** Follow the underlying concepts, as well as the surface implementation, of the guidelines. They describe a tested and proven interface, familiar and friendly to millions of users. If you go beyond the guidelines, make it a natural extension.
- **Design data structures before writing code:** Your menus, windows, controls, dialog boxes, and alerts influence program structure so strongly that they should be carefully planned and defined at the beginning. You'll save yourself wasteful rewriting and awkward patching if your code organization flows naturally from the design of your data structures.
- **Test for errors:** Make it a habit to put error-testing code after toolbox calls. It can help inform the user and can keep your program from doing harmful things to the user's system or data.
- **Save and restore:** When a subroutine accesses the desktop, it may not always know the exact state of things. Note that many HodgePodge routines start with a GetPort call to save the current state of the desktop, and end by restoring the desktop (with a SetPort call). It's another good habit to get into.
- **Lock handles while in use:** If your program has allocated a piece of memory accessed by a handle, be sure to lock it just before using it. A lot of memory errors are caused by trying to access data that has been moved.
- **Unlock handles when not in use:** Don't prevent the Memory Manager from doing its job.
- **Dispose handles when finished:** Don't prevent the Memory Manager from doing its job.
- **Make it easy to translate:** If you want to appeal to international markets, remember to place in one or more individual data areas all text that is to be displayed, so that it may be found and modified easily.
- **Design for "Undo":** Consider including a facility that allows the user to reverse his actions to undo a mistake. Your customers will be eternally grateful.

Join APDA

If you are already a member of the Apple Programmer's and Developer's Association (APDA), you know that it is the fastest way to get the most recent software, documentation, and other information of interest to developers. If you are not a member, it's easy to join.

APDA is a membership organization for both professional and advanced amateur programmers and developers. It was founded by Apple Computer and the A.P.P.L.E. Co-op near Seattle, Washington; its purpose is to publicize and distribute programming tools and technical documentation for Apple computers.

APDA serves as a "one-stop shopping center." It offers both finished products from Apple and other vendors, and prerelease versions of many Apple development tools and documents. Some small-volume products, not suitable for the retail market, are available only through APDA. Other products, scheduled for the retail market, are offered through APDA in prerelease versions, on an "as-is" (no support) basis.

If you join APDA, you will receive quarterly catalogs (and more frequent updates) of the available material for both Apple II and Macintosh development. Membership is open to all interested parties. Yearly dues are $20.00.

Write to

Apple Programmer's and Developer's Association
290 SW 43rd Street
Renton, WA 98055

(206) 251-6548

Become an Apple Developer

If you are a developer with a product soon to reach the commercial market, you may want to become an Apple Certified Developer. As a Certified Developer, you will receive monthly mailings including a newsletter, Apple II and Macintosh Technical Notes, pertinent Developer Program information, and all the latest news relating to Apple products. You will have access to our Developer Hotline for general developer information.

Once you are certified, Apple's Developer Technical Support staff can provide technical assistance during your product's evolution. Our Technical Support engineers will answer your development questions within 24 hours by electonic mail.

The Certified Developer program is for professional hardware and software developers who plan to have a finished *commercial* product within 18 months. If you fit this description and are interested, please write for an application. You will need to submit information on previous products and your present business plan along with your completed application.

Write to

Developer Programs
Apple Computer, Inc.
20525 Mariani Avenue, M.S. 27W
Cupertino, California 95014

(408) 973-4897

Licensing Apple software

If the software you write uses all or part of some Apple software (such as ProDOS 16 or the Apple IIGS Toolbox), you will need to license the use of that software from Apple Computer. You needn't license any parts of HodgePodge you use, but you will need to license any system software that accompanies or is incorporated into your application.

A modest yearly fee authorizes you to use Apple software in your product. There are no royalties. Please contact

Software Licensing
Apple Computer, Inc.
20525 Mariani Avenue, M.S. 28B
Cupertino, CA 95014
Attn: Software Licensing Program

(408) 973-4667

Appendixes

Appendix A

Converting Macintosh Programs to the Apple IIGS

If you have written a desktop application for the Macintosh, you may be able to convert it to run on the Apple IIGS without completely rewriting it. On a conceptual level, the task should be rather simple—after all, program organization and toolbox capabilities are similar for both computers. But when it comes to implementation, there are many differences that require careful attention to details of coding. This appendix notes some of the details to keep in mind when converting Macintosh programs to the Apple IIGS.

High-level languages

Programming in a high-level language can insulate you from many of the differences among machines. However, the individual toolbox calls are different enough between the Macintosh and the Apple IIGS that in most cases it will not be possible just to recompile Macintosh code and expect it to run on the Apple IIGS.

The best approach is probably to regard the conversion process algorithmically, rather than literally. In other words, don't expect that you will be able to drop a whole program or even any one routine, unchanged, into an Apple IIGS program. Use your Macintosh program's organization as a framework into which to place individually converted routines. Even though most of the organization and much of the original code can be translated exactly, you'll have to locate those statements, calls, and structures that are incompatible with the Apple IIGS environment.

This doesn't *necessarily* mean pouring over the source code line-by-line. In general, you might be able to port well-behaved high-level code, just by carefully locating and modifying tool calls and any code that accesses toolbox data structures.

Of course, if you have a routine that makes no tool calls, accesses no tool structures, and otherwise makes no Macintosh-specific assumptions, you may indeed be able to convert it simply by recompiling it.

Assembly language

Approaching the conversion process algorithmically rather than literally is even more critical when converting programs written in assembly language. Besides toolbox differences, you are faced with fundamentally different microprocessor architectures and instruction sets, very different memory maps, and a host of other low-level differences between the two types of computer. The only possible aproach is to think of your Macintosh program as an organizational shell in which every routine will need extensive revision to convert correctly.

Here are just a few of the differences to keep in mind.

- **Registers:** The 65816 does not have nearly the number of registers that the 68000 has, so you will have to store more of your variables in memory—usually local memory (direct page).
- **Direct page:** Direct page is an Apple IIGS concept that can be very useful, especially if you are constructing tables in memory and accessing them by offsets. If your Macintosh program allocates such data structures on the heap, you can gain efficiency by putting them onto the Apple IIGS direct page.

- **Stack:** Your stack on the Apple IIGS is likely to be smaller than what you are used to on the Macintosh. More of your variables and data structures will be allocated in other parts of memory.
- **Memory space and segmentation:** Your program may have to run in less space on an Apple IIGS than it may be used to on a Macintosh. Therefore, segmentation can be very important. Break the program into segments, and use as many dynamic segments as possible.
- **Video display:** The Apple IIGS offers you two different Super Hi-Res graphics modes—320 and 640 pixels across. Both use color, but neither has square pixels.

Toolbox differences

If you compare the Macintosh and Apple IIGS toolboxes, you'll see that many routines have identical names and function in the same way. Many others do not, however, so watch out for differences when using the tools. In particular, the required parameters and the order of the parameters may differ between the Macintosh and Apple IIGS versions of a particular call. Be sure to look up each routine in the *Apple IIGS Toolbox Reference* before using it.

Some groups of tool calls are more alike than others. For example, many QuickDraw calls are identical or very similar in both environments. Thus, graphic routines might be relatively simple to translate. On the other hand, calls that directly access or manipulate memory, such as Memory Manager calls and handle manipulations, can operate very differently in the two environments—even when they look the same. Be careful.

Also keep in mind that the records that describe toolbox structures such as GrafPorts and controls are different. Fields that exist in one environment may not be in the other. So be particularly careful if you access data structures directly.

Some specific recommendations on how to handle toolbox differences follow.

Resources

To a Macintosh programmer, the term *resource* means something much more specific than a useful item. Resources are certain types of data structures, easily accessible by the programmer, that help to separate code from static data and make program modification simpler.

The Apple IIGS has no predefined structures like resources, and no Resource Manager or resource editors for manipulating them. So, in conversion, you will have to move your resources from the resource fork of your file into your program code, either as separate data segments or files, or merged into the execution stream. The Pascal version of the sample program HodgePodge shows several ways to do this:

Pascal HodgePodge is listed in Appendix G. Furthermore, individual routines are listed and described throughout the book. See Table 2-1.

- **Icons:** You can define your icons by directly creating a pattern in memory, as HodgePodge does with the Apple icon in `InitGlobals` (file `HP.PAS`).
- **Text strings:** Instead of a string or string list resource, you can define your strings in initialization routines (as HodgePodge does with its menu strings), or in the individual routines in which they are needed (as HodgePodge does with prompt strings in the Standard File dialog boxes).

  Remember, keeping all your strings easily accessible will make the program more convenient to translate or otherwise modify.
- **Window and dialog box templates:** The templates (DLOG, WIND, ALRT, and DITL resources on the Macintosh) used to define your windows and dialog boxes, and the controls and items within them, must be defined within the body of your Apple IIGS code.

  Each time it opens a window, HodgePodge defines and initializes a parameter list that controls the window's appearance (part of the routine `DoTheOpen` in `WINDOW.PAS`). When it creates an alert box, it calls a routine (`MakeATemplate` in `DIALOG.PAS`) that defines the characteristics of an alert box and two items within it.

Other resources in your Macintosh program will need to be converted similarly.

TaskMaster or GetNextEvent?

The Apple IIGS offers at least one very useful event-handling capability not yet available on the Macintosh: *TaskMaster.* TaskMaster automatically handles many standard events for standard types of windows—resizing, dragging, scrolling, updating and activating, and so on.

On the other hand, the Apple IIGS also supports "normal" event-handling with GetNextEvent, just as on the Macintosh. So it might seem more efficient to keep that same GetNextEvent organization when converting an existing Macintosh program.

Usually it is not. Unless your program constructs unconventional windows or handles them in an unusual manner, it is probably best to change from GetNextEvent to TaskMaster when making the conversion. Using TaskMaster may allow you to eliminate entire routines from your progam, routines that would otherwise need individual attention to convert correctly.

HodgePodge, for example, has no update routine, no activate routine, no scrolling procedure, no window-dragging or -resizing routines, and yet it supports windows that do all those things. It may greatly simplify your conversion to switch to TaskMaster.

QuickDraw II

QuickDraw II on the Apple IIGS is quite similar to QuickDraw on the Macintosh, apart from extensions to support Apple IIGS color display. However, keep the following in mind:

- The conceptual drawing space for QuickDraw II has boundary coordinates -16K, -16K, 16K, 16K, compared to -32K,-32K and 32K,32K on the Macintosh.
- QuickDraw II's pixel images are similar to Macintosh QuickDraw's *bit images,* but pixels are described by more than one bit each. Bit images such as icons will have to be converted to pixel images, with either two or four bits per pixel.

  Icons are not as restricted on the Apple IIGS as they are on the Macintosh. Besides having color, they may be of arbitrary height and width, rather than 32 pixels (bits) on a side.

- ☐ You won't need to change most drawing commands—your black-and-white Macintosh drawings will convert directly to white-and-black drawings on the Apple IIGS screen.

  There will be some change in aspect ratio of images and drawn objects in transferring to the Apple IIGS screen, and significant changes in overall size—Super Hi-Res pixels are not square and are significantly larger than Macintosh screen pixels.

- ☐ Text drawing and text measurement on the Apple IIGS are similar to their treatment on the Macintosh. The Apple IIGS font definition is similar to that of the Macintosh, and a simple conversion algorithm allows the IIGS to use any font developed for the Macintosh. Most Macintosh QuickDraw text calls are duplicated precisely in QuickDraw II.

  Some calls have been added to handle the `CString` data type (a sequence of characters terminated by a 0 byte).

  QuickDraw II does not scale text—the Font Manager does. In general, the interaction between the Apple IIGS Font Manager and QuickDraw II is different from the close relationship between the Font Manager and QuickDraw on the Macintosh. Font selection on the Apple IIGS requires a little more care than on the Macintosh.

File system differences

ProDOS 16 is the Apple IIGS operating system for desktop applications. There are ProDOS 16 calls equivalent to most Macintosh File Manager calls, but some parameters are different or are used differently. If your Macintosh application makes File Manager calls, they will have to be translated to ProDOS 16 calls.

On the other hand, if your program is written in a high-level language and uses only that language's file access facilities, you might not have to do any translating at all. On recompiling under a IIGS development environment, your file calls will be translated.

As noted under "Resources" earlier in this appendix, files do not have separate resource and data forks. Data stored as resources in your Macintosh files will have to be redefined and stored as standard ProDOS 16 files.

If your program handles all its file access through Standard File Operations, it will not have to manipulate pathnames explicitly. Just as on the Macintosh, the Standard File Operations Tool Set on the Apple IIGS takes care of all that. But if you do access files by name, please note these differences from the Macintosh file system:

- Filenames under ProDOS 16 are more restricted than on the Macintosh. Only the characters A–Z, 1–9, and the period (.) are permitted, and the maximum length is 15 characters.
- ProDOS 16 permits you to define up to 9 *prefixes,* for convenient simultaneous access to files in several different subdirectories.
- ProDOS 16 uses a *hierarchical file system,* in which files are grouped into subdirectories and accessed by *pathname.* The present Macintosh file system is also hierarchical, but if you have an early Macintosh program written for the *flat file system,* you may have to modify it to account for pathnames instead of just filenames.

Other toolbox differences

As you get involved in the conversion process, you will of course discover many other differences, some subtle and some obvious, between the Macintosh and Apple IIGS toolboxes. There are far too many to list in this appendix, but here is a sample:

- **Memory Manager:** The Apple IIGS Memory Manager is conceptually very similar to the Macintosh Memory Manager. However, because of the 65C816 microprocessor and the architecture of the Apple IIGS, the Apple IIGS Memory Manager's calls are very different, and its internal data structures totally different, from those of the Macintosh. Pay extra-close attention to converting Memory Manager calls and manipulating its data structures such as pointers and handles.
- **Window Manager/Control Manager:** Windows and controls can be handled differently in several ways, largely because of the Window Manager routine TaskMaster. The Apple IIGS has window types that include scroll bars (*frame scroll bars*) manipulated automatically by TaskMaster. The use of frame scroll bars greatly simplifies window handling.

- **Frame scroll bars:** If you use TaskMaster and have it manipulate frame scroll bars, remember that the scroll bars are part of the window frame, not the content region. That is, unlike standard scroll bars on a Macintosh window, they are *outside* the window's port rectangle. That may affect your clipping and drawing commands.
- **Desk Accessories:** If you are converting a Macintosh desk accessory, it will become a *new desk accessory* on the Apple IIGS.
- **Standard File Operations:** The *Disk* button on the Apple IIGS works differently from the *Drive* button on the Macintosh. When a user clicks the Disk button, Standard File first looks at the disk in the same drive the current disk is in. If the current disk is no longer in that drive, the disk in that drive becomes the current disk. If the current disk is still there, the Disk button moves to the next disk in the ProDOS chain. The Disk button works this way because a user can change disks without the system's knowledge.
- **Printing:** On the Apple IIGS, the Choose Printer function is part of the Print Manager, rather than part of the Chooser desk accessory as on the Macintosh. To support printing, you will need to add a Choose Printer menu item to the File menu, and create a short routine to handle it.

Appendix B

Enhancing Standard Apple II Programs

If you have written a ProDOS 8-based program for a standard Apple II computer (64K Apple II Plus, Apple IIe, or Apple IIc), you should be able to run it without modification on the Apple IIGS. The only noticeable difference will be its faster execution because of the greater clock speed of the Apple IIGS—and even that difference can be eliminated if you wish. However, the program will not be able to take advantage of any advanced Apple IIGS features such as its large memory, the toolbox, the mouse-based interface, and the new graphics and sound abilities.

This appendix discusses some of the basic alterations you can make to upgrade a ProDOS 8 application for various execution modes on the Apple IIGS. Depending on the program's size and structure and the new features you wish to install, those changes may range from minor to drastic.

❖ *High-level languages:* This discussion is primarily about assembly-language programs. If you have a standard Apple II program written in a compiled BASIC or other high-level language, converting it to run in native mode on the Apple IIGS may require nothing more than recompiling it on an equivalent Apple IIGS development system. Accessing the toolbox may then be as easy as adding the calls to your original source code.

Conceptual differences

For the purpose of program conversion, there are perhaps three main areas of difference between traditional Apple II computers and the Apple IIGS:

- **Hardware execution modes:** The 65C816 microprocessor executes in both native mode and 6502 emulation mode. In fact, there are at least three modes to consider:
  - Emulation mode (e flag, m flag, and x flag set). The processor functions like a 6502.
  - Native mode with the m flag and x flag set. The processor has all 65C816 features, but the accumulator and index registers remain 8 bits wide.
  - Full native mode (e flag, m flag, and x flag cleared). All 65C816 features are available, and the accumulator and index registers are 16 bits wide.

  65816 assembly language is described in the *Apple IIGS Programmer's Workshop Assembler Reference.*

  The 65816 microprocessor adds several new addressing modes and instructions to those of the 6502. All 6502 and 65C02 instructions are still available, but the new larger registers and relocatable stack and direct page add flexibility and power to the system.

- **Tool sets:** The toolbox is the essence of what makes the Apple IIGS more powerful and convenient than other Apple II computers. To write the kinds of programs described in this book, you need access to the toolbox. Tool calls can be made while in full native mode only.

  Relocatable code and the Memory Manager are discussed in Chapter 6.

  The Apple IIGS also provides a sophisticated loader and a software memory manager. To take full advantage of the system, you should write *relocatable* code, and request any memory you need through Memory Manager calls. Otherwise your program will be incompatible with other programs in memory, such as desk accessories and memory-resident utilities.

- **Operating systems:** The Apple IIGS comes equipped with two operating systems: ProDOS 8 and ProDOS 16. Unaltered standard-Apple II applications can run on the Apple IIGS only under ProDOS 8. They cannot access tool sets or ProDOS 16. They can make ProDOS 8 calls only while in emulation mode. The ProDOS 8 global page is supported, but again only in emulation mode.

ProDOS 8 is discussed in the *ProDOS 8 Technical Reference Manual.*

ProDOS 16 calls can be made from either emulation or native mode, but ProDOS 16 is not available to programs launched under ProDOS 8. ProDOS 16 is loaded into memory only when a native-mode, ProDOS 16-based application is launched. The ProDOS 8 global page is not available under ProDOS 16.

What does all this mean? It means that at least parts of your program must be modified for native-mode operation if you want to use Apple IIGS features. There are several approaches you can take:

- ☐ You can convert your program to a *hybrid application:* it runs in emulation mode, under ProDOS 8, but switches to native mode to make tool calls.
- ☐ You can insert parts of your original code, unchanged or largely unchanged, into a new program that runs in native mode under ProDOS 16.
- ☐ You can convert your entire program to run in native mode under ProDOS 16.
- ☐ You can start from scratch, writing a brand new Apple IIGS application that replaces your original program.

The rest of this appendix briefly discusses each of the above possibilities.

Write a hybrid application

It is possible to run your standard Apple II program under ProDOS 8 in emulation mode on the Apple IIGS, but modify it so that, at specific points, it switches to native-mode operation. A program that does this is called a *hybrid application.*

Writing a hybrid application is usually undertaken to access the greater memory capacity and higher execution speed of the Apple IIGS, but it is also the simplest way to access the toolbox. Using this technique, you can write an application that runs on both a standard Apple II and an Apple IIGS—it can use toolbox features when it determines that it is running on an Apple IIGS.

Writing a hybrid application is not easy, and the results for toolbox access are not always entirely satisfactory. You'll need to address at least these issues:

- **Loading RAM patches:** If your program is self-booting (starts up directly under ProDOS 8) on the Apple IIGS, ProDOS 16 and the System Loader will not have been activated. Therefore RAM tool sets and RAM patches to the ROM tool sets will not be in place. There are several possible responses to this problem:
  - Do without the patches or RAM-based tools.
  - Write your own RAM-based tool set, convert it to ProDOS 8 binary format, and load and install it yourself. See "Writing Your Own Tool Set," in the *Apple IIGS Toolbox Reference.*
  - Allow your program to be launched only from a ProDOS 16-based finder or launcher, after the normal ProDOS 16 boot sequence has loaded all the RAM patches and RAM tool sets.

See "Setting Up Direct-Page/Stack Space", in Chapter 6.

- **Switching stacks and zero pages:** You have a standard stack and zero-page available in emulation mode, but you also need a direct-page/stack space for use by tool sets in native mode. Set it up as needed. When switching from emulation mode to native mode and back, you must save the current value of the stack pointer, and set the stack pointer to the proper value for the mode you are about to enter. Likewise, the direct register is set to zero upon entering emulation mode; you must save its value before switching to emulation and restore it upon returning.

  For detailed instructions on saving and restoring the proper environment while switching execution modes, see "Notes for Programmers" in the *Apple IIGS Firmware Reference.*

- **Staying in bank $00 or disabling interrupts:** Any code that your program calls while in emulation mode must be in bank $00, or else interrupts must be disabled. The Program Bank register is not saved or restored when an interrupt occurs in emulation mode.

Insert parts of your 6502 code

Because the 65C816 processor recognizes the 6502 instruction set, it may be possible to use significant sections of your code, unchanged or only slightly modified, in a native-mode, ProDOS 16-based application. That is, instead of making a hybrid application, you might write a new Apple IIGS application, but save time by incorporating as much of your older, 6502-based code as possible. In most cases this option is far better than writing a hybrid application; it puts ProDOS 16 and the tool sets much more directly at your program's disposal.

How successful you can be depends greatly on the specific content of your existing code. Routines that draw to the screen or otherwise duplicate the tasks performed by tool sets may not be worth converting to native-mode execution. Code that uses absolute address references or that must itself occupy specific addresses will be incompatible with native-mode memory management. Instructions that can't reach everywhere in the 16-megabyte Apple IIGS memory space (such as JSR rather than JSL) can cause a lot of problems, depending on where your code and data are and what system features you need to access.

In spite of these and other problems, it may be possible to use large portions of certain types of 6502-based code, relatively unchanged, in native-mode Apple IIGS applications. Here are just a few considerations.

- **Register width:** In most cases your 6502 code will require short (8-bit) accumulator and index registers when running in native mode. That is, the m- and x-bits need to be set (=1) when the e-bit is cleared (=0). However, see the next note.
- **Stack manipulation:** The stack pointer value is commonly saved and restored with the instruction pair TSX. . .TXS. If performed in 8-bit mode, this sequence destroys the high-order byte of the stack pointer. To be safe, *do all stack manipulation with 16-bit registers.*
- **Firmware entry points:** Replace all calls to specific firmware entry points with FWEntry tool calls. FWEntry allows you, while in native mode, to make calls to (6502) code that executes in emulation mode; it saves and restores the Data Bank and Direct registers.

The FWEntry call is part of the Miscellaneous Tool Set. See the *Apple IIGS Toolbox Reference.*

- **Data and buffer allocation:** Remove absolute addresses that define your data buffers or other entry points. For example, if your program reserves a 4K buffer space with an equate such as `BUFFER EQU $8000`, replace that with something such as `BUFFER DS $1000`, which reserves a $1000-byte buffer but doesn't require it to start at address $8000.
- **Input/output:** I/O in a standard Apple II computer takes place by accessing locations in the $Cxxx address space (*I/O memory*). In the Apple IIGS, I/O memory exists only in banks $00, $01, $E0, and $E1. Therefore, if your code is running anywhere in expansion RAM, it cannot perform I/O unless data accesses to $Cxxx are made in *long* addressing mode, to access the proper bank.

  However, the timing of much I/O is critical and, because a long-addressing load instruction takes an extra cycle to execute, you may not be able to change the addressing mode.

  One way around this is to set the data bank register to $00 before executing the I/O instructions. Then, however, any other data in the same bank as your code becomes inaccessible—but that may not be a problem in your particular case.

  There are many other alternatives, including creative use of the direct page and isolating timing-critical code, that can be useful in various individual situations. Every situation is unique—feel free to be creative.

Rewrite it to run under ProDOS 16

Modifying your entire program for full 16-bit native mode operation on the Apple IIGS is a more ambitious task, but it may well be worth it for the greater number of features you can access. In order to run entirely in native mode, under ProDOS 16 and with the tool sets always available, your program needs to consider at least the following points.

- **Managing memory:** Because the Apple IIGS supports segmented load files, one of the first decisions to make is whether and how to segment the program (both the original program and any added parts). First, make your code relocatable so the Memory Manager can control where it is loaded. You'll need to specify memory-block attributes in addition to modifying your code as described in the previous section, "Insert Parts of Your 6502 Code."

Memory management under native-mode operation on the Apple IIGS is completely different from standard-Apple II methods. If your program allocates its own memory space and marks it off in the ProDOS 8 global page bit map, the enhanced version must be altered so that it requests all needed space from the Memory Manager.

- **Converting operating-system calls:** For most ProDOS 8 calls, there is an equivalent ProDOS 16 call with the same name. Still, each call block must be modified for ProDOS 16, and each parameter block must be reconstructed in the ProDOS 16 format.

  For other ProDOS 8 calls, a ProDOS 16 near-equivalent performs a slightly different task, and the original code will have to be changed to account for that.

  Refer to the detailed descriptions in Chapters 9 through 13 of the *Apple IIGS ProDOS 16 Reference* to see which ProDOS 16 calls are different from their ProDOS 8 counterparts.

  Yet other ProDOS 8 calls have no equivalent in ProDOS 16. If your program uses any of these calls, they will have to be replaced as appropriate.

- **Removing global page references:** Any access your original program makes to the ProDOS 8 global page must be replaced by appropriate ProDOS 16 or toolbox calls.

- **Converting stack and zero page:** Under ProDOS 16 in native mode, you are not constrained to the fixed stack and zero-page locations provided by ProDOS 8 in emulation mode. You may either let ProDOS 16 assign you a default 1K direct-page/stack space, or you may define a direct-page/stack segment in your object code. In either case, the loader may place the segment anywhere in bank $00—you cannot expect any specific address to be within the space.

  See "Setting Up Direct-Page/Stack Space," in Chapter 6.

- **Assembling:** Once your source code has been modified and augmented as desired, you need to reassemble it. You must use an assembler (or compiler, for high-level languages) that produces object files in Apple IIGS object module format (OMF); otherwise the program cannot be properly linked and loaded for execution. Using a different assembler may mean that, in addition to modifying your program code, you'll have to change some directives to follow the syntax of the new assembler.

  Object module format is documented in the *Apple IIGS Programmer's Workshop Reference*.

  If you have been using the EDASM assembler, you will not be able to use it to write Apple IIGS programs. Instead, you can use the Apple IIGS Programmer's Workshop (APW). APW is a set of development programs that allow you to produce and edit source files, assemble/compile object files, and link them into proper OMF load files.

  APW is discussed in Chapter 7.

After your revised program is linked, assign it the proper Apple IIGS application file type (normally $B3) with the APW FileType command.

Start from scratch

In the long run, this is the best alternative in most cases. Combing through your code line-by-line to make all the conversions described in the previous sections—even if it works—will probably yield a product that's only half successful. Why not start fresh, maintaining your original design and concepts but writing new code that truly takes advantage of the power and convenience of the Apple IIGS?

The purpose of this book has been to show you that it is both easy and rewarding to write desktop applications for the Apple IIGS. It has also shown you that such applications have a structure, an approach to the hardware, and a user interface that are fundamentally different from those of traditional Apple II software. Don't confine yourself unnecessarily; a clean slate is the best way to start. Take advantage of the freedom the Apple IIGS gives you!

Appendix C

Files on an Apple IIGS System Disk

A **system disk** is a 3.5-inch disk, 5.25-inch disk, or hard disk that has the files necessary for an Apple IIGS to start up when turned on or rebooted. It also has any files needed to support the specific application programs on the disk. This appendix shows you what files a system disk must have.

Because not all applications have the same needs, not all system disks are alike. In particular, there are *complete system disks* and *application system disks.*

Complete system disk

Every Apple IIGS user (and programmer) needs at least one complete system disk. It is a pool of system software resources, and may contain files missing from some application system disks. Table C-1 lists the contents of a complete system disk.

❖ *Note:* The word *complete* doesn't mean that the system disk has all the files that may be on your system disk—only that it has all the available system resources. For example, most system disks include files containing disk utility programs or finder-style program launchers. Those programs aren't considered here.

Table C-1
Contents of a complete system disk

Directory/File	Description
PRODOS	A routine that loads the proper operating system and selects an application, both at boot time and whenever an application quits.
SYSTEM/	A subdirectory containing the following files:
P8	The ProDOS 8 operating system.
P16	The ProDOS 16 operating system and Apple IIGS System Loader.
START	The first program executed: typically a program launcher or finder.
LIBS/	A subdirectory containing the standard system libraries.
TOOLS/	A subdirectory containing all RAM-based tool sets.
FONTS/	A subdirectory containing all fonts.
DESK.ACCS/	A subdirectory containing all desk accessories.
DRIVERS/	A subdirectory containing printer and port drivers.
SYSTEM.SETUP/	A subdirectory containing system initialization programs.
TOOL.SETUP	A permanent initialization file containing patches to ROM and a program to install them. This is the only required file in the SYSTEM.SETUP/ subdirectory; it is executed before any others that may be in the subdirectory.
ATINIT	A permanent initialization file that initializes the AppleTalk network.
ATLOAD.0	Another file for AppleTalk intialization.
BASIC.SYSTEM	The Applesoft BASIC system interface program.
APPLETALK/	A subdirectory containing files supporting the built-in Appletalk network interface.

The complete system disk is an 800K byte, double-sided 3.5-inch disk; the required files will not fit on a 140K, single-sided 5.25-inch disk. However, see "Application System Disks" (next).

When you boot a complete system disk, it executes the file `SYSTEM/START`.

The SYSTEM.SETUP/ subdirectory

The `SYSTEM.SETUP/` subdirectory may contain several different types of files, all of which are loaded at boot time. They include the following.

- **TOOL.SETUP:** This file must always be present; it is executed before any others in `SYSTEM.SETUP/`. `TOOL.SETUP` installs and initializes any RAM patches to ROM-based tool sets. After `TOOL.SETUP` is finished, ProDOS 16 loads and executes the remaining files in the `SYSTEM.SETUP/` subdirectory, which may belong to any of the categories listed below.
- **Permanent initialization files (filetype $B6):** These files are loaded and executed just like standard applications (type $B3), but they are not shut down when finished. They also must have certain characteristics:
  - They must be loaded in nonspecial memory.
  - They cannot permanently allocate any stack/direct-page space.
  - They must terminate with an RTL (Return from subroutine Long) rather than a QUIT.
- **Temporary initialization files (type $B7):** These files are loaded and executed just like standard applications (type $B3), and they are shut down when finished. They must terminate with an RTL rather than a QUIT.

Although they are loaded and installed in the system at the same time as the files in `SYSTEM.SETUP/`, desk accessories actually reside in the subdirectory `DESK.ACCS/`. There are two types.

- **New desk accessories (type $B8):** These files are loaded but not executed. They are put in nonspecial memory.
- **Classic desk accessories (type $B9):** These files are loaded but not executed. They are put in nonspecial memory.

Application system disks

Each application program or group of related programs comes on its own application system disk. The disk has all of the system files needed to run that application, but it may not have all the files present on a complete system disk. Different applications may have different system files on their application system disks.

Table C-2 shows which files must be present on all application system disks, and which files are needed only for particular applications. In some very restricted instances, it may be possible to fit an application and its required system files onto a single-sided (140K) 5.25-inch disk; most applications, however, require at least one double-sided (800K) 3.5-inch disk.

Table C-2
Required contents of an application system disk

Directory/File	Required?
`PRODOS`	Yes
`SYSTEM/`	Yes
`    P8`	(Required if the application runs under ProDOS 8)
`    P16`	Yes
`    START`	(Required if a START file, such as a finder, is to be used)
`    LIBS/`	(Required if system library routines are needed)
`    TOOLS/`	(Required if the application needs RAM-based tools)
`    FONTS/`	(Required if the application needs fonts)
`    DRIVERS`	(Required if the application does any printing or serial communication)
`    DESK.ACCS/`	(Required if desk accessories are to be provided)
`    SYSTEM.SETUP/`	Yes
`        TOOL.SETUP`	Yes
`BASIC.SYSTEM`	(Required if the application is written in Applesoft BASIC)
`APPLETALK`	(Required if the application supports printing to a LaserWriter or otherwise uses AppleTalk)

Important The files PRODOS, P8, and P16 all have version numbers. Whenever it loads an operating system (at startup or when launching an application), PRODOS checks the P8 or P16 version number against its own. If the numbers do not match, it is a fatal error. Be careful not to construct an application system disk using incompatible versions of PRODOS, P8, and P16.

Appendix D

HodgePodge Organization

This appendix presents three topics related to the organization of the sample program *HodgePodge*.

- ☐ It lists all HodgePodge routines and their source files for all three languages.
- ☐ It diagrams the routines that execute when HodgePodge opens a window.
- ☐ It discusses and lists HodgePodge's error-handling procedures.

HodgePodge subroutines

Table D-1 lists all HodgePodge routines. Column 1 lists, in alphabetical order, each routine in the Pascal version. Column 2 shows what source file each Pascal routine is in. Columns 3 and 4 name the source files containing the equivalent C and 65816 assembly-language routines. Column 5 gives the number of the chapter in which the Pascal version of each routine is discussed and listed. Column 6 briefly notes what each routine does.

Table D-1
HodgePodge routines (complete)

Routine	Pascal file	C file	Assembly file	Listed in ...	Function
AddToMenu	MENU.PAS	MENU.CC	MENU.ASM	Chap. 5	adds an item
AdjWind	WINDOW.PAS	WINDOW.CC	WINDOW.ASM	Chap.5	deletes an item
AskUser	PAINT.PAS	WINDOW.CC	WINDOW.ASM	Chap. 5	which file to open
CheckDiskError	DIALOG.PAS	DIALOG.CC	DIALOG.ASM	App. D	error alert box
CheckFrontW	EVENT.PAS	EVENT.CC	EVENT.ASM	App. G	adjusts menu items
CheckToolError	DIALOG.PAS	DIALOG.CC*	DIALOG.ASM	App. D	system failure
DisableAll	EVENT.PAS	EVENT.CC	EVENT.ASM*	App. G	adjusts menu items
DisableItems	EVENT.PAS	EVENT.CC	EVENT.ASM*	App. G	adjusts menu items
DispFontWindow	FONT.PAS	FONT.CC	FONT.ASM	Chap. 2	calls text-draw
DoAboutItem	DIALOG.PAS	DIALOG.CC	DIALOG.ASM	Chap. 4	"About" box
DoChooseFont	FONT.PAS	FONT.CC	FONT.ASM	Chap. 3	user selects font
DoChooserItem	PRINT.PAS	PRINT.CC	PRINT.ASM	Chap. 5	selects printer
DoCloseItem	WINDOW.PAS	WINDOW.CC*	WINDOW.ASM	Chap. 2	closes a window
DoMenu	MENU.PAS	MENU.CC	MENU.ASM	Chap. 2	dispatches menus
DoOpenItem	MENU.PAS	WINDOW.CC	WINDOW.ASM	Chap. 4	to open a window
DoPrintItem	PRINT.PAS	PRINT.CC	PRINT.ASM	Chap. 5	calls printing
DoQuitItem	MENU.PAS	EVENT.CC	EVENT.ASM	App. G	sets quit variable
DoSaveItem	PAINT.PAS	WINDOW.CC	WINDOW.ASM	Chap. 5	to save a file
DoSetMono	FONT.PAS	FONT.CC	FONT.ASM	App. G	toggles menu item
DoSetUpItem	PRINT.PAS	PRINT.CC	PRINT.ASM	Chap. 5	user page-setup
DoTheOpen	WINDOW.PAS	WINDOW.CC	WINDOW.ASM	Chap. 4	opens a window
DoWindow	MENU.PAS	WINDOW.CC	WINDOW.ASM	App. D	brings window to front
DrawTopWindow	PRINT.PAS	PRINT.CC	PRINT.ASM	Chap. 5	printing routine
EnableItems	EVENT.PAS	EVENT.CC*	EVENT.ASM*	App. G	adjusts menu items
FindMaxWidth	**	WINDOW.CC	WINDOW.ASM	App. E, F	sizes font window
HideAllWindows	WINDOW.PAS	WINDOW.CC	WINDOW.ASM	App. G	closes windows
HidePleaseWait	DIALOG.PAS	DIALOG.CC	DIALOG.ASM	Chap. 4	hides "wait" dialog
HodgePodge	HP.PAS	HP.CC*	HP.ASM	Chap. 2	main program
InitGlobals	GLOBALS.PAS	HP.H*	GLOBALS.ASM*	Chap. 2	initializes variables
LoadOne	PAINT.PAS	EVENT.CC	IO.ASM	Chap. 6	reads a picture file
MainEvent	EVENT.PAS	EVENT.CC*	EVENT.ASM	Chap. 2	main event loop

Table D-1 (continued)
HodgePodge routines (complete)

Routine	Pascal file	C file	Assembly file	Listed in ...	Function
MakeATemplate	DIALOG.PAS	DIALOG.CC*	DIALOG.ASM*	Chap. 4	creates alert items
ManyWindDialog	DIALOG.PAS	DIALOG.CC	DIALOG.ASM	App. G	caution alert
MountBootDisk	DIALOG.PAS	DIALOG.CC	DIALOG.ASM	App. D	asks user for disk
OpenFilter	PAINT.PAS	WINDOW.CC	WINDOW.ASM	Chap. 6	alters file display
OpenWindow	WINDOW.PAS	WINDOW.CC	WINDOW.ASM*	Chap. 4	to open a window
Paint	PAINT.PAS	WINDOW.CC	WINDOW.ASM	Chap. 2	calls picture-draw
PaintIt	PAINT.PAS	WINDOW.CC	WINDOW.ASM	Chap. 3	draws picture
SaveOne	PAINT.PAS	EVENT.CC	IO.ASM	Chap. 6	saves a picture file
SetUpDefault	PRINT.PAS	PRINT.CC	PRINT.ASM	Chap. 2	makes print record
SetUpForAppW	EVENT.PAS	EVENT.CC	EVENT.ASM	App. G	adjusts menu items
SetUpForDAW	EVENT.PAS	EVENT.CC	EVENT.ASM	App. G	adjusts menu items
SetUpMenus	MENU.PAS	MENU.CC	MENU.ASM	Chap. 2	makes menu bar
SetUpWindows	WINDOW.PAS	WINDOW.CC*	GLOBALS.ASM*	Chap. 2	sets size & loc.
ShowFont	FONT.PAS	FONT.CC	FONT.ASM	Chap. 3	draws text
ShowPleaseWait	DIALOG.PAS	DIALOG.CC	DIALOG.ASM	Chap.4	does "wait" dialog
ShutDownTools	HP.PAS	HP.CC	INIT.ASM	Chap. 2	shuts down
StartUpTools	HP.PAS	HP.CC	INIT.ASM	Chap. 2	starts all tools

* Name or content of routine is slightly different from the Pascal version.

** Does not exist in the Pascal version.

Execution sequence: opening a window

When a window is opened in HodgePodge, several routines are called in sequence, starting with `DoOpenItem`. The execution sequence starts out in the same way whether the window to be opened is a font window or a picture window.

The routines involved with opening a window are described in several different chapters in this book. To help you follow the sequence, we diagram the sequence of subroutine calls here, for both font windows and picture windows.

Opening a font window

A font window is opened when the user chooses Display Font from the Fonts menu. That causes execution to pass to the routine `DoOpenItem`, which calls `OpenWindow`. `OpenWindow` first calls `DoChooseFont`, then `DoTheOpen` to actually open the window.

After `OpenWindow` is finished, `DoOpenItem` calls `AddToMenu`, and then execution passes back to the main event loop. See Figure D-1.

❖ *Note:* The dimmed boxes in Figure D-1 represent routines called to open a picture window (Figure D-2).

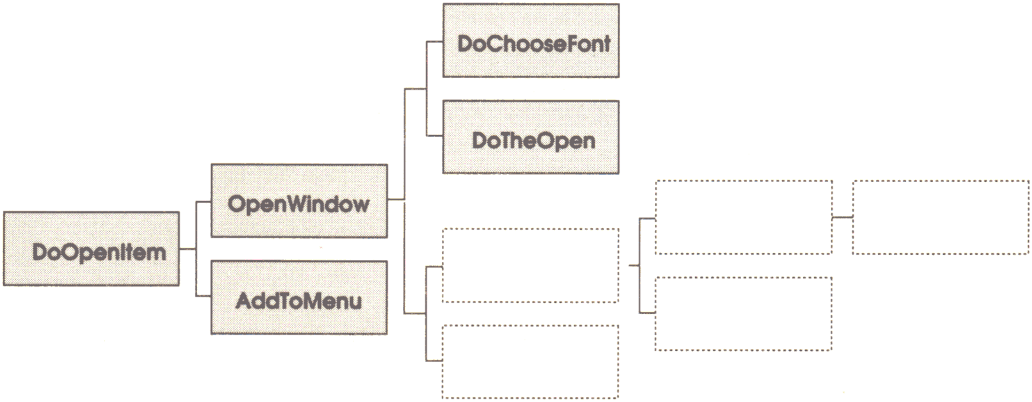


Figure D-1
Execution sequence: opening a font window

Opening a picture window

A picture window is opened when the user selects Open from the File menu. Just as when a font window is opened, execution passes to the routine `DoOpenItem`, and to `OpenWindow`.

In this case `OpenWindow` calls `AskUser`. `AskUser` first calls SFGetFile—part of the Apple IIGS Toolbox, not HodgePodge. SFGetFile calls the HodgePodge routine `OpenFilter` while it is displaying filenames. Once a filename is chosen, `AskUser` calls `LoadOne` to open the file. `OpenWindow` then calls `DoTheOpen` to actually open the window.

After OpenWindow is finished, DoOpenItem calls AddToMenu, and then execution passes back to the main event loop. See Figure D-2.

❖ *Note:* The dimmed boxes in Figure D-2 represent routines called to open a font window (Figure D-1).

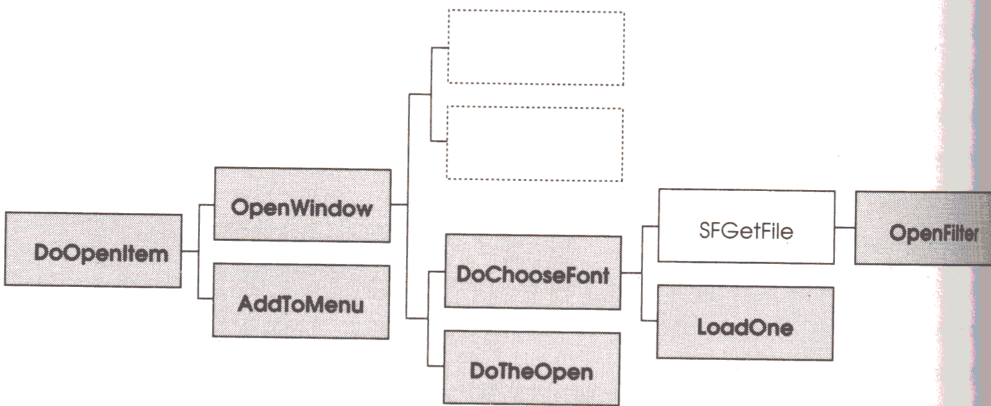


Figure D-2
Execution sequence: opening a picture window

Error handling

HodgePodge has three routines that handle error conditions: CheckToolError, MountBootDisk, and CheckDiskError. This section lists them and discusses what they do.

CheckToolError

CheckToolError is called only when the program is starting up. It is a very simple error handler, because any error it detects is made fatal, and because it puts up no message box for the user. In general, CheckToolError cannot put up a dialog box because the Dialog Manager may not have been started when CheckToolError is called.

`CheckToolError` is called after each tool startup call. It checks the value of the global variable `toolErrorNum`; if the number is nonzero an error has occurred. In that case `CheckToolError` calls the System Failure Manager, which puts up the "sliding apple" error screen and halts execution.

CheckToolError is in the source file DIALOG.PAS.

❖ *Input:* `CheckToolError` has a single input parameter: an integer *location number* that specifies what part of the program made the call. Each call to `CheckToolError` passes a different integer. The integers have no significance or purpose other than helping the programmer locate the part of the source code that generated the error.

```
procedure CheckToolError (Where: Integer);            {begin CheckToolError…}

var    toolErrorSave: Integer;
       deathMsg    : String;                          {string to display}

begin
  toolErrorSave := ToolErrorNum;                      {save the error number}
  deathMsg    :=                                      {This is the message with…}
       ' At $XXXX; Could not handle error $';         {…a dummy location number}

  if toolErrorSave <> 0 then                          {If there HAS been an error…}
   begin
   Int2Hex (Where,StringPtr(Longint                   {…convert loc. no. to a string…}
                     (@deathMsg)+6),4);               {…and insert it in message}
   SysFailMgr (toolErrorSave,deathMsg);               {Then go to system failure}
   end;                                               {end of IF error nonzero}
end;                                                  {End of CheckToolError}
```

MountBootDisk

MountBootDisk is in the source file DIALOG.PAS.

`MountBootDisk` is called during the loading of RAM-based tool sets, if the disk containing the tool sets is not already on line. `MountBootDisk` makes use of the Tool Locator routine TLMountVolume, which displays a dialog box prompting the user to remount the boot volume. See Figure D-3.

```
function MountBootDisk : integer;                    {begin MountBootDisk...}

var     promptStr : String;
        okStr     : String;
        cancelStr : String;
        volStr    : String;
        gbvParams : PathnameRec;

begin
  promptStr := 'Please insert the disk';             {string to appear in box}
  okStr     := 'OK';                                 {title of OK button (=1)}
  cancelStr := 'Shutdown';                           {title of Cancel button (=2)}
  gbvParams.pathName := @volStr;                     {define pointer to volume name}

  GET_BOOT_VOL (gbvParams);                          {find the boot volume name}

  MountBootDisk := TLMountVolume (174,30,            {Call Tool Locator's mount-volume...}
                        promptStr,volStr,            {...routine; it returns the number of...}
                        okStr,cancelStr);            {...the button user selects (1 or 2)}
end;                                                 {End of MountBootDisk}
```

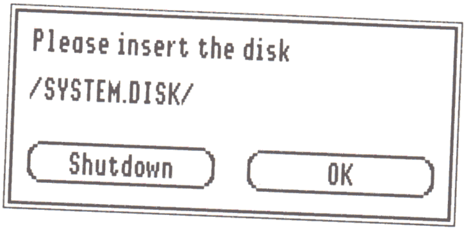


Figure D-3
TLMountVolume screen display

❖ *Note:* The TLMountVolume dialog box shown in Figure D-3 is not a real dialog box—the Tool Locator doesn't use the Dialog Manager because it can't assume that the Dialog Manager is available.

CheckDiskError

`CheckDiskError` is HodgePodge's primary example of an error-handling routine. It is called after every ProDOS 16 disk-access call.

`CheckDiskError` notes whether the previous operation caused an error and, if so, puts up a stop alert and returns TRUE as the function result. Otherwise it just returns with a value of FALSE.

CheckDiskError is in the source file DIALOG.PAS.

- *Input:* `CheckDiskError` has a single input parameter: an integer *location number* that specifies what part of the program made the call. Each call to `CheckDiskError` passes a different integer. The integers have no significance or purpose other than helping the programmer locate the part of the source code that generated the error.

```
function CheckDiskError(Where:Integer)
                             :Boolean;               {Begin CheckDiskError...}

var     itemClicked : Integer;                      {which button user clicks}
        ourAlert   : AlertTemplate;                 {defined in DIALOG.PAS}
        ourErrStr  : Str255;                        {error number to display}
        ourWhereStr : Str255;                       {our internal error code}
        ourString  : Str255;                        {error message}
        diskErrNum : Integer;                       {error number}

begin
  diskErrNum    := ToolErrorNum;                    {Save the global error number}
  CheckDiskError := (diskErrNum <> 0);              {Assign function result:
                                                     = TRUE if error nonzero}
  ourErrStr   := 'XXXX';                            {dummy chars. to set length byte}
  ourWhereStr  : = 'XX';                            {dummy chars. to set length byte}
  if diskErrNum <> 0 then
   begin
   Int2Hex (diskErrNum,StringPtr(
            LongInt(@ourErrStr)+1),4);              {Get ASCII string of error no.}
   Int2Hex (Where,StringPtr(
            LongInt(@ourWhereStr)+1),2);            {Get ASCII string of our code no.}
   ourString := concat ('Disk Error $',             {Build our error message...}
                        ourErrStr,
                        ' occurred at $',
                        ourWhereStr,
                        '.');
   MakeATemplate (@ourAlert,@ourString);            {Build a template for the alert}
   InitCursor;                                      {restore arrow cursor}
   itemClicked :=StopAlert(@ourAlert,NIL);          {Bring up the alert and take
                                                     the user's input}
   end;                                             {end of IF error nonzero}
end;                                                {End of CheckDiskError}
```

The alert box put up by CheckDiskError is shown in Figure 4-12.

Note that `CheckDiskError` calls `MakeATemplate` to define the features (text message and an OK button) the alert box will have. `MakeATemplate` is described under "Constructing Dialog Boxes and Alerts" in Chapter 4.

Appendix E

HodgePodge Source Code: Assembly Language

HP.ASM (main program)

```
***************************************************************************
*                                                                         *
*        HodgePodge:  An example Apple IIGS Desktop application           *
*                                                                         *
*     Written in 65816 Assembler by the Apple IIGS Development Team       *
*                                                                         *
*            Copyright (c) 1986-87 by Apple Computer, Inc.                *
*                         All Rights Reserved                             *
*                                                                         *
*   ----------------------------------------------------------------     *
*                                                                         *
*          This program and its derivatives are licensed only for        *
*          use on Apple computers.                                        *
*                                                                         *
*          Works based on this program must contain and                   *
*          conspicuously display this notice.                             *
*                                                                         *
*          This software is provided for your evaluation and to           *
*          assist you in developing software for the Apple IIGS           *
*          computer.                                                      *
*                                                                         *
*          This is not a distribution license. Distribution of           *
*          this and other Apple software requires a separate              *
*          license. Contact the Software Licensing Department of          *
*          Apple Computer, Inc. for details.                              *
*                                                                         *
*          DISCLAIMER OF WARRANTY                                         *
*                                                                         *
*          THE SOFTWARE IS PROVIDED "AS IS" WITHOUT                       *
*          WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED,               *
*          WITH RESPECT TO ITS MERCHANTABILITY OR ITS FITNESS             *
*          FOR ANY PARTICULAR PURPOSE.  THE ENTIRE RISK AS TO             *
*          THE QUALITY AND PERFORMANCE OF THE SOFTWARE IS WITH            *
*          YOU.  SHOULD THE SOFTWARE PROVE DEFECTIVE, YOU (AND            *
*          NOT APPLE OR AN APPLE AUTHORIZED REPRESENTATIVE)               *
*          ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING,             *
*          REPAIR OR CORRECTION.                                          *
*                                                                         *
*          Apple does not warrant that the functions                      *
*          contained in the Software will meet your requirements          *
*          or that the operation of the Software will be                  *
*          uninterrupted or error free or that defects in the             *
*          Software will be corrected.                                    *
*                                                                         *
*          SOME STATES DO NOT ALLOW THE EXCLUSION                         *
*          OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY              *
*          NOT APPLY TO YOU.  THIS WARRANTY GIVES YOU SPECIFIC            *
*          LEGAL RIGHTS AND YOU MAY ALSO HAVE OTHER RIGHTS                *
*          WHICH VARY FROM STATE TO STATE.                                *
*                                                                         *
*   ----------------------------------------------------------------     *
*                                                                         *
*   ASM65816 Code file "HP.ASM" -- Main routine and COPY's for other files *
*                                                                         *
***************************************************************************
```

```
***************************************************************************
*                                                                         *
*                    Version 1.0 -- August 1987                          *
*                                                                         *
***************************************************************************



             ABSADDR ON
             KEEP    HP
             MCOPY   HP.MACROS

**************************************************************
*
*  The main program
*
**************************************************************
HodgePodge     START
               using GlobalData

;------------------------------------------------
;
; Global equates used throughout the program.
;
True           gequ $8000
False          gequ $0000


;------------------------------------------------
;
; Set the data bank to code bank so I can use absolute
; addressing.
;
               phk
               plb

;------------------------------------------------
;
; Save address of D for use later
;
               tdc
               sta MyZP

;------------------------------------------------
;
; Load Init everything.
;
               pha
               PushWord #$0080          ; mode to use for QD
               jsl StartupTools
               pla                               ;Necessary because StartUpTools
;                                                ;uses Pascal calling convention
;                                                ;leaving input params on stack
               pla
               bne AllDone

               jsr SetupMenus


;------------------------------------------------
;
; Initialize system flags.
;
               stz LastWType
               stz QuitFlag
               stz Windex
```

```
;-----------------------------------------------------
;
; Zero the print record handle.
;
               stz PrintRecord
               stz PrintRecord+2

;-----------------------------------------------------
;
; Take events until user quits.
;
               jsr MainEvent

;-----------------------------------------------------
;
; All is done, let's shut down.
;
AllDone        anop
               jsl ShutDownTools

               PushLong PrintRecord      ; get rid of print record handle
               _DisposeHandle            ; if PrintRecord has zero in it
;                                        ; dispose handle will fail but
;                                        ; we don't care.

               _Quit QuitParams

               END



               COPY  7/E16.WINDOW
               COPY  7/E16.DIALOG

               COPY  INIT.ASM
               COPY  EVENT.ASM
               COPY  MENU.ASM
               COPY  WINDOW.ASM
               COPY  DIALOG.ASM
               COPY  FONT.ASM
               COPY  PRINT.ASM
               COPY  IO.ASM
               COPY  GLOBALS.ASM
```

INIT.ASM (Initialization)

```
****************************************************************************
*                                                                          *
*       HodgePodge:  An example Apple IIGS Desktop application             *
*                                                                          *
*                                                                          *
*           Copyright (c) 1986-87 by Apple Computer, Inc.                  *
*                       All Rights Reserved                                *
*                                                                          *
*                                                                          *
*  --------------------------------------------------------------------    *
*                                                                          *
*    ASM65816 Code file "INIT.ASM" -- Toolbox startup/shutdown routines    *
*                                                                          *
****************************************************************************

***************************************************************
*
* INIT.ASM
*
* Contains the following global data
*
*   MyID            Variable holding userid of this program
*
*   ThisMode        Variable holding mode used to start
*                   QuickDraw
*   OrigPort        Variable holding pointer to original
*                   port that QuickDraw has when started up.
*
* Contains the following private data
*
*   ZPHandle        Holds handle to memory that is used
*                   as direct page for the tools.
*   ZPPtr           Pointer to above memory.
*
* Contains the following public procedures.
*
*   function  StartupTools  (ModeToUse : SCB_type) : integer;
*
*      Starts up the tools (initializing quickdraw with the specified
*      mode) and initializes the global variables above.
*
*   procedure  ShutdownTools;
*      Shuts things down, undoing what was done above.
*
* Uses the MountBootDisk dialog routine to have the user put the
* system disk on line.
*
* Uses the CheckToolError dialog routine to cause a system death
* (bouncing apple) if the A register is nonzero.  The X register is
* assumed to contain a "Where" value.
*
* Change History
*
* June   1987  Steven E. Glass
* August 1987  Ben Koning
*
* Modified to use the C calling convention so that can be used by
* both C and TMLPascal.  (Input parameters are not removed from
* the stack.)
*
***************************************************************
```

```
InitDummy       START

                COPY 7/E16.MEMORY

                END

****************************************************************
*
* StartupTools
*
*     Input:    ModeToUse  -- $0080 for 640 mode
*     Output:   ErrorCode  -- Error if nonzero
*                             (NOTE: DIFFERENT FROM C AND PASCAL VERSIONS)
*
* Calling Sequence:
*
*         pha                    ; space for output
*         PushWord #Mode         ; Mode to use for QD
*         jsl StartupTools
*         plx                    ; remove input parameter
*         pla                    ; get func result
*         bne MustQuit
*
* This is a subroutine to load and startup all the tools
* an application generally needs.  This routine also gets the
* space in bank zero that the tools use for direct page.  The
* only time an error code other than zero is returned is when
* the boot disk is not on line and the user asks to cancel
* rather than to put it on line.
*
* Order of work:
*
* 1)  Start
*
*      Tool Locator, Memory Manager, Misc Tools
*      QuickDraw, Event Manager
*
* 2)  When these are running, the "One moment please" string is
* displayed and LoadTools is called.
*
* QuickDraw and the Event Manager are started up first
* because if the LoadTools call returns a VolNotFound error
* we need to have the volume mounted.  This is done with
* the TLMountVolume call which requires both QuickDraw and
* the Event Manager to be active.
*
* 3)  Next I start up
*
*      Window Manager, Control Manager,
*      Menu Manager, LineEdit, Dialog Manager
*
* 4)  After these are initialized, I setup and draw the
* menu bar and display a message to the user before I
* initialize the rest (Standard File, Font Manager,
* QuickDraw Auxiliary and print manager).
*
****************************************************************
StartupTools    START
                using InitData

ModeToUse       equ   $5
ResultCode      equ   $7
```

```
;-----------------------------------------------
;
; Direct Page use.  The following equates
; describe how the direct pages are assigned
; to the tools below.
;
DPForQuickDraw equ $000                 ; needs 3
DPForEventMgr  equ $300                 ; needs 1
DPForCtlMgr    equ $400                 ; needs 1
DPForLineEdit  equ $500                 ; needs 1
DPForMenuMgr   equ $600                 ; needs 1
DPForStdFile   equ $700                 ; needs 1
DPForFontMgr   equ $800                 ; needs 1
DPForPrintMgr  equ $900                 ; needs 2

TotalDP        equ $B00

;-----------------------------------------------
;
; Just in case this routine is called when the
; data bank is set somewhere else we set it
; right here.
;
               phb
               phk
               plb

;-----------------------------------------------
;
; Copy the input parameter into the global
; data area and initialize the result code
; assuming all is well.
;
               lda ModeToUse,s
               sta ThisMode

               lda #0
               sta ResultCode,s

;-----------------------------------------------
;
; Start with TLStartup
;
               _TLStartup               ; Tool Locator


;-------------------------------------------------
;
; Initialize the memory manager.
;

               PushWord #0
               _MMStartup
               ldx #1
               jsr CheckToolError

               pla
               sta MyID

;-------------------------------------------------
;
; Initialize misc tools.
;
               _MTStartup
               ldx #2
               jsr CheckToolError
```

```
;--------------------------------------------------------
;
; First get some memory for the zero page we need!
;
               pha                          ; space for handle
               pha
               PushLong #TotalDP
               PushWord MyID
               PushWord #attrBank+attrPage+attrFixed+attrLocked
               PushLong #0
               _NewHandle

               ldx #3
               jsr CheckToolError


;--------------------------------------------------
;
; Take the resulting handle (still on the stack)
; and dereference it, putting the pointer into
; ZPPtr.
;
               phd                          ; save current D
               tsc                          ; turn stack into direct page
               tcd
               lda [3]                      ; deref the pointer
               sta ZPPtr                    ; we know that high word is 0
               pld                          ; restore direct page

               pla                          ; put handle into storage
               sta ZPHandle
               pla
               sta ZPHandle+2

;----------------------------------------------------------------
;
; Note that width on startup is 320 to allow doubling the
; screen width when doing best printing.
;

               lda ZPPtr
               clc
               adc #DPForQuickDraw
               pha
               PushWord ThisMode
               PushWord #320                ; max size of scan line in bytes
               PushWord MyID
               _QDStartup

               ldx #4
               jsr CheckToolError

               PushLong #0
               _GetPort
               PullLong OrigPort

               ldy #640
               lda ThisMode
               cmp #$80
               beq okmode
               ldy #320
okmode         anop
               sty MaxX

               lda ZPPtr
               clc
               adc #DPForEventMgr
               pha
```

```
          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

          ldx #5
          jsr CheckToolError

;-------------------------------------------------------
;
; Put up a string telling user that something is
; happening.
;
          PushWord #20
          PushWord #20
          _MoveTo

          PushWord #0
          _SetBackColor

          PushWord #$F
          _SetForeColor

          PushLong #MomentStr
          _DrawString

          _ShowCursor


;-------------------------------------------------------
;
; Make the LoadTools call
;
LoadAgain _GET_FILE_INFO ParamBlock       ;Try to find the directory
          bcc OkToLoad                    ;*/SYSTEM/TOOLS/.  Ok? Go load.

          jsr MountBootDisk               ;Else, display psuedo-dialog
          cmp #1                          ;Did they select "OK"?
          beq LoadAgain                   ;Yes, so try it again.

;                                         ;Else, they selected "Cancel".
          sta ResultCode,s                ;So return result code
          brl GetOut                      ;and leave this routine.

OkToLoad  PushLong #ToolTable             ;Push address of tool table
          _LoadTools                      ;Attempt to load them (should
          bcc ToolsLoaded                 ;work).  If ok, go on.

          ldx #6                          ;If error happened anyway,
          jsr CheckToolError              ;we'll just die here.

;-------------------------------------------------------
;
; The tools are loaded so start them up.
;
ToolsLoaded  anop
          _QDAuxStartup           ; QuickDraw Auxiliary

          _WaitCursor             ; With QDAux started we can show the
;                                 ; watch cursor

          PushWord MyID           ; Window Manager
          _WindStartup

          ldx #7
```

```
        jsr CheckToolError

        PushLong #$0000         ; display desktop
        _RefreshDeskTop

        PushWord MyID           ; Control Manager
        lda ZPPtr
        clc
        adc #DPForCtlMgr
        pha
        _CtlStartup

        ldx #8
        jsr CheckToolError

        PushWord MyID           ; LineEdit
        lda ZPPtr
        clc
        adc #DPForLineEdit
        pha
        _LEStartup

        ldx #9
        jsr CheckToolError

        PushWord MyID           ; Dialog Manager
        _DialogStartup

        ldx #10
        jsr CheckToolError

        PushWord MyID           ; Menu Manager
        lda ZPPtr
        clc
        adc #DPForMenuMgr
        pha
        _MenuStartup

        ldx #11
        jsr CheckToolError

        _DeskStartup            ; Desk Manager

        jsr ShowPleaseWait      ; Message for user

        ldx #12
        jsr CheckToolError

        PushWord MyID           ; Standard File
        lda ZPPtr
        clc
        adc #DPForStdFile
        pha
        _SFStartup

        ldx #13
        jsr CheckToolError

        PushWord #$8000         ; display file names in all caps
        _SFAllCaps

        PushWord MyID           ; Font Manager
        lda ZPPtr
        clc
        adc #DPForFontMgr
        pha
        _FMStartup
```

```
            ldx #14
            jsr CheckToolError

            PushWord MyID           ; Print Manager
            lda ZPPtr
            clc
            adc #DPForPrintMgr
            pha
            _PMStartup

            ldx #15
            jsr CheckToolError

            jsr HidePleaseWait

            _InitCursor             ; reset cursor to arrow cursor


;-----------------------------------------------------
;
; All is done.  We must clean up the stack and get out
;
GetOut      anop
            plb                     ; restore dbr
            rtl                     ; all done.

MomentStr   str 'One moment please...'

MaxX        ds 2

ParamBlock  dc i4'PathName'         ;ProDOS/16 Parameter block
            ds 2                    ;With pathname as input; rest of the
            ds 2                    ;fields will be set as output.
            ds  4
            ds 2
            ds 2
            ds 2
            ds 2
            ds 2
            ds  4
PathName    str '*/SYSTEM/TOOLS'

            END

*****************************************************************
*
* PublicInitData
*
* These are global variables available to the main program.
*
*****************************************************************

PublicInitDATA DATA ~Global

;---------------------------------------------
;
; Public Variables
;

MyID        ENTRY
            ds 2

ThisMode    ENTRY
            ds 2

OrigPort    ENTRY
            ds 4
            END
```

```
InitData        PrivDATA

ZPHandle        ds 4

ZPPtr           ds 4

ToolTable       anop
StartTable      anop
                dc i'(EndTable-StartTable)/4'
                dc i'1,$0101'               ; tool locator
                dc i'2,$0101'               ; memory manager
                dc i'3,$0101'               ; misc tools
                dc i'4,$0101'               ; quickdraw
                dc i'5,$0100'               ; desk manager
                dc i'6,$0100'               ; event manager
                dc i'14,$0103'              ; window manager
                dc i'15,$0103'              ; menu manager
                dc i'16,$0103'              ; control manager
                dc i'18,$0100'              ; quickdraw aux
                dc i'19,$0100'              ; print manager
                dc i'20,$0100'              ; line edit
                dc i'21,$0100'              ; dialog manager
                dc i'22,$0102'              ; scrap manager
                dc i'23,$0100'              ; standard file
                dc i'27,$0100'              ; Font manager
                dc i'28,$0100'              ; List manager
EndTable        anop
                END


*****************************************************************
*
* ShutDownTools
*
*   Inputs:   None
*
*   Outputs:  None
*
*
* Shuts down every thing started up in InitTools
*
*****************************************************************

ShutDownTools   START
                using InitData


                _DeskShutdown               ; shut this first so that other tools
;                                           ; are still around (close DA's)
                _FMShutDown
                _PMShutdown
                _SFShutDown
                _DialogShutdown
                _LEShutdown
                _MenuShutDown
                _WindShutDown
                _CtlShutdown
;                                           ; this is shut down after window mgr
;                                           ; because window mgr makes contol
                                            ; manager calls at shutdown time.

                _EMShutDown
                _QDAuxShutdown
                _QDShutDown
```

```
_MTShutdown

PushLong ZPHandle        ; get rid of handle for direct
_DisposeHandle           ; page

PushWord MyID
_MMShutdown

_TLShutdown

rtl

END
```

MENU.ASM (menus)

```
************************************************************************
*                                                                      *
*         HodgePodge:  An example Apple IIGS Desktop application       *
*                                                                      *
*                                                                      *
*             Copyright (c) 1986-87 by Apple Computer, Inc.            *
*                           All Rights Reserved                        *
*                                                                      *
*                                                                      *
*   -----------------------------------------------------------------  *
*                                                                      *
*   ASM65816 Code file "MENU.ASM" -- Menu initialization and dispatching.  *
*                                                                      *
************************************************************************


**************************************************************
* Menu item ID's                                             *
**************************************************************

MenuIDs         start

AppleMenuID     gequ 1
FileMenuID      gequ 2
EditMenuID      gequ 3
WindowsMenuID   gequ 4
FontsMenuID     gequ 5

UndoID          gequ 250                 ; These next 6 are standard and
CutID           gequ 251                 ; required for DA support under
CopyID          gequ 252                 ; TaskMaster.
PasteID         gequ 253
ClearID         gequ 254
CloseWID        gequ 255

AboutID         gequ 256                 ; These are our own responsibility
QuitID          gequ 257
OpenWID         gequ 258
SaveID          gequ 259
ChooseID        gequ 260
SetupID         gequ 261
PrintID         gequ 262
ShowFontID      gequ 263
MonoID          gequ 264
                end

***********************************************************
*
* DoMenu
*
* Called when TaskMaster tells me that a menu item has
* been selected.
*
***********************************************************
```

```
DoMenu          START
                using GlobalDATA

                lda TaskDATA
                cmp #299                ; 299 is dummy do nothing - ignore
                beq Unhilite            ; do nothing
                bge DoWItem             ; 300 and up are added windows
                sec
                sbc #UndoID
                asl a
                tax
                jsr (MenuTable,x)


Unhilite        anop
                PushWord #False         ; draw normal
                PushWord TaskDATA+2     ; which menu
                _HiliteMenu

                rts


MenuTable       anop
                dc i'ignore'            ; Edit items
                dc i'ignore'
                dc i'ignore'
                dc i'ignore'
                dc i'ignore'
                dc i'doCloseItem'
                dc i'doAboutItem'
                dc i'doQuitItem'
                dc i'doOpenItem'
                dc i'doSaveItem'
                dc i'doChooserItem'
                dc i'doSetupItem'
                dc i'doPrintItem'
                dc i'doOpenItem'
                dc i'doSetMono'

DoWitem         anop

                sec
                sbc #300
ProcessW        anop                    ; In A is window number

                asl a
                asl a                   ; times 4 to index window list
                tax
                lda windowlist,x
                sta WhichWindow
                lda windowlist+2,x
                sta whichwindow+2
                jsr dowindow

                jmp  unhilite      ; done with it

                END
```

```
***************************************************************
*
* SetupMenus
*
* Now build the menu bar by inserting the six menus
* (back to front).
*
***************************************************************
SetupMenus      START
                using MenuDATA


                PushLong #0                 ; space for return
                PushLong #FontsMenu
                _NewMenu
                PushWord #0
                _InsertMenu

                PushLong #0                 ; space for return
                PushLong #WindowsMenu
                _NewMenu
                PushWord #0
                _InsertMenu

                PushLong #0                 ; space for return
                PushLong #EditMenu
                _NewMenu
                PushWord #0
                _InsertMenu

                PushLong #0                 ; space for return
                PushLong #FileMenu
                _NewMenu
                PushWord #0
                _InsertMenu

                PushLong #0                 ; space for return
                PushLong #AppleMenu
                _NewMenu
                PushWord #0
                _InsertMenu
;-----------------------------------------------------
;
; Call the desk accessory manager to install the
; list of NDAs in the system.
;
                PushWord #1
                _FixAppleMenu

;-----------------------------------------------------
;
; Finish off getting the menu bar ready.
;
                PushWord #0
                _FixMenuBar
                pla                         ;Discard menu bar height

                PushWord #10                ;Set starting position of menu
                _SetMTitleStart

                _DrawMenuBar                ;Actually draw the menu bar

                rts


                END
```

```
*****************************************************************
*
* AddToMenu:
*
*   Use the fact that the last SFGTEFILE returned in REPLY record
*   the name of the file and the state of the request.  Set PrintAvail.
*
*****************************************************************
AddToMenu       START
                using GlobalDATA

                lda #1
                sta PrintAvail              ;Set PrintAvail flag to allow printing

                pushlong #0                 ;it's the front window we're adding in
                _FrontWindow
                pla
                sta whichwindow
                plx
                stx whichwindow+2             ;get result for pushing in a sec.

                PUSHLONG #0                 ;space for result
                PUSHLONG whichwindow
                _GetWrefCon                 ;refcon has handle to data

                pla
                sta Temphandle
                plx
                stx TempHandle+2

                jsr Deref                   ; dereference
                sta 0
                stx 2

                PushWord  Windex            ;font's size
                PushLong  #Iddgt            ;ptr to string
                PushWord #2                 ;length of string
                PushWord #0                 ;unsigned integer
                _Int2Dec                    ;convert size into an ASCII string

                lda iddgt
                ora #'00'
                sta iddgt

                ldy #oLength                ;get names length
                lda [0],y                   ;find end of string to slide stuff
                and #$FF
                clc
                adc #6
                tay                         ; y index off ids is where we store
                ldx #0                      ; x index off idn is where we load

cpyidlp         lda idn,x
                sta [0],y
                iny
                iny
                inx
                inx
                cpx #6                      ; do 6 bytes
                bne cpyidlp

                lda 0                       ;now pt. to itemlist loc. for insert
                clc
                adc #4
                tax
                lda 2
                adc #0
                pha
```

```
                phx
                PUSHWORD #$FFFF
                PUSHWORD #WindowsMenuID
                _InsertMItem

                lda windex                ; if first time, omit dummy 299
                bne NotFirstTime
                PushWord #299             ; 299 is dummy item to delete
                _DeleteMItem               ; it's gone, now add next one

                Pushword #$ff7f
                PushWord #WindowsMenuID
                _SetMenuFlag

                Lda #True
                Sta NeedToUpdate

NotFirstTime    lda #0                    ; re-calc size
                pha
                pha
                PushWord #WindowsMenuID
                _CalcMenuSize

                lda Windex                ; save off window Pointer for menu stuff
                asl a
                asl a
                tax                       ;*4 for WINDOWLIST index
                lda whichwindow
                sta WindowList,x
                tay
                lda whichwindow+2
                sta WindowList+2,x

                inc windex                ; bump counter for next add on

                lda temphandle
                ldx temphandle+2
                jsr unlock                ; ok, let this loose again

                rts

idn             dc  c'\N3'                ; "\N3nn" will slide in behind it
iddgt           dc  c'00'                 ;00->15 slides into nn
idcr            dc  i1'13'                ;and finally a carriage return
iddmy           dc  i1'0'                 ;a dummy so we slide exactly 8

                END
```

```
***************************************************************
*
* Menu Data
*
***************************************************************
MenuData        DATA


Return          equ 13


AppleMenu       dc c'>>@\XH',i'AppleMenuID',i1'RETURN'
                dc c'==About HodgePodge...\H',i'AboutID',i1'RETURN'
                dc c'==-\N500D\0',i1'RETURN'
                dc c'.'

EditMenu        dc c'>>  Edit  \DH',i'EditMenuID',i1'RETURN'
                dc c'==Undo\*ZzH',i'UndoID',i1'RETURN'
                dc c'==-\N500D\0',i1'RETURN'
                dc c'==Cut\*XxH',i'CutID',i1'RETURN'
                dc c'==Copy\*CcH',i'CopyID',i1'RETURN'
                dc c'==Paste\*VvH',i'PasteID',i1'RETURN'
                dc c'==Clear\H',i'ClearID',i1'RETURN'
                dc c'.'

FileMenu        dc c'>>  File  \H',i'FileMenuID',i1'RETURN'
                dc c'==Open...\*OoH',i'OpenWID',i1'RETURN'
                dc c'==Close\DH',i'CloseWID',i1'RETURN'
                dc c'==Save As...\DH',i'SaveID',i1'RETURN'
                dc c'==-\N500D\0',i1'RETURN'
                dc c'==Choose Printer...\H',i'ChooseID',i1'RETURN'
                dc c'==Page Setup...\DH',i'SetupID',i1'RETURN'
                dc c'==Print...\D*PpH',i'PrintID',i1'RETURN'
                dc c'==-\N500D\0',i1'RETURN'
                dc c'==Quit\*QqH',i'QuitID',i1'RETURN'
                dc c'.'

WindowsMenu     dc c'>>  Window  \DH',i'WindowsMenuID',i1'RETURN'

OrigItem        ENTRY
                dc c'==No Windows allocated\N299',i1'RETURN'
                dc c'.'

FontsMenu       dc c'>>  Fonts  \H',i'FontsMenuID',i1'RETURN'
                dc c'==Display Font...\*FfH',i'ShowFontID',i1'RETURN'

MonoPropItem    ENTRY
                dc c'==Display Font as Mono-spaced\*MmH',i'MonoID',i1'RETURN'
                dc c'.'
MonoStr         dc c'==Display Font as Mono-spaced\H',i'MonoID',i1'RETURN'
PropStr         dc c'==Display Font as Proportional\*MmH',i'MonoID',i1'RETURN'


******NOTE: 300 is starting number for a building list - used in AddToMenu
******      299 is the dummy one that is deleted when we get a real one

                END
```

EVENT.ASM (main event loop)

```
*****************************************************************
*                                                               *
*        HodgePodge:  An example Apple IIGS Desktop application *
*                                                               *
*                                                               *
*            Copyright (c) 1986-87 by Apple Computer, Inc.      *
*                        All Rights Reserved                    *
*                                                               *
*   ----------------------------------------------------------  *
*                                                               *
*   ASM65816 Code file "EVENT.ASM" -- TaskMaster call; Dispatching to all *
*                             other routines; Menu dimming.     *
*****************************************************************


*****************************************************************
*
* Event
*
* This contains the main event loop.
*
*****************************************************************

MainEvent       START
                using GlobalData

Again           anop

                lda QuitFlag            ;Has Quit been select?
                bne AllDone             ;... if so, stop the loop.

                jsr CheckFrontW         ;Handle the menu dis/enable

                PushWord #0
                PushWord #$FFFF
                PushLong #EventRecord
                _TaskMaster

                pla
                beq Again               ;No event? loop.

                asl a                   ;Multiply by two...
                tax                     ;use for index into...
                jsr (TaskTable,x)       ;dispatch table to execute events.

                bra Again               ;Loop.

AllDone         rts

TaskTable       anop
;------------------------------------------------
;
; Event manager events
;
                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'DoActivate'        ; 8 activate
            dc i'ignore'            ; 9 switch
            dc i'ignore'            ; 10 desk acc
            dc i'ignore'            ; 11 device driver
            dc i'ignore'            ; 12 ap
            dc i'ignore'            ; 13 ap
            dc i'ignore'            ; 14 ap
            dc i'ignore'            ; 15 ap

;-----------------------------------------------
;
; Task master events

            dc i'ignore'            ; 0 in desk
            dc i'DoMenu'            ; 1 in MenuBar
            dc i'ignore'            ; 2 in system window
            dc i'ignore'            ; 3 in content of window
            dc i'ignore'            ; 4 in drag
            dc i'ignore'            ; 5 in grow
            dc i'DoCloseItem'       ; 6 in goaway -- same as "Close" item
            dc i'ignore'            ; 7 in zoom
            dc i'ignore'            ; 8 in info bar
            dc i'DoMenu'            ; 9 in special menu item
            dc i'ignore'            ; 10 in OpenNDA
            dc i'ignore'            ; 11 in frame
            dc i'ignore'            ; in drop

            END

***************************************************************
*
* CheckFrontW
*
* Checks to see if front window has changed and if
* so deals with various menu enables and disables.
* called by main event loop, and activate events.
*
***************************************************************
CheckFrontW     Start
                using MenuData
                using GlobalData

                PushLong #0
                _FrontWindow
                PullLong ThisWindow     ;get the current front window.

                lda ThisWindow          ;Check to see if it is
                cmp LastWindow          ;still the same window as
                bne Changed             ;last time
                lda ThisWindow+2
                cmp LastWindow+2
                bne Changed

Exit1           rts                     ;No Change No problem....Else.

Changed         anop
                lda ThisWindow          ;LastWindow := ThisWindow
                sta LastWindow
                lda ThisWindow+2
                sta LastWindow+2

                jsr TypeThisW           ;set ThisWType=type of the new front win

                lda ThisWType           ;arriving here, the window has changed.
```

```
                cmp LastWType               ;it's type may not have changed.
                beq Exit1                   ;Branch taken if the latter is true.

!ok so start changing menus

                cmp #0                      ;is there a front window
                bne ThereIs1                ;take this branch if there is.

                jsr SetupForNoW             ;if no front window then disable
                bra FinishUp                ;various thing I care about and go
!                                           ;Finish up

ThereIs1        ANOP
                cmp #1                      ;is it a system (Da)
                bne NotSysW                 ;taken if not.

                jsr SetUpForDaW             ;else it is a da. do what's needed
                bra FinishUp                ;and do the exit stuff

NotSysW         jsr SetUpForAppW            ;A-reg = Wtype. Go deal w/menu stuff

! And drop into exit stuff

FinishUp        lda NeedToUpdate            ;has the menu bar changed
                beq ReallyDone              ;taken if not. else

                _DrawMenuBar                ;we need to re-draw the menu
                stz NeedToUpdate            ;and say we did it.

ReallyDone      lda ThisWType               ;LastWType := ThisWType
                sta LastWType
                rts

* figure out the type of the front window.
* 0= there is no window. 1 = it's a da window. 2 = App Font Win. 3= App Pic Win.

TypeThisW       anop
                lda ThisWindow              ;was there a window at all ?
                ora ThisWindow+2
                sta ThisWType               ;if no front window then ThisWtype=0
                beq DoneEarly               ;taken if there really was no front win

                PushWord #0                 ;get and save wuther or not
                PushLong ThisWindow         ;this is a
                _GetSysWFlag                ;system window or not.
                pla
                beq WasApp                  ;0 means not a sys window

                lda #1                      ;it's a sys (da) window so
                sta ThisWType               ;set lastwtype = 1
DoneEarly       rts

WasApp          Anop                        ;it's an app win. find out what kind.

                PushLong #0                 ;space for get ref con in a sec
                PushLong ThisWindow         ;else I have the window ptr

                _GetWrefCon                 ;get refcon it has handle to data

                pla                         ;recon handle to
                sta Temp                    ;temp and A/X
                plx
                stx Temp+2

                jsr deref                   ;lock it down for a sec

                sta 0
                stx 2
```

```
            ldy #oFlag                  ;check if picture
            lda [0],y                   ;get window type
            beq PicW

            lda #2                      ;it's a font window so...
            sta ThisWType               ;say so and
            bra OuttaHere               ;split

PicW        lda #3                      ;it's a pic window. so
            sta ThisWType               ;say so and split.

OuttaHere   lda Temp
            ldx Temp+2
            jsr Unlock                  ;unlock the refcon handle.'
            rts

Temp        ds 4


ThisWindow  ds 4
LastWindow  ds 4

            END

****************************************************************
*
* doQuitItem
*
* Sets quit flag.
*
****************************************************************
doQuitItem  START
            using GlobalDATA

            lda #True
            sta QuitFlag

            rts
            END

******************************************************************
*
* DoActivate
*
* Handles activation of windows and adjusts the edit meun
* based on window type.
*
******************************************************************
DoActivate  Start
            using GlobalData

            lda EventModifiers
            and #1
            beq end                     ;don't care about deactivate ?

            jsr CheckFrontW
end         rts
            END
```

```
*******************************************************************
*
* SetUpForAppW
*
* Sets the edit menu items up for the application window:
* that is disabling them. And sets the other file menu items
* accordingly.
*
*******************************************************************
SetUpForAppW    Start
                Using GlobalData
                Using MenuData

                PushLong #0             ;get ready to call changeMitems
                PushWord #SaveID        ;We gonna do save item. but we need

                lda ThisWType           ;to figure out whether it should be
                cmp #3                  ;enabled or not. is it a font window ?
                bne NoSaveEnable        ;if so dont enable the save item.

                PushWord #True          ;else push true for enable
                bra Cont

NoSaveEnable    PushWord #False

Cont            PushWord #CloseWID
                PushWord #True

                lda PrintAvail
                beq SkipPrint

                PushWord #PrintID
                PushWord #True
                PushWord #SetUpID
                PushWord #True
SkipPrint       jsr ChangeMItems

                lda LastWType
                cmp #1                  ;was it a da last ?
                bne Exit                ;if not we don't need to do whats next

                PushWord #$0080         ;disable edit menu
                PushWord #EditMenuID
                _SetMenuFlag
                lda #True               ;set update flag so I only redraw
                sta NeedToUpdate        ;the menu bar once

Exit            rts

                END

*******************************************************************
*
* SetUpForNoW
*
* Sets the edit menu items up for the desk acc window:
* that is enabling edit menu, and close in file menu.
* accordingly.
*
*******************************************************************
SetUpForNoW     START
                Using GlobalData
                Using MenuData

                PushLong #0             ;end of list mark first...
                PushWord #SaveID        ;disble save
                PushWord #False         ;i desire disable.
```

```
PushWord #PrintID
            PushWord #False
            PushWord #SetUpID
            PushWord #False
            PushWord #CloseWID
            PushWord #False         ;enable
            jsr ChangeMItems

            lda LastWType           ;what was it last
            cmp #1                  ;was it a da last ?
            bne Exit                ;if not we don't need to do whats next

            PushWord #$0080         ;disable edit menu
            PushWord #EditMenuID
            _SetMenuFlag
            lda #True               ;set update flag so I only redraw
            sta NeedToUpdate        ;the menu bar once

Exit        rts
            End
******************************************************************
*
* SetUpForDaW
*
* Sets the edit menu items up for the desk acc window:
* that is enabling edit menu, and close in file menu.
* accordingly.
*
******************************************************************
SetUpForDaW START
            Using GlobalData
            Using MenuData

            PushLong #0             ;end of list mark first...
            PushWord #SaveID        ;disble save
            PushWord #False         ;i desire disable.
            PushWord #PrintID
            PushWord #False
            PushWord #SetUpID
            PushWord #False

            PushWord #CloseWID
            PushWord #True          ;enable
            jsr ChangeMItems

            lda LastWType           ;what was it last
            cmp #1                  ;was it a da window last ?
            beq Exit                ;if so we don't need to do whats next

            PushWord #$ff7f         ;enable edit menu
            PushWord #EditMenuID
            _SetMenuFlag
            lda #True               ;set update flag so I only redraw
            sta NeedToUpdate        ;the menu bar once

Exit        rts
            END
```

```
*****************************************************************
*
* ChangeMItems
*
* Enables/Disables the various menu items according to the
* flags pushed on stack.
*
* Entry Stack Looks like:
*
*               0                       ;long indicator of end of items
*               ItemID                  ;word item id
*               Enable/Disable Flag     ;(word) true = enable
*               .
*               ItemID                  ;word item id
*               Enable/Disable Flag     ;(word) true = enable
*               .
*               Return                  ;word
*       Sp =>
*****************************************************************
ChangeMItems   Start

               PullWord RtaTemp         ;save return

Lp             lda 3,s                  ;check for end of list mark
               beq Done                 ;if so split

               pla
               bne DoEnable             ;taken if we should enable items

               _DisableMItem            ;else disable them
               bra Lp                   ;and start over

DoEnable       _EnableMItem             ;enable item
               bra Lp                   ;one more time

Done           PullLong                 ;pull end of list mark

               PushWord RtaTemp         ;push return address
               rts

RtaTemp        ds 2
EnableFlag     ds 2


               END
```

WINDOW.ASM (windows)

```
**************************************************************************
*                                                                        *
*       HodgePodge:  An example Apple IIGS Desktop application          *
*                                                                        *
*                                                                        *
*          Copyright (c) 1986-87 by Apple Computer, Inc.                 *
*                      All Rights Reserved                               *
*                                                                        *
*                                                                        *
*  ------------------------------------------------------------------    *
*                                                                        *
*        ASM65816 Code file "WINDOW.ASM" -- Open/Close windows           *
*                                                                        *
**************************************************************************



***************************************************************
*
* HideAllWindows
*
***************************************************************

HideAllWindows START
              using GlobalDATA

              stz VIndex              ;index for list of what was vis.

HideLoop      PushLong #0             ;hide 'em all, looks neater
              _FrontWindow
              ldx VIndex
              pla
              sta VTable,x
              pla
              sta VTable+2,x
              cmp #0
              bne dohid
              lda Vtable,x
              bne dohid
              rts                     ;all vis. windows hidden now

doHid         pha
              lda Vtable,x
              pha
              _HideWindow
              Lda Vindex
              clc
              adc #4
              sta Vindex
              bra HideLoop

              END
```

```
****************************************************************
*
* DoOpenItem :
*
*    1) Make sure not too many windows open already -- may show dialog
*
*    2) Call AddToMenu to add its name into the "windows" menu list
*
*
****************************************************************

DoOpenItem      START

                using GlobalDATA
                using FontDATA

                lda Windex                  ;Check if too many windows open already
                cmp #LastWind               ;... otherwise "window" menu overflows!
                bcc OkToOpen                ;No, so go ahead and try to open one
                jsr ManyWindDialog          ;Yes, so confront user with dialog box
                sec                         ;Set carry because it didn't happen
Done            rts

OkToOpen        jsr OpenWindow
                bcs Done                    ;if we didn't open, don't add it
                jmp AddToMenu               ;Add it to the menu list and exit

                END


****************************************************************
*
* DoSaveitem :
*
*
****************************************************************

DoSaveItem      START
                using GlobalDATA
                using IOData


                pushlong #0                 ;it's the front window we're saving
                _FrontWindow
                pla
                sta whichwindow
                plx
                stx whichwindow+2             ;get result for pushing in a sec.

                PUSHLONG #0                 ;space for result
                PUSHLONG whichwindow
                _GetWrefCon                 ;refcon has handle to data

                pla
                plx

                jsr deref

                sta 0
                sta Refptr
                stx 2
                stx Refptr+2

                ldy #oFlag                  ;check if picture
                lda [0],y
                beq oktosav                 ;save only type 0 windows
                rts
```

```
oktosav         PUSHLONG #0             ;space for result
                PUSHLONG whichwindow
                _GetWTitle

                pla
                sta NamePtr
                plx
                stx NamePtr+2

                PushWord #20            ; x loc
                PushWord #20            ; y loc
                PushLong #Prompt2       ; prompt string pointer
                PushLong NamePtr        ; File name
                Pushword #15            ; Max file name length
                PushLong #reply         ; reply list result
                _SFPutFile

                lda r_good              ; <> 0 means OK to load it
                bne Saveitoff

                rts


Saveitoff       anop

                _WaitCursor

                lda Refptr
                sta 0
                lda Refptr+2
                sta 2

                ldy #oHandle
                lda [0],y
                sta PicHandle
                iny
                iny
                lda [0],y
                sta PicHandle+2         ; this de-refd, is the data to write oute pichandle (we'll de-allocate)

                lda PicHandle
                ldx PicHandle+2
                jsr DeRef
                sta PicDestOUT
                stx PicDestOUT+2        ; now pointing to what we write

                lda #R_fullPN           ; put pointer to name in i/o param block
                sta NamePtr
                lda #^R_FullPN
                sta NamePtr+2


nopack          Jsr SaveOne
                Bcs OuttaHere

fixnm           lda refptr              ; now fix up name
                clc
                adc #oLength            ; where the name will go
                sta 0                   ; save in 0,2 also for later indirect
                sta refptr
                lda refptr+2
                adc #0
                sta 2
                sta refptr+2

                lda r_Fname
                and #$00FF
                tay
```

```
                sep #%00100000
                longa off
cpynm           lda r_Fname,y
                sta [0],y
                dey
                bpl cpynm
                rep #%00100000
                longa on

                pushlong refptr         ;(it points to string now, remember?)
                pushlong whichwindow
                _SetWTitle

                lda #0                  ; re-calc size
                pha
                pha
                PushWord #WindowsMenuId
                _CalcMenuSize

OuttaHere       lda PicHandle
                ldx PicHandle+2
                jsr Unlock

                _InitCursor

                rts
refptr          ds   4
                END


****************************************************************
*
* OpenWindow:
*
*    1) Call SFGETFILE to get name of file to display in window
*       (or the dialog to select font if Display Font call)
*
*    2) Gets memory for, and loads the picture/font data into memory
*
*    3) Allocates a new window
*          a) puts handle to MyWindowInfo in WrefCon
*          b) note that routine to draw picture contents is set to "PAINT"
*          c) note for font draw contents is "DISPFONTWINDOW"
*
*    The definition of MyWindowInfo is contained in global data
*
* If the menu manager is being used to add itemlist items with the file
* name, it will squeeze the \N etc. together (see AddToMenu).  In any
* case, the file name string for the window title can still be found
* starting at this area+5
*
* returns:  carry set - didn't open it (user cancelled SFGETFILE)
*           carry clear - window opened
*
****************************************************************
OpenWindow      START
                using GlobalDATA
                using IOData
                using FontDATA
                using WindowData

                lda TaskData
                cmp #ShowFontID         ; is it open for font window?
                bne AskUser
                jsr DoChooseFont
                bcs stp
                jmp DoTheOpen
stp             rts                     ;cancelled choose font
```

```
********
*
*  call SFGETFILE to request the file name
*
********
AskUser         lda #20
                pha                     ; x loc.
                lda #20
                pha                     ; y loc.
                PushLong #Prompt        ; prompt string pointer
                PushLong #OpenFilter    ; Do dimmed display of unloadables
                PushLong #0             ; list of types to include -- 0 for all
                PushLong #reply         ; reply list result
                _SFGetFile
                lda r_good              ; <>0 means OK to load it
                bne loaditup
                sec                     ; carry set return: didn't open
HandleError     rts

*********
*
* Get space for the picture file
*
********

LoadItUp        anop

                PushLong #0             ; space
                PushLong #$8000         ; size
                PushWord MyID           ; id
                PushWord #$0000         ; no restrictions
                PushLong #0             ; loc not important
                _NewHandle

                pla
                sta PicHandle
                plx
                stx PicHandle+2

                bcs HandleError         ; if error occured from no handle

                jsr Deref               ; derefence handle (in a,x)

                sta PicDestIN           ; put pointer in i/o param block
                stx PicDestIN+2

DoTheOpen       anop
                PushLong #0             ; space
                PushLong #MyWinfoSize   ; size
                PushWord MyID           ; id
                PushWord #$C000         ; fixed and locked
                PushLong #0             ; loc not important
                _NewHandle

                pla
                sta refcon
                plx
                stx refcon+2

                bcs HandleError

                jsr deref               ;de ref. for storing stuff into

                sta refptr
                sta 0
                stx refptr+2
                stx 2
```

```
*********
*
* Start by assuming this will be a picture window (not a font window).
* We set the address of the drawing routine to PAINT and set the flag
* in MyWindowInfo record to 0 indicating picture.
*
*********

         lda #Paint                    ; first the address of the Paint
         sta DrawRtn                   ; routine
         lda #^Paint
         sta DrawRtn+2
         ldy #oFlag                    ; Now set the flag field
         lda #0
         sta [0],y


;-------------------------------------------------------------
;
; Now we see if that silly assumption above was correct.
;
         lda TaskData                  ; look at the menu item that
         cmp #ShowFontID               ; brought us here.
         bne setIO                     ; not the font one so go on

         lda #1                        ; fix the flag field
         ora MonoFlag                  ; set bit 1 if monospaced font
         sta [0],y                     ; (y still set)
         lda DesiredFont               ; put the chosen fontid where
         sta PicHandle                 ; we will later put it in
         lda DesiredFont+2             ; the MyWindowInfo record
         sta PicHandle+2

         lda #DispFontWindow           ; finally, fix the pointer to the
         sta DrawRtn                   ; drawing routine
         lda #^DispFontWindow
         sta DrawRtn+2
         jmp DoMovNam

SetIO    lda #R_fullPN                 ; put pointer to name in i/o param block
         sta NamePtr
         lda #^R_FullPN
         sta NamePtr+2

*********
*
* load picture in "NamePtr" into "PicDest"
*
*********


               jsr LoadOne             ; load it
               bcc DoMovNam

IOError        anop                    ; There was an error loading the file
               PushLong RefCon         ; so dispose of the memory that we
               _DisposeHandle          ; allocated while trying to create
               PushLong PicHandle      ; this window
               _DisposeHandle
               sec
               rts
```

```
**********
*
* Move the files name into the param block
*
**********

DoMovNam        lda refptr              ;use zero page for indirect stores
                sta 0
                lda refptr+2
                sta 2

                lda pichandle           ;into the recfon area (refptr)
                ldy #oHandle
                sta [0],y
                iny
                iny
                lda pichandle+2
                sta [0],y

                ldy #oBlank             ; Put blank in record at blank field
                lda #'  '               ; (note this 16 bit store will over-
                sta [0],y               ; write the length field but we don't
;                                       ; care since we fix it below.

                lda refptr
                clc
                adc #oLength            ; where the name will go
                sta windaddr
                sta 0                   ; save in 0,2 also for later indirect
                lda refptr+2
                adc #0
                sta windaddr+2
                sta 2

                lda r_Fname
                and #$00FF
                cmp #MaxNameSize
                bmi NameLenOK
                lda #MaxNameSize
                sep #%00100000
                sta r_Fname
                rep #%00100000
NameLenOK       tay
                sep #%00100000
                longa off
cpynm           lda r_Fname,y
                sta [0],y
                dey
                bpl cpynm
                rep #%00100000
                longa on

                ldy #350                ;adjust max siz
                ldx #640                ;adjust pixel count
                stx DataWidth
                stx mcw
                sty IsizPos+6

;------------------------------------------------------------
;
; Set up the DataHeight based on the type of window it is.
;
                lda #188                ; assume picture and make 200 the max
                sta DataHeight          ; height
                lda RefPtr              ; now see what it really is
                sta 0
                lda RefPtr+2
                sta 2
```

```
                lda TaskData
                cmp #OpenWID
                bne NotisPicture
                jmp IsPicture

                PushLong OrigPort         ; Use the original port obtained during
                _SetPort                  ; startup to make sure a port is set
;                                         ; for the following text size calcs

NotIsPicture    PushLong #0               ; save this on the stack
                _GetFontID

                ldy #oFontID+2            ; now install the font that will
                lda [0],y                 ; be used in the current port
                pha
                dey
                dey
                lda [0],y
                pha
                PushWord #0
                _InstallFont

                PushLong #FIRecord        ; get the font info so can get
                _GetFontInfo              ; ascent and descent.

                PushLong #0               ; space for result
                lda ascent                ; now multiply sum of ascent &
                clc                       ; descent by num lines to draw
                adc Descent
                pha
                PushWord #NumLines+2
                _Multiply
                pla                       ; put result in DataHeight
                sta DataHeight
                pla                       ; strip off high word of nothing

                jsr FindMaxWidth

                PushWord #0               ; using saved fontid on stack
                _InstallFont              ; re-install the orig font

IsPicture       anop
*********
*
*  offset upperleft corner for opening of window
*
*********
                ldx #0
MovOff          lda ISizPos,x
                clc
                adc Wyoffset
                sta SizPos,x
                lda ISizPos+2,x
                clc
                adc Wxoffset
                sta SizPos+2,x
                inx
                inx
                inx
                inx
                cpx #8
                bne MovOff

                lda WxOffSet              ;adjust offsets
                clc
                adc #20
                sta WxOffset
                lda WyOffset
```

```
            clc
            adc #12
            cmp #120                    ;if we get too low, start at top
            bne DoYset
            lda #12
doYset      sta WyOffset



**********
*
* Now, Finally, create the new window
*
**********

Finally     PushLong #0                 ; space for result
            pushlong #WindowParamBlock
            _NewWindow

            pla
            sta whichwindow
            pla
            sta whichwindow+2

            PushLong OrigPort           ; Use the original port obtained during
            _SetPort                    ; startup to make sure a port is set

            lda PicHandle               ; unlock handle
            ldx PicHandle+2
            jsr Unlock


**********
*
* Force origin boundaries (see Manual definition of Window Mgr's SetOriginMask)
*
**********

            PushWord #$FFFE
            PushLong  whichwindow
            _SetOriginMask

            clc                         ; carry clear return: we opened it
            rts
            end

*****************************************************************
*
* WindowData
*
*****************************************************************
WindowData      data
WindowParamBlock anop
            dc i2'WindowEnd-WindowParamBlock'
            dc i2'FTitle+FClose+FRScroll+FBScroll+FGrow+FZoom+FMove+FVis'
windaddr    dc i4'0'                    Ptr to title
refcon      dc i4'0'                    RefCon
            dc i2'0,0,0,0'              Full Size (0= default)
            dc i4'0'                    Color Table Pointer
            dc i2'0'                    Vertical origin
            dc i2'0'                    Horizontal origin
DataHeight  dc i2'200'                  Data Area Height
DataWidth   dc i2'640'                  Data Area Width
            dc i2'200'                  Max Cont Height
McW         dc i2'640'                  Max Cont Width
            dc i2'4'                    Number of pixels to scroll vertically.
            dc i2'16'                   Number of pixels to scroll horizontally.
            dc i2'40'                   Number of pixels to page vertically.
```

```
                dc i2'160'
                dc i4'0'                    Number of pixels to page horizontally.
                dc i2'0'                    Infomation bar text string.
                dc i4'0'                    Info bar height
                dc i4'0'                    DefProc.
DrawRtn         dc i4'Paint'                Routine to draw info. bar.
SizPos          dc i2'0,0,0,0'              Routine to draw content.
                dc i4'$FFFFFFFF'            Size/pos of content
                dc i4'0'                    Plane to put window up in.
WindowEnd       anop                        Address for window record (0 to allocate)

Refptr          ds    4                        ;refcon pointer to 20 bytes
ISizPos         dc    i'20,10,80,350'          ;Size/pos of content


FiRecord        anop
Ascent          ds 2
Descent         ds 2
Leading         ds 2
WidMax          ds 2

                END

******************************************************************
*
* OpenFilter
*
*   This routine is passed to SFGetFile to filter out the filetypes
*   that are loadable by us.
*
* On entry, the stack looks like this:
*
*          |         previous contents          |
*          |------------------------------------|
*          |          space for result          |  word
*          |------------------------------------|
*          |  pointer to directory entry        |  long
*          |------------------------------------|
*          |          return address            |  3 bytes
*          |------------------------------------|
*          |                                    |  <- SP
******************************************************************

OpenFilter      start
                using GlobalData

                phb
                phk                         ; save DBR (and even out RTL addr)
                plb                         ; set DBR back to this bank
                pla
                sta RtnAddr                 ; save the return address
                pla
                sta RtnAddr+2

                tdc
                sta DPSave                  ; save the ROM's ZP
                lda MyZP
                tcd                         ; and swap in ours

                pla
                sta 0                       ; now get the pointer to the
                pla                         ; directory entry
                sta 2

                ldx #1
                                            ; assume visible and dimmed
                ldy #$10
                                            ; look at the filetype byte
```

```
            lda [0],y
            and #$00FF              ; don't look at the entire word

            cmp #$C1
            bne NotPicFile          ; pass on all others
            ldx #2                  ; show it as a selectable entry

NotPicFile  txa
            sta 1,s                 ; pass it back on the stack

            lda DPSave              ; point back to the old DP
            tcd

            lda RtnAddr+2           ; and put the return address back
            pha
            lda RtnAddr
            pha
            plb                     ; restore old DBR

            rtl

DPSave      ds 2
RtnAddr     ds 4

            end

****************************************************************
*
* FindMaxWidth - this routine finds out how wide the window
*       should be for the currently installed font.
*
****************************************************************

FindMaxWidth  start
            using WindowData
            using FontData
            using GlobalData

            PushWord #0             ; save prev set mono/pro flag
            _GetFontFlags

            ldy #oFlag              ; keep the result on the stack while
            lda [0],y               ; we set it to what we want (as
            lsr a                   ; defined by its window type set up
            and #$0001              ; when we open this window)
            pha

            _SetFontFlags

            stz MaxSoFar
            lda #1
            sta LineCounter
LineLoop    anop
*
            PushWord #0             ; space for width result.
            phk                     ; Get a pointer to the current line.
            phk                     ; The upper word is the same as the
            pla                     ; program bank.
            and #$00FF
            pha
            lda LineCounter         ; The lower word is stored in a table.
            asl a
            tax
            lda LineTable,x
            pha

            _StringWidth
*
```

```
                pla                         ; How does this line compare with the
                cmp MaxSoFar                ; previous longest line?
                bcc LessThan
                sta MaxSoFar                ; > or =, so save it as record holder.

LessThan        anop
                inc LineCounter             ; bump current line
                lda LineCounter
                cmp #NumLines
                bcc LineLoop

                lda MaxSoFar                ; Get the width of longest line.
                clc                         ; Add in room for left and right margins
                adc #10
                sta DataWidth

                _SetFontFlags               ; restore old settings

                rts
LineCounter     ds 2
MaxSoFar        ds 2
                end

****************************************************************
*
* DoCloseItem
*
*   Close a window, and dispose of extra data (in WrefCon)
*   and remove it from window list.  If no windows, then dim "Window"
*   menu and disallow printing.
*
****************************************************************
DoCloseItem     START
                using GlobalDATA

                pushlong #0                 ;it's the front window we're deleting
                _FrontWindow                ;...so get its GrafPortPtr
                pla
                sta whichwindow
                pla
                sta whichwindow+2           ; get result for pushing in a sec.
                ora WhichWindow             ; was there one?
                bne ThereIsOne
GotIt           rts                         ; quit now

ThereIsOne      PushLong WhichWindow        ; if it is a system window, this will
                _CloseNDAByWinPtr           ; close it
                bcc GotIt                   ; no error so done

; Must be one of mine.
dothecls        PUSHLONG #0                 ;space for result
                PUSHLONG whichwindow
                _GetWrefCon                 ;refcon has handle to data

                pla
                sta temp2Handle
                plx
                stx Temp2Handle+2           ;the refcon to de-allocate

                jsr deref
                sta 0
                stx 2

                ldy #oHandle
                lda [0],y
                sta PicHandle
                iny
                iny
```

```
            lda [0],y
            sta PicHandle+2         ; the pichandle (we'll de-allocate)

            ldy #oFlag              ; check if picture or font
            lda [0],y
            beq itsapic
            stz PicHandle           ; flag so we don't dispose
            stz PicHandle+2


ItsAPic     jsr AdjWind             ; goes and pulls window from WindowList

            clc                     ; position returned in a-reg.
            adc #300                ; start at 300
            sta IDdelete            ; the MenuID to de-allocate

            lda windex              ; if only one, we must be special
            cmp #1
            bne MoreThanOne

            pushlong #origitem      ; We're now deleting the only window
            pushword #0             ; left.
            pushword #WindowsMenuID
            _InsertMItem            ; add old "no windows" menu item.

            Pushword #$0080         ;Disable windows menu
            PushWord #WindowsMenuID
            _SetMenuFlag

            Lda #True
            Sta NeedToUpdate

            stz PrintAvail          ; Disallow printing

            lda #20                 ; reset start loc for window sizing
            sta WxOffset
            lda #12
            sta WyOffSet


MoreThanOne lda IDdelete
            pha                     ;now delete this item from menus
            _DeleteMItem
            dec windex

            lda windex              ;now, renumber list
            beq nomore              ; none left, skip

            sta IdCounter           ; counts how many
            lda #300                ; always the starting no.
            sta IDstart             ; will be first
            sta IDNew               ; and the new one
back        lda IdStart
            cmp IdDelete            ; is it the one we deleted?
            bne DoIt                ; nope, go re-set ID
            inc Idstart             ; yes, skip over it
            bra back
DoIt        pushword IdNew
            pushword IdStart
            _SetMItemId              ; reset
            inc IdStart
            inc IdNew
            dec IdCounter
            bne back

NoMore      lda #0                  ; re-calc size
            pha
```

```
                pha
                PushWord #WindowsMenuID
                _CalcMenuSize

                Pushlong Temp2Handle     ;get rid of refcon area
                _DisposeHandle

                lda PicHandle            ;is it font
                bne dodisp
                lda PicHandle+2
                beq skipdisp

DoDisp          Pushlong PicHandle       ;get rid of picture area
                _DisposeHandle

SkipDisp        PushLong WhichWindow     ;get rid of window
                _CloseWindow

skip            rts
****************************
*
* AdjWind finds and deletes a window list item which matches
* "WhichWindow" and returns in a-reg. where it's position was
*
* Note: it's optimized to find things near end of list
*       (if you'd prefer the other end, you'd need some different logic,
*        but here, generally, you'll open, look at it, and close it, so
*        this method seems best)
*
****************************
AdjWind         lda Windex
                tay                      ;use this to count thru
                dec a
                asl a                    ; pt. before last for end (a-2)*4
                asl a
                sta IDCounter
adjloop         dey
                bmi AdjDone
                tya
                asl a
                asl a
                tax
                lda WindowList,x         ;get the pointer (uniqueness exists)
                cmp WhichWindow
                bne adjloop
                lda WindowList+2,x
                cmp WhichWindow+2
                beq shoveChk
                bra Adjloop

ShoveIt         lda WindowList+4,x       ;now shove things up
                sta WindowList,x
                lda WindowList+6,x
                sta WindowList+2,x
                inx
                inx
                inx
                inx
ShoveChk        cpx IdCounter
                bne shoveit
AdjDone         tya
                rts

IdNew           ds 2
IdStart         ds 2
IdCounter       ds 2
IDdelete        ds 2
                END
```

```
****************************************************************
*
* Paint
*
* This draws picture in the window when task master calls.
*
****************************************************************
Paint          START
               using GlobalData
**********
*
*  get my own zero page
*
**********
               phb
               phk
               plb
               phd
               lda MyZP
               tcd

**********
*
* get the correct window port (got here from within taskmaster)
*
**********

               pushlong #0
               _GetPort
               plx
               ply                      ;get result for pushing in a sec.

               PUSHLONG #0              ;space for result
               phy
               phx                      ; saved the port here
               _GetWrefCon              ;refcon has handle to data

               pla
               sta Temphandle
               plx
               stx TempHandle+2

               jsr Deref                ; dereference
               sta 0
               stx 2

               ldy #oHandle             ; get handle to pic data
               lda [0],y
               sta picptr
               pha
               iny
               iny
               lda [0],y
               sta picptr+2
               tax
               pla

               jsl PaintIt

               lda TempHandle
               ldx TempHandle+2
               jsr Unlock

               pld
               plb

               rtl
               END
```

```
*****************************************************************
*
* PaintIt
*
* The routine which actually does the painting when passed the
* the handle to the picture in the a & x registers.
*
*****************************************************************
PaintIt         START
                using GlobalData

                phx                         ; save this on stack
                pha

                jsr deref                   ;deref. picture handle
                sta picptr
                stx picptr+2

                PushLong #SrcLocInfo
                PushLong #SrcRect
Pshmor          PushWord #0                 ; x
                PushWord #0                 ; y
                PushWord #0                 ; copy
                _PPToPort

                pla
                plx
                jsr Unlock

                rtl

                END

*****************************************************************
*
* DoGoAway -- not necessary because we handle it the same as
*             DoCloseItem.
*
*****************************************************************



*****************************************************************
*
* DoWindow
*
* Selects and shows window in response to menu selection.
*
*****************************************************************
DoWindow        START
                using GlobalDATA

                PUSHLONG WHICHWINDOW        ; select first so it only redraws
                _SelectWindow               ; once

                PUSHLONG WHICHWINDOW
                _ShowWindow

                rts


                END
```

DIALOG.ASM (dialog boxes)

```
****************************************************************************
*                                                                          *
*        HodgePodge:  An example Apple IIGS Desktop application            *
*                                                                          *
*                                                                          *
*            Copyright (c) 1986-87 by Apple Computer, Inc.                 *
*                          All Rights Reserved                             *
*                                                                          *
*                                                                          *
*  --------------------------------------------------------------------   *
*                                                                          *
*  ASM65816 Code file "DIALOG.ASM" -- Various dialogs taking modal control *
*                                                                          *
****************************************************************************


**************************************************************
*
* ManyWindDialog -- Warning that too many windows are open.
*
**************************************************************

ManyWindDialog START
               using GlobalData

               pha
               pushlong #OurAlert
               pushlong #$0000
               _CautionAlert
               pla                         ; get the item hit
               rts

OurAlert       dc i'30,120,80,520'         ; bounds rect
               dc i'2374'                  ; id
               dc h'80'
               dc h'80'
               dc h'80'
               dc h'80'
               dc i4'item1'
               dc i4'item2'
               dc i4'0000'

item1          dc i2'1'                    ; id
               dc i2'25,320,00,00'         ; bounds rect for button
               dc i2'ButtonItem'           ; type
               dc i4'But1'                 ; item descreptor
               dc i2'00'                   ; item value
               dc i2'0'                    ; item flag
               dc i4'0'                    ; item color

item2          dc i2'1348'                 ; id
               dc i2'11,72,200,640'        ; bounds rect for message
               dc i2'StatText+$8000'       ; type + disabled
               dc i4'Msg'                  ; item descreptor
               dc i2'00'                   ; item value
               dc i2'0'                    ; item flag
               dc i4'0'                    ; item color

But1           str 'OK'
Msg            str 'No more windows, please.'

               end
```

```
************************************************************************
*
* DoAboutItem
*
* Brings up about box and waits until button press until
* it puts it away. Shows how to build a dialog window by hand.
*
************************************************************************

DoAboutItem     START
                using globalData

                PushLong #0             ; get space for Icon
                PushLong #34*16+8       ; #lines * bytes/line + rect
                PushWord MyId
                PushWord #0             ; don't care where it goes
                PushLong #0
                _NewHandle
                pla
                plx
                bcc ok
                lda #$81                ; out of memory
                ldx #1
                jmp CheckDiskError      ; Go and tell user error message,
;                                       ; and use its RTS to exit from here.

ok              anop
                sta AppleIconH
                stx AppleIconH+2

                jsr deref

                sta 0
                stx 2

                ldy #0                  ;move Icon to new space
Copy640         lda AppleIcon640,y
                sta [0],y
                iny
                iny
                cpy #34*16+8
                bne Copy640

FixDBox         ldx #320-180
                lda #320+180

JoinRect        stx DRect+2
                sta DRect+6

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

                pla
                sta MDialogPtr
                pla
                sta MDialogPtr+2

                PushLong MDialogPtr
                PushWord #1
                PushLong #ButtonRect
                PushWord #ButtonItem
                PushLong #ButtonText
```

```
            PushWord #0
            PushWord #0
            PushLong #0
            _NewDItem

            PushLong MDialogPtr
            PushWord #2
            PushLong #AppleIconRect
            PushWord #IconItem+ItemDisable
            PushLong AppleIconH
            PushWord #0
            PushWord #0
            PushLong #0
            _NewDItem

            PushLong MDialogPtr
            PushWord #4
            PushLong #TextRect
            PushWord #LongStatText2+ItemDisable
            PushLong #StartOfText
            PushWord #EndOfText-StartOfText
            PushWord #0
            PushLong #0
            _NewDItem

DoModal     PushWord #0                ; result
            PushLong #0                ; no filterproc
            _ModalDialog
            pla                        ; chuck the item hit

            PushLong MDialogPtr
            _CloseDialog

            PushLong AppleIconH
            _DisposeHandle

            rts

DRect       dc i'20,10,192,320-10'

AppleIconH     ds 4
AppleIconRect  dc i'135,20,0,0'

AppleIcon640   anop
               dc i'0,0,34,64'
```

```
                dc  h'00000000000000000000000000000000'
                DC  h'0FFFFFFFFFFFFFFFFFFFFFFFFFFFFFF0'
                DC  h'0F0000000000000000000000000000F0'
                DC  h'0F0FFFFFFFFFFFFFFFFFFFFFFFFFF0F0'
                DC  h'0F0FFFFFFFFFFFFFFFFFF88FFFFFF0F0'
                DC  h'0F0FFFFFFFFFFFFFFFF8888FFFFFF0F0'
                DC  h'0F0FFFFFFFFFFFFFFF88888FFFFFF0F0'
                DC  h'0F0FFFFFFFFFFFFFF88888FFFFFFF0F0'
                DC  h'0F0FFFFFFFFFFFFF888888FFFFFFF0F0'
                DC  h'0F0FFFFFFFFFFFFF88888FFFFFFFF0F0'
                DC  h'0F0FFFFFFFFFFFFF8888FFFFFFFFF0F0'
                DC  h'0F0FFFFFF8888FFF88FF8888FFFFF0F0'
                DC  h'0F0FFFF88888888FFF88888888FFF0F0'
                DC  h'0F0FFF888888888888888888888FF0F0'
                DC  h'0F0FFeeeeeeeeeeeeeeeeeeeeFFFF0F0'
                DC  h'0F0FFeeeeeeeeeeeeeeeeeeeFFFFF0F0'
                DC  h'0F0FFeeeeeeeeeeeeeeeeeeFFFFFF0F0'
                DC  h'0F0FF666666666666666666FFFFFF0F0'
                DC  h'0F0FF666666666666666666FFFFFF0F0'
                DC  h'0F0FF666666666666666666FFFFFF0F0'
                DC  h'0F0FF4444444444444444444FFFFF0F0'
                DC  h'0F0FF44444444444444444444FFFF0F0'
                DC  h'0F0FFF444444444444444444444FF0F0'
                DC  h'0F0FFF555555555555555555555FF0F0'
                DC  h'0F0FFF555555555555555555555FF0F0'
                DC  h'0F0FFFF5555555555555555555FFF0F0'
                DC  h'0F0FFFF1111111111111111111FFF0F0'
                DC  h'0F0FFFFF11111111111111111FFFF0F0'
                DC  h'0F0FFFFFF111111FFF111111FFFFF0F0'
                DC  h'0F0FFFFFFF1111FFFFF1111FFFFFF0F0'
                DC  h'0F0FFFFFFFFFFFFFFFFFFFFFFFFFF0F0'
                DC  h'0F0000000000000000000000000000F0'
                DC  h'0FFFFFFFFFFFFFFFFFFFFFFFFFFFFFF0'
                DC  h'00000000000000000000000000000000'
TextRect        dc i2'4,10,135,340'
StartOfText     anop
                dc h'01',c'J',i'1'               ; center
                dc h'01',c'S',i'$0008'           ; Outline
                dc c'HodgePodge',h'0D'
                dc h'01',c'S',i'$0000'           ; plain
                dc c'A potpourri of routines that '
                dc c'demonstrate many features of '
                dc c'the Apple IIGS Tools.'
                dc h'0D'
                dc h'0D'
                dc c'By the Apple IIGS developement team including:'
                dc h'0D'
                dc h'0D'
                dc c'E. Berns, A. Cabral, C. Ewy, '
                dc c'S. Glass,'
                dc h'0D'
                dc c'D. Hitchens, B. Koning, S. Lee, '
                dc c'P. McDonald,'
                dc h'0D'
                dc c'B. Marks, D. Oliver, G. Ortiz, '
                dc c'and K. Rollin.'
                dc h'0D'
                dc h'0D'
                dc c'Copyright Apple Computer, Inc., '
                dc c'1986-87,'
                dc h'0D'
                dc c'All Rights Reserved',h'0D'
                dc c'v4.0        '
                dc c'&SysDate'
EndOfText       anop
```

```
ButtonRect      dc i'153,205,0,0'

ButtonText      str 'OK'

MDialogPtr      ds 4

                END

***************************************************************
*
* ShowPleaseWait / HidePleaseWait
*
* Brings up a window and immediately puts message in it
* (without waiting for update event).
*
***************************************************************

ShowPleaseWait START
               using globalData

               PushLong #0              ; save the current port
               _GetPort

               pla
               sta SavePort
               pla
               sta SavePort+2

               PushLong #0
               PushLong #DialogTemplate
               _GetNewModalDialog

               pla
               sta MsgWinPtr
               pla
               sta MsgWinPtr+2

               PushLong MsgWinPtr       ; begin the updating process
               _BeginUpdate

               PushLong MsgWinPtr
               _DrawDialog

               PushLong MsgWinPtr
               _EndUpdate

               rts

HidePleaseWait ENTRY
               PushLong MsgWinPtr       ; hide the window
               _CloseDialog

               PushLong SavePort        ; restore the port
               _SetPort

               rts

MsgWinPtr      ds 4

DialogTemplate anop
               dc i'30,120,80,520'      ; bounding box
               dc i'True'               ; visible
               dc i4'0'                 ; refcon
               dc i4'item1'
               dc i4'0000'
```

```
item1           anop
                dc i2'1348'             ; id
                dc i2'19,70,200,640'    ; bounds rect for text
                dc i2'StatText'         ; type
                dc i4'Msg'              ; item descriptor
                dc i2'00'               ; item value
                dc i2'0'                ; item flag
                dc i4'0'                ; item color
Msg             str 'Please wait while we set things up.'

                END


****************************************************************
*
* MountBootDisk
*
* This is a routine that is called whenever the application
* needs to get something off the boot volume and the
* boot volume is not on line.
*
* This can occur when loading fonts, tools or drivers.
*
****************************************************************

MountBootDisk   START

                _Set_Prefix SetPrefixParams
                _Get_Prefix GetPrefixParams

                PushWord #0             ;Space for result
                PushWord #174           ;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'
OKStr           str 'OK'
CancelStr       str 'Shutdown'

GetPrefixParams dc i'7'
                dc i4'VolStr'

SetPrefixParams dc i'7'
                dc i4'BootStr'

VolStr          ds 16

BootStr         str '*/'

                END
```

```
***************************************************************
*
* CheckToolError
*
* Cause system death if A register is nonzero and carry set;
* otherwise, it just returns.
*
* Error code to make part of string is in A register.
* "Where" number to make part of string is in X register.
*
***************************************************************

CheckToolError START
               bcs RealDeath              ;If a tool error didn't happen
               rts                        ;then just return

RealDeath      pha                        ;Save error code for now

               pea 0                      ;Convert the "Where" debug trace
               pea 0                      ;number to a four-digit ASCII hex
               phx                        ;string.
               _Hexit
               pla
               sta codes
               pla
               sta codes+2

               pla                        ;Restore error code

               pha                        ;Exit to system failure handler
               PushLong #DeathMsg         ;(bouncing apple)
               _SysFailMgr

DeathMsg       anop
               dc i1'EndMsg-StartMsg'
StartMsg       dc c' At $'
Codes          ds 4
               dc c'; Could not handle error $'
EndMsg         anop

               END


***************************************************************
*
* CheckDiskError -- Display stop alert dialog if ProDOS error happened.
*                   We sniff the A register to see if an error occurred,
*                   and assume the X register to be loaded with a
*                   "where" value, used to locate bugs.
*
***************************************************************

CheckDiskError START
               using GlobalData

               phx                        ; Save the Where value
               pha                        ; Save the error number

               _InitCursor                ; Set pointer--looks better than watch

               pla                        ; Restore the error number
               pha                        ; Convert the error message
               PushLong #OurErrStr        ; to an ASCII string 4 chars long
               PushWord #4
               _Int2Hex
```

```
              pla                       ; Do this just for clarity (note that
              pha                       ; Where value is already on stack!)
              PushLong #OurWhereStr     ; Convert the Where value
              PushWord #2               ; to an ASCII string 2 chars long
              _Int2Hex

              pha                       ; Space for result
              pushlong #OurAlert        ; Pointer to template
              pushlong #$0000           ; Standard Filter procedure
              _StopAlert                ; Draw box and wait for mouse OK press
              pla                       ; Get the item hit (the OK button)

              sec                       ; Set the error flag
              rts                       ; Return to caller

OurAlert      dc i'30,120,80,520'       ; bounds rect
              dc i'6666'                ; id
              dc h'80'
              dc h'80'
              dc h'80'
              dc h'80'
              dc i4'OKButton'
              dc i4'Message'
              dc i4'0000'

OKButton      dc i2'1'                  ; id
              dc i2'25,320,00,00'       ; bounds rect for button
              dc i2'ButtonItem'         ; type
              dc i4'OKName'             ; item descreptor
              dc i2'00'                 ; item value
              dc i2'0'                  ; item flag
              dc i4'0'                  ; item color

Message       dc i2'1348'               ; id
              dc i2'11,72,200,640'      ; bounds rect for static text
              dc i2'StatText+$8000'     ; type + disable flag
ErrMsgPtr     dc i4'Msg'                ; item descreptor
              dc i2'00'                 ; item value
              dc i2'0'                  ; item flag
              dc i4'0'                  ; item color

OKName        str 'OK'

Msg           dc i1'EndMsg-StartMsg'
StartMsg      dc c'Disk error $'
OurErrStr     ds 4
              dc c' occurred at $'
OurWhereStr   ds 2
              dc c'.'
              dc h'0D'
EndMsg        anop

               END
```

FONT.ASM (fonts)

```
*************************************************************************
*                                                                       *
*       HodgePodge:  An example Apple IIGS Desktop application          *
*                                                                       *
*                                                                       *
*           Copyright (c) 1986-87 by Apple Computer, Inc.               *
*                        All Rights Reserved                            *
*                                                                       *
*                                                                       *
*   -------------------------------------------------------------------  *
*                                                                       *
* ASM65816 Code file "FONT.ASM" -- Choose font; display font window contents *
*                                                                       *
*************************************************************************



************************************************************
*
* FontData
*
************************************************************
FontDATA  DATA

FontWinPtr  ds 4

DesiredFont     dc i4'$0800FFFE'          ; System Font size 8
MonoFlag        dc i2'0'                  ; start out showing proportional

CurFontInfo     anop
CFAscent        ds 2
CFDescent       ds 2
CFMaxWid        ds 2
CFLeading       ds 2


CurHeight       ds 2
LineCounter     ds 2

CurPos          ds 4


NumLines        equ 13

LineTable       dc i'Line0,Line1,Line2,Line3,Line4'
                dc i'Line5,Line6,Line7,Line8,Line9'
                dc i'Line10,Line11,Line12,Line12,Line12'


Line0  ds 30                              ; max name len is 25 + 1 for length
;                                         ; and 4 for size info
Line1  str ''
Line2  str 'The quick brown fox jumped over the lazy dog.'
Line3  str 'She sells sea shells down by the sea shore.'
Line4  str ''

Line5    dc h'20'
  dc h'00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F'
  dc h'10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F'
```

```
Line6  dc h'20'
  dc h'20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F'
  dc h'30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F'
Line7  dc h'20'
  dc h'40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F'
  dc h'50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F'
Line8  dc h'20'
  dc h'60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F'
  dc h'70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F'
Line9  dc h'20'
  dc h'80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F'
  dc h'90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F'
Line10 dc h'20'
  dc h'A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF'
  dc h'B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF'
Line11 dc h'20'
  dc h'C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF'
  dc h'D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF'
Line12 dc h'20'
  dc h'E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF'
  dc h'F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF'
   END

*****************************************************************
*
* DoChooseFont
*
*****************************************************************
DoChooseFont START
          using GlobalDATA
          using FontDATA

          PushLong #0
          _GetPort

          PushLong #TempPort
          _OpenPort

          PushLong #0                    ; space for result
          PushLong DesiredFont
          PushWord #0
          _ChooseFont

          lda 1,s
          ora 3,s
          bne ItChanged

          pla                            ; ChooseFont returned a 0000, so the
          pla                            ; font hasn't changed

          PushLong #TempPort
          _ClosePort

          _SetPort

          sec                            ; bad return
          rts


ItChanged anop
          pla
          sta DesiredFont
          pla
          sta DesiredFont+2

          pushword #0                    ;space for result
          PushWord DesiredFont
```

```
        PushLong #R_Fname
        _GetFamInfo
        pla                             ;ignore result

        lda DesiredFont+3               ; get font size in a reg
        and #$00FF
        pha
        lda #^R_FName                   ; high word of pointer to name
        pha
        lda R_FName
        and #$00FF
        inc a
        adc #R_FName
        pha                             ; low word of pointer
        PushWord #4                     ; output length
        PushWord #0                     ; not signed
        _Int2Dec

        lda R_FName                     ; bump the length
        inc a
        inc a
        inc a
        inc a
        sta R_FName

        PushLong #TempPort
        _ClosePort

        _SetPort

        clc                             ;good return
        rts

TempPort ds $AA                         ; size of graph port

   END

****************************************************************
*
* DispFontWindow
*
****************************************************************
DispFontWindow START
        using FontDATA
        using GlobalData

**********
*
*  get my own zero page
*
**********
        phb
        phk
        plb

        phd
        lda MyZP
        tcd

**********
*
* get the correct window port (got here from within taskmaster)
*
**********
```

```
         pushlong #0
         _GetPort
         plx
         ply                              ;get result for pushing in a sec.

         PUSHLONG #0                      ;space for result
         phy
         phx                              ; saved the port here
         _GetWRefCon

         pla
         sta Temphandle
         plx
         stx TempHandle+2

         jsr Deref                        ; de reference
         sta 0
         stx 2

         ldy #oFontID+2                   ; get the font ID
         lda [0],y
         tax
         dey
         dey
         lda [0],y

         jsl ShowFont

         lda TempHandle
         ldx TempHandle+2
         jsr Unlock

         pld
         plb

         rtl

         END

*****************************************************************
*
* ShowFont
*
* Common routine to actually draw the contents of the window.
* This routine is called with the font to install in the
* in the A & X registers.
*
*****************************************************************
ShowFont START
         using GlobalData
         using FontData

         phx                              ; save copy on stack
         pha

         phx                              ; install the font
         pha
         PushWord #0
         _InstallFont

         PushLong #CurFontInfo            ; Get its size info
         _GetFontInfo

         stz LineCounter                  ; zero the line counter

         clc                              ; calculate the line separation
         lda CFAscent
```

```
        adc CFDescent
        adc CFLeading
        sta CurHeight

        PushWord #0                 ; start the pen position at 0,0
        PushWord #0
        _MoveTo

        plx                         ; get fontid off stack

        pushword #0                 ; space for result
        phx                         ; family number was in x
        PushLong #Line0
        _GetFamInfo
        pla                         ; ignore result

        pla                         ; high word of font id
        xba                         ; size in high byte
        and #$00FF
        pha
        lda #^Line0                  ; high word of pointer to name
        pha
        lda Line0
        and #$00FF
        inc a
        adc #Line0
        pha                         ; low word of pointer
        PushWord #4                 ; output length
        PushWord #0                 ; not signed
        _Int2Dec

        lda Line0                   ; bump the length
        inc a
        inc a
        inc a
        inc a
        sta Line0

        PushWord #0                 ; save prev set mono/pro flag
        _GetFontFlags

        ldy #oFlag                  ; keep the result on the stack while
        lda [0],y                   ; we set it to what we want (as
        lsr a                       ; defined by its window type set up
        and #$0001                  ; when we opened this window)
        pha
        _SetFontFlags

LineLoop anop

        PushLong #CurPos            ; get the current position
        _GetPen

        PushWord #5                  ; reset x position
        lda CurPos
        clc
        adc CurHeight
        pha                         ; and y position
        _MoveTo

        lda LineCounter             ; draw current line
        asl a
        tax
        phk
        phk
        lda 1,s
        and #$00FF
        sta 1,s
```

```
        lda LineTable,x
        pha
        _DrawString

        inc LineCounter                 ; bump current line
        lda LineCounter
        cmp #NumLines
        bcc LineLoop

        _SetFontFlags                   ; restore from saved position

        rtl
        END

*****************************************************************
*
* doSetMono
*
*****************************************************************

doSetMono       START
                using FontData
                using MenuData

                lda MonoFlag
                eor #$02
                sta MonoFlag

                beq ChangeToMono        ;Change message to show effect in
                PushLong #PropStr       ;NEXT selection of this menu item
                bra PushID
ChangeToMono    PushLong #MonoStr
PushID          PushWord #MonoID
                _SetMItem
                rts


                END
```

PRINT.ASM (printing)

```
***************************************************************************
*                                                                         *
*       HodgePodge:  An example Apple IIGS Desktop application            *
*                                                                         *
*                                                                         *
*           Copyright (c) 1986-87 by Apple Computer, Inc.                 *
*                        All Rights Reserved                              *
*                                                                         *
*                                                                         *
*   ---------------------------------------------------------------------  *
*                                                                         *
*   ASM65816 Code file "PRINT.ASM" -- Print dialogs; Print Manager calls  *
*                                                                         *
***************************************************************************


**************************************************************
*
* DoChooserItem
*
* This is the routine that handles the Choose Printer
* menu item.
*
**************************************************************
DoChooserItem  START
               using GlobalData

               PushWord #0
               _PrChooser
               pla

               rts

               END


**************************************************************
*
* DoSetupItem
*
* This is the routine that handles the page setup item.
*
**************************************************************
DoSetupItem    START
               using GlobalData

               lda PrintRecord
               ora PrintRecord+2
               bne AlreadyThere

               jsr SetupDefault

AlreadyThere   anop
               pha
               PushLong PrintRecord
               _PrValidate
               pla
               Pushword #0
               Pushlong PrintRecord
```

```
                _prStlDialog
                pla

                rts

                END

*****************************************************************
*
* SetupDefault
*
* This routine creates the default PrintRecord.  Puts handle
* in PrintRecord.
*
*****************************************************************
SetupDefault    START
                using GlobalDATA

                PushLong #0
                PushLong #140
                Pushword MyID
                PushWord #$8010
                PushLong #0
                _newHandle
                pla
                sta PrintRecord
                pla
                sta PrintRecord+2

AlreadyThere    anop
                PushLong PrintRecord
                _prdefault

                rts


                END

*****************************************************************
*
* DoPrintItem
*
* This is the routine that handles the print item in the
* file menu.
*
*****************************************************************
DoPrintItem     START
                using GlobalData

                pha                         ; get the current port
                pha
                _GetPort

                pha                         ; first see if there is a window
                pha                         ; to print.
                _FrontWindow                ; and save pointer to it now
                pla                         ; before any dialogs are displayed!
                sta WindowToPrint
                pla
                sta WindowToPrint+2

                ora WindowToPrint
                bne SomethingToPrint
                brl SkipIt

SomethingToPrint anop
                lda PrintRecord
```

```
            ora PrintRecord+2
            bne AlreadySet

            jsr SetupDefault

AlreadySet  anop
            pha                         ; space for result
            PushLong PrintRecord
            _PrValidate
            pla                         ; ignore result since all is well now

            PushWord #0
            PushLong PrintRecord
            _PrJobDialog
            pla

            bne continue
            brl skipit

continue    anop

            _WaitCursor

            PushLong #0
            PushLong PrintRecord
            PushLong #0
            _PrOpenDoc
            pla
            sta PrintPort
            pla
            sta PrintPort+2

            PushLong PrintPort
            PushLong #0
            _PrOpenPage

            jsr DrawTopWindow

            PushLong PrintPort
            _PrClosePage

            PushLong PrintPort
            _PrCloseDoc

            PushLong PrintRecord
            PushLong #0
            PushLong #0
            _PrPicFile

            _InitCursor

SkipIt      anop

            _SetPort                    ; restore original port

            rts

            END
```

```
*****************************************************************
*
* DrawTopWindow
*
*
*
*****************************************************************
DrawTopWindow   START
                using GlobalDATA
                pha                         ; space for result of GetWRefCon call
                pha
                PushLong WindowToPrint
                _GetWRefCon

                pla
                sta TheRefCon
                plx
                stx TheRefCon+2
                jsr Deref
                sta 0
                stx 2
                ldy #oFlag
                lda [0],y
                beq UsePaint


                ldy #oFontID+2
                lda [0],y
                tax
                dey
                dey
                lda [0],y
                jsl ShowFont

                bra AllDone

UsePaint        anop
                ldy #oHandle+2              ; get handle to pic data
                lda [0],y
                tax
                dey
                dey
                lda [0],y
                jsl PaintIt


AllDone         lda TheRefCon
                ldx TheRefCon+2
                jsr Unlock

                rts


theRefCon       ds 4

WindowToPrint   ENTRY
                ds 4


                END
```

IO.ASM (pictures and files)

```
*****************************************************************************
*                                                                           *
*        HodgePodge:  An example Apple IIGS Desktop application             *
*                                                                           *
*                                                                           *
*            Copyright (c) 1986-87 by Apple Computer, Inc.                  *
*                          All Rights Reserved                              *
*                                                                           *
*                                                                           *
*   -----------------------------------------------------------------------  *
*                                                                           *
* ASM65816 Code file "IO.ASM" -- Picture Load and Save stuff calling ProDOS  *
*                                                                           *
*****************************************************************************


****************************************************************
*
* LoadOne
*
* Loads the picture whose path name is passed in
*
*    NamePtr
*
* to address passed in
*
*    PicDestIN
*
****************************************************************
LoadOne       START
              using IOData

              _OPEN OpenParams
              bcc cont1
              jmp Error1

cont1         anop
              lda OpenID
              sta ReadID
              sta CloseID

              _READ ReadParams
              bcc cont2
              jmp Error1

cont2         anop
              _Close CloseParams

              clc
              rts
              end
```

```
****************************************************************
*
* SaveOne
*
* Saves the picture whose path name 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 CreateParms
                bcc cont0
                jmp Error1

Cont0           _OPEN OpenParams
                bcc cont1
                jmp Error1

cont1           anop
                lda OpenID
                sta WriteID
                sta CloseID

                _WRITE WriteParams
                bcc cont2
                jmp Error1

cont2           anop
                _Close CloseParams

                clc
                rts
                end

****************************************************************
*
* Error1 -- handle disk error during read or write
*
****************************************************************
Error1          START
                using  IOData
                pha
                _Close CloseParams
                pla
                jsr    CheckDiskError
                rts

                END
```

GLOBALS.ASM (global data)

```
***********************************************************************
*                                                                     *
*       HodgePodge:  An example Apple IIGS Desktop application        *
*                                                                     *
*                                                                     *
*           Copyright (c) 1986-87 by Apple Computer, Inc.             *
*                        All Rights Reserved                          *
*                                                                     *
*                                                                     *
*   -----------------------------------------------------------------  *
*                                                                     *
*  ASM65816 Code file "GLOBALS.ASM" -- Global variables and misc. routines  *
*                                                                     *
***********************************************************************



**************************************************************
*
* GlobalDATA
*
**************************************************************
GlobalData      DATA


Prompt          dc i1'19',c'Load which Picture:'
Prompt2         dc i1'19',c'Save which Picture:'
Wxoffset        dc i'20'     ; offset for upperleft window corner
Wyoffset        dc i'12'     ; offset for upperleft window corner

nullRect        dc i'0,0,0,0'

reply           anop                         ;SF GET/PUT FILE record
r_good          dc i2'0'
r_type          dc i2'0'
r_auxtyp        dc i2'0'
r_fname         ds 16
r_fullpn        ds 128

QuitParams      dc i4'0'
                dc i'$4000'                  ; am restartable in memory


; ToolTable       dc i'11'
;               dc i'4,$0101'                ; quickdraw
;               dc i'5,$0100'                ; desk manager
;               dc i'6,$0100'                ; event manager
;               dc i'14,$0103'               ; window manager from disk!
;               dc i'15,$0103'               ; menu manager from Disk!
;               dc i'16,$0103'               ; control manager form disk!
;               dc i'20,$0100'               ; line edit
;               dc i'21,$0100'               ; dialog manager from disk!
;               dc i'23,$0100'               ; standard files from disk!
;               dc i'27,$0100'               ; Font manager
;               dc i'28,$0000'               ; List manager

; ThisMode        dc i'$0080'                  ;init mode: 640


SrcLocInfo      dc i'$80'                    ;PPtoPort 640 parms
PicPtr          ds 4
```

```
                dc i'160'
                dc i'0,0,200,640'

SrcRect         dc i'0,0,200,640'

EventRecord     anop
EventWhat       ds 2
EventMessage    ds 4
EventWhen       ds 4
EventWhere      ds 4
EventModifiers  ds 2

TaskDATA        ds 4
TaskMask        dc i4'$0FFF'

QuitFlag        ds 2
DialogPtr       ds 4
Windex          ds 2                    ;Index to next avail.window ID
LastWind        gequ 15                 ;Maximum number of windows open
MyZP            ds 2
; ZpHandle        ds 4
; MyID            ds 2

Vindex          ds 2                    ;index used to list of what WAS visible
Vtable          ds 16*4                 ;list of what WAS vis. when Hiding all
WindowList      ds 16*4                 ;all windows handle go into this list
WhichWindow     ds 4                    ;will contain window pointer, cur. window
TempHandle      ds 4                    ;some temp handles
Temp2Handle     ds 4
PicHandle       ds 4                    ; handle to picture data
SavePort        ds 4                    ;Save current port env for ShowPlWait
SaveType        ds 2
ActivateFlag    ds 2                    ;flag for check front window.
NeedToUpdate    ds 2                    ;used to prevent multi menu redraws
ThisWType       ds 2
LastWtype       ds 2
PrintAvail      dc i'0'                 ;Make sure this starts as 0

PrintRecord     ds 4                    ; handle to print record
PrintPort       ds 4                    ; pointer to printing GrafPort.

VolNotFound     gequ $45                ; prodos error

;------------------------------------------------------------
;
; MyWindowInfo
;
; This is the data structure used for the windows we
; allocate.
;
MaxNameSize     equ 29                  ; largest name we allow

oHandle         equ 0
oBlank          equ oHandle+4
oLength         equ oBlank+1
oName           equ oLength+1
oMMStuff        equ oName+MaxNameSize
oFlag           equ oMMStuff+6
oExtra          equ oFlag+1
;
oFontID         equ oHandle             ; if the type is for font,
;                                       ; the first field is a FontID
;                                       ; rather than the handle to picture
;                                       ; data.

MyWinfoSize     equ oExtra+4

                END
```

```
*****************************************************************
*
* IOData
*
*****************************************************************
IOData      DATA

CreateParms anop
NameC    dc    i4'0'
         dc    i2'$00C3'                ; DRNWR
CType    dc    i2'$0006'                ; BIN
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

PReadParams    anop                     ; for reading a packed file
PReadID        ds 2
PReadLoc       ds 4
PReadSize      ds 4                     ; this many bytes
               ds 4                     ; how many xfered

MarkParams     anop
MarkID         ds 2
CurrentMark    anop
Mark           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
```

```
****************************************************************
*
* Ignore
*
* Does not do a whole lot.
*
****************************************************************
Ignore          START
                rts
                END

****************************************************************
*
* Deref
*
* Derefs and locks the handle passed in a,x.  Result passed back
* in a,x. Trashes 0 on zp.
*
****************************************************************
Deref           START
                sta 0
                stx 2
                ldy #4
                lda [0],y
                ora #$8000
                sta [0],y
                dey
                dey
                lda [0],y
                tax
                lda [0]
                rts

                END

****************************************************************
*
* Unlock
*
* Unlocks the handle passed in x and a.  0 is trashed on zp.
*
****************************************************************
Unlock          START

                sta 0
                stx 2
                ldy #4
                lda [0],y
                and #$7FFF
                sta [0],y
                rts

...............END                                    END
```

Appendix F

HodgePodge Source Code: C

HP.CC (main program)

```
/*********************************************************************
*                                                                    *
*      HodgePodge:  An example Apple IIGS Desktop application       *
*                                                                    *
*            Written by the Apple IIGS Development Team             *
*                                                                    *
*      C Versionn 4.0                                               *
*                                                                    *
*           Copyright (c) 1986-87 by Apple Computer, Inc.           *
*                       All Rights Reserved                         *
*                                                                    *
*  ----------------------------------------------------------------  *
*                                                                    *
*     This program and its derivatives are licensed only for        *
*     use on Apple computers.                                       *
*                                                                    *
*     Works based on this program must contain and                  *
*     conspicuously display this notice.                            *
*                                                                    *
*     This software is provided for your evaluation and to          *
*     assist you in developing software for the Apple IIGS          *
*     computer.                                                     *
*                                                                    *
*     This is not a distribution license. Distribution of           *
*     this and other Apple software requires a separate             *
*     license. Contact the Software Licensing Department of         *
*     Apple Computer, Inc. for details.                             *
*                                                                    *
*     DISCLAIMER OF WARRANTY                                        *
*                                                                    *
*     THE SOFTWARE IS PROVIDED "AS IS" WITHOUT                      *
*     WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED,              *
*     WITH RESPECT TO ITS MERCHANTABILITY OR ITS FITNESS            *
*     FOR ANY PARTICULAR PURPOSE.  THE ENTIRE RISK AS TO            *
*     THE QUALITY AND PERFORMANCE OF THE SOFTWARE IS WITH           *
*     YOU.  SHOULD THE SOFTWARE PROVE DEFECTIVE, YOU (AND           *
*     NOT APPLE OR AN APPLE AUTHORIZED REPRESENTATIVE)              *
*     ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING,            *
*     REPAIR OR CORRECTION.                                         *
*                                                                    *
*     Apple does not warrant that the functions                     *
*     contained in the Software will meet your requirements         *
*     or that the operation of the Software will be                 *
*     uninterrupted or error free or that defects in the            *
*     Software will be corrected.                                   *
*                                                                    *
*     SOME STATES DO NOT ALLOW THE EXCLUSION                        *
*     OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY             *
*     NOT APPLY TO YOU.  THIS WARRANTY GIVES YOU SPECIFIC           *
*     LEGAL RIGHTS AND YOU MAY ALSO HAVE OTHER RIGHTS               *
*     WHICH VARY FROM STATE TO STATE.                               *
*                                                                    *
*  ----------------------------------------------------------------  *
*                                                                    *
*        Source file HP.CC -- Startup and Shutdown routines         *
*                                                                    *
*********************************************************************/

#include <types.h>
#include <prodos.h>
#include <misctool.h>
#include <quickdraw.h>
```

```
#include <qdaux.h>
#include <window.h>
#include <memory.h>
#include <dialog.h>
#include <menu.h>
#include <control.h>
#include <desk.h>
#include <event.h>
#include <lineedit.h>
#include <misctool.h>
#include <locator.h>
#include <stdfile.h>
#include <print.h>
#include <font.h>
#include <intmath.h>
#include <list.h>
#include <scrap.h>
#include "hp.h"

extern int _toolErr;

boolean ToolsFound     = FALSE,                /* Do we have tools ? */
       AllOk           = FALSE,
       PManagerFound   = TRUE;                 /* assume the PM is there */

int MyID;
int ThisMode = 0x80;                           /* init mode = 640 */

int ToolTable [] = {14,                 /* Number of items */
                  4,  0x100,          /* QuickDraw II */
                  5,  0x100,          /* Desk Manager */
                  6,  0x100,          /* Event Manager */
                  14, 0x100,          /* Window Manager */
                  15, 0x100,          /* Menu Manager */
                  16, 0x100,          /* Control Manager */
                  18, 0x100,          /* QuickDraw Auxiliary */
                  19, 0x000,          /* Print Manager */
                  20, 0x100,          /* Line Edit */
                  21, 0x100,          /* Dialog Manager */
                  22, 0x100,          /* Scrap Manager */
                  23, 0x100,          /* Standard File  */
                  27, 0x100,          /* Font Manager */
                  28, 0x000};         /* List Manager */



char **y,*z;


GrafPortPtr OrigPort;




/* This is the routine that will do the initialization of tools, will allocate
  memory and all related tasks
*/



boolean StartUpTools ()

{
  static char     SysToolsDirStr [] = "\p*/SYSTEM/TOOLS";
  static FileRec  ParamBlock = { SysToolsDirStr , NULL };

  TLStartUp ();                                /* for calling tools */
```

```
   CheckToolError (1);

   MyID = MMStartUp();                          /* ID for all transactions */
   CheckToolError (2);

   MTStartUp();                                 /* Misc. Tools  */
   CheckToolError (3);                          /* Make sure all is OK      */

   y = NewHandle (0xB00L,                       /* Eleven pages         */
                  MyID,                         /* put it to my name    */
                  attrBank +
                  attrPage +
                  attrFixed +
                  attrLocked,
                  0L);                          /* don't care           */
   CheckToolError (4);

   z = *y;                                      /* deref handle     */

   QDStartUp ((int) z,ThisMode,MAXSCAN,MyID);
   CheckToolError (5);
   OrigPort = GetPort ();

   EMStartUp((int) z + 0x300,20,0,640,0,200,MyID);  /* Event Manager */
   CheckToolError (6);

   MoveTo       (20,20);
   SetBackColor (0);
   SetForeColor (15);
   DrawString   ("\pOne Moment Please... ");
   ShowCursor   ();

TryAgain:
   GET_FILE_INFO (&ParamBlock);
   if (_toolErr)
       if (MountBootDisk () == 1)
           goto TryAgain;
       else
           return (false);              /* Exit function unsuccessfully */

   LoadTools (ToolTable);               /* Now it's ok to do this */
   CheckToolError (7);

   QDAuxStartUp ();
   CheckToolError (8);

   WaitCursor ();                       /* Show wristwatch cursor */

   WindStartUp (MyID);
   CheckToolError (9);

   RefreshDesktop (NULL);

   CtlStartUp (MyID,(int) z + 0x400);
   CheckToolError (10);

   LEStartUp (MyID, (int) z + 0x500);
   CheckToolError (11);

   DialogStartUp (MyID);
   CheckToolError (12);

   MenuStartUp (MyID, (int) z + 0x600);
   CheckToolError (13);

   DeskStartUp();                               /* All we need is init'ed now */
   CheckToolError (14);
```

```
   ShowPleaseWait ();

   SFStartUp(MyID, (int) z + 0x700);
   CheckToolError (15);
   SFAllCaps (true);

   FMStartUp(MyID, (int) z + 0x800);          /* the watch cursor is up */
   CheckToolError (16);                       /* while we count the fonts */

   ListStartup ();        /* >!< Note, not ListStartUp with upper case "U"! */
   CheckToolError (17);

   ScrapStartUp ();
   CheckToolError (18);

   PMStartUp (MyID, (int) z + 0x900);
   CheckToolError (19);

   HidePleaseWait ();      /* Remove dialog box */
   InitCursor ();          /* Show arrow cursor */

   return (true);          /* Exit function successfully */
}


ShutDownTools ()

{
   DeskShutDown ();

   if (WindStatus () != 0)
      HideAllWindows ();

   ListShutDown   ();
   FMShutDown     ();
   ScrapShutDown  ();
   PMShutDown     ();
   SFShutDown     ();
   MenuShutDown   ();
   DialogShutDown ();
   LEShutDown     ();
   CtlShutDown    ();
   WindShutDown   ();
   QDAuxShutDown  ();
   EMShutDown     ();
   QDShutDown     ();
   MTShutDown     ();

   if (MMStatus () != 0)
   {
      DisposeHandle (y);
      MMShutDown    (MyID);
   }
   TLShutDown ();
}


/* MAIN program routine */

main ()
{
   if (StartUpTools ())                /* Try to initialize tools          */
   {
      SetUpMenus ();
      MainEvent  ();
   }
   ShutDownTools ();                   /* Shutdown tools even if didn't run */
}
```

MENU.CC (menus)

```
/**********************************************************************
*                                                                    *
*      HodgePodge:  An example Apple IIGS Desktop application        *
*                                                                    *
*                                                                    *
*                                                                    *
*                                                                    *
*                                                                    *
*           Copyright  (c) 1986-87 by Apple Computer, Inc.           *
*                        All Rights Reserved                         *
*                                                                    *
*  ---------------------------------------------------------------   *
*                                                                    *
*   Source file MENU.CC -- Menu bar inserting/deleting / vectoring   *
*                                                                    *
**********************************************************************/

#include <types.h>
#include <menu.h>
#include <desk.h>
#include <window.h>
#include <memory.h>
#include <intmath.h>
#include <misctool.h>
#include <texttool.h>
#include "hp.h"

/* Bunch of routines defined somewhere else */
extern DoCloseItem();
extern DoAboutItem();
extern DoQuitItem();
extern DoOpenItem();
extern DoSaveItem();
extern DoChooserItem();
extern DoSetUpItem();
extern DoPrintItem();
extern DoChangeRes();
extern DoOpenItem();
extern DoSetMono();
extern DoShowVers();

extern WmTaskRec TheEvent;
extern GrafPortPtr WhichWindow;
extern GrafPortPtr WindowList[16];
extern int Windex;

char IDStr[8] = "\\N300\r";
extern char str[];


/* Here we have all defines for all the menus */


char *Menus[] = {
/* Fonts menu */
">>  Fonts  \\N6\r\
==Display Font ...\\*FfN264\r\
==Display Font as Mono-spaced\\*MmN265\r.",  /* compiler adds '0' at the end */

/* Windows menu */
">>  Window  \\DN5\r\
==No WIndows Allocated\\N299\r.",
```

```
/* Edit Menu */
">>  Edit  \\DN3\r\
==Undo\\*ZzN250\r\
==-\\N298D\r\
==Cut\\*XxN251\r\
==Copy\\*CcN252\r\
==Paste\\*VvN253\r\
==Clear\\N254\r.",

/* File Menu */
">>  File  \\N2\r\
==Open ...\\*OoN258\r\
==Close\\DN255\r\
==Save As ...\\DN259\r\
==-\\N298D\r\
==Choose Printer...\\N260\r\
==Page Setup ...\\DN261\r\
==Print ...\\D*PpN262\r\
==-\\N298D\r\
==Quit\\*QqN257\r.",

/* Apple Menu */
">>@\\XN1\r\
==About HodgePodge...\\N256\r\
==-\\N298D."};


AddToMenu()
{
DataRecPtr dereftemp;
DataRecHandle TempHandle;
int index,i;

   WhichWindow = FrontWindow();
   TempHandle = (DataRecHandle) GetWRefCon(WhichWindow);

   dereftemp = *TempHandle;
   HLock(TempHandle);

   Int2Dec(Windex,IDStr + 3,2,0);
   IDStr[3] |= 0x30;
   IDStr[4] |= 0x30;

   index = (dereftemp -> Str[0]) + 1;
   for (i=0;i <=6;i++)
     dereftemp -> Str[i + index] = IDStr[i];


   InsertMItem(&(dereftemp -> Blank),0xffff,WindowsMenuID);

   if (!(Windex))                                  /* this is the first window */
     {
       DeleteMItem(299);                           /* Token item   */
       SetMenuFlag(0xff7f,WindowsMenuID);          /* highlight the menu   */
       DrawMenuBar();
     }
   CalcMenuSize(0L,WindowsMenuID);
   WindowList[Windex] = WhichWindow;
   Windex++;
   HUnlock(TempHandle);
}

DoWItem()
{
  WhichWindow=WindowList[(TheEvent.wmTaskData&0xffff) - 300 ];
  DoWindow();
  HiliteMenu(FALSE, TheEvent.wmTaskData/0xffff);
```

```
}

DoMenu()
{

        switch (TheEvent.wmTaskData & 0xffff)
          {
            case UndoID        : break;    /* we do nothing with */
            case CutID         : break;
            case CopyID        : break;
            case PasteID       : break;
            case ClearID       : break;            /* these !     */
            case CloseWID      : DoCloseItem();
                                 break;
            case AboutID       : DoAboutItem();
                                 break;
            case QuitID        : DoQuitItem();
                                 break;
            case OpenWID       : DoOpenItem();
                                 break;
            case SaveID        : DoSaveItem();
                                 break;
            case ChooseID      : DoChooserItem();
                                 break;
            case SetUpID       : DoSetUpItem();
                                 break;
            case PrintID       : DoPrintItem();
                                 break;
            case ShowFontID    : DoOpenItem();
                                 break;
            case MonoID        : DoSetMono();
                                 break;
            case 299           : break;
            default            : DoWItem();
          }

        HiliteMenu(FALSE, TheEvent.wmTaskData/0xffff);
}


SetUpMenus ()
{

   int MenuLooper;

   SetMTitleStart (10);                        /* Set starting pos of menus */

   for (MenuLooper = 0;   MenuLooper < NUM_MENUS;   MenuLooper++)
       InsertMenu (NewMenu (Menus [MenuLooper]),0);

   FixAppleMenu (AppleMenuID);       /* Add NDA's, if any */
   FixMenuBar   ();                  /* Set sizes of menus */
   DrawMenuBar  ();                  /* Draw the menu bar on the screen */
}
```

EVENT.CC (main event loop)

```
/*************************************************************************
*                                                                        *
*     HodgePodge:  An example Apple IIGS Desktop application             *
*                                                                        *
*                                                                        *
*                                                                        *
*                                                                        *
*                                                                        *
*          Copyright (c) 1986-87 by Apple Computer, Inc.                 *
*                      All Rights Reserved                               *
*                                                                        *
*  ---------------------------------------------------------------      *
*                                                                        *
*   Source file EVENT.CC -- Main event loop and window activation        *
*                                                                        *
*************************************************************************/

#include <types.h>
#include <memory.h>
#include <window.h>
#include <prodos.h>
#include <misctool.h>
#include <texttool.h>
#include <menu.h>
#include "hp.h"

extern int _toolErr;
extern PManagerFound;

int QuitFlag = 0;
GrafPortPtr LastWindow = NIL,
        ThisWindow = NIL;

int ActivateFlag;


struct HandleRec {
                  char *ptrpart;
                  int flags;
                };
/* for Open

typedef struct OpenRec {
    Word     openRefNum;
    Ptr      openPathname;
    Handle   iOBuffer;
    } OpenRec, *OpenRecPtr, **OpenRecHndl;
 */

OpenRec MyOpenParams = {0,0,0L};

/* for Read, Write, Close, Flush

typedef struct FileIORec {
    Word     fileRefNum;
    Ptr      dataBuffer;
    Longint  requestCount;
    Longint  transferCount;
    } FileIORec, *FileIORecPtr, **FileIORecHndl;  */
```

```
FileIORec ReadParams = {
                                  0,                          /* fileRefNum       */
                                  0L,                         /* dataBuffer       */
                                  0x8000L,                     /* requestCount     */
                                  0L};                        /* transferCount    */

FileIORec WriteParams = {
                                  0,                          /* fileRefNum       */
                                  0L,                         /* dataBuffer       */
                                  0x8000L,                     /* requestCount     */
                                  0L};                        /* transferCount    */

FileIORec CloseParams;                                        /* most remains unused */

/* for Create, SetFileInfo, GetFileInfo

typedef struct FileRec {
      Ptr       pathname;
      Word      fAccess;
      Word      fileType;
      Longint   auxType;
      Word      storageType;
      Word      createDate;
      Word      createTime;
      Word      modDate;
      Word      modTime;
      Longint   blocksUsed;
    } FileRec, *FileRecPtr, **FileRecHndl;      */

FileRec CreateParams = {
                                0L,
                                0x00c3,
                                0x0006,
                                0L,
                                1,
                                0,0,
                                0,0,
                                0L};

/* for Destroy, ChangePath, ClearBackupBit, GetPathname, GetBootVol

typedef struct PathNameRec {
      Ptr       pathname;
      Ptr       newPathname;
    } PathNameRec, *PathNameRecPtr, **PathNameRecHndl;                */

PathNameRec DestParams = {0L,0L};

WmTaskRec TheEvent;

MainEvent()
{
int MyEvent;

   TheEvent.wmTaskMask = 0x00000fff;  /* initialize mask */

   do
      {
        do
           {
             ActivateFlag = 0;
             CheckFrontW ();
             MyEvent = TaskMaster(0xFFFF, &TheEvent);
           }
        while(!MyEvent);
        switch (MyEvent)
        {
```

```
      case  activateEvt : DoActivate ();
                          break;

      case 17:                        /* in menu */

              DoMenu();
              break;

      case 22:                          /* in goaway */

              DoCloseItem();
              break;

      case 25:                          /* in special menu item */

             DoMenu();
             break;
      }
    }
  while (!(QuitFlag));
}

DoQuitItem()
{
  QuitFlag = 0x8000;                          /* simple uh? */
}

/* Check if the front window has changed and react accordingly */

CheckFrontW()
{

DataRecHandle TempDataHand;
DataRecPtr TempDataPtr;

  ThisWindow = FrontWindow();
  if (!(ThisWindow == LastWindow))
    {
     if (LastWindow = ThisWindow)             /* at least one window */
       {
         if (!(GetSysWFlag(ThisWindow)))
           {
             SetUpForAppW();
             if (ActivateFlag)
               TempDataHand = (DataRecHandle)GetWRefCon(TheEvent.wmTaskData);
             else
               TempDataHand = (DataRecHandle)GetWRefCon(ThisWindow);
             TempDataPtr = *TempDataHand;
             HLock(TempDataHand);
             if (TempDataPtr -> Flag)
               DisableMItem (SaveID);
             else
               EnableMItem  (SaveID);
             HUnlock(TempDataHand);
           }
         else
           SetUpForDaW();
       }
     else
       DisableAll();
    }
}

DoActivate()
{
  if (TheEvent.wmModifiers & 1)
    {
      ActivateFlag = 1;
```

```
        CheckFrontW ();
      }
}


/* Disable items not applicable */

DisableAll ()
{
   SetMenuFlag  (0x0080,EditMenuID);                /* disable */
   DrawMenuBar  ();
   DisableItems ();
}

SetUpForAppW ()
{
   SetMenuFlag (0x0080,EditMenuID);
   DrawMenuBar ();
   EnableItems ();
}

SetUpForDaW()
{
   DisableItems();
   EnableMItem(CloseWID);

   SetMenuFlag(0xff7f,EditMenuID);
   DrawMenuBar();
}

EnableItems()
{
   EnableMItem(SaveID);
   EnableMItem(CloseWID);
   if (PManagerFound)
      {
        EnableMItem(PrintID);                    /* don't enable if printing */
        EnableMItem(SetUpID);                    /* is out of the question!  */
      }
}

DisableItems()
{
   DisableMItem(SaveID);
   DisableMItem(CloseWID);
   DisableMItem(PrintID);                        /* who cares!? */
   DisableMItem(SetUpID);
}

/* Now some I/O stuff, this file is just Ok for it */

boolean LoadOne()
{
   OPEN(&MyOpenParams);
   if (_toolErr)
      {
        CheckDiskError (1);
        return(FALSE);                           /* couldn't open */
      }
   else
      {
        ReadParams.fileRefNum = MyOpenParams.openRefNum;
        CloseParams.fileRefNum = MyOpenParams.openRefNum;
        READ(&ReadParams);
        if (_toolErr)
          {
            CLOSE(&CloseParams);
            CheckDiskError(2);
```

```
            return(FALSE);
          }
        else
          {
            CLOSE(&CloseParams);
            return(TRUE);
          }
      }
}

SaveOne()
{
   CreateParams.pathname = MyOpenParams.openPathname;
   DestParams.pathname = MyOpenParams.openPathname;
   CloseParams.fileRefNum = MyOpenParams.openRefNum;
   CreateParams.fileType = 0xc1;
   CreateParams.auxType = 0;

   DESTROY(&DestParams);

   CREATE(&CreateParams);
   if (_toolErr)
     {
       CLOSE(&CloseParams);
       CheckDiskError(3);
     }
   else
     {
       OPEN(&MyOpenParams);
       if (_toolErr)
         {
           CLOSE(&CloseParams);
           CheckDiskError(4);
         }
       else
         {
           WriteParams.fileRefNum = MyOpenParams.openRefNum;
           WRITE(&WriteParams);
           if (_toolErr)
             {
               CLOSE(&CloseParams);
               CheckDiskError(5);
             }
           else
             CLOSE(&CloseParams);
         }
     }
}
```

WINDOW.CC (windows)

```
/*************************************************************************
*                                                                        *
*      HodgePodge:  An example Apple IIGS Desktop application           *
*                                                                        *
*                                                                        *
*                                                                        *
*                                                                        *
*                                                                        *
*           Copyright (c) 1986-87 by Apple Computer, Inc.               *
*                        All Rights Reserved                            *
*                                                                        *
*  -------------------------------------------------------------------  *
*                                                                        *
*          Source file WINDOW.CC -- Window opening / closing            *
*                                                                        *
*************************************************************************/

#include <types.h>
#include <quickdraw.h>
#include <window.h>
#include <stdfile.h>
#include <prodos.h>
#include <memory.h>
#include <qdaux.h>
#include <font.h>
#include <menu.h>
#include <desk.h>
#include <misctool.h>
#include <texttool.h>
#include <intmath.h>
#include "hp.h"

extern GrafPortPtr OrigPort;

extern char str[];
/* stuff to define the window data structure, defined in HP.H
typedef struct DataRec {
                    handle PicHand;
                    char Blank;
                    char Str[30];
                    char MMStuff[6];
                    Byte Flag;
                    char Extra;
                    } DataRec, *DataRecPtr, **DataRecHandle;
*/

DataRecHandle MyDataHandle;
DataRecPtr RefPtr;

/* This structure is defined in window.h
typedef struct WmTaskRec {
     Word         wmWhat;
     DblWord      wmMessage;
     DblWord      wmWhen;
     Point        wmWhere;
     Word         wmModifiers;
     DblWord      wmTaskData;
     DblWord      wmTaskMask;
     } WmTaskRec, *WmTaskRecPtr, **WmTaskRecHndl;   */

extern WmTaskRec TheEvent;
extern char *LineTable[];
```

```
extern int _toolErr;
extern int MyID;
extern int ThisMode;

extern GetPutTemplate SFP640Temp;                    /* templates for StdFile */

char origitem[] = "==No Windows Allocated\\N299\r"; /* first item in windows */

extern int MonoFlag;                            /* see Font.c   */
extern FontID DesiredFont;                        /*  "   "   "   */

extern int Paint();                                    /* window content proc */
extern int DispFontWindow();                    /* Window def Proc for fonts */
pascal int OpenFilter();

/* typedef struct ParamList {
    Integer     paramLength;
    Word        wFrameBits;
    Ptr         wTitle;
    long        wRefCon;
    Rect        wZoom;
    Ptr         wColor;
    Integer     wYOrigin;
    Integer     wXOrigin;
    Integer     wDataH;
    Integer     wDataW;
    Integer     wMaxH;
    Integer     wMaxW;
    Integer     wScrollVer;
    Integer     wScrollHor;
    Integer     wPageVer;
    Integer     wPageHor;
    DblWord     wInfoRefCon;
    Integer     wInfoHeight;
    Ptr         wFrameDefProc;
    Ptr         wInfoDefProc;
    Ptr         wContDefProc;
    Rect        wPosition;
    WPortPtr    wPlane;
    WindRecPtr  wStorage;
    } ParamList, *ParamListPtr, **ParamListHndl;  */

ParamList MyWindow = {
                      sizeof(MyWindow),       /* Record size      */
                      0xdda0,                 /* Frame   dda0     */
                      0L,                     /* Ptr to title     */
                      0L,                     /* RefCon           */
                      0,0,0,0,                /* Full size ( 0 --> default)*/
                      0L,                     /* Color Table Ptr  */
                      0,                      /* Vertical Origin  */
                      0,                      /* Horizontal Origin    */
                      200,                    /* Data area hight  */
                      640,                    /* Data area width  */
                      200,                    /* Max cont height  */
                      640,                    /* max cont width   */
                      4,                      /* pixels to scroll vert    */
                      16,                     /* pixels to scroll horz    */
                      40,                     /* pixels to page vert      */
                      160,                    /* pixels to page horz      */
                      0L,                     /* info bar string      */
                      0,                      /* info bar height      */
                      01,                     /* def proc ptr         */
                      0L,                    /* info bar def proc */
                      Paint,                  /* Content def proc     */
                      0,0,0,0,                /* size/pos of content  */
                      -1L,                    /* plane of window      */
                      0};                     /* Wind Rec add         */
```

```
extern FileIORec ReadParams;
extern FileIORec WriteParams;

Rect ISizPos = {20,10,80,350};

/*
typedef struct FontInfoRecord {
     integer     ascent;
     integer     descent;
     integer     widMax;
     integer     leading;
    } FontInfoRecord, *FontInfoRecPtr, **FontInfoRecHndl;
*/

FontInfoRecord FIRecord;

SFReplyRec MyReply = {0,0,0," "," "};

extern OpenRec MyOpenParams;                    /* OpenRec {
                                                     int openRefNum;
                                                     ptr openPathname;
                                                     long iOBuffer;
                                                     } */

char Prompt1 [] = "\pLoad which picture:";
char Prompt2 [] = "\pSave which picture:";

int Wxoffset = 20;
int Wyoffset = 12;

handle PicHandle;                                /* handle to picture data */

extern boolean OpenWindow();                     /* to add window to menu */
extern boolean AskUser();
extern boolean LoadItUp();
extern boolean DoTheOpen();

GrafPortPtr WhichWindow;                              /* current window handle  */

GrafPortPtr WindowList[16];                           /* list of window handles */

int vIndex;                                      /* index to:    |         */
                                                 /*             \ /        */
GrafPortPtr vTable[16];                             /* list of what was visible */

int Windex = 0;                                 /* index to next avail wind id*/


LocInfo SrcInfo640 = {
                           0x80,
                           /* used to be byte here */
                           0L,
                           160,
                           {0,0,200,640}
                     };

Rect SrcRect640    = {0,0,200,640};



/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */
/* Now the real stuff                                               */

/* Procedure to Close windows, we close them from the back.
   Things move faster this way.
 */
```

```
HideAllWindows()
{
  vIndex = 0;                                  /* init index */
  if (vTable[vIndex] = FrontWindow())          /*at least one window */
    {
      for (vIndex;vTable[vIndex + 1] = GetNextWindow(vTable[vIndex]);vIndex++);
      for (vIndex;vIndex >= 0;vIndex--)
        HideWindow(vTable[vIndex]);
    }
}



/* DoOpenItem:
  1) Make sure that there aren't too many windows open already;
  2) Call OpenWindow to let the user see it; if successful,
  3) Call AddToMenu to add the name to the windows menu list.
*/

DoOpenItem()
{
  if (Windex == NUM_WINDOWS)
     ManyWindDialog ();
  else
     if (OpenWindow ())
         AddToMenu ();
}



/* OpenWindow:
  1) Calls SFGetFile to get name of file to display in window
     (or the dialog to select font if needed)
  2) Gets memory for, and loads the picture/font data into memory
  3) Allocates a new window
      a)puts handle to MyWindowInfo in WrefCon
      b) note that wContDefProc is set to "Paint"
      c) for fonts wContDefProc is set to "DispFontWindow"

  The definition of MyWindowInfo is global data.
*/



boolean OpenWindow()
{
  if ((TheEvent.wmTaskData & 0xFFFF) == ShowFontID)
  {
     if (DoChooseFont())
       if (DoTheOpen ())
         return (TRUE);
       else
         return (FALSE);
     else
       return (FALSE);
  }
  else
  {
    if (AskUser())
      return(TRUE);
    else
      return(FALSE);
  }
}
```

```
/*
typedef struct SFReplyRec  {
      Boolean     good;
      Word     fileType;
      Word     auxFileType;
      char     filename[16];
      char     fullPathname[129];
      } SFReplyRec,  *SFReplyRecPtr ;
*/

boolean AskUser()
{
   SFGetFile(20,20,Prompt1,OpenFilter,0L,&MyReply);
   if (MyReply.good)
     if (LoadItUp())
       return(TRUE);
     else
       return(FALSE);
   else
       return(FALSE);
}

boolean LoadItUp()
{
   WaitCursor();

   PicHandle = NewHandle(0x8000L,MyID,0,0L);
   if (_toolErr)
     return(FALSE);
   else
     {
       ReadParams.dataBuffer = *PicHandle;
       HLock(PicHandle);
       if (DoTheOpen())
         return(TRUE);
       else
         return(FALSE);
     }
}

boolean DoTheOpen()
{
int aux1,aux2;
ptr aux;

boolean IOError = FALSE;

int i;
                                        /* there is always a need */
long FIDAux;

  MyDataHandle = (DataRecHandle)NewHandle((long)(sizeof(DataRec)) ,
                                        MyID,0xc000,0L);
  MyWindow.wRefCon = (long)MyDataHandle;

  if (_toolErr)
    return(FALSE);
  else
    {
      RefPtr = *MyDataHandle;
      HLock(MyWindow.wRefCon);

/* The assumption is that the window is for a picture (not a font) */

      MyWindow.wContDefProc = (VoidProcPtr) Paint;
      RefPtr -> Flag = 0;             /* picture flag */
```

```
if ((TheEvent.wmTaskData & 0xffff) == ShowFontID)  /* were we right? */
  {
   RefPtr -> Flag = 0x1 | MonoFlag;  /*   No! so change    */
   PicHandle = (handle) ((DesiredFont.famNum) +
                        (DesiredFont.fontSize  * 0x1000000) +
                        (DesiredFont.fontStyle * 0x10000));
                                /* everything to font   */
                                          /* display      */
   MyWindow.wContDefProc = (VoidProcPtr) DispFontWindow;

  }
else
  {
   MyOpenParams.openPathname = MyReply.fullPathname;

     if (!(LoadOne()))
       IOError = TRUE;
  }                                       /* end of picture stuff */
   if (IOError)
     {
      DisposeHandle(MyWindow.wRefCon);
      DisposeHandle(PicHandle);
      return(FALSE);
     }
   else
     {
      RefPtr -> PicHand = PicHandle;
      RefPtr -> Blank  =  0x20;
      MyWindow.wTitle = RefPtr -> Str;

      if (!(MyReply.filename[0] <= MaxNameSize))
        MyReply.filename[0] = MaxNameSize;
      for (i=MyReply.filename[0];i>=0;i--)
        RefPtr -> Str[i] = MyReply.filename[i];
        MyWindow.wDataW = 640;
        MyWindow.wMaxW = 640;
        ISizPos.h2 = 350;

      MyWindow.wDataH = 200;                /* in case is a picture */

      SetPort (OrigPort);

      if ((TheEvent.wmTaskData & 0xFFFF) == ShowFontID)
        {
         FIDAux = GetFontID();

         InstallFont(PicHandle,0);

         GetFontInfo(&FIRecord);

         MyWindow.wDataH =
              ((FIRecord.ascent + FIRecord.descent) * (NumLines + 1));

         FindMaxWidth();

         InstallFont(FIDAux,0);
        }
           /* windows have to offset evenly */
      MyWindow.wPosition.v1 = Wyoffset+ISizPos.v1;
      MyWindow.wPosition.h1 = Wxoffset+ISizPos.h1;
      MyWindow.wPosition.v2 = Wyoffset+ISizPos.v2;
      MyWindow.wPosition.h2 = Wxoffset+ISizPos.h2;

      Wxoffset += 20;

      if ((Wyoffset += 12) > 120)
        Wyoffset = 12;
```

```
            WhichWindow = NewWindow(&MyWindow);

            SetPort (OrigPort);

            HUnlock(PicHandle);

            SetOriginMask(0xFFFE,WhichWindow);

            InitCursor();

            return(TRUE);                     /* finally! */

          }
     }
}

void DoSaveItem()
{
DataRecHandle AuxHandle;

Pointer AuxPtr;

int i;

   WhichWindow = FrontWindow();

   AuxHandle = (DataRecHandle) GetWRefCon(WhichWindow);

   RefPtr = *AuxHandle;
   HLock(AuxHandle);


   if (!(RefPtr -> Flag))                      /* Save only type 0 windows */
     {
       MyOpenParams.openPathname = GetWTitle(WhichWindow);
       SFPutFile (20,20,Prompt2,MyOpenParams.openPathname,15,&MyReply);
       if (MyReply.good)                      /* <> 0 --> OK to save it */
         {
           WaitCursor();
           PicHandle = RefPtr -> PicHand;
           WriteParams.dataBuffer = *PicHandle;
           HLock(PicHandle);
           MyOpenParams.openPathname = MyReply.fullPathname;
           SaveOne();                         /* save the picture */
           for (i = MyReply.filename[0];i >= 0; i--)
              RefPtr -> Str[i] = MyReply.filename[i];
           SetWTitle(RefPtr -> Str,WhichWindow);
           HUnlock(PicHandle);
           CalcMenuSize(0L,WindowsMenuID);
           InitCursor();
         }
     }
}

/* This routine finds out how wide the window should be for the
   current font
*/

FindMaxWidth()

{
int tempFlags;
int LineCounter;
int aux;
```

```
   MyWindow.wDataW = 0;
   tempFlags = GetFontFlags();
   SetFontFlags((RefPtr -> Flag >> 1) & 1);
   for (LineCounter = 1;LineCounter < NumLines;LineCounter++)
     if ( (aux = StringWidth(LineTable[LineCounter])) > MyWindow.wDataW)
       MyWindow.wDataW = aux;
   MyWindow.wDataW += 10;
   SetFontFlags(tempFlags);                    /* put flags back */
}

/* Close a window and dispose of extra-data (in WRefCom)
   and remove it from window list
*/

DoCloseItem()
{
DataRecHandle tempHand2;

DataRecPtr tempPtr2;

int IDDelete;
int Counter;
int IDStart;
int IDNew;

   if (WhichWindow = FrontWindow())
     {
       CloseNDAByWinPtr(WhichWindow);   /* if it's a sys wind this is enough*/
       if (_toolErr)                   /* error means wasn't a system window */
         {
           tempHand2 = (DataRecHandle) GetWRefCon(WhichWindow);
           tempPtr2 = *tempHand2;               /* deref    */
           HLock(tempHand2);                   /* and lock it  */
           PicHandle = tempPtr2 -> PicHand; /* handle to get rid of*/

           if (tempPtr2 -> Flag)            /* ~0 -->  font */
             PicHandle = NIL;                  /* so, don't dispose    */
           IDDelete = AdjWind() + 300;          /* take it out of list  */
           if (Windex == 1)                     /* one wind is special case */
             {
               InsertMItem(origitem,0,WindowsMenuID); /* no windows message*/
               SetMenuFlag(0x0080,WindowsMenuID);      /* disable windows */
               DrawMenuBar();

               Wxoffset = 20;                  /* reset start */
               Wyoffset = 12;                  /* for window sizing */
             }
           DeleteMItem(IDDelete);              /* off the menu */
           Windex--;


           if (Counter = Windex)
             {
               IDStart = 300;
               IDNew = 300;                    /* starting point */

               while (Counter)
                 {
                   if (IDStart != IDDelete)
                     {
                       SetMItemID(IDNew,IDStart);
                       IDNew++;
                       Counter--;
                     }
                   IDStart++;
                 }
             }
```

```
                CalcMenuSize(0L,WindowsMenuID);
                DisposeHandle(tempHand2);
                if (PicHandle)
                  DisposeHandle(PicHandle);
                CloseWindow(WhichWindow);
            }
        }
    }

/* AdjWind() finds and deletes a window list item which matches
   "WhichWindow" and returns where it position was.
*/

AdjWind()
{
int IDCounter,i,y;

   i = Windex -1;
   IDCounter = i;

   while (!((WhichWindow == WindowList[i]) || i < 0))
     i--;
   y = i;

   while (!(y == IDCounter))
     {
       WindowList[y] = WindowList[ y + 1];
       y++;
     }
   return(i);
}

/* This procedure gets called when task master feels is time to
   draw the picture.
*/

Paint()
{
DataRecHandle auxHandle;
GrafPortPtr auxPtr;
DataRecPtr DataPtr;

   auxPtr = GetPort();                                   /* get current port */
   auxHandle = (DataRecHandle) GetWRefCon(auxPtr);            /* handle to data */
   DataPtr = *auxHandle;
   HLock(auxHandle);

   PaintIt(DataPtr -> PicHand);         /* (handle *) */

   HUnlock(auxHandle);
}

/* This is the routine that actually does the painting after it
   receives the handle to the picture.
*/

PaintIt(Painthand)
handle Painthand;

{
ptr auxPtr2;

   auxPtr2 = *Painthand;                          /* deref */
```

```
   HLock(Painthand);
   SrcInfo640.ptrToPixImage = auxPtr2;
   PPToPort(&SrcInfo640,&SrcRect640,0,0,0);
   HUnlock(Painthand);
}

DoGoAway()
{
  HideWindow(TheEvent.wmTaskData);
}

DoWindow()
{
  SelectWindow(WhichWindow);
  ShowWindow(WhichWindow);
}




pascal int OpenFilter (DirEntry)

ptr DirEntry;

/* Filter function called by the Standard File Operations' SFGetFile
   dialog to determine whether a filename should be dimmed or not. */

{
   if ((*(DirEntry + 0x10) & 0x00FF) == 0xC1)  /* Type $C1: picture file */
       return (2);                             /* ... so it's undimmed.  */
   else
       return (1);                             /* Else show it dimmed.   */
}
```

DIALOG.CC (dialog boxes)

```
/************************************************************************
 *                                                                      *
 *      HodgePodge:  An example Apple IIGS Desktop application          *
 *                                                                      *
 *                                                                      *
 *                                                                      *
 *                                                                      *
 *                                                                      *
 *           Copyright (c) 1986-87 by Apple Computer, Inc.              *
 *                       All Rights Reserved                            *
 *                                                                      *
 *  -----------------------------------------------------------------   *
 *                                                                      *
 *      Source file DIALOG.CC -- Dialogs and error trapping             *
 *                                                                      *
 ************************************************************************/

#include <types.h>
#include <quickdraw.h>
#include <qdaux.h>
#include <memory.h>
#include <dialog.h>
#include <prodos.h>
#include <texttool.h>
#include <stdfile.h>
#include <window.h>
#include <locator.h>
#include <intmath.h>
#include <misctool.h>
#include "hp.h"


extern int              _toolErr;
extern int              MyID;
extern GrafPortPtr      OrigPort;



GrafPortPtr             MsgWindPtr;



/* Data structure for "About HodgePodge..." dialog box: */

static char             OKStr [] = "\pOK";

Rect                    DRect = {20,190,192,450};

Rect                    AppleIconRect = {135,20,0,0};
```

```
char AppleIcon640[] = {0x00,0x00,0x00,0x00,0x22,0x00,0x40,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x0f,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xf0,
0x0f,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xf0,
0x0f,0x0f,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xf0,0xf0,
0x0f,0x0f,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xf8,0x8f,0xff,0xff,0xf0,0xf0,
0x0f,0x0f,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xf8,0x88,0x8f,0xff,0xff,0xf0,0xf0,
0x0f,0x0f,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0x88,0x88,0x8f,0xff,0xff,0xf0,0xf0,
0x0f,0x0f,0xff,0xff,0xff,0xff,0xff,0xff,0xf8,0x88,0x88,0xff,0xff,0xff,0xf0,0xf0,
0x0f,0x0f,0xff,0xff,0xff,0xff,0xff,0xff,0x88,0x88,0x88,0xff,0xff,0xff,0xf0,0xf0,
0x0f,0x0f,0xff,0xff,0xff,0xff,0xff,0xff,0x88,0x88,0x8f,0xff,0xff,0xff,0xf0,0xf0,
0x0f,0x0f,0xff,0xff,0xff,0xff,0xff,0xff,0x88,0x88,0xff,0xff,0xff,0xff,0xf0,0xf0,
0x0f,0x0f,0xff,0xff,0xf8,0x88,0x8f,0xff,0x88,0xff,0x88,0x88,0xff,0xff,0xf0,0xf0,
0x0f,0x0f,0xff,0xf8,0x88,0x88,0x88,0x8f,0xff,0x88,0x88,0x88,0x88,0xff,0xf0,0xf0,
0x0f,0x0f,0xff,0x88,0x88,0x88,0x88,0x88,0x88,0x88,0x88,0x88,0x88,0x8f,0xf0,0xf0,
0x0f,0x0f,0xfe,0xee,0xee,0xee,0xee,0xee,0xee,0xee,0xee,0xee,0xef,0xff,0xf0,0xf0,
0x0f,0x0f,0xfe,0xee,0xee,0xee,0xee,0xee,0xee,0xee,0xee,0xee,0xff,0xff,0xf0,0xf0,
0x0f,0x0f,0xfe,0xee,0xee,0xee,0xee,0xee,0xee,0xee,0xee,0xef,0xff,0xff,0xf0,0xf0,
0x0f,0x0f,0xf6,0x66,0x66,0x66,0x66,0x66,0x66,0x66,0x66,0x6f,0xff,0xff,0xf0,0xf0,
0x0f,0x0f,0xf6,0x66,0x66,0x66,0x66,0x66,0x66,0x66,0x66,0x6f,0xff,0xff,0xf0,0xf0,
0x0f,0x0f,0xf6,0x66,0x66,0x66,0x66,0x66,0x66,0x66,0x66,0x6f,0xff,0xff,0xf0,0xf0,
0x0f,0x0f,0xf4,0x44,0x44,0x44,0x44,0x44,0x44,0x44,0x44,0x44,0xff,0xff,0xf0,0xf0,
0x0f,0x0f,0xf4,0x44,0x44,0x44,0x44,0x44,0x44,0x44,0x44,0x44,0x4f,0xff,0xf0,0xf0,
0x0f,0x0f,0xff,0x44,0x44,0x44,0x44,0x44,0x44,0x44,0x44,0x44,0x44,0x4f,0xf0,0xf0,
0x0f,0x0f,0xff,0x55,0x55,0x55,0x55,0x55,0x55,0x55,0x55,0x55,0x55,0x5f,0xf0,0xf0,
0x0f,0x0f,0xff,0x55,0x55,0x55,0x55,0x55,0x55,0x55,0x55,0x55,0x55,0x5f,0xf0,0xf0,
0x0f,0x0f,0xff,0xf5,0x55,0x55,0x55,0x55,0x55,0x55,0x55,0x55,0x55,0xff,0xf0,0xf0,
0x0f,0x0f,0xff,0xf1,0x11,0x11,0x11,0x11,0x11,0x11,0x11,0x11,0x11,0xff,0xf0,0xf0,
0x0f,0x0f,0xff,0xff,0x11,0x11,0x11,0x11,0x11,0x11,0x11,0x11,0x1f,0xff,0xf0,0xf0,
0x0f,0x0f,0xff,0xff,0xf1,0x11,0x11,0x1f,0xff,0x11,0x11,0x11,0xff,0xff,0xf0,0xf0,
0x0f,0x0f,0xff,0xff,0xfF,0x11,0x11,0xff,0xff,0xf1,0x11,0x1F,0xff,0xff,0xf0,0xf0,
0x0f,0x0f,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xf0,0xf0,
0x0f,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xf0,
0x0f,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xf0,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
};



Rect Text1Rect = {12,4,200,256};

/* This is LONGSTATTEXT2-formatted type text: */
char Text1 [] = "\1J\1\0\
\1S\010\0\
HodgePodge in C\
\r\
\r\
\1S\0\0\
A potpourri of routines that demonstrates many \
features of the Apple IIGS tools.\
\r\
\r\
By the Apple IIGS Development Team\
\r\
\r\
Copyright Apple Computer, Inc., 1986-1987,\
\r\
All rights reserved\
\r\
v4.0   October 1987";


Rect ButtonRect = {153,180,0,0};
```

```
/* Dialog Template data structure for "Please wait while ..." dialog: */

char PlsWtMsg [] = "\pPlease wait while we set things up.";

ItemTemplate PlsWtItem =   {1348,
                            19,70,200,640,
                            statText,
                            PlsWtMsg,
                            0,
                            0,
                            NULL};

DialogTemplate PlsWtTemp = {30,120,80,520,
                            true,
                            NULL,
                            &PlsWtItem,
                            NULL};




/* Alert Template data structure for too many windows and disk error alerts: */

ItemTemplate OurAlertItem1 = {1,
                              25,320,0,0,
                              buttonItem,
                              OKStr,
                              0,
                              0,
                              NULL};

ItemTemplate OurAlertItem2 = {1348,
                              11,72,200,640,
                              statText,
                              NULL,          /* ItemDescr -- will fill it in */
                              0,
                              0,
                              NULL};

AlertTemplate OurAlertTemp = {30,120,80,520,
                              6666,
                              0x80,0x80,0x80,0x80,
                              &OurAlertItem1,
                              &OurAlertItem2,
                              NULL};




CheckToolError (Where)

/* CheckToolError checks to see if the last tool call completed successfully.
   If so, then it just returns.  If not, we crash using the System Death
   Handler (bouncing apple). */

   int Where;

{
   static char DeathMsg [] = "\p At $XXXX; Could not handle error $";
   int  ToolErrorSave;
```

```
  ToolErrorSave = _toolErr;
  if (ToolErrorSave)
  {
    Int2Hex    (Where,DeathMsg + 6,4);
    SysFailMgr (ToolErrorSave,DeathMsg);
  }
}


boolean CheckDiskError (Where)

/* This routine checks if the last ProDOS operation caused an error.  If so,
   then we change the cursor to the arrow cursor, put up a stop alert dialog
   box with the text of the error message, which then waits for the user's
   OK click, and then change the cursor back to the wristwatch.  If there
   was no disk error, then we do nothing.  We also return TRUE or FALSE
   depending on whether an error actually occurred or not. */

int Where;

{
  int DiskErrNum;

  DiskErrNum = _toolErr;                                  /* Save this first */
  if (DiskErrNum)
  {
      OurAlertItem2.itemDescr = "\pDisk Error $XXXX occurred at $XXXX.";
      Int2Hex (DiskErrNum,                                   /* Put ASCII  */
               OurAlertItem2.itemDescr + 13,
               4);
      Int2Hex (Where,                                        /* Put ASCII  */
               OurAlertItem2.itemDescr + 31,
               4);
      InitCursor ();                              /* Set arrow cursor     */
      StopAlert  (&OurAlertTemp,NULL);            /* Draw dialog & wait   */
      /* Do not restore watch cursor */
  }
  return (DiskErrNum);                          /* Assign function result */
}



ManyWindDialog ()

/* Displays caution alert dialog with a message about no more windows
   being allowed open.  Handles mouse events until OK button is clicked.
   Then the dialog box is removed and we return. */

{
  OurAlertItem2.itemDescr = "\pNo more windows, please.";   /* Set string */
  CautionAlert  (&OurAlertTemp,NULL);           /* Do draw, wait, undraw. */
}



DoAboutItem ()

/* Function DoAboutItem shows how to build a dialog box manually. */

{
  handle      AppleIconH;
  GrafPortPtr MdialogPtr;

  AppleIconH = NewHandle (552L,MyID,0,0L);        /* Allocate memory */
  CheckToolError          (50);                   /* Hope it was ok  */
  HLock                   (AppleIconH);           /* Freeze handle   */
  PtrToHand     (AppleIcon640,AppleIconH,552L);   /* Move icon image */
```

```
    MdialogPtr = NewModalDialog (&DRect,TRUE,0L);   /* Draw dialog box */

    /* Install and draw items in the dialog box: */
    NewDItem (MdialogPtr,1,&ButtonRect,buttonItem,OKStr,0,0,NULL);
    NewDItem (MdialogPtr,3,&AppleIconRect,iconItem+itemDisable,
              AppleIconH,0,0,0L);
    NewDItem (MdialogPtr,4,&Text1Rect,longStatText2+itemDisable,Text1,
              sizeof (Text1) - 1,0,0L);

    ModalDialog    (NULL);            /* Track the mouse inside the box */
    CloseDialog    (MdialogPtr);      /* Remove the box from the screen */
    DisposeHandle  (AppleIconH);      /* Deallocate memory              */
}




/* ShowPleaseWait / HidePleaseWait */

/* Brings up a window and puts a message on it
   witout waiting for Update Event              */

ShowPleaseWait ()
{
   OrigPort    = GetPort ();
   MsgWindPtr = GetNewModalDialog (&PlsWtTemp);
   BeginUpdate (MsgWindPtr);                        /* begin Update process */
   DrawDialog  (MsgWindPtr);
   EndUpdate   (MsgWindPtr);
}



HidePleaseWait ()
{
   CloseDialog(MsgWindPtr);
   SetPort(OrigPort);
}




MountBootDisk ()

/* MountBootDisk is called whenever the application requires
   something from the boot volume and it is not online         */

{
   static char          PromptStr []    =   "\pPlease insert the disk",
                        OKStr     []    =   "\pOK",
                        CancelStr []    =   "\pShut Down",
                        VolStr     [256];
   static PathNameRec   GBVParams       =  { VolStr,NULL };

   GET_BOOT_VOL (&GBVParams);
   return (TLMountVolume (174,30,PromptStr,VolStr,OKStr,CancelStr));
}
```

FONT.CC (fonts)

```
/***********************************************************************
*                                                                      *
*     HodgePodge:  An example Apple IIGS Desktop application           *
*                                                                      *
*                                                                      *
*                                                                      *
*                                                                      *
*                                                                      *
*           Copyright (c) 1986-87 by Apple Computer, Inc.              *
*                       All Rights Reserved                            *
*                                                                      *
*  -----------------------------------------------------------------   *
*                                                                      *
*   Source file FONT.CC -- Choosing font, font window defproc          *
*                                                                      *
***********************************************************************/

#include <types.h>
#include <quickdraw.h>
#include <font.h>
#include <intmath.h>
#include <stdfile.h>
#include <window.h>
#include <memory.h>
#include <menu.h>
#include <texttool.h>
#include "hp.h"

extern SFReplyRec MyReply;
/*
typedef struct SFReplyRec  {
     Boolean     good;
     Word       fileType;
     Word       auxFileType;
     char       filename[16];
     char       fullPathname[129];
     } SFReplyRec,  *SFReplyRecPtr ;
*/


extern int _toolErr;

ptr FontWinPtr;

/*
typedef struct FontID {
    Word       famNum;
    Byte       fontStyle;
    Byte       fontSize;
   } FontID, *FontIDPtr, **FontIDHndl;
*/

FontID DesiredFont = { 0xfffe,00,0x08};

int MonoFlag = 0;

/*
typedef struct FontInfoRecord {
    integer      ascent;
    integer      descent;
    integer      widMax;
    integer      leading;
```

```
    } FontInfoRecord, *FontInfoRecPtr, **FontInfoRecHndl;
*/

FontInfoRecord CurrFont;

int CurrHeight,LineCounter;

/*
typedef struct Point {
     Integer   v;
     Integer   h;
    } Point, *PointPtr, **PointHndl;
*/

Point CurrPos;

char Line0[30] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,  /*   Namelength + 1  */
                  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}; /* + 4 for size info */
char Line1[] = "\0";
char Line2[] = "\pThe quick brown fox jumps over the lazy dog.";
char Line3[] = "\pShe sells sea shells down by the sea shore.";
char Line4[] = "\0";

char Line5[] = {32,
                 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13,14,15,
                16,17,18,19,20,21,22,23,24,25,25,27,28,28,30,31,0};

char Line6[] = {32,
                32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,
                48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,0};

char Line7[] = {32,
                64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,
                80,81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,0};

char Line8[] = {32,
                96, 97, 98, 99,100,101,102,103,104,405,106,107,108,109,110,111,
              112,113,114,115,116,117,118,119,120,121,122,123,124,125,126,127,
              0};

char Line9[] = {32,
              128,129,130,131,132,133,134,135,136,137,138,139,140,141,142,143,
              144,145,146,147,148,149,150,151,152,153,154,155,156,157,158,159,
              0};

char Line10[]= {32,
              160,161,162,163,164,165,166,167,168,169,170,171,172,173,174,175,
              176,177,178,179,180,182,182,183,184,185,186,187,188,189,190,191,
              0};

char Line11[]= {32,
              192,193,194,195,196,197,198,199,200,201,202,203,204,205,206,207,
              208,209,210,211,212,213,214,215,216,217,218,219,220,221,222,223,
              0};

char Line12[]= {32,
              224,225,226,227,228,229,230,231,232,233,234,235,236,237,238,239,
              240,241,242,243,244,245,246,247,248,249,250,251,252,253,254,255,
              0};

char *LineTable[] = {Line0,Line1,Line2,Line3,Line4,Line5,Line6,Line7,
                     Line8,Line9,Line10,Line11,Line12};

char ProMsg[32] = "==Display Font as Proportional\r";
char MonoMsg[31]= "==Display Font as Mono-spaced\r";

/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
```

```
DoChooseFont()
{
int whocares;

GrafPortPtr oldPort;
int tempPort[85];                         /* port size in bytes / 2 */

/*
typedef struct FontID {
    Word     famNum;
    Byte     fontStyle;
    Byte     fontSize;
   } FontID, *FontIDPtr, **FontIDHndl;
*/

long tempFont;

  oldPort = GetPort();

  OpenPort(tempPort);

  if (tempFont = ChooseFont(DesiredFont,0))          /* font changed */
    {
      DesiredFont.famNum = (Word)(tempFont & 0xffff);
      DesiredFont.fontStyle = (Byte)((tempFont >> 16 ) & 0xff);
      DesiredFont.fontSize = (Byte)(tempFont >> 24);
      whocares = GetFamInfo(DesiredFont.famNum,
                       MyReply.filename);/* ignore result */
      Int2Dec(DesiredFont.fontSize,    /* size of font  */
            ((MyReply.filename)+(MyReply.filename[0])+1),/* position*/
            4,                          /* length of result */
            0);                              /* not signed */
      MyReply.filename[0] +=4;               /* new legth    */
      ClosePort(tempPort);
      SetPort(oldPort);
      return(TRUE);                          /* new stuff */
    }
  else
    {
    ClosePort(tempPort);
    SetPort(oldPort);
    return(FALSE);                      /* No change */
    }
}

DispFontWindow()

{

FontID fontId;                  /* Dont need it */

FDataRecHandle FontHand;
FDataRecPtr FontPtr;

GrafPortPtr tempPort;

  tempPort = GetPort();                   /* get curr port */
  FontHand = (FDataRecHandle) GetWRefCon(tempPort); /* get handle to data */
  FontPtr = *FontHand;                    /* dereference */
  HLock(FontHand);
  ShowFont(FontPtr -> FID,FontPtr);
  HUnlock(FontHand);
}

ShowFont(fontId,FontPtr)

FontID fontId;
FDataRecPtr FontPtr;
```

```
{
word tempFlags;

   InstallFont(fontId,0);
   GetFontInfo(&CurrFont);
   CurrHeight = CurrFont.ascent + CurrFont.descent + CurrFont.leading;
   MoveTo(0,0);                                    /* start pen position */

   GetFamInfo(fontId.famNum,Line0);                /* ignore result */
   Int2Dec(fontId.fontSize,                   /* size of font  */
              (Line0)+Line0[0]+1,             /* pointer to end*/
              4,                                        /* length of result */
              0);                                            /* not signed */
     Line0[0] +=4;                              /* new length    */
   tempFlags = GetFontFlags();
   SetFontFlags((((FontPtr -> Flag)) >> 1) & 1);


   for (LineCounter = 0;LineCounter < NumLines;LineCounter++)
     {
       GetPen(&CurrPos);
       MoveTo(5,CurrHeight + CurrPos.v);              /* reset x and y    */
       DrawString(LineTable[LineCounter]);
     }

   SetFontFlags(tempFlags);
}

DoSetMono()

{

   if (MonoFlag ^=0x02)
     SetMItem(ProMsg,MonoID);
   else
     SetMItem(MonoMsg,MonoID);
}
```

PRINT.CC (printing)

```
/*********************************************************************
*                                                                    *
*     HodgePodge:  An example Apple IIGS Desktop application         *
*                                                                    *
*                                                                    *
*                                                                    *
*                                                                    *
*                                                                    *
*          Copyright (c) 1986-87 by Apple Computer, Inc.             *
*                      All Rights Reserved                           *
*                                                                    *
*  ---------------------------------------------------------------   *
*                                                                    *
*          Source file PRINT.CC -- Printing stuff                    *
*                                                                    *
*********************************************************************/

#include <types.h>
#include <memory.h>
#include <quickdraw.h>
#include <window.h>
#include <print.h>
#include <qdaux.h>
#include <font.h>
#include "hp.h"

extern int MyID;

GrafPortPtr WindowToPrint = NIL;

handle PrintRecord = NIL;

GrafPortPtr PrintPort;

/* Coose Printer Item handler */

DoChooserItem()
{
   PrChooser();
}

/* Routine to handle page setup item */

DoSetUpItem()
{
   if (!(PrintRecord))
     SetUpDefault();
   PrStlDialog(PrintRecord);
}

/* routine to create default print record */

SetUpDefault()
{
     PrintRecord = NewHandle(140L,MyID,0x8010,0L);
     PrDefault(PrintRecord);
}

/* Now the menu item "Print" item */
```

```
DoPrintItem()
{
    if (WindowToPrint = FrontWindow())              /* is there a window to print? */
      {
        if (!(PrintRecord))
          SetUpDefault();
        if (PrJobDialog(PrintRecord))
          {
            WatchCursor();
            PrintPort = PrOpenDoc(PrintRecord,0L);
            PrOpenPage(PrintPort,0L);
            DrawTopWindow();
            PrClosePage(PrintPort);
            PrCloseDoc(PrintPort);
            PrPicFile(PrintRecord,0L,0L);
            InitCursor();
          }
      }
}

DrawTopWindow()
{

DataRecHandle TheRefCon = NIL;                      /* we use slightly different */
DataRecPtr auxPtr;                                  /* structures for pictures   */

FDataRecHandle FontHandle;                          /* and for fonts             */
FDataRecPtr FontPtr;

   TheRefCon = (DataRecHandle)GetWRefCon(WindowToPrint);
   HLock(TheRefCon);
   auxPtr = *TheRefCon;
   if (auxPtr -> Flag)                              /* non_zero --> font */
     {
       FontHandle = (FDataRecHandle)GetWRefCon(WindowToPrint);    /* again          */
       HLock(FontHandle);                                /* to pass     */
       FontPtr = *FontHandle;                            /* the right   */
       ShowFont(FontPtr -> FID,FontPtr);                 /* stuff       */
       HUnlock(FontHandle);
     }
   else                                             /* Picture Window */
     PaintIt(auxPtr -> PicHand);
   HUnlock(TheRefCon);
}
```

HP.H (global data)

```
#include <types.h>
#include <quickdraw.h>
#include <font.h>

#define SCREENMODE 0x80                       /* 640 mode */
#define MAXSCAN    160
#define QDAuxTool 18                          /* Auxiliary Quickdraw  */
#define PManager  19                          /* Print Manager Tool Number */
#define MinVer     0                          /* Minimun Version for them */
#define VolNotFound 0x45

#define NUM_MENUS      5                      /* Number of menus */
#define NUM_WINDOWS    15                     /* Maximum number of windows */

/* Menus related defines */
#define AppleMenuID     1
#define FileMenuID      2
#define EditMenuID      3
#define ModeMenuID      4
#define WindowsMenuID   5
#define FontsMenuID     6

#define UndoID          250               /* These next 6 are standard and */
#define CutID           251               /* required for DA support under */
#define CopyID          252               /* TaskMaster.                   */
#define PasteID         253
#define ClearID         254
#define CloseWID        255

#define AboutID         256            /* These are our own responsibility */
#define QuitID          257
#define OpenWID         258
#define SaveID          259
#define ChooseID        260
#define SetUpID         261
#define PrintID         262
#define ModeID          263
#define ShowFontID      264
#define MonoID          265


/* some font and window handling stuff */

#define MaxNameSize 29


#define NumLines 13

typedef struct DataRec {
                  char **PicHand;
                  char Blank;
                  char Str[30];
                  char MMStuff[6];
                  short Flag;
                  char Extra;
                  } DataRec, *DataRecPtr, **DataRecHandle;
```

```
/* same thing as DataRec but for FONT windows */

typedef struct FontDataRec {
                FontID FID;         /* This is Pic handle in DataRec */
                char Blank;
                char Str[30];
                char MMStuff[6];
                Byte Flag;
                char Extra;
                } FDataRec, *FDataRecPtr, **FDataRecHandle;

typedef int PackedData[320];

typedef struct DirEntry
                  {
                  int PackedBytes;
                  word Mode;
                  } DirEntry;

typedef struct MainBlk {
                long SizeOfBlock;
                char IDStr[5];
                word MasterMode;
                int PixelsPerScanLine;
                int NumPallets;
                int  PalletArray[16][16];
                int NumScanLines;
                DirEntry ScanLineDir[200];
                PackedData PackedScanLines[200];
                } MainBlk,*MainBlkPtr,**MainBlkHandle;

/* all the files for this program include HP.H, but not all do the same with
   DIALOG.H that is why:
*/

#ifndef __dialog__

typedef struct ItemTemplate  {
      Word      itemID;      /* ItemTemplate -  */
      Rect      itemRect;     /* ItemTemplate -  */
      Word      itemType;     /* ItemTemplate -  */
      Pointer   itemDescr;     /* ItemTemplate -  */
      Word      itemValue;     /* ItemTemplate -  */
      Word      itemFlag;     /* ItemTemplate -  */
      Pointer   itemColor;     /* pointer to appropreate ctl color table */
      } ItemTemplate,  *ItemTempPtr,  **ItemTempHndl ;
#endif

/* Here we define the dialog templates used for Standard File Get and Put
   calls.
*/

#ifndef GetPutListLength
#define  GetPutListLength  0xF   /* Set to 15 which is the max */
#endif

typedef struct GetPutTemplate  {
      Rect      gpBoundsRect;
      Boolean      gpVisible;
      LongWord      gpRefCon;
      ItemTempPtr      gpItemList[GetPutListLength];
      } GetPutTemplate,  *GetPutTempPtr ;
```

Appendix G

HodgePodge Source Code: Pascal

HP.PAS (main program)

```
program HodgePodge;

{+-----------------------------------------------------------------------------+
 |                                                                             |
 |       HodgePodge:  An example Apple IIGS Desktop application                |
 |                                                                             |
 |            Written by the Apple IIGS Development Team                       |
 |            Translated to TML Pascal by TML Systems, Inc.                    |
 |                                                                             |
 |           Copyright (c) 1986-87 by Apple Computer, Inc.                     |
 |                         All Rights Reserved                                 |
 |                                                                             |
 |    ---------------------------------------------------------------------    |
 |                                                                             |
 |          This program and its derivatives are licensed only for             |
 |          use on Apple computers.                                            |
 |                                                                             |
 |          Works based on this program must contain and                       |
 |          conspicuously display this notice.                                 |
 |                                                                             |
 |          This software is provided for your evaluation and to               |
 |          assist you in developing software for the Apple IIGS               |
 |          computer.                                                          |
 |                                                                             |
 |          This is not a distribution license. Distribution of                |
 |          this and other Apple software requires a separate                  |
 |          license. Contact the Software Licensing Department of              |
 |          Apple Computer, Inc. for details.                                  |
 |                                                                             |
 |          DISCLAIMER OF WARRANTY                                             |
 |                                                                             |
 |          THE SOFTWARE IS PROVIDED "AS IS" WITHOUT                           |
 |          WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED,                   |
 |          WITH RESPECT TO ITS MERCHANTABILITY OR ITS FITNESS                 |
 |          FOR ANY PARTICULAR PURPOSE.  THE ENTIRE RISK AS TO                 |
 |          THE QUALITY AND PERFORMANCE OF THE SOFTWARE IS WITH                |
 |          YOU.  SHOULD THE SOFTWARE PROVE DEFECTIVE, YOU (AND                |
 |          NOT APPLE OR AN APPLE AUTHORIZED REPRESENTATIVE)                   |
 |          ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING,                 |
 |          REPAIR OR CORRECTION.                                              |
 |                                                                             |
 |          Apple does not warrant that the functions                          |
 |          contained in the Software will meet your requirements              |
 |          or that the operation of the Software will be                      |
 |          uninterrupted or error free or that defects in the                 |
 |          Software will be corrected.                                        |
 |                                                                             |
 |          SOME STATES DO NOT ALLOW THE EXCLUSION                             |
 |          OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY                  |
 |          NOT APPLY TO YOU.  THIS WARRANTY GIVES YOU SPECIFIC                |
 |          LEGAL RIGHTS AND YOU MAY ALSO HAVE OTHER RIGHTS                    |
 |          WHICH VARY FROM STATE TO STATE.                                    |
 |                                                                             |
 |    ---------------------------------------------------------------------    |
 |                                                                             |
 |    Pascal UNIT "HP.PAS" : Main routine and tool init/shutdown routines      |
 |                                                                             |
 +-----------------------------------------------------------------------------+}
```

```
USES
    HPIntfData,         {HodgePodge Apple IIGS Toolbox Interface Units}
    HPIntfProc,
    HPIntfPdos,

    Globals,            {HodgePodge Code Units}
    Dialog,
    Font,
    Paint,
    Window,
    Print,
    Menu,
    Event;

function StartUpTools : boolean;

  {Routine to start up the Apple IIGS toolbox.  We attempt to start up all
   the managers that we need, checking each time if an error occurred during
   startup.  True/false is returned by this routine depending on its success.
   If the RAM-based tools cannot be loaded, the user is prompted to install
   a system disk and is given the option of trying again or exitting.  The
   latter option exits this procedure with a False result.  Tool startup
   errors result in a call to the system death handler (the bouncing apple),
   with a code showing where we died as well as the actual tool error number.}

  const DPForQuickDraw = $000;    {Needs 3 pages}
        DPForEventMgr  = $300;    {Needs 1}
        DPForCtlMgr    = $400;    {Needs 1}
        DPForLineEdit  = $500;    {Needs 1}
        DPForMenuMgr   = $600;    {Needs 1}
        DPForStdFile   = $700;    {Needs 1}
        DPForFontMgr   = $800;    {Needs 1}
        DPForPrintMgr  = $900;    {Needs 2}
        TotalDP        = $B00;    {Total direct page space}

  var toolRec    : ToolTable;
      paramBlock : FileRec;
      baseDP     : integer;

  label 1;  {Just for once, let's commit the cardinal sin of using the GOTO!}

  begin   {of StartUpTools}

    StartUpTools := true;              {Assume all is well at first}

    TLStartUp;                                {Init Tool Locator   }
    CheckToolError ($1);

    MyMemoryID := MMStartUp;                  {Init Memory Manager}
    MTStartUp;                                {Init Misc Tools     }
    CheckToolError ($2);

    {Allocate memory space in bank 0 for direct-page use by GS Tools:}
    ToolsZeroPage :=
        NewHandle (TotalDP,                      {Allocate memory}
                   MyMemoryID,                   {Process (user) ID}
                   attrBank+attrFixed+attrLocked+attrPage, {Attributes}
                   Ptr (0));                     {Start in bank 0   }
    CheckToolError ($3);
    baseDP := LoWord (ToolsZeroPage^);

    QDStartUp
        (baseDP + DPForQuickDraw,                {Address of zpag # 3 }
         ScreenMode,                             {640 mode            }
         MaxScan,                                {Horizontal line size}
         MyMemoryID);                            {Process (user) ID   }
    CheckToolError ($4);
```

```
    EMStartUp
        (baseDP + DPForEventMgr,                    {Address of zpag # 4 }
         20,                                        {Event queue size    }
         0,                                         {X min clamp         }
         MaxX,                                      {X max clamp         }
         0,                                         {Y min clamp         }
         200,                                       {Y max clamp         }
         MyMemoryID);                               {Process (user) ID   }
    CheckToolError ($5);

    {Give a message while we load RAM based tools:}
    MoveTo          (20,20);
    SetBackColor    (0);
    SetForeColor    (15);
    DrawString      ('One Moment Please...');

    ShowCursor;

    {Now load RAM based tools (and RAM patches to ROM tools!):}
    toolRec.NumTools := 14;
    toolRec.Tools[1].TSNum := 4;                    {QuickDraw II    }
    toolRec.Tools[1].MinVersion := 0;
    toolRec.Tools[2].TSNum := 5;                    {Desk Manager    }
    toolRec.Tools[2].MinVersion := 0;
    toolRec.Tools[3].TSNum := 6;                    {Event Manager   }
    toolRec.Tools[3].MinVersion := 0;
    toolRec.Tools[4].TSNum := 14;                   {Window Manager }
    toolRec.Tools[4].MinVersion := 0;
    toolRec.Tools[5].TSNum := 15;                   {Menu Manager    }
    toolRec.Tools[5].MinVersion := 0;
    toolRec.Tools[6].TSNum := 16;                   {Control Manager}
    toolRec.Tools[6].MinVersion := 0;
    toolRec.Tools[7].TSNum := 18;                   {QuickDraw Aux   }
    toolRec.Tools[7].MinVersion := 0;
    toolRec.Tools[8].TSNum := 19;                   {Print Manager   }
    toolRec.Tools[8].MinVersion := 0;
    toolRec.Tools[9].TSNum := 20;                   {Line Edit       }
    toolRec.Tools[9].MinVersion := 0;
    toolRec.Tools[10].TSNum := 21;                  {Dialog Manager }
    toolRec.Tools[10].MinVersion := 0;
    toolRec.Tools[11].TSNum := 22;                  {Scrap Manager   }
    toolRec.Tools[11].MinVersion := 0;
    toolRec.Tools[12].TSNum := 23;                  {Standard File   }
    toolRec.Tools[12].MinVersion := 0;
    toolRec.Tools[13].TSNum := 27;                  {Font Manager    }
    toolRec.Tools[13].MinVersion := 0;
    toolRec.Tools[14].TSNum := 28;                  {List Manager    }
    toolRec.Tools[14].MinVersion := 0;

1:
    paramBlock.pathname := @'*/SYSTEM/TOOLS';       {Make sure tools avail}
    GET_FILE_INFO (paramBlock);
    if toolErr <> 0 then
        if MountBootDisk = 1 then
            goto 1
        else begin
            StartUpTools := false;
            Exit;
        end;

    LoadTools       (toolRec);                      {Load the tools I need}
    CheckToolError ($6);

    WindStartUp     (MyMemoryID);                   {Init Window Manager }
    CheckToolError ($7);
```

```
   RefreshDesktop (nil);                            {Draw the desktop     }

   CtlStartUp                                       {Init Control Manager}
       (MyMemoryID,                                 {Process (user) ID    }
        baseDP + DPForCtlMgr);                      {Address of zpag # 5 }
   CheckToolError ($8);

   LEStartUp                                        {Init Line Edit       }
       (baseDP + DPForLineEdit,                     {Address of zpag # 6 }
        MyMemoryID);                                {Process (user) ID    }
   CheckToolError ($9);

   DialogStartUp                                    {Init Dialog Manager }
       (MyMemoryID);                                {Process (user) ID    }
   CheckToolError ($A);

   MenuStartUp                                      {Init Menu Manager    }
       (MyMemoryID,                                 {Process (user) ID    }
        baseDP + DPForMenuMgr);                     {Address of zpag # 7 }
   CheckToolError ($B);

   DeskStartUp;                                     {Init Desk Manager    }
   CheckToolError ($C);

   ShowPleaseWait;                                  {Put up dialog box    }

   SFStartUp                                        {Init Standard File  }
       (MyMemoryID,                                 {Process (user) ID    }
        baseDP + DPForStdFile);                     {Address of zpag # 8 }
   CheckToolError ($D);
   SFAllCaps (true);                        {I want filenames in all caps}

   QDAuxStartUp;                                    {Init QuickDraw Auxil}
   CheckToolError ($E);
   WaitCursor;                                      {Wristwatch cursor    }

   FMStartUp                                        {Init Font Manager    }
       (MyMemoryID,                                 {Process (user) ID    }
        baseDP + DPForFontMgr);                     {Address of zpag # 9 }
   CheckToolError ($F);

   ListStartUp;                                     {Init List Manager    }
   CheckToolError ($10);

   ScrapStartUp;                                    {Init Scrap Manager   }
   CheckToolError ($11);

   PMStartUp                                        {Init Print Manager   }
       (MyMemoryID,                                 {Process (user) ID    }
        baseDP + DPForPrintMgr);                    {Address of zpag # 10}
   CheckToolError ($12);

   HidePleaseWait;                                  {Remove dialog box    }
   InitCursor;                                      {Normal arrow cursor }

end;    {of StartUpTools}
```

```
procedure ShutDownTools;

    {Routine to shut down all the tools we used in reverse order of startup.
     Only tools which are currently active are shut down; this facilitates
     recovery from an error condition from StartUpTools.}

    begin   {of ShutDownTools}

        DeskShutDown;
        if WindStatus <> 0 then
            HideAllWindows;     {Close all windows only if OK!  Takes some time.}
        ListShutDown;
        FMShutDown;
        ScrapShutDown;
        PMShutDown;
        QDAuxShutDown;
        SFShutDown;
        MenuShutDown;
        DialogShutDown;
        LEShutDown;
        CtlShutDown;
        WindShutDown;
        EMShutDown;
        QDShutDown;
        MTShutDown;
        if MMStatus <> 0 then begin
            DisposeHandle (ToolsZeroPage);  {Deallocate tool directpage space}
            MMShutDown    (MyMemoryID);     {Do this only if OK!}
        end;
        TLShutDown;

    end;    {of ShutDownTools}



BEGIN  {of MAIN program HodgePodge}

    InitGlobals;              { Initialze our globals, menus, etc. }

    if StartUpTools then begin      { Initialize IIGS Tools }
        SetUpDefault;               { Set up print dialog }
        SetUpMenus;                 { Set up menus }
        SetUpWindows;               { Set up windows }
        MainEvent;                  { Use application }
    end;

    ShutDownTools;            { Shut down IIGS Tools }

END.   {of MAIN program HodgePodge}
```

MENU.PAS (menus)

```
UNIT Menu;

{+--------------------------------------------------------------------------+
 |                                                                          |
 |         HodgePodge:  An example Apple IIGS Desktop application           |
 |                                                                          |
 |              Written by the Apple IIGS Development Team                  |
 |              Translated to TML Pascal by TML Systems, Inc.                |
 |                                                                          |
 |             Copyright (c) 1986-87 by Apple Computer, Inc.                |
 |                          All Rights Reserved                             |
 |                                                                          |
 |                   --------------------------------                       |
 |                                                                          |
 |     Pascal UNIT "MENU.PAS" : Menu bar setup and menu item handling       |
 |                                                                          |
 +--------------------------------------------------------------------------+}


INTERFACE

USES
      HPIntfData,           {HodgePodge Apple IIGS Toolbox Interface Units}
      HPIntfProc,
      HPIntfPdos,

      Globals,              {HodgePodge Code Units}
      Dialog,
      Font,
      Paint,
      Window,
      Print;


procedure DoMenu;       {Execute a menu item}
procedure SetUpMenus;   {Install menus and redraw menu bar}


IMPLEMENTATION


procedure AddToMenu;

  {Private routine to add a new window item to the "Windows" menu after a
   new window has been drawn.  Increments the variable WIndex, a count of
   the number of windows currently open.}

  var theWindow    : GrafPortPtr;
      myDataHandle : WindDataH;

  begin   {of AddToMenu}
      theWindow             := FrontWindow;
      WindowList [WIndex] := theWindow;

      myDataHandle := WindDataH (GetWRefCon (theWindow));

      InsertMItem (@myDataHandle^^.menuStr [1],$FFFF,WindowsMenuID);
```

```
      if WIndex = 0 then begin                   {This is the first window}
           DeleteMItem (NoWindowsItem);           {Remove the "filler" item}
           SetMenuFlag ($FF7F,WindowsMenuID);  {Highlight the menu}
           DrawMenuBar;
       end;

       CalcMenuSize (0,0,WindowsMenuID);
       Inc (WIndex);
   end;    {of AddToMenu}
procedure DoOpenItem;

   {Private routine which is called when the "Open..." item from the "File"
    menu OR the "Display Font..." item from the "Fonts" menu is selected
    (OpenWindow will determine which one it was).  If too many windows are
    already open, then a dialog is displayed.}

   begin   {of DoOpenItem}
       if WIndex < LastWind then
           if OpenWindow then
               AddtoMenu
           else
       else
           ManyWindDialog;
   end;    {of DoOpenItem}



procedure DoQuitItem;

   {Private routine to set Done flag if the "Quit" item was selected}

   begin   {of DoQuitItem}
       Done := true;
   end;    {of DoQuitItem}



procedure DoWindow (itemNum: integer);

   {Private routine which brings a specific window to the front of the
    desktop, in response to a selection from the "Windows" menu.}

   var theWindow: GrafPortPtr;

   begin   {of DoWindow}
       theWindow := WindowList [itemNum - FirstWindItem];
       SelectWindow (theWindow);
       ShowWindow   (theWindow);
   end;    {of DoWindow}
```

```
procedure DoMenu;

  {Procedure to handle all menu selections.  Examines the Event.TaskData
   menu item ID word from TaskMaster (from Event Manager) and calls the
   appropriate routine.  While the routine is running the menu title is
   still highlighted.  After the routine returns, we unhilight the
   menu title.}

  var menuNum : integer;
      itemNum : integer;

  begin   {of DoMenu}

      menuNum := HiWord (Event.wmTaskData);
      itemNum := LoWord (Event.wmTaskData);

      case itemNum of
          AboutItem    :  DoAboutItem;
          OpenItem     :  DoOpenItem;
          CloseItem    :  DoCloseItem;
          SaveAsItem   :  DoSaveItem;
          ChoosePItem  :  DoChooserItem;
          PageSetItem  :  DoSetupItem;
          PrintItem    :  DoPrintItem;
          QuitItem     :  DoQuitItem;
          UndoItem     :  ;
          CutItem      :  ;
          CopyItem     :  ;
          PasteItem    :  ;
          ClearItem    :  ;
          FontItem     :  DoOpenItem;
          MonoItem     :  DoSetMono;
      otherwise
          DoWindow (itemNum);
      end;

      HiliteMenu (false,menuNum);     {Unhighlight the menu title}

  end;    {of DoMenu}


procedure SetUpMenus;

  {Procedure to install our menu titles and their items in the system menu
   bar and to redraw it so we can see them.}

  var height : integer;

  begin   {of SetUpMenus}
      SetMTitleStart(10);                       {Set Starting position of menu}

      InsertMenu (NewMenu (@FontMenuStr   [1]),0);     {Fonts Menu  }
      InsertMenu (NewMenu (@WindowMenuStr [1]),0);     {Window Menu }
      InsertMenu (NewMenu (@EditMenuStr   [1]),0);     {Edit Menu   }
      InsertMenu (NewMenu (@FileMenuStr   [1]),0);     {File Menu   }
      InsertMenu (NewMenu (@AppleMenuStr  [1]),0);     {Apple Menu  }

      FixAppleMenu (AppleMenuID);               {Add DAs to apple menu    }
      height := FixMenuBar;                     {Set sizes of menus       }
      DrawMenuBar;                              {...and draw the menu bar!}
  end;    {of SetUpMenus}

END.
```

EVENT.PAS (main event loop)

```
UNIT Event;

{+--------------------------------------------------------------------------------+
 |                                                                                |
 |        HodgePodge:  An example Apple IIGS Desktop application                 |
 |                                                                                |
 |             Written by the Apple IIGS Development Team                        |
 |             Translated to TML Pascal by TML Systems, Inc.                     |
 |                                                                                |
 |            Copyright (c) 1986-87 by Apple Computer, Inc.                      |
 |                          All Rights Reserved                                  |
 |                                                                                |
 |                   --------------------------------                            |
 |                                                                                |
 |    Pascal UNIT "EVENT.PAS" : Event loop and dispatching routine               |
 |                                                                                |
 +--------------------------------------------------------------------------------+}


INTERFACE

USES
       HPIntfData,          {HodgePodge Apple IIGS Toolbox Interface Units}
       HPIntfProc,
       HPIntfPdos,

       Globals,             {HodgePodge Code Units}
       Dialog,
       Font,
       Paint,
       Window,
       Print,
       Menu;


procedure MainEvent;   {Main event handling loop which repeats until Quit}


IMPLEMENTATION


procedure MainEvent;

{Main event handling routine which loops until the Done flag is set by
    selection of the "Quit" item.  We call the Window Manager's TaskMaster
    routine, which calls the Event Manager's GetNextEvent routine and
    handles window resize tracking/resizing, window movement tracking/resizing,
    window activiation (bringing to front by clicking on an inactive window),
    among other things.  TaskMaster returns control to us when the user has
    clicked a window's GoAway check box, or when the user has selected a menu
    item, either with the mouse or with an equivalent Solid-Apple keystroke
    sequence.}

  var code : integer;
```

```
procedure CheckFrontW;

   {Check to whom the front window belongs to (us or a Desk Accessory (DA)),
    and if it belongs to us, whether it is appropriate to disable (dim) certain
    menu items (such as the Save item) or to enable them.  Private routine.}

      var theWindow    : GrafPortPtr;
          myDataHandle : WindDataH;



   procedure DisableItems;

          {Private routine to disable (dim) certain menu titles}

      begin   {of DisableItems}
          DisableMItem (SaveAsItem);
          DisableMItem (CloseItem);
          DisableMItem (PrintItem);
          DisableMItem (PageSetItem);
      end;    {of DisableItems}



   procedure EnableItems;

      {Private routine to enable (undim) certain menu titles}

      begin   {of EnableItems}
          EnableMItem (SaveAsItem);
          EnableMItem (CloseItem);
          EnableMItem (PrintItem);
          EnableMItem (PageSetItem);
      end;    {of EnableItems}



   procedure DisableAll;

      {Private routine to disable all menu titles for Desk Accessory (DA)}

      begin   {of DisableAll}
          SetMenuFlag ($0080,EditMenuID);
          DrawMenuBar;
          DisableItems;
      end;    {of DisableAll}
```

```
        procedure SetUpForAppW;

            {Called if an application window (ours) is the frontmost window.
             Private routine.}

            begin   {of SetUpForAppW}
                SetMenuFlag ($0080,EditMenuID);
                DrawMenuBar;
                EnableItems;
            end;    {of SetUpForAppW}


        procedure SetUpForDAW;

            {Called if a Desk Accessory's window is the frontmost window. Private.}

            begin   {of SetUpForDAW}
                DisableItems;
                EnableMItem (CloseItem);
                SetMenuFlag ($FF7F,EditMenuID);
                DrawMenuBar;
            end;    {of SetUpForDAW}


        begin   {of CheckFrontW}
            theWindow := FrontWindow;

            if theWindow = lastWindow then
                Exit;

            if theWindow = nil then
                DisableAll
            else begin
                if GetSysWFlag (theWindow) = true then
                    SetUpforDAW
                else begin
                    SetUpforAppW;
                    myDataHandle := WindDataH (GetWRefCon (theWindow));
                    if myDataHandle^^.Flag = 1 then
                        DisableMItem (SaveAsItem)
                end;
            end;

            lastWindow := theWindow;
        end;    {of CheckFrontW}


    begin   {of MainEvent}
        Event.wmTaskMask := $00001FFF;  {Allow TaskMaster to do everything}
        Done             := false;      {Done flag will be set by Quit item}

        repeat
            CheckFrontW;
            code := TaskMaster ($FFFF,Event);
            case code of
                wInGoAway   : DoCloseItem;
                wInSpecial,
                wInMenuBar  : DoMenu;
            end;
        until Done;
    end;    {of MainEvent}


END.
```

WINDOW.PAS (windows)

```
UNIT Window;

{+--------------------------------------------------------------------------+
 |                                                                          |
 |        HodgePodge:  An example Apple IIGS Desktop application            |
 |                                                                          |
 |             Written by the Apple IIGS Development Team                   |
 |             Translated to TML Pascal by TML Systems, Inc.                |
 |                                                                          |
 |            Copyright (c) 1986-87 by Apple Computer, Inc.                 |
 |                          All Rights Reserved                             |
 |                                                                          |
 |                    -------------------------------                       |
 |                                                                          |
 |    Pascal UNIT "WINDOW.PAS" : Routines to open and close windows         |
 |                                                                          |
 +--------------------------------------------------------------------------+}


INTERFACE

USES
      HPIntfData,          {HodgePodge Apple IIGS Toolbox Interface Units}
      HPIntfProc,
      HPIntfPdos,

      Globals,             {HodgePodge Code Units}
      Dialog,
      Paint,
      Font;


procedure DoCloseItem;             {Closes current frontmost window        }
procedure HideAllWindows;          {Closes all windows on the desktop      }
function  OpenWindow : boolean;    {Tries to open a font or picture window }
procedure SetUpWindows;            {Initialize variables for stacking windows}


IMPLEMENTATION



var
      myWind      :   ParamList;
      wXoffset    :   integer;
      wYoffset    :   integer;
      iSizPos     :   Rect;


procedure DoCloseItem;

   {This procedure closes the frontmost window and deallocates all of its
    associated storage.  NDA windows are supported for when this procedure
    is called by HideAllWindows when exitting HodgePodge.}

   var theWindow    : GrafPortPtr;
       myDataHandle : WindDataH;
```

```
procedure AdjWind (theWindow: GrafPortPtr);

    {Finds the window designated by theWindow and removes it from the
     WindowList and returns the position in the window list where it was
     found.  Private function.}

    var  i       : integer;
         theOne  : integer;

    begin    {of AdjWind}

        {Find the index of the grafportptr of the window being deleted:}
        i := firstWind;
        while WindowList [i] <> theWindow do
            Inc (i);
        theOne   := i;

        {Remove corresponding item from the WINDOW-menu:}
        if WIndex = 1 then begin                    {Last window--special case}
            InsertMItem (@NoWindStr [1],FirstWindItem + theOne,WindowsMenuID);
            SetMenuFlag ($0080,WindowsMenuID);
            DrawMenuBar;
            wXoffset := 20;
            wYoffset := 12;
        end;
        DeleteMItem    (FirstWindItem + theOne);
        CalcMenuSize   (0,0,WindowsMenuID);

        {Physically delete (scroll) the grafportptr of the ill-fated window:}
        Inc (i);
        while i < LastWind do begin
            WindowList [i - 1] := WindowList [i];
            Inc (i);
        end;

        {Renumber the WINDOW-menu items:}
        for i := theOne to LastWind do
            SetMItemID (FirstWindItem+i-1 {new ID}  ,  FirstWindItem+i {old ID});

    end;    {of AdjWind}


  begin    {of DoCloseItem}
      theWindow := FrontWindow;
      CloseNDAbyWinPtr (theWindow);
      if isToolError then begin                  {It wasn't an NDA window}
          AdjWind         (theWindow);           {Update WINDOW menu}
          myDataHandle    := WindDataH (GetWRefCon (theWindow));
          DisposeHandle   (Handle (myDataHandle));  {Deallocate storage}
          CloseWindow     (theWindow);           {Remove the window}
          Dec             (WIndex);              {Index into window list}
      end;
  end;    {of DoCloseItem}

procedure HideAllWindows;

   {Repeatedly call DoCloseItem to close the frontmost window (which has the
    effect of making the next deeper level window the frontmost one) until
    there is no frontmost window anymore; ie, there are no more windows.}

  begin    {of HideAllWindows}
      while FrontWindow <> nil do
          DoCloseItem;
  end;    {of HideAllWindows}
```

```
function  OpenWindow : boolean;

  {Tries to open either a font or picture window, depending on the
   Event.TaskData returned from TaskMaster (which got it from the
   Event Manager).  True/false is returned depending on whether a
   window was actually opened.  Note the way in which the different
   functions are called in the if-then-else structure below.  Each
   function tries to do what its name implies, and the true/false
   result that each returns is used to determine if the next logical
   function should be called.}


  function DoTheOpen: boolean;

     {This function tries to open a window and returns true/false depending on
      its success.}

     var theWindow    : GrafPortPtr;
         myDataHandle : WindDataH;
         theMenuStr   : Str255;
         ourFontInfo  : FontInfoRecord;

     begin   {of DoTheOpen}
         DoTheOpen := false;

         myDataHandle := WindDataH (NewHandle (sizeof (WindDataRec),
                                              MyMemoryID,
                                              attrLocked + attrFixed,
                                              Ptr (0)));
         if isToolError then
             Exit;

         with myWind do begin
             paramLength  := sizeof (ParamList);
             wFrameBits   := $DDA0;
             wRefCon      := longint (myDataHandle);
             SetRect      (wZoom,0,26,620,190);
             wColor       := nil;
             wYOrigin     := 0;
             wXOrigin     := 0;
             wDataH       := 188;
             wDataW       := 640;
             wMaxH        := 200;
             wMaxW        := 640;
             wScrollVer   := 4;
             wScrollHor   := 16;
             wPageVer     := 40;
             wPageHor     := 160;
             wInfoRefCon  := 0;
             wInfoHeight  := 0;
             wFrameDefProc:= nil;
             wInfoDefProc := nil;
             wPlane       := -1;
             wStorage     := nil;
         end;

         theMenuStr := concat ('==',
                               myReply.filename,
                               '\N',
                               IntToString (FirstWindItem + WIndex),
                               '\0.');

         with myDataHandle^^ do begin
             Name    := myReply.filename;
             MenuStr := theMenuStr;
             MenuID  := FirstWindItem + WIndex;
         end;
```

```
        if LoWord (Event.wmTaskData) = FontItem then begin
            {We're opening a font window:}
            myWind.wContDefProc := @DispFontWindow;
            with myDataHandle^^ do begin
                flag    := 1;
                theFont := DesiredFont;
                isMono  := isMonoFont;
            end;
            InstallFont (DesiredFont,0);
            GetFontInfo (ourFontInfo);
            MyWind.wDataH := 15 {NumLines+2} *
                (OurFontInfo.ascent + ourFontInfo.descent);
            {Call to a FindMaxWidth procedure would be placed here to set
             the MyWind.wDataW field to length of the longest line of text.}
        end else begin
            {We're opening a picture window:}
            myWind.wContDefProc := @Paint;
            with myDataHandle^^ do begin
                flag := 0;
                pict := PictHndl;
            end;
        end;

        with myWind do begin
            wTitle := @myDataHandle^^.Name;
            SetRect (wPosition,wXoffset + iSizPos.h1,
                               wYoffset + iSizPos.v1,
                               wXoffset + iSizPos.h2,
                               wYoffset + iSizPos.v2);
        end;

        wXoffset := wXoffset + 20;  {Update globals which offset new window pos}
        wYoffset := wYoffset + 12;
        if wYoffset > 120 then      {Cause stacking effect}
             wYoffset := 12;

        {Now create the window:}
        theWindow := NewWindow      (myWind);
        SetPort                     (theWindow);
        SetOriginMask               ($FFFE,theWindow);

        InitCursor;                 {Go back to the arrow cursor}
        DoTheOpen := true;          {Indicate successful completion}
    end;    {of DoTheOpen}


   begin   {of OpenWindow}
      OpenWindow := false;
      if LoWord (Event.wmTaskData) = FontItem then begin
          if DoChooseFont then
              if DoTheOpen then
                  OpenWindow := true
      end else begin
          if AskUser then
              if DoTheOpen then
                  OpenWindow := true
      end;
   end;    {of OpenWindow}

procedure SetUpWindows;

   begin   {of SetUpWindows}
      wXoffset := 20;               {Initial window position offset used for}
      wYoffset := 12;               {...stacking the windows.}
      SetRect  (iSizPos,10,20,350,80);
   end;    {of SetUpWindows}

END.
```

DIALOG.PAS (dialog boxes)

```
UNIT Dialog;

+------------------------------------------------------------------------------+
|                                                                              |
|         HodgePodge:  An example Apple IIGS Desktop application               |
|                                                                              |
|              Written by the Apple IIGS Development Team                      |
|              Translated to TML Pascal by TML Systems, Inc.                   |
|                                                                              |
|             Copyright (c) 1986-87 by Apple Computer, Inc.                    |
|                           All Rights Reserved                                |
|                                                                              |
|                     --------------------------------                         |
|                                                                              |
|     Pascal UNIT "DIALOG.PAS" : Dialog and Alert box drawing routines         |
|                                                                              |
+------------------------------------------------------------------------------+}


INTERFACE

USES
      HPIntfData,          {HodgePodge Apple IIGS Toolbox Interface Units}
      HPIntfProc,
      HPIntfPdos,

      Globals;             {HodgePodge Code Unit}



var
      currentItem1 : ItemTemplate;
      currentItem2 : ItemTemplate;
      origPort     : GrafPortPtr;
      msgWindPtr   : GrafPortPtr;


procedure DoAboutItem;                                 {About item in apple menu}
procedure ShowPleaseWait;                              {Please Wait during init }
procedure HidePleaseWait;                              {Erase the above         }
function  CheckDiskError (Where : integer) : boolean;  {Alert if ProDOS error   }
procedure CheckToolError (Where : integer);            {Death if tool error     }
function  MountBootDisk : integer;                     {Ask boot disk; 1 if OK  }
procedure ManyWindDialog;                              {Waits until OK clicked  }
```

```
procedure MakeATemplate (TheTemplate : AlertTempPtr; TheStr : StringPtr);

  {Private routine which creates an alert template.}

  begin   {of MakeATemplate}
      with TheTemplate^ do begin
          SetRect (atBoundsRect,120,30,520,80);
          atAlertID := 1500;
          atStage1  := $80;
          atStage2  := $80;
          atStage3  := $80;
          atStage4  := $80;
          atItem1   := @currentItem1;
          atItem2   := @currentItem2;
          atItem3   := nil;
      end;
      with currentItem1 do begin
          ItemId    := 1;
          SetRect   (ItemRect,320,25,0,0);
          ItemType  := 10; {Buttton item constant >!<}
          ItemDescr := @'OK';
          ItemValue := 0;
          ItemFlag  := 0;
          ItemColor := nil;
      end;
      with currentItem2 do begin
          ItemId    := 2;
          SetRect   (ItemRect,72,11,639,199);
          ItemType  := 15 + $8000; {Disabled Static Text item constant >!<}
          ItemDescr := Pointer (TheStr);
          ItemValue := 0;
          ItemFlag  := 0;
          ItemColor := nil;
      end;
  end;    {of MakeATemplate}


procedure ShowPleaseWait;

  {Displays "Please Wait..." dialog box on the screen.}

  var r : rect;

  begin   {of ShowPleaseWait}
      origPort   := GetPort;
      msgWindPtr := GetNewModalDialog (@PlsWtTemp);
      SetRect  (r,70,19,640,200);
      NewDItem (msgWindPtr,1502,r,15,@'Please wait while we set things up.',
          0,0,Pointer(0));
      BeginUpdate (msgWindPtr);
      DrawDialog  (msgWindPtr);
      EndUpdate   (msgWindPtr);
  end;    {of ShowPleaseWait}


procedure HidePleaseWait;

  {Removes "Please Wait..." dialog box from the screen.}

  begin   {of HidePleaseWait}
      CloseDialog (msgWindPtr);
      SetPort (origPort);
  end;    {of HidePleaseWait}
```

```
function CheckDiskError (Where : integer) : boolean;

{This routine checks if the last ProDOS operation caused an error.  If so,
 then we change the cursor to the arrow cursor, put up a stop alert
 dialog box with the text of the error message, change the cursor back to
 the wristwatch, and return TRUE as the function result.  If there was no
 disk error, then we simply return with FALSE.}

   var itemClicked : integer;
       ourAlert    : AlertTemplate;
       ourErrStr   : str255;
       ourWhereStr : str255;
       ourString   : str255;
       diskErrNum  : integer;

   begin   {of CheckDiskError}

       diskErrNum     := toolErr;     {Use the std C-like toolerr var for P/16}
       CheckDiskError := (diskErrNum <> 0);            {Assign function result}
       ourErrStr      := 'XXXX';                       {Set string length byte}
       ourWhereStr    := 'XX';                         {Set string length byte}
       if diskErrNum <> 0 then begin
           {*** If desired, get disk err string here}
           Int2Hex (diskErrNum,
                    StringPtr (longint (@ourErrStr) + 1),
                    4);                                {Get ASCII error # str }
           Int2Hex (Where,
                    StringPtr (longint (@ourWhereStr) + 1),
                    2);                                {Get ASCII where # str }
           ourString := concat ('Disk Error $',        {Build our error mesg  }
                                ourErrStr,
                                ' occurred at $',
                                ourWhereStr,
                                '.');
           MakeATemplate (@ourAlert,@ourString);       {Build our alert tmplt }
           InitCursor;                                 {Set arrow cursor      }
           itemClicked := StopAlert (@ourAlert,nil);   {Draw dialog & wait    }
           {Do not restore watch cursor  }
       end;

   end;    {of CheckDiskError}



procedure ManyWindDialog;

   {Displays alert dialog (triangle with "!") with a message about no more
    windows being allowed open.  Handles mouse events until the OK button
    is clicked.  Then the dialog box is removed and we return.}

   var ourAlert    : AlertTemplate;
       ourString   : str255;
       itemClicked : integer;

   begin   {of ManyWindDialog}
       ourString     := 'No more windows, please.';
       MakeATemplate (@ourAlert,@ourString);
       itemClicked   := CautionAlert (@ourAlert,nil);
   end;    {of ManyWindDialog}
```

```
procedure CheckToolError (Where : integer);

   {This routine checks if the last tool called returned an error code.
    If not, then we just return.  Else, we exit to the system death
    handler routine which prints our string showing where we bombed.  The
    death manager adds the tool error code to the end of the string, and
    puts the bouncing apple on the screen.}

   var     toolErrorSave   :   integer;
           deathMsg        :   string;

   begin   {of CheckToolError}
       toolErrorSave := ToolErrorNum;
       deathMsg      := ' At $XXXX; Could not handle error $';

       if toolErrorSave <> 0 then begin
           {Add the hex-in-ascii number to the string:}
           Int2Hex (Where,StringPtr (longint (@deathMsg)+6),4);

           {Halt with our death message string and tool error code:}
           SysFailMgr (toolErrorSave,deathMsg);
       end;
   end;    {of CheckToolError}



function MountBootDisk : integer;

   var
       promptStr   :   string;
       okStr       :   string;
       cancelStr   :   string;
       volStr      :   string;
       gbvParams   :   PathNameRec;

   begin   {of MountBootDisk}
       promptStr := 'Please insert the disk';
       okStr     := 'OK';
       cancelStr := 'Shut Down';
       gbvParams.pathname := @volStr;

       GET_BOOT_VOL (gbvParams);
       MountBootDisk := TLMountVolume (174,30,promptStr,volStr,okStr,cancelStr);
   end;    {of MountBootDisk}



procedure DoAboutItem;

   var aboutDlog  : GrafPortPtr;
       r          : Rect;
       itemHit    : integer;
       appleIconP : Ptr;
       appleIconH : Handle;

   begin   {of DoAboutItem}
       {Draw the dialog box on the screen:}
       SetRect  (r,146,20,495,192);
       aboutDlog := NewModalDialog (r,true,0);

       {Add an OK button to it:}
       SetRect  (r,270,153,0,0);
       NewDItem (aboutDlog,1,r,ButtonItem,@'OK',0,0,nil);

       {Add the Apple logo to it:}
       SetRect     (r,20,135,0,0);
       appleIconP := @AppleIcon;
       appleIconH := @appleIconP;
```

```
    NewDItem (aboutDlog,3,r,IconItem + ItemDisable,appleIconH,0,0,nil);

    {Simply write the text rather than create a bunch of dialog items:}
    SetPort         (aboutDlog);
    SetForeColor    (0);
    SetBAckColor    (15);
    MoveTo          (40,17);
    SetTextFace     (8);
    DrawString      ('                   HodgePodge');
    SetTextFace     (0);
    MoveTo          (40,27);
    DrawString      ('       A potpourri of routines that');
    MoveTo          (40,37);
    DrawString      ('       demonstrate many features of');
    MoveTo          (40,47);
    DrawString      ('             the Apple IIGS Tools.');
    MoveTo          (40,67);
    DrawString      ('     By the Apple IIGS Development Team');
    MoveTo          (36,77);
    DrawString      ('Translated to TML Pascal by TML Systems');
    MoveTo          (40,87);
    DrawString      ('      Copyright Apple Computer, Inc.');
    MoveTo          (40,117);
    DrawString      ('       1986-87, All rights reserved');
    MoveTo          (40,127);
    DrawString      ('              v4.0    23-Sep-87');

    {Let Dialog Manager handle all events until the OK button is clicked:}
    itemHit := ModalDialog (nil);

    {Remove the dialog box from the screen:}
    CloseDialog (aboutDlog);
  end;    {of DoAboutItem}


END.
```

FONT.PAS (fonts)

```
UNIT Font;

{+---------------------------------------------------------------------------+
 |                                                                           |
 |        HodgePodge:  An example Apple IIGS Desktop application             |
 |                                                                           |
 |             Written by the Apple IIGS Development Team                    |
 |             Translated to TML Pascal by TML Systems, Inc.                 |
 |                                                                           |
 |            Copyright (c) 1986-87 by Apple Computer, Inc.                  |
 |                          All Rights Reserved                              |
 |                                                                           |
 |                   --------------------------------                        |
 |                                                                           |
 |        Pascal UNIT "FONT.PAS" : Font window drawing routines              |
 |                                                                           |
 +---------------------------------------------------------------------------+}


INTERFACE

USES
       HPIntfData,          {HodgePodge Apple IIGS Toolbox Interface Units}
       HPIntfProc,
       HPIntfPdos,

       Globals;             {HodgePodge Code Unit}


procedure DispFontWindow;                       {Draw font window contents       }
function  DoChooseFont: boolean;                {Dialog for asking font size, etc.}
procedure DoSetMono;                            {Sets flag and affects menu item  }
procedure ShowFont (theFontID: FontID; isMono: boolean);  {Actually draw font}


IMPLEMENTATION



procedure DispFontWindow;

   {This is a Definition Procedure used to draw the contents of a Font
    window.}

   var tmpPort      : GrafPortPtr;
       myDataHandle : WindDataH;

   begin    {of DispFontWindow}
       tmpPort      := GetPort;
       myDataHandle := WindDataH (GetWRefCon (tmpPort));
       with myDataHandle^^ do
            ShowFont (theFont,isMono);
   end;     {of DispFontWindow}
```

```
function DoChooseFont: boolean;

 {Display the Font Manager's dialog for the user to select a Font,
  font size, and font style.

  The function returns true if a font was chosen, else false if the Cancel
  button is pressed in the dialog.  If a font is chosen, its FontID information
  is returned in the global variable DesiredFont.  In addition, the
  global myReply.filename contains a string which is the font's file name.

  Because the call to ChooseFont actually changes the font of the current
  port, we must first save the current port and open a dummy one do that
  our current port is not affected.}

   var theFont     :    FontID;
       dummy       :    integer;
       tmpPort     :    GrafPortPtr;
       tmpPortRec  :    GrafPort;
       famName     :    Str255;

   begin   {of DoChooseFont}
       tmpPort := GetPort;
       OpenPort (@tmpPortRec);            {Save current port and open new one}

       theFont := ChooseFont (DesiredFont,0);  {Do standard dialog box}

       if longint (theFont) = 0 then      {Cancel was chosen}
           DoChooseFont := false
       else begin
           DesiredFont := theFont;        {Update global DesiredFont}
           dummy := GetFamInfo (DesiredFont.famNum,famName);
           myReply.filename :=
               concat (famName,
                       ' ',
                       IntToString (DesiredFont.fontSize));
           DoChooseFont := true;
       end;

       ClosePort (@tmpPortRec);
       SetPort (tmpPort);                  {Restore current port}

   end;    {of DoChooseFont}



procedure DoSetMono;

   {This procedure flips the flag indicating whether we are currently
    displaying a font in mono-spacing or not, and updates the
    font menu item accordingly.}

   begin   {of DoSetMono}
       if isMonoFont then
           SetMItem (MonoStr,MonoItem)
       else
           SetMItem (ProStr,MonoItem);
       isMonoFont := not isMonoFont;
   end;    {of DoSetMono}
```

```
procedure ShowFont (theFontID: FontID; isMono: boolean);

   var fontInfo    : FontInfoRecord;
       currHeight : integer;
       i,j         : integer;
       theCh       : integer;
       currPt      : Point;
       fontStr     : Str255;

   begin   {of ShowFont}
       InstallFont (theFontID,0);
       GetFontInfo (fontInfo);
       currHeight := fontInfo.ascent + fontInfo.descent + fontInfo.leading;

       i := GetFamInfo (theFontID.famNum,fontStr);
       fontStr := concat (fontStr,' ',IntToString (theFontID.fontSize));

       i := GetFontFlags;
       if isMono then
           i := BitOr  (i,$0001)           {Set bottom bit}
       else
           i := BitAnd (i,$0000);          {Clear bottom bit}
       SetFontFlags(i);

       MoveTo     (5,currHeight);
       DrawString (fontStr);

       MoveTo     (5,currHeight * 3);
       DrawString ('The quick brown fox jumps over the lazy dog.');
       MoveTo     (5,currHeight * 4);
       DrawString ('She sells sea shells down by the sea shore.');

       MoveTo     (5,currHeight * 5);

       for i := 0 to 7 do begin
           GetPen (currPt);
           MoveTo (5,currPt.v + currHeight);
           theCh := i * 32;
           for j := 1 to 32 do begin
               fontStr [j] := chr (theCh);
               inc (theCh);
           end;
           fontStr [0] := chr (32);
           DrawString (fontStr);
       end;
   end;    {of ShowFont}


END.
```

PRINT.PAS (printing)

```
UNIT Print;

{+-----------------------------------------------------------------------------+
 |                                                                             |
 |        HodgePodge:  An example Apple IIGS Desktop application              |
 |                                                                             |
 |             Written by the Apple IIGS Development Team                     |
 |             Translated to TML Pascal by TML Systems, Inc.                  |
 |                                                                             |
 |            Copyright (c) 1986-87 by Apple Computer, Inc.                   |
 |                           All Rights Reserved                              |
 |                                                                             |
 |                   --------------------------------                         |
 |                                                                             |
 |       Pascal UNIT "PRINT.PAS" : Window content printing routines           |
 |                                                                             |
 +-----------------------------------------------------------------------------+}


INTERFACE

USES
       HPIntfData,           {HodgePodge Apple IIGS Toolbox Interface Units}
       HPIntfProc,
       HPIntfPdos,

       Globals,              {HodgePodge Code Units}
       Dialog,
       Font,
       Paint;


procedure DoChooserItem;    {Show standard chooser dialog to select options}
procedure DoSetupItem;      {Show standard page setup dialog to sel options}
procedure DoPrintItem;      {Print contents of current window to printer   }
procedure SetUpDefault;     {Create and initialize THPrint record          }


IMPLEMENTATION



var  printHndl:     PrRecHndl;  {Private print record handle for Print Manager}


procedure DoChooserItem;

   {Display the Chooser Dialog for the user to select which printer and
    printer connection to use.}

   var dummy: boolean;

   begin   {of DoChooserItem}
       dummy := PrChooser;
   end;    {of DoChooserItem}
```

```
  procedure DoPrintItem;

     {Print the contents of the front window to the selected printer.}

     var  prPort     : GrafPortPtr;
          theWindow : GrafPortPtr;


     procedure DrawTopWindow (theWindow: GrafPortPtr);

        {This private procedure determines what type of window theWindow is and
         calls the appropriate procedure to draw its contents to the current port
         which is now the printer port created in DoPrintItem.}

        var myDataHandle: WindDataH;

        begin   {of DrawTopWindow}
            myDataHandle := WindDataH (GetWRefCon (theWindow));
            with myDataHandle^^ do
                if Flag = 0 then
                    PaintIt (pict)
                else
                    ShowFont (theFont,isMono);
        end;    {of DrawTopWindow}


     begin   {of DoPrintItem}
         theWindow := FrontWindow;
         if theWindow <> nil then
             if PrJobDialog (printHndl) then begin
                 WaitCursor;
                 prPort := PrOpenDoc (printHndl,nil);
                 PrOpenPage          (prPort,nil);
                 DrawTopWindow       (theWindow);
                 PrClosePage         (prPort);
                 PrCloseDoc          (prPort);
                 PrPicFile           (printHndl,nil,nil);
                 InitCursor;
             end;
     end;    {of DoPrintItem}


  procedure DoSetupItem;

     {Display the Page Setup dialog for the user to choose the print mode,
      number of pages, etc. to print.}

     var dummy: boolean;

     begin   {of DoSetupItem}
         dummy := PrStlDialog (printHndl);
     end;    {of DoSetupItem}


  procedure SetUpDefault;

     {Create and initialize a THPrint record which is required for all
      printing operations.}

     begin   {of SetUpDefault}
         printHndl := PrRecHndl (NewHandle (140,
                                 myMemoryID,
                                 attrNoCross + attrLocked,
                                 Ptr (0)));
         PrDefault (printHndl);
     end;    {of SetUpDefault}

END.
```

PAINT.PAS (pictures and files)

```
UNIT Paint;

{+--------------------------------------------------------------------------+
 |                                                                          |
 |        HodgePodge:  An example Apple IIGS Desktop application            |
 |                                                                          |
 |             Written by the Apple IIGS Development Team                   |
 |             Translated to TML Pascal by TML Systems, Inc.                |
 |                                                                          |
 |            Copyright (c) 1986-87 by Apple Computer, Inc.                 |
 |                          All Rights Reserved                             |
 |                                                                          |
 |                    -------------------------------                       |
 |                                                                          |
 |  Pascal UNIT "PAINT.PAS" : Bitmapped picture load/save and window drawing |
 |                                                                          |
 +--------------------------------------------------------------------------+}


INTERFACE

USES
       HPIntfData,          {HodgePodge Apple IIGS Toolbox Interface Units}
       HPIntfProc,
       HPIntfPdos,

       Globals,             {HodgePodge Code Units}
       Dialog;


function  AskUser : boolean;                     {Load a new picture from disk}
procedure DoSaveItem;       {If paint window in front, do Std File dialog & save}
procedure Paint;                                 {Draw picture window contents}
procedure PaintIt (pict: Handle);                       {Do Paint's dirty work}



IMPLEMENTATION


{$DefProc }
function OpenFilter (DirEntry : longint) : integer;

   {Filter function called by the Standard File Operations' SFGetFile
    dialog to determine whether a filename should be dimmed or not.}

   type
     BytePtr = ^byte;
   var
     fileTypePtr : BytePtr;

   begin   {of OpenFilter}
       fileTypePtr := Pointer (DirEntry + $10);
       if (BitAND(FileTypePtr^,$00FF) = $C1) then { Unpacked Picture File type }
           OpenFilter := 2
       else
           OpenFilter := 1;
   end;    {of OpenFilter}
```

```
function AskUser : boolean;

  var ourTypeList : TypeListPtr;



  function  LoadOne : boolean;

    {Private procedure which actually loads a picture from disk}

    var openBlk : OpenRec;
        readBlk : FileIORec;

    begin   {of LoadOne}
        LoadOne := false;

        WaitCursor;
        PictHndl := NewHandle ($8000,
                               MyMemoryID,
                               0,
                               Ptr (0));
        if isToolError then
            Exit;

        HLock (PictHndl);

        openBlk.openPathname := @myReply.fullpathname;
        openBlk.ioBuffer     := nil;
        OPEN (openBlk);
        if CheckDiskError (27) then
            Exit;

        readBlk.dataBuffer   := PictHndl^;
        readBlk.requestCount := $8000;
        readBlk.fileRefNum   := openBlk.openRefNum;
        READ (readBlk);
        if CheckDiskError (28) then
           Exit;

        CLOSE (readBlk);
        HUnLock  (PictHndl);

        LoadOne := true;
    end;    {of LoadOne}



 begin   {of AskUser}

    SFGetFile (20,
               20,
               'Load which picture:',
               @OpenFilter,
               NIL,
               MyReply);

    AskUser := false;
    if myReply.good then
        if LoadOne then
            AskUser := true;
 end;    {of AskUser}
```

```
procedure DoSaveItem;

   {This procedure is called to save the contents of a "Paint" window
    to a disk file.  NOTE:  This routine is ONLY called when a "Paint"
    window is in front due to enabling/disabling the menu items.}

   var theWindow:    GrafPortPtr;
       myDataHandle: WindDataH;
       i:            integer;


   procedure SaveOne (pict: Handle);

      {Private procedure which actually does the picture save}

      var destroyBlk  :  PathNameRec;
          createBlk   :  FileRec;
          openBlk     :  OpenRec;
          writeBlk    :  FileIORec;

      begin   {of SaveOne}
          destroyBlk.pathname := @myReply.fullpathname;
          DESTROY (destroyBlk);

          createBlk.pathname    := @myReply.fullpathname;
          createBlk.fAccess     := $C3;      {DRbWR, see ProDOS16 docs}
          createBlk.fileType    := $C1;      {Unpacked file}
          createBlk.auxType     := 0;        {-nothing-}
          createBlk.storageType := 1;        {Seedling file}
          createBlk.createDate  := 0;
          createBlk.createTime  := 0;
          CREATE (createBlk);
          if CheckDiskError (25) then
              Exit;

          openBlk.openPathname := @myReply.fullpathname;
          openBlk.ioBuffer     := nil;
          OPEN (openBlk);

          writeBlk.dataBuffer   := pict^;
          writeBlk.requestCount := $8000;
          writeBlk.fileRefNum   := openBlk.openRefNum;
          WRITE (writeBlk);
          if CheckDiskError (26) then
              Exit;

          CLOSE (writeBlk);
      end;    {of SaveOne}


   begin   {of DoSaveItem}
       theWindow := FrontWindow;
       myDataHandle := WindDataH (GetWRefCon (theWindow));
       SFPutFile (20,
                  20,
                  'Save which picture:',
                  myDataHandle^^.Name,
                  15,
                  myReply);

       if myReply.good then begin
           WaitCursor;
           SaveOne (myDataHandle^^.pict);
           with myDataHandle^^ do begin
               {Change name of correct menu item:}
               Name := myReply.filename;
```

```
                  MenuStr := concat ('=',
                                     myReply.filename,
                                     '\N',
                                     IntToString (MenuID),
                                     '\0.');

                  for i := FirstWind to LastWind do
                      if WindowList [i] = theWindow then
                          Leave;  {Exit loop}
                  SetMItem (MenuStr,FirstWindItem + i);  {New menu name}
              end;

              {Set window title to field in refcon, NOT to myReply.filename!!}
              SetWTitle (myDataHandle^^.Name,theWindow);

              CalcMenuSize (0,0,WindowsMenuID);
              InitCursor;
          end;

      end;    {of DoSaveItem}


  procedure Paint;

      {This is a Definition Procedure used to draw the contents of a Font window}

      var tmpPort      : GrafPortPtr;
          myDataHandle : WindDataH;

      begin   {of Paint}
          tmpPort      := GetPort;
          myDataHandle := WindDataH (GetWRefCon (tmpPort));
          PaintIt (myDataHandle^^.pict);
      end;    {of Paint}


  procedure PaintIt (pict: Handle);

      {Procedure to actually draw the picture in memory to the window.}

      var srcLoc  : LocInfo;
          srcRect : Rect;

      begin   {of PaintIt}
          HLock (pict);

          with srcLoc do begin
              portSCB             := $0080;
              ptrToPixImage       := pict^;
              lwidth              := 160;
              SetRect             (boundsRect,0,0,640,200);
              BoundsRect.v1       := 0;
          end;

          SetRect  (srcRect,0,0,640,200);
          PPToPort (srcLoc,
                    srcRect,
                    0,
                    0,
                    srcCopy);

          HUnLock (pict);
      end;    {of PaintIt}

END.
```

GLOBALS.PAS (global data)

```
UNIT Globals;

{+--------------------------------------------------------------------------+
 |                                                                          |
 |        HodgePodge:  An example Apple IIGS Desktop application           |
 |                                                                          |
 |             Written by the Apple IIGS Development Team                   |
 |             Translated to TML Pascal by TML Systems, Inc.                |
 |                                                                          |
 |            Copyright (c) 1986-87 by Apple Computer, Inc.                 |
 |                          All Rights Reserved                             |
 |                                                                          |
 |                  --------------------------------                        |
 |                                                                          |
 |     Pascal UNIT "GLOBALS.PAS" : Global data structs and init routine     |
 |                                                                          |
 +--------------------------------------------------------------------------+}


INTERFACE

USES
       HPIntfData,          {HodgePodge Apple IIGS Toolbox Interface Units}
       HPIntfProc,
       HPIntfPdos;

const
       ScreenMode          = $80;   {640 mode}
       MaxX                = 640;   {Max X clamp (should correspond to ScreenMode}
       MaxScan             = 160;   {Max size of scan line}

       AppleMenuID         = 300;
           AboutItem       = 301;
       FileMenuID          = 400;
           OpenItem        = 401;
           CloseItem       = 255;   {For DA's}
           SaveAsItem      = 403;
           ChoosePItem     = 405;
           PageSetItem     = 406;
           PrintItem       = 407;
           QuitItem        = 409;
       EditMenuID          = 500;
           UndoItem        = 250;    {For DA's}
           CutItem         = 251;    {For DA's}
           CopyItem        = 252;    {For DA's}
           PasteItem       = 253;    {For DA's}
           ClearItem       = 254;    {For DA's}
       WindowsMenuID       = 600;
           NoWindowsItem = 601;
           FirstWindItem = 2000;     {Allocated dynamically starting at 2000}
       FontsMenuID         = 700;
           FontItem        = 701;
           MonoItem        = 702;

       FirstWind           = 0;      {Lower bound of WindowList}
       LastWind            = 15;     {Upper bound of WindowList}
```

```
type
    WindDataH   = ^WindDataP;              {RefCon data for our windows}
    WindDataP   = ^WindDataRec;            {...carried along with wind data}
    WindDataRec = record
                    Name:    Str255;
                    MenuStr: Str255;
                    MenuID:  integer;
                    Flag:    integer;             {0 = Paint, 1 = Font}
                    case integer of
                        0     : (theFont: FontID;
                                  isMono :  boolean);
                        1     : (pict:    Handle);
                  end;


var
    MyMemoryID     : integer;       {Application ID assigned by Memory Mgr}
    Done           : boolean;       {True when quitting}
    ToolsZeroPage  : Handle;        {Handle to Zero page memory for Tools}
    Event          : WmTaskRec;     {All events are returned here}

    AppleMenuStr   : Str255;        {For creating menus}
    FileMenuStr    : Str255;
    EditMenuStr    : Str255;
    WindowMenuStr  : Str255;
    FontMenuStr    : Str255;

    NoWindStr      : String [40];   {Menu Item String for "No Windows..."}
    MonoStr        : String [40];
    ProStr         : String [40];

    LastWindow     : GrafPortPtr;   {The Front Window last time through event l
    dummy          : integer;       {*** >!< Possible compiler bug?}
    DesiredFont    : FontID;        {Indicates current selected font}
    isMonoFont     : boolean;       {Flag indicates if mono spacing selected}
    myReply        : SFReplyRec;
    PictHndl       : Handle;

    WIndex         : integer;       {Count of number of windows open}
    WindowList     :                {List of up to 15 open windows}
      array [firstWind..lastWind] of GrafPortPtr;

    PlsWtTemp      : DialogTemplate;
    PlsWtItem      : ItemTemplate;

    AppleIcon      : record
                        boundsRect : Rect;
                        data       : array [1..34] of
                                        packed array [1..16] of byte;
                     end;


procedure InitGlobals;                                    {Setup variables}
PROCEDURE HPStuffHex (thingPtr : Ptr; s : Str255);        {Store hex}
```

```
IMPLEMENTATION



PROCEDURE HPStuffHex (thingPtr : Ptr; s : Str255);

   {>!< This routine will be implemented in TML Pascal V1.1.  For now,
    we define it ourselves.  StuffHex stores bytes (expressed as a string
    of hexadecimal digits) into any data structure, and is based on the
    StuffHex procedure in Macintosh QuickDraw.  The resolution of this
    routine is on byte boundaries.}

   var iterator    : integer;
       stringIndex : integer;

   begin   {of HPStuffHex}
       for iterator := 0 to Length (s) - 1 do begin
           stringIndex := (iterator * 2) + 1;
           thingPtr^ := Hex2Int (StringPtr (longint (@s) + stringIndex),2);
           thingPtr := pointer (longint (thingPtr) + 1);
       end;
   end;    {of HPStuffHex}



procedure InitGlobals;

   {Initialize global data variables, including the PlsWtTemp used by
    ShowPleaseWaitDialog, the menu strings used by the menu bar setup
    routines in MENU.PAS, and the apple icon used by the "about..."
    item dialog routine in DIALOG.PAS}

   begin   {of InitGlobals}
       with PlsWtTemp do begin
           SetRect     (dtBoundsRect,120,30,520,80);
           dtVisible  := true;
           dtRefCon   := 0;
           dtItemList [0] := pointer (0); {We will insert ptr to item here}
           dtItemList [1] := nil;         {Null-terminated}
       end;

       AppleMenuStr := concat ('>>@\N300X\0',
                               '==About HodgePodge...\N301\0',
                               '==-\N302D\0.');
       FileMenuStr  := concat ('>>  File  \N400\0',
                               '==Open...\N401*Oo\0',
                               '==Close\N255D\0',
                               '==Save As...\N403D\0',
                               '==-\N404D\0',
                               '==Choose Printer...\N405\0',
                               '==Page Setup...\N406D\0',
                               '==Print...\N407*PpD\0',
                               '==-\N408D\0',
                               '==Quit\N409*Qq\0.');
       EditMenuStr  := concat ('>>  Edit  \N500D\0',
                               '==Undo\N250*Zz\0',
                               '==-\N501D\0',
                               '==Cut\N251*Xx\0',
                               '==Copy\N252*Cc\0',
                               '==Paste\N253*Vv\0',
                               '==Clear\N254\0.');
       WindowMenuStr := concat('>>  Window  \N600D\0',
                               '== No Windows Allocated\N601D\0.');
       FontMenuStr   := concat ('>>  Fonts  \N700\0',
                               '==Display Font...\N701*Ff\0',
                               '==Display Font as Mono-spaced\N702*Mm\0.');

       LastWindow    := nil;
```

```
    NoWindStr      := '==No Windows Allocated\N601D\0.';

    MonoStr        := '=Display Font as Mono-spaced';
    ProStr         := '=Display Font as Proportional';
    isMonoFont     := false;

    with DesiredFont do begin
        famNum     := $FFFE;
        fontStyle  := 0;
        fontSize   := 8;
    end;

    WIndex := 0;                        {No windows open yet}

    SetRect    (AppleIcon.boundsRect,0,0,64,34);
    HPStuffHex (@AppleIcon.data[1], '00000000000000000000000000000000');
    HPStuffHex (@AppleIcon.data[2], '0FFFFFFFFFFFFFFFFFFFFFFFFFFFFFF0');
    HPStuffHex (@AppleIcon.data[3], '0F0000000000000000000000000000F0');
    HPStuffHex (@AppleIcon.data[4], '0F0FFFFFFFFFFFFFFFFFFFFFFFFFF0F0');
    HPStuffHex (@AppleIcon.data[5], '0F0FFFFFFFFFFFFFFFFFF88FFFFFF0F0');
    HPStuffHex (@AppleIcon.data[6], '0F0FFFFFFFFFFFFFFFF8888FFFFFF0F0');
    HPStuffHex (@AppleIcon.data[7], '0F0FFFFFFFFFFFFFFF88888FFFFFF0F0');
    HPStuffHex (@AppleIcon.data[8], '0F0FFFFFFFFFFFFFF88888FFFFFFF0F0');
    HPStuffHex (@AppleIcon.data[9], '0F0FFFFFFFFFFFFF888888FFFFFFF0F0');
    HPStuffHex (@AppleIcon.data[10],'0F0FFFFFFFFFFFFF88888FFFFFFFF0F0');
    HPStuffHex (@AppleIcon.data[11],'0F0FFFFFFFFFFFFF8888FFFFFFFFF0F0');
    HPStuffHex (@AppleIcon.data[12],'0F0FFFFFF8888FFF88FF8888FFFFF0F0');
    HPStuffHex (@AppleIcon.data[13],'0F0FFF888888888FFF88888888FFF0F0');
    HPStuffHex (@AppleIcon.data[14],'0F0FFF888888888888888888888FF0F0');
    HPStuffHex (@AppleIcon.data[15],'0F0FFeeeeeeeeeeeeeeeeeeeeFFFF0F0');
    HPStuffHex (@AppleIcon.data[16],'0F0FFeeeeeeeeeeeeeeeeeeeFFFFF0F0');
    HPStuffHex (@AppleIcon.data[17],'0F0FFeeeeeeeeeeeeeeeeeeFFFFFF0F0');
    HPStuffHex (@AppleIcon.data[18],'0F0FF666666666666666666FFFFFF0F0');
    HPStuffHex (@AppleIcon.data[19],'0F0FF666666666666666666FFFFFF0F0');
    HPStuffHex (@AppleIcon.data[20],'0F0FF666666666666666666FFFFFF0F0');
    HPStuffHex (@AppleIcon.data[21],'0F0FF4444444444444444444FFFFF0F0');
    HPStuffHex (@AppleIcon.data[22],'0F0FF44444444444444444444FFFF0F0');
    HPStuffHex (@AppleIcon.data[23],'0F0FFF444444444444444444444FF0F0');
    HPStuffHex (@AppleIcon.data[24],'0F0FFF555555555555555555555FF0F0');
    HPStuffHex (@AppleIcon.data[25],'0F0FFF555555555555555555555FF0F0');
    HPStuffHex (@AppleIcon.data[26],'0F0FFFF5555555555555555555FFF0F0');
    HPStuffHex (@AppleIcon.data[27],'0F0FFFF1111111111111111111FFF0F0');
    HPStuffHex (@AppleIcon.data[28],'0F0FFFFF11111111111111111FFFF0F0');
    HPStuffHex (@AppleIcon.data[29],'0F0FFFFFF111111FFF111111FFFFF0F0');
    HPStuffHex (@AppleIcon.data[30],'0F0FFFFFFF1111FFFFF1111FFFFFF0F0');
    HPStuffHex (@AppleIcon.data[31],'0F0FFFFFFFFFFFFFFFFFFFFFFFFFF0F0');
    HPStuffHex (@AppleIcon.data[32],'0F0000000000000000000000000000F0');
    HPStuffHex (@AppleIcon.data[33],'0FFFFFFFFFFFFFFFFFFFFFFFFFFFFFF0');
    HPStuffHex (@AppleIcon.data[34],'00000000000000000000000000000000');

  end;    {of InitGlobals}

END.
```

Glossary

absolute: Characteristic of a load segment or other program code that must be loaded at a specific address in memory and never moved. Compare **relocatable, position-independent.**

absolute addressing: an addressing mode in which instruction operands are interpreted as literal addresses.

access (or **access byte):** An attribute of a ProDOS file that controls whether the file may be read from, written to, renamed, or backed up.

accumulator: The register in a computer's central processor or microprocessor where most computations are performed.

activate: To make active. A control or window may be activated. Compare **enable.**

activate event: a window event that occurs when a window is made either active or inactive.

active: Able to respond to the user's mouse or keyboard actions. Controls and windows that are active are displayed differently from **inactive** items.

ADB: See **Apple Desktop Bus.**

address bus: The bus that carries addresses from the CPU to components under its control.

advanced linker (APW): One aspect of the linker supplied with APW. The operation of the advanced linker is programmable. Compare **standard linker.**

alert: A warning or report of an error in the form of an alert box, a sound from the computer's speaker, or both.

alert box: A special type of dialog box that appears on the screen to give a warning or to report an error message during use of an application.

alert window: The window in which an alert box appears. One of the two predefined window formats. Compare **document window.**

analog RGB: A type of color video consisting of separate analog signals from the red, green, and blue color primaries. The intensity of each primary can vary continuously, making possible many shades and tints of colors. Compare **TTL RGB.**

Apple Desktop Bus (ADB): An input bus, with its own protocol and electrical characteristics, that provides a method of connecting input devices such as keyboards and mouse devices to personal computers.

Apple Desktop Bus Tool Set: The Apple IIGS tool set that facilitates an application's interaction with devices connected to the Apple Desktop Bus.

Apple key: A modifier key on the Apple IIGS keyboard, marked with both an Apple icon and a *spinner*, the icon used on the equivalent key on some Macintosh keyboards. It performs the same functions as the Open Apple key on standard Apple II machines.

AppleTalk network: A local area network developed by Apple Computer, Inc.

Apple II: A family of computers, including the original Apple II, the Apple II Plus, the Apple IIe, the Apple IIc, and the Apple IIGS. Compare **standard Apple II.**

Apple IIc: A transportable personal computer in the Apple II family, with a disk drive, serial ports, and 80-column display capability built in.

Apple IIe: A personal computer in the Apple II family with seven expansion slots and an auxiliary memory slot that allow the user to enhance the computer's capabilities with peripheral memory and video enhancement cards.

Apple IIGS: The most advanced computer in the Apple II family. It features expanded memory, advanced sound and graphics, and the **Apple IIGS Toolbox** of programming routines.

Apple IIGS Debugger: A 65816 machine language code debugger for the Apple IIGS computer.

Apple IIGS Programmer's Workshop (APW): A multilanguage development environment for writing Apple IIGS desktop applications.

Apple IIGS Toolbox: An extensive set of routines that facilitate writing desktop applications and provide easy program access to many Apple IIGS hardware and firmware features.

Apple II Plus: A personal computer in the Apple II family with expansion slots that allow the user to enhance the computer's capabilities with peripheral cards.

application: A stand-alone program that performs a specific function, such as word-processing, drawing, or telecommunications. Compare, for example, **desk accessory** or **device driver.**

application-defined event: Any of four types of events available for applications to define and respond to as desired.

application prefix: The ProDOS 16 **prefix** number 1/. It specifies the directory of the currently running application.

application window: A window in which an application's document appears.

APW: See **Apple IIGS Programmer's Workshop.**

APW Assembler: The 65816 assembly-language assembler provided with the Apple IIGS Programmer's Workshop.

APW C Compiler: The C-language compiler provided with the Apple IIGS Programmer's Workshop.

APW Editor: The program within the Apple IIGS Programmer's Workshop that allows you to enter, modify, and save source files for all APW languages.

APW Linker: The linker supplied with the Apple IIGS Programmer's Workshop.

APW Shell: The programming environment of the Apple IIGS Programmer's Workshop—it provides facilities for file manipulation and program execution, and supports shell applications.

APW utility program: Any of various Shell applications supplied with the Apple IIGS Programmer's Workshop that function as APW Shell commands.

arc: A portion of an **oval;** one of the fundamental shapes drawn by QuickDraw II.

A register: See **accumulator.**

ascent: In a font, the distance between the base line and the ascent line.

ascent line: A horizontal line that coincides with the tops of the tallest characters in a font. See also **base line, descent line.**

ASCII: Acronym for *American Standard Code for Information Interchange,* pronounced "ASK-ee." A code in which the numbers from 0 to 127 stand for text characters. ASCII code is used to represent text inside a computer and to transmit text between computers or between a computer and a peripheral device.

assembler: A language translator that converts a program written in assembly language into an equivalent program in machine language. The opposite of a **disassembler.**

attributes word: Determines how memory blocks are allocated and maintained. Most of the attributes are defined at allocation time and can't be changed after that; other attributes can be modified after allocation.

auto-key: A keyboard feature and an event type, in which a key being held down continuously is interpreted as a rapid series of identical keystrokes.

auxID: A subfield of the **User ID.** An application may place any value it wishes into the auxID field.

auxiliary type: A secondary classification of ProDOS files. A file's auxiliary type field may contain information of use to the applications that read it. Compare **file type.**

background: The pixels within a character or other screen object that are not part of the object itself.

background color: The color of background pixels in text; by default it is black.

background pattern: The pattern QuickDraw II uses to erase objects on the screen.

background pixels: In a character image, the pixels that are not part of the character itself.

background procedure: A procedure run by the Print Manager whenever the Print Manager has directed output to the printer and is waiting for the printer to finish.

backup bit: A bit in a file's access byte that tells backup programs whether the file has been altered since the last time it was backed up.

bank: A 64K (65,536-byte) portion of the Apple IIGS internal memory. An individual bank is specified by the value of the 65816 microprocessor's bank register.

bank-switched memory: On Apple II computers, the part of **language card** memory in which two 4K portions of memory share the same address range ($D000 to $DFFF).

bank $00: The first bank of memory in the Apple IIGS. In emulation mode, it is equivalent to *main memory* in an Apple IIe or Apple IIc computer.

base line: A horizontal line that coincides with the bottom of the main body of each character in a font. Character **descenders** extend below the base line.

BASIC: Acronym for *Beginners All-purpose Symbolic Instruction Code*. BASIC is a high-level programming language designed to be easy to learn. Applesoft BASIC is built into the Apple IIGS firmware.

batch: A mode of executing a computer program in which all code and data required by the program are loaded into the computer at the beginning, the program is run, and all results are output at the end. Batch mode is non-interactive.

binary file: (1) A file whose data is to be interpreted in binary form. Machine-language programs and pictures are stored in binary files. Compare **text file.** (2) A file in binary file format.

binary file format: The ProDOS 8 loadable file format, consisting of one absolute memory image along with its destination address. A file in binary file format has ProDOS file type $06 and is referred to as a BIN file. The System Loader cannot load BIN files.

bit: A contraction of *binary digit*, the smallest representation of data in a digital computer.

bit plane: A method of representing images in computer memory. In a bit plane, consecutive bits in memory specify adjacent pixels in the image; if more than one bit is required to completely specify the state of a pixel, more than one bit plane is used for the image. Compare **chunky pixels.**

block: (1) A unit of data storage or transfer, typically 512 bytes. (2) A contiguous region of computer memory of arbitrary size, allocated by the Memory Manager. Also called a *memory block.*

block device: A device that transfers data to or from a computer in multiples of one block (512 bytes) of characters at a time. Disk drives are block devices. Also called *block I/O device.*

Boolean logic: A mathematical system in which every expression evaluates to one of two values, usually referred to as TRUE or FALSE.

Boolean variable: A variable that can have one of two values, usually referred to as TRUE or FALSE.

boot prefix: The ProDOS 16 **prefix** number */. It specifies the name of the volume from which the currently running version of ProDOS 16 was started up.

boundary rectangle: A rectangle, defined as part of a QuickDraw II **LocInfo** record, that encloses the active area of the pixel image and imposes a coordinate system on it. Its upper-left corner is always aligned on the first pixel in the pixel map.

boundsRect: The GrafPort field that defines the port's boundary rectangle.

breakpoint: A machine-language instruction in a program that causes execution to halt.

buffer: A holding area of the computer's memory where information can be stored by one program or device and then read, perhaps at a different rate, by another; for example, a print buffer.

Busy flag: A feature that informs the Scheduler whether a currently needed resource is busy or available.

button: (1) A pushbutton-like image in a dialog box where the user clicks to designate, confirm, or cancel an action. Compare **radio button, check box.** (2) A button on a mouse or other pointing device.

byte: A unit of information consisting of eight bits. A byte can have any value between 0 and 255, which may represent an instruction, letter, number, punctuation mark, or other character. See also **bit, kilobyte, megabyte.**

C: A high-level programming language. One of the languages available for the Apple IIGS Programmer's Workshop.

cancel: To stop an operation, such as the setting of page-setup values in a dialog box, without saving any results produced up to that point.

Cancel: One of two predefined item ID numbers for dialog box buttons (Cancel = 2). Compare **OK.**

card: See **peripheral card.**

caret: A symbol that indicates where something should or will be inserted in text. On the screen it designates the **insertion point,** and is usually a vertical bar (|).

carry flag: A status bit in the microprocessor indicating whether an accumulator calculation has resulted in a carry out of the register.

CDA: See **classic desk accessory.**

c flag: See **carry flag.**

character: Any symbol that has a widely understood meaning and thus can convey information. Some characters—such as letters, numbers, and punctuation—can be displayed on the monitor screen and printed on a printer. Most characters are represented in the computer as one-byte values.

character device: A device that transfers data to or from a computer as a stream of individual characters. Keyboards and printers are character devices.

character image: An arrangement of bits that defines a character in a font.

character origin: The point on the **base line** used as a reference location for drawing a character.

character width: The number of pixels the pen position is to be advanced after the character is drawn.

check box: A small box associated with an option in a dialog box. When the user clicks the check box, that may change the option or affect related options.

Choose Printer: A part of the Print Manager that lets the user select a printer or port for printing.

chunkiness: The number of bits required to describe the state of a pixel in a pixel image.

chunky pixels: A method of representing images in computer memory. In chunky pixel organization, a number of consecutive bits in memory combine to specify the state of a single pixel in the image. Consecutive *groups* of bits (the size of the group is equal to the image's **chunkiness**) define adjacent pixels in the image. Compare **bit plane.**

clamp values: The x- and y-limits, in terms of pixels, on cursor position controlled by mouse movement.

classic desk accessory (CDA): Desk accessories designed to execute in a non-desktop, non-event-based environment. Compare **new desk accessory.**

click: To position the pointer on something, and then to press and quickly release the button on the mouse or other pointing device.

clip: To restrict drawing to within a particular boundary; any drawing attempted outside that boundary does not occur.

Clipboard: The holding place for what the user last cut or copied; a buffer area in memory. Information on the Clipboard can be inserted (pasted) into documents. In memory, the contents of the Clipboard are called the **desk scrap.**

clipping region: The region to which an application limits drawing in a GrafPort.

clock: (1) The timing circuit that controls execution of a microprocessor. Also called the *system clock*. (2) An integrated circuit, often with battery-backup memory, that gives the current date and time. Also called the *clock-calendar*.

clock speed: The frequency of the system clock signal in megahertz.

close box: The small white box on the left side of the title bar of an active window. Clicking it closes the window.

CMOS: Abbreviation for *complementary metal-oxide semiconductor*, one of several methods of making integrated circuits out of silicon. CMOS devices are characterized by their low power consumption. CMOS techniques are derived from MOS techniques.

color table: One of 16 possible lookup tables in Apple IIGS memory, that lists the available color values for a scan line.

command line: (1) In APW, the line of text with which the user invokes a procedure or function or executes a program. The command line often includes both the name of the function to execute and a list of parameters to be passed to the function. (2) The line on the screen on which a command is entered.

command-line interface: The type of interface between user and program in which information is passed in a command line.

compaction: The rearrangement of allocated blocks in memory to open up larger contiguous areas of free space.

compiler: A program that produces object files (containing machine-language code) from source files written in a high-level language such as C. Compare **assembler.**

content region: The area in a window in which an application presents information to the user.

control: An object in a window with which the user, using the mouse, can cause instant action with visible results or change settings to modify a future action.

controlling program: A program that loads and runs other programs, without itself leaving memory. A controlling program is responsible for shutting down its subprograms and freeing their memory space when they are finished. A shell, for example, is a controlling program.

Control Manager: The Apple IIGS tool set that manages **controls.**

Control Panel: A desk accessory that lets the user change certain system parameters, such as speaker volume, display colors, and configuration of slots and ports.

coordinate plane: A two-dimensional grid defined by QuickDraw II. All drawing commands are located in terms of coordinates on the grid.

coordinates: X-Y locations on the QuickDraw II coordinate plane. Most QuickDraw routines accept and return coordinates in the order (Y,X).

copy: To duplicate something by selecting it and choosing Copy from the Edit menu. A copy of the selected portion is placed on the Clipboard, without affecting the original selection.

creation date: An attribute of a ProDOS file; it specifies the date on which the file was first created.

creation time: An attribute of a ProDOS file; it specifies the time at which the file was first created.

C string: An ASCII character string terminated by a null character (ASCII value = 0).

cursor: A symbol displayed on the screen marking where the user's next action will take effect or where the next character typed from the keyboard will appear.

cut: To remove something by selecting it and choosing Cut from the Edit menu. The cut portion is placed on the Clipboard.

data area: A document as viewed in a window. The data area is the entire document, only a portion of which (the **visible region**) may be seen in the window at any one time.

data bank register: A register in the 65816 processor that contains the high-order byte of the 24-bit address that references data in memory.

data bus: A set of the electrical conductors that carry data from one internal part of the computer to another.

data structure: A specifically formatted item of data or a form into which data may be placed.

DB register: See **data bank register.**

debugger: A utility used for software development that allows you to analyze a program for errors that cause it to malfunction. For example, it may allow you to step through execution of the program one instruction at a time.

default prefix: The pathname prefix attached by ProDOS 16 to a partial pathname when no prefix number is supplied by the application. The default prefix is equivalent to prefix number 0/.

definition procedure: A routine that defines the characteristics of some desktop feature such as a window or control. For example, TaskMaster needs a pointer to a *window-content definiton procedure* (`wContDefProc`) in order to draw the contents of windows that it manipulates.

DefProc: See **definition procedure.**

dereference: To substitute a pointer for a memory handle, or a value for a pointer. When you dereference a memory block's handle, you access the block directly (through its master pointer) rather than indirectly (through its handle).

descender: Any part of a character that lies below the base line (such as the tail on a lowercase "p") .

descent: In a font, the distance between the base line and the descent line.

descent line: A horizontal line that coincides with the bottom of the character descender that extends farthest below the **base line.** See also **ascent line, font height.**

desk accessory: A "mini-application" that is available to the user regardless of whether another application is running. The Apple IIGS supports two types of desk accessories: **classic desk accessories** and **new desk accessories.**

desk accessory event: An event that occurs when the user enters the special keystroke (Control–Apple–Escape) to invoke a **classic desk accessory.**

Desk Manager: The Apple IIGS tool set that executes **desk accessories** and enables applications to support them.

desk scrap: A piece of data, maintained by the Scrap Manager, taken from one application and available for insertion into another.

desktop: The visual interface between the computer and the user—the menu bar and the gray (or solid-colored) area on the screen. In many applications the user can have a number of **documents** on the desktop at the same time.

desktop interface: See **desktop.**

destination: See **destination location.**

destination location: The location (memory buffer or portion of the QuickDraw II coordinate plane) *to* which data such as text or graphics is copied. Compare **source location.** See also **destination rectangle.**

destination rectangle: The rectangle (on the QuickDraw II coordinate plane) in which text or graphics are drawn when transferred from somewhere else. Compare **source rectangle.**

development environment: A program or set of programs that allows you to write applications. It typically consists of a text editor, an assembler or compiler, a linker, and support programs such as a debugger.

device: A piece of hardware used in conjunction with a computer and under the computer's control. Also called a *peripheral device* because such equipment is often physically separate from, but attached to, the computer.

device driver: A program that handles the transfer of data to and from a peripheral device, such as a printer or disk drive.

device-driver event: An event generated by a device driver.

dial: An indicator on the screen that displays a quantative setting or value. Usually found in analog form, such as a fuel gauge or thermometer. A scroll bar is a standard type of dial.

dialog: See **dialog box.**

dialog box: A box on the screen that contains a message requesting more information from the user. See also **alert.**

Dialog Manager: The Apple IIGS tool set that manipulates **dialog boxes** and **alerts,** which appear on the screen when an application needs more information to carry out a command or when the user needs to be notified of an important situation.

dialog record: Information describing a dialog window that is maintained by the Dialog Manager.

dialog window: The window in which a dialog box appears.

digital oscillator chip (DOC): An integrated circuit in the Apple IIGS that contains 32 digital oscillators, each of which can generate a sound from stored digital waveform data.

digital RGB video monitor: A type of RGB video display in which the intensities of the red, green, and blue signals are fixed at discrete values.

dim: On the Apple IIGS desktop, to display a control or menu item in gray rather than black, to notify the user that the item is **inactive.**

direct page: A page (256 bytes) of bank $00 of Apple IIGS memory, any part of which can be addressed with a short (one-byte) address because its high-order address byte is always $00 and its middle address byte is the value of the 65816 direct register. Co-resident programs or routines can have their own direct pages at different locations. The direct page corresponds to the 6502 processor's **zero page.** The term direct page is often used informally to refer to any part of the **direct-page/stack space.**

direct-page/stack segment: A program segment that is used to initialize the size and contents of an application's stack and direct page.

direct-page/stack space: A single block of memory that contains an application's stack and direct page.

direct register: A hardware register in the 65816 processor that specifies the start of the direct page.

disable: To make unresponsive to user actions. A dialog box control that is disabled does nothing when selected or manipulated by the user. In appearance, however, it is identical to an enabled control. Compare **inactive.**

disabled menu: A menu that can be pulled down, but whose items are dimmed and not selectable.

disassembler: A program that converts machine-language code in memory into assembly-language instructions. Opposite of **assembler.**

disk operating system: An operating system whose principle function is to manage disk-based file access.

disk port: The connector on the rear panel of the Apple IIGS for attaching disk drives.

Disk II: A type of disk drive made and sold by Apple Computer, Inc., for use with the Apple II, II Plus, and IIe computers. It uses 5.25-inch disks.

display mode: A specification for the way in which a video display functions, including such parameters as whether displaying text or graphics, available colors, and number of pixels. The Apple IIGS has two text display modes (40 column and 80 column), two standard Apple II graphics display modes (Hi-Res and Double Hi-Res), and two new Super Hi-Res graphics display modes (320 mode and 640 mode).

display rectangle: A rectangle that determines where an item is displayed within a dialog box.

dispose: To permanently deallocate (a memory block). The Memory Manager disposes of a memory block by removing its master pointer. Any handle to that pointer will then be invalid. Compare **purge.**

dithering: A technique for alternating the values of adjacent pixels to create the optical effect of intermediate values. Dithering can give the effect of shades of gray on a black-and-white display, or more colors on a color display.

DOC: See **digital oscillator chip.**

document: A file created by an application.

document window: A window that displays a document. One of the two predefined window formats. Compare **alert window.**

dormant: Said of a program that is not being executed, but whose essential parts are all in the computer's memory. A dormant program may be quickly restarted because it need not be reloaded from disk.

double-click: To position the pointer where you want an action to take place, and then press and release the mouse button twice in quick succession without moving the mouse.

draft printing: The print method that the LaserWriter uses. QuickDraw II calls are converted directly into command codes the printer understands, which are then immediately used to drive the printer. Compare **spool printing.**

drag: To position the pointer on something, press and hold the mouse button, move the mouse, and release the mouse button. When you release the mouse button, you either confirm a menu selection or move an object to a new location.

drag area: A subregion in a window (usually the title bar) in which the mouse pointer must be placed before the user can drag the window.

draw: In QuickDraw II, to color pixels in a pixel image.

drawing environment: The complete description of how and where drawing may take place. Every open window on the Apple IIGS screen is associated with a **GrafPort** record, which specifies the window's drawing environment. Same as **port, graphic port.**

drawing mask: An 8-bit by 8-bit pattern that controls which pixels in the QuickDraw pen will be modified when the pen draws.

drawing mode: One of eight possible interactions between pixels in QuickDraw II's pen **pattern** and pixels already on the screen that fall under the pen's path. In COPY mode, for example, pixels already on the screen are ignored. In XOR mode, on the other hand, bits in pixels on the screen are XOR'd with bits in pixels in the pen; the resulting pixels are drawn on the screen.

drawing pen: See **pen.**

D register: See **direct register.**

driver: See **device driver.**

dynamic segment: A load segment capable of being loaded during program execution. Compare **static segment.**

edit record: A complete text editing environment in the Line Edit Tool Set, which includes the text to be edited, the GrafPort and rectangle in which to display the text, the arrangement of the text within the rectangle, and other editing and display information.

e flag: One of three flag bits in the 65816 processor that programs use to control the processor's operating modes. The setting of the e flag determines whether the processor is in native mode (6502), or emulation mode (65816). See also **m flag** and **x flag.**

emulate: To operate in a way identical to a different system. For example, the 65816 microprocessor in the Apple IIGS can carry out all the instructions in a program originally written for an Apple II that uses a 6502 microprocessor, thus emulating the 6502.

emulation mode: The 8-bit configuration of the 65816 processor in which it functions like a 6502 processor in all respects except clock speed.

enable: To make responsive to user manipulation. A dialog or menu that is enabled can be selected by the user. Enabling does not affect how an item is displayed. Compare **activate.**

end-of-file: See **EOF.**

EOF: The logical size of a ProDOS 16 file; it is the number of bytes that may be read from or written to the file.

erase: In QuickDraw II, to color an area with the **background pattern.**

error: The state of a computer after it has detected a fault in one or more commands sent to it. Also called *error condition.*

error message: A message issued by the system or application program when it has encountered an abnormal situation or an error in data.

event: A notification to an application of some occurrence (such as an interrupt generated by a keypress) to which the application may want to respond.

event code: A numeric value assigned to each event by the Event Manager. Compare **task code.**

event-driven: A kind of program that responds to user inputs in real time by repeatedly testing for **events.** An event-driven program does nothing until it detects an event such as a click of the mouse button.

Event Manager: An Apple IIGS tool set that detects events as they happen, and passes the events on to the application or to the appropriate event handler, such as **TaskMaster.**

event mask: A parameter passed to an Event Manager routine to specify which types of events the routine should apply to.

event queue: A list of pending events maintained by the Event Manager.

event record: The internal representation of an event, through which your program learns all pertinent information about that event.

execution mode: One of two general states of execution of the 65816 processor—**native mode** and 6502 **emulation mode.**

extended task event record: A data structure based on the event record that contains information used and returned by **TaskMaster.**

FALSE: Zero. The result of a Boolean operation. Opposite of **TRUE.**

file: Any named, ordered collection of information stored on a disk. Application programs and operating systems on disks are examples of files; so also are text or graphics created by applications and saved on disks. Text and graphics files are also called **documents**.

file level: See **system file level.**

file mark: See **Mark.**

filename: The string of characters that identifies a particular file within its directory. ProDOS filenames may be up to 15 characters long. Compare **pathname.**

file type: An attribute of a ProDOS file that characterizes its contents and indicates how the file may be used. On disk, file types are stored as numbers; in a directory listing, they are often displayed as three-character or single-word mnemonic codes.

fill mode: A display option in Super Hi-Res 320 mode. In fill mode, pixels in memory with the value 0 are automatically assigned the color of the previous nonzero pixel on the scan line; the program thus need assign explicit pixel values only to *change* pixel colors.

firmware: Programs stored permanently in ROM; most provide an interface to system hardware. Such programs (for example, the Monitor program) are built into the computer at the factory. They can be executed at any time but cannot be modified or erased. Compare **hardware** and **software.**

fixed: Not movable in memory once allocated. Also called *immovable.* Program segments that must not be moved are placed in fixed memory blocks. Opposite of **movable.**

fixed-address: A memory block that must be at a specified address when allocated.

fixed-bank: A block of memory that must start in a specified bank.

flag: A variable whose value (usually 1 or 0, standing for *true* or *false*) indicates whether some condition holds or whether some event has occurred. A flag is used to control the program's actions at some later time.

folder: See **subdirectory.**

font: In typography, a complete set of type in one size and style of character. In computer usage, a collection of letters, numbers, punctuation marks, and other typographical symbols with a consistent appearance; the size and style can be changed readily. See also **font scaling.**

font family: All fonts that share the same name but may vary in size or style. For example, all fonts named Helvetica are in the same family, even though that family contains Helvetica, Helvetica Narrow and Helvetica Bold.

font height: The vertical distance from a font's ascent line to its descent line.

font ID: A number that specifies a font by family, style, and size.

font scaling: A process by which the Font Manager creates a font at one size by enlarging or reducing characters in an existing font of another size.

font strike: A 1 bit/pixel pixelmap consisting of the character images of every defined character in the font, placed sequentially in order of increasing ASCII code.

foreground color: The color of the foreground pixels in text; by default it is white.

foreground pixels: In a **character image,** the pixels corresponding to the character itself.

frame region: The part of a window that surrounds the window's **content region** and contains standard window controls.

full pathname: The complete name by which a file is specified, starting with the volume directory name. A full pathname always begins with a slash (/), because a volume directory name always begins with a slash. See also **pathname.**

Function Pointer Table (FPT): A table, maintained by the Tool Locator, that points to all routines in a given tool set.

general logic unit: See **GLU.**

GetNextEvent: The Event Manager call that an application can make on each cycle through its main event loop. Compare **TaskMaster.**

global coordinates: The coordinate system assigned to a pixel image (such as screen memory) to which QuickDraw II draws. In global coordinates, the origin (upper-left corner) of the pixel image's boundary rectangle has the value (0,0). Compare **local coordinates.**

global symbol: A label in a segment that may be referenced by other segments. Compare with **local symbol, private symbol.**

GLU: Abbreviation of *general logic unit,* a class of custom integrated circuits used as interfaces between different parts of the computer.

go-away area: A subregion in a window frame, corresponding to the **close box.** Clicking inside this region of the active window makes the window close or disappear.

GrafPort: A data structure (record) that specifies a complete drawing environment, including such elements as a pixel image, boundaries within which to draw, a character font, patterns for drawing and erasing, and other pen characteristics.

graphic interface: An interface between computer and user in which all screen drawing or other output, including text, is done by graphic routines. Desktop programs use a graphic interface. Compare **text-based interface.**

graphic port: A specification for how and where QuickDraw II draws. A graphic port is defined by its **GrafPort** record; an application may have more that one graphic port open at one time, each defined by its own GrafPort. Same as **drawing environment.**

grow area: A window-frame subregion in which dragging changes the size of the window.

handle: See **memory handle.**

hardware: In computer terminology, the machinery that makes up a computer system. Compare **firmware, software.**

Heartbeat Interrupt Task queue: A list of tasks, such as cursor-movement updating or checking stack size, to be performed during **vertical blanking.** Heartbeat tasks are manipulated by the Miscellaneous Tool Set.

Heartbeat routines: Routines that execute at some multiple of the *heartbeat interrupt signal*, which occurs during the vertical blanking interval (every 1/60 of a second).

hex: See **hexadecimal.**

hexadecimal: The representation of numbers in the base-16 system, using the ten digits 0 through 9 and the six letters A through F. Each hexadecimal digit corresponds to a sequence of four binary digits (**bits**). Hexadecimal numbers are usually preceded by a dollar sign ($).

hide: To make invisible (but not necessarily to discard) an object on the screen such as a window.

highlight: To make something visually distinct. For example, when a button on a dialog box is selected, it appears as light letters on a dark background, rather than dark-on-light. An active window or control is highlighted differently than an inactive one.

HodgePodge: A sample Apple IIGS desktop application; the program described in this book.

horizontal blanking: The interval between the drawing of each scan line on a video display.

Human Interface Guidelines: Apple Computer's set of conventions and suggestions for writing desktop programs. Programs that follow the Human Interface Guidelines present a consistent and friendly interface to users.

IC: See **integrated circuit.**

icon: An image that graphically represents an object, a concept, or a message.

i flag: A bit in the 65816 microprocessor's Processor Status register that, if set to 1, disables interrupts.

image: A representation of the contents of memory. A code image consists of machine-language instructions or data that may be loaded unchanged into memory. See also **pixel image.**

inactive: Controls that have no meaning or effect in the current context, such as an *Open* button when no document has been selected to be opened. These inactive controls are not affected by the user's mouse actions and are dimmed on the screen. Compare **disable.**

index register: A register in a computer processor that holds an index for use in indexed addressing. The 6502 and 65816 microprocessors used in the Apple II family of computers have two index registers, called the **X register** and the **Y register.**

information bar: An optional component of a window. If present, the information bar appears just below the title bar. It may contain any information the application that created the window wishes.

initialization file: A program (in the `SYSTEM.SETUP` subdirectory of the boot disk) that is loaded and executed at system startup, independently of any application.

initialization segment: A segment in an initial load file that is loaded and executed independently of the rest of the program. It is commonly executed first, to perform any initialization that the program may require.

input/output: See **I/O.**

insertion point: The place in a document where something will be added; it is selected by clicking and is normally represented by a blinking vertical bar.

instrument: A data structure, used by the Note Sequencer and Synthesizer, that specifies such parameters as the amplitude envelope, pitchbend and vibrato characteristics, and the specific waveforms that characterize the sound to be played.

integer: A whole number in fixed-point form.

Integer Math Tool Set: The Apple IIGS tool set that performs simple mathematical functions on integers and other fixed-point numbers and converts numbers to their ASCII string equivalents.

integrated circuit: An electronic circuit—including components and interconnections—entirely contained in a single piece of semiconducting material, usually silicon. Often referred to as a *chip*.

interface: (1) The general form of interaction between a user and a computer. (2) In programming, the compile-time and runtime linkage between your program and toolbox routines.

interface library: A set of variable definitions and data-structure definitions that link a program (such as a C application) with software written in another language (such as the Apple IIGS Toolbox).

interrupt: A temporary suspension in the execution of a main program that allows the computer to perform some other task, typically in response to a signal from a peripheral device or other source external to the computer.

interrupt environment: The machine state, including register length and contents, within which the interrupt handler executes.

invert: To highlight by changing white pixels to black and vice versa.

I/O: Input/Output. A general term that encompasses input/output activity, the devices that accomplish it, and the data involved.

I/O space: The portion of the memory map in a standard Apple II (and in banks $00, $01, $E0, and $E1 of an Apple IIGS) with addresses between $C000 and $CFFF. Programs perform I/O by writing to or reading from locations in this I/O space.

item: A component of a dialog box, such as a button, text field, or icon.

item ID: A unique number that defines an item in a dialog box and allows further reference to it.

item line: The line of text that defines a menu item's name and appearance.

item list: A list of information about all the items in a dialog or alert box.

item type: Identifies the type of dialog item, usually represented by a predefined constant (such as `editLine`) or a series of constants (such as `editLine+itemDisable`).

Job dialog box: A dialog box presented when the user selects Print from the File menu.

joystick: A peripheral device with a lever, typically used to move creatures and objects in game programs; a joystick can also be used in applications such as computer-aided design and graphics programs.

JSL: Jump to Subroutine (Long), a 65816 assembly-language instruction that requires a long (3-byte) address. JSL can be used to transfer execution to code in another memory bank.

JSR: Jump to Subroutine, a 6502 and 65816 assembly-language instruction that requires a 2-byte address.

Jump Table: A table constructed in memory by the System Loader. The Jump Table contains all references to dynamic segments that may be called during execution of the program.

K: See **kilobyte.**

keyboard equivalent: The combination of the Apple key and another key, used to invoke a menu item from the keyboard.

key-down: An event type caused by the user's pressing any character key on the keyboard or keypad. 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.

kilobyte (K): A unit of measurement consisting of 1024 (2^{10}) bytes. In this usage, *kilo* (from the Greek, meaning a thousand) stands for 1024. Thus, 64K memory equals 65,536 bytes. See also **megabyte.**

kind: See **segment kind.**

language card: Memory with addresses between $D000 and $FFFF in any Apple II-family computer. It includes two RAM banks in the $D*xxx* space, called **bank-switched memory.** The *language card* was originally a peripheral card for the 48K Apple II or Apple II Plus that expanded the computer's memory capacity to 64K and provided space for an additional dialect of BASIC.

leading: (Pronounced LED-ing.) The space between lines of text. It is the number of pixels vertically between the descent line of one character and the ascent line of the character immediately beneath it.

length byte: The first byte of a **Pascal string.** It specifies the length of the string, in bytes.

library (or library file): An object file containing program segments, each of which can be used in any number of programs. The linker can search through the library file for segments that have been referenced in the program source file.

library dictionary segment: The first segment of a library file; it contains a list of all the symbols in the file together with their locations in the file. The linker uses the library dictionary segment to find the segments it needs.

line: In QuickDraw II, the straight-line trajectory between two points on the coordinate plane. The line is specified by its starting and ending points.

LineEdit Tool Set: The Apple IIGS tool set that provides simple text-editing functions. It is used mostly in dialog boxes.

line height: The total amount of vertical space from line to line in a text document. Line height is the sum of **ascent, descent,** and **leading.**

LinkEd: A command language that can be used to control the APW Linker.

linker: A program that combines files generated by compilers and assemblers, resolves all symbolic references, and generates a file that can be loaded into memory and executed.

Lisa: A model of Apple computer; the first computer that offered windows and the use of a mouse to choose commands. The Lisa is now known as the Macintosh XL.

list: See **list control.**

list control: A custom control created by the List Manager. It is a scrollable, vertical arrangement of similar items on the screen; the items are *selectable* by the user.

List Manager: The Apple IIGS tool set that allows an application to present the user with a list from which to choose. For example, the Font Manager uses the List Manager to arrange lists of fonts.

load: To transfer information from a peripheral storage medium (such as a disk) into main memory for use—for example, to transfer a program into memory for execution.

load file: The output of the linker. Load files contain memory images that the system loader can load into memory, together with **relocation dictionaries** that the loader uses to relocate references.

load segment: A segment in a load file. Any number of object segments can go into the same load segment.

local coordinates: A coordinate system unique to each GrafPort and independent of the **global coordinates** of the pixel image that the port is associated with. For example, local coordinates do not change as a window is dragged across the screen; global coordinates do not change as a window's contents are scrolled.

local symbol: A label defined only within an individual segment. Other segments cannot reference the label. Compare with **global symbol.**

LocInfo: Acronym for *location information*. The data structure (record) that ties the coordinate plane to an individual pixel image in memory.

lock: To prevent a memory block from being moved or purged. A block may be locked or unlocked by a call to the Memory Manager.

long (or **long word):** On the Apple IIGS, a 32-bit (4-byte) data type.

Macintosh: A family of Apple computers; for example, the Macintosh 512K and the Macintosh Plus. Macintosh computers have high-resolution screens and use mouse devices for choosing commands and for drawing pictures.

macro: A single keystroke or command that a program replaces with several keystrokes or commands. For example, the APW Editor allows you to define macros that execute several editor keystroke commands; the APW Assembler allows you to define macros that execute instructions and directives. Macros are almost like higher-level language instructions, making assembly-language programs easier to write and complex keystrokes easier to execute.

macro library: A file of related macros.

main event loop: The central routine of an event-driven program. During execution, the program continually cycles through the main event loop, branching off to handle events as they occur and then returning to the event loop.

mainID: A subfield of the **User ID.** Each running program is assigned a unique mainID.

manager: See **tool set.**

Mark: The current position in an open file. It is the point in the file at which the next read or write operation will occur.

mask: (n) A parameter, typically one or more bytes long, whose individual bits are used to permit or block particular features. See, for example, **event mask.** (v) To apply a mask.

master color value: A 2-byte number that specifies the relative intensities of the red, green, and blue signals output by the Apple IIGS video hardware.

Lisa: A model of Apple computer; the first computer that offered windows and the use of a mouse to choose commands. The Lisa is now known as the Macintosh XL.

list: See **list control.**

list control: A custom control created by the List Manager. It is a scrollable, vertical arrangement of similar items on the screen; the items are *selectable* by the user.

List Manager: The Apple IIGS tool set that allows an application to present the user with a list from which to choose. For example, the Font Manager uses the List Manager to arrange lists of fonts.

load: To transfer information from a peripheral storage medium (such as a disk) into main memory for use—for example, to transfer a program into memory for execution.

load file: The output of the linker. Load files contain memory images that the system loader can load into memory, together with **relocation dictionaries** that the loader uses to relocate references.

load segment: A segment in a load file. Any number of object segments can go into the same load segment.

local coordinates: A coordinate system unique to each GrafPort and independent of the **global coordinates** of the pixel image that the port is associated with. For example, local coordinates do not change as a window is dragged across the screen; global coordinates do not change as a window's contents are scrolled.

local symbol: A label defined only within an individual segment. Other segments cannot reference the label. Compare with **global symbol.**

LocInfo: Acronym for *location information.* The data structure (record) that ties the coordinate plane to an individual pixel image in memory.

lock: To prevent a memory block from being moved or purged. A block may be locked or unlocked by a call to the Memory Manager.

long (or **long word):** On the Apple IIGS, a 32-bit (4-byte) data type.

Macintosh: A family of Apple computers; for example, the Macintosh 512K and the Macintosh Plus. Macintosh computers have high-resolution screens and use mouse devices for choosing commands and for drawing pictures.

macro: A single keystroke or command that a program replaces with several keystrokes or commands. For example, the APW Editor allows you to define macros that execute several editor keystroke commands; the APW Assembler allows you to define macros that execute instructions and directives. Macros are almost like higher-level language instructions, making assembly-language programs easier to write and complex keystrokes easier to execute.

macro library: A file of related macros.

main event loop: The central routine of an event-driven program. During execution, the program continually cycles through the main event loop, branching off to handle events as they occur and then returning to the event loop.

mainID: A subfield of the **User ID.** Each running program is assigned a unique mainID.

manager: See **tool set.**

Mark: The current position in an open file. It is the point in the file at which the next read or write operation will occur.

mask: (n) A parameter, typically one or more bytes long, whose individual bits are used to permit or block particular features. See, for example, **event mask.** (v) To apply a mask.

master color value: A 2-byte number that specifies the relative intensities of the red, green, and blue signals output by the Apple IIGS video hardware.

master User ID: The value of a User ID, *disregarding the contents of the auxID field.* If an application allocates various memory blocks and assigns them unique ID's consisting of different auxID values added to its own User ID, then all will share the same Master User ID and all can be purged or disposed with a single call.

Mb: See **megabyte.**

megabyte (Mb): A unit of computer memory or disk drive capacity that equals 1,048,576 bytes.

megahertz (MHz): A unit of measurement of frequency, equal to 1,000,000 *hertz* (cycles per second); abbreviated *MHz*.

memory block: See **block** (2).

memory expansion card: A memory card that increases Apple IIGS internal memory capacity beyond 256K, up to 8 megabytes

memory fragmentation: A condition in which free (unallocated) portions of memory are scattered because of repeated allocation and deallocation of blocks by the Memory Manager.

memory handle: A number that identifies a memory block. A handle is a pointer to a pointer—it is the address of a master pointer, which in turn contains the address of the block.

memory image: See **image.**

Memory Manager: The Apple IIGS tool set that manages memory use. The Memory Manager keeps track of how much memory is available, and allocates memory **blocks** to hold program segments or data.

Memory Segment Table: A linked list in memory, created by the loader, that allows the loader to keep track of the segments that have been loaded into memory.

menu: A list of choices presented by a program, from which the user can select an action. See also **pull-down menu.**

menu bar: The horizontal strip at the top of the screen that contains menu titles for the pull-down menus.

menu ID: A number in the menu record that identifies an individual menu.

menu line: A line of text plus code characters that defines the appearance of a particular menu title.

Menu Manager: The Apple IIGS tool set that maintains the pull-down menus and the items in the menus.

m flag: One of 3 flags in the 65816 microprocessor's Processor Status register that controls execution mode. When the m flag is set to 1, the accumulator is 8 bits wide; otherwise, it is 16 bits wide.

MHz: See **megahertz.**

microprocessor: A central processing unit that is contained in a single integrated circuit. The Apple IIGS uses a 65816 microprocessor.

minipalette: In Super Hi-Res 640 mode, a quarter of the **color table.** Each pixel in 640 mode can have one of four colors specified in a minipalette.

Miscellaneous Tool Set: The Apple IIGS tool set that includes mostly system-level routines that must be available for other tool sets.

missing symbol: In a font, the symbol substituted for any ASCII value for which the font does not have a defined symbol. In the Apple IIGS system font, the missing symbol is a box containing a question mark.

modal dialog box: A dialog box that puts the machine in a state such that the user cannot execute functions outside of the dialog box, until the dialog box is closed.

mode: A state of a computer or system that determines its behavior. A manner of operating.

modeless dialog box: A dialog box that lets the user take other action besides responding to the dialog box. Compare **modal dialog box.**

modification date: An attribute of a ProDOS file; it specifies the date on which the content of the file was last changed.

modification time: An attribute of a ProDOS file; it specifies the time at which the content of the file was last changed.

Monitor program: A firmware program built into the ROM of Apple II computers, used for directly inspecting or changing the contents of main memory and for operating the computer at the machine-language level.

monospaced: Said of a font whose character widths are all identical. Compare **proportionally spaced.**

MOS: Abbreviation for *metal-oxide semiconductor,* a method of fabricating integrated-circuits on silicon by using layers of silicon dioxide in the make-up of the devices. Compare **CMOS.**

mouse: A small device that the user moves around on a flat surface next to the computer. The mouse controls a pointer on the screen whose movements correspond to those of the mouse. The pointer selects operations, moves data, and draws graphic objects.

mouse button: A button on a mouse device with which the user selects objects on the screen.

mouse-down: An action or an event, signifying that the user has pressed the mouse button.

mouse-up: An action or an event, signifying that the user has released the mouse button.

movable: Able to be moved to different memory locations during program execution (a memory block attribute).

native mode: The 16-bit operating configuration of the 65816 microprocessor.

NDA: See **new desk accessory.**

new desk accessory (NDA): A desk accessory designed to execute in a desktop, event-driven environment. Compare **classic desk accessory.**

newline read mode: A file-reading mode in which each character read from the file is compared to a specified character (called the *newline character*); if there is a match, the read is terminated. Newline mode is typically used to read individual lines of text, with the newline character defined as a carriage return.

NewWindow parameter list: A template describing the features of a window that is to be created. A pointer to a NewWindow parameter list is a required input to the NewWindow call.

NIL: Pointing to a value of 0. A memory handle is NIL if the address it points to is filled with zeros. Handles to purged memory blocks are NIL. Compare **null.**

Note Sequencer: The Apple IIGS tool set that makes it possible to play music asynchronously in programs.

Note Synthesizer: An Apple IIGS tool set that facilitates creation and manipulation of musical notes.

null: Zero. A pointer is null if its value is all zeros. Compare **NIL.**

null event: An event reported when there are no other events to report.

null prefix: A prefix of zero length (and therefore nonexistent).

object file: The output from an assembler or compiler, and the input to a linker. It contains machine-language instructions as well as other information. Also called *object program* or *object code.* In APW an object file cannot be loaded into memory. Compare **source file, load file.**

object module format (OMF): The file format used in Apple IIGS object files, library files, and load files.

object segment: A segment in an object file.

offset: The number of character positions or memory locations away from a point of reference.

OK: One of two predefined item ID numbers for dialog box buttons (OK = 1). Compare **Cancel.**

OMF: See **object module format.**

operating environment: The overall hardware and software setting within which a program runs. Also called *execution environment.*

operating system: A general-purpose program that organizes the actions of the various parts of the computer and its peripheral devices. See also **disk operating system.**

origin: (1) The first memory address of a program or of a portion of one. The first instruction to be executed. (2) The location (0,0) on the QuickDraw II coordinate plane, in either **global coordinates** or **local coordinates.** (3) The upper-left corner of any rectangle (such as a boundary rectangle or port rectangle) in QuickDraw II. (4) See **character origin.**

oscillator: A device that generates a vibration. In the Apple IIGS Digital Oscillator Chip, an oscillator is an address generator that points to the next data byte in memory that represents part of a particular sound wave.

oval: A circle or ellipse, one of the fundamental classes of objects drawn by QuickDraw II.

overlay: One of a set of program segments meant to alternately occupy the same memory space. Use of overlays is one way to minimize the amount of memory a program needs.

pack: To compress data into a smaller space to conserve storage space.

page: (1) A portion of memory 256 bytes long and beginning at an address that is an even multiple of 256. Memory blocks whose starting addresses are an even multiple of 256 are said to be *page-aligned.* (2) (usually capitalized) An area of main memory containing text or graphic information being displayed on the screen.

page-aligned: Starting at a memory address that is an even multiple of 256 (a memory block attribute). See **page** (1).

palette: The full set of colors available for an individual screen pixel.

parameter: A value passed to or from a function or other routine.

parameter RAM: RAM on the Apple IIGS clock chip. A battery preserves the clock settings and the RAM contents when the power is off. Control Panel settings are kept in battery RAM.

partial pathname: A **pathname** that includes the **filename** of the desired file but excludes the volume directory name (and possibly one or more of the subdirectories in the path). It is the part of a pathname following a **prefix**—a prefix and a partial pathname together constitute a **full pathname.** A partial pathname does not begin with a slash because it has no volume directory name.

Pascal: A high-level programming language. Named for the philosopher and mathematician Blaise Pascal.

Pascal string: An ASCII character string preceded by a single byte whose numerical value is the number of characters in the string. Compare **C string.**

paste: To place the desk scrap (contents of the Clipboard—whatever was last cut or copied) at the insertion point.

patch: To replace one or more bytes in memory or in a file with other values. The address to which the program must jump to execute a subroutine is *patched* into memory at load time, when the System Loader performs **relocation** on a file.

pathname: A name that specifies a file. It is a sequence of one or more **filenames** separated by slashes, tracing the path through subdirectories that a program must follow to locate the file. See **full pathname, partial pathname, prefix.**

pathname prefix: See **prefix.**

Pathname Table: A table constructed in memory by the System Loader. The Pathname Table contains cross-references between load files referenced by number (in the **Jump Table**) and by pathname (in the file directory).

pattern: (1) An 8-by-8 pixel image, used to define a repeating design (such as stripes) or color. (2) A series of commands to the Note Synthesizer.

PB register: See **program bank register.**

PC register: A register within the 65816 microprocessor that keeps track of the memory address of the next instruction to be executed. PC stands for *program counter.*

pen: The conceptual tool with which QuickDraw II draws shapes and characters. Each GrafPort has its own pen.

pen location: The position (on the coordinate plane) at which the next character or line will be drawn.

pen pattern: See **pattern (1).**

peripheral card: A hardware device placed inside a computer, and connected to one of the computer's peripheral expansion slots. Peripheral cards perform a variety of functions, from controlling a disk drive to providing a clock/calendar.

peripheral device: See **device.**

phrase: In music synthesis, a set of pointers to **patterns** that make it easy to build repetitive, complex passages out of simple patterns.

picture: A saved sequence of QuickDraw drawing commands (and, optionally, picture comments) that you can play back later with a single procedure call; also, the image resulting from these commands.

pixel: Short for *picture element.* The smallest dot you can draw on the screen. Also a location in video memory that corresponds to a point on the graphics screen when the viewing window includes that location. In the Super Hi-Res display on the Apple IIGS, each pixel is represented by either two or four bits. See also **pixel image.**

pixel image: A graphics image picture consisting of a rectangular grid of pixels.

plain-styled: Said of a font or character that is not bold, italicized, underlined, or otherwise styled apart from ordinary text.

plane: The front-to-back position of a window, compared to other windows on the desktop.

point: A unit of measurement for type; 12 points equal 1 pica, and 6 picas equal 1 inch; thus, 1 point equals $^1/_{72}$ inch.

pointer: (1) An item of information consisting of the memory address of some other item. For example, the 65816 stack register contains a pointer to the top of the stack. (2) The mouse pointer, an arrow-shaped cursor whose screen location is controlled by mouse movements.

pointing device: Any device, such as a mouse, graphics tablet, or light pen, that can be used to specify locations on the computer screen.

polygon: Any sequence of connected lines.

port: (1) A socket on the back panel of the computer where the user can plug in a cable to connect a peripheral device, another computer, or a network. (2) A **graphic port (GrafPort).**

portRect: The GrafPort field that defines the port's **port rectangle.**

port rectangle: A rectangle that describes the active region of a GrafPort's pixel map—the part that QuickDraw II can draw into. The **content region** of a window on the desktop corresponds to the window's port rectangle.

position-independent: Said of code that can execute, without modification of any kind, at any location in memory. Compare **absolute, relocatable.**

post: To place an event in the event queue for later processing.

prefix: A **pathname** starting with a volume name and ending with a subdirectory name. It is the part of a full pathname that precedes a **partial pathname**—a prefix and a partial pathname together constitute a full pathname. A prefix always starts with a slash (/) because a volume directory name always starts with a slash.

prefix number: A code used to represent a particular prefix. Under ProDOS 16, there are nine prefix numbers, each consisting of a numeral followed by a slash: 0/, 1/,…,8/, and */.

P register: See **status register.**

printing loop: The page-by-page cycle that an application goes through when it prints a document.

Print Manager: The Apple IIGS tool set that allows an application to use standard QuickDraw II routines to print text or graphics on a printer.

print record: A record containing all the information needed by the Print Manager to perform a particular printing job.

private scrap: A buffer (and its contents) set up by an application for cutting and pasting, analogous to but apart from the **desk scrap.**

private symbol: A label in a segment that may be referenced by other segments in the same file, but not by segments in other files.

processor status register: See **status register.**

ProDOS: A family of disk operating systems developed for the Apple II family of computers. *ProDOS* stands for *Professional Disk Operating System*, and includes both ProDOS 8 and ProDOS 16.

ProDOS 8: A disk operating system developed for standard Apple II computers. It runs on 6502-series microprocessors and on the Apple IIGS when the 65816 processor is in 6502 **emulation mode.**

ProDOS 16: A disk operating system developed for 65816 **native-mode** operation on the Apple IIGS. It is functionally similar to **ProDOS 8** but more powerful.

program bank register: The 65816 register whose contents form the high-order byte of all 3-byte code address operands.

program counter: See **PC register.**

program status register: See **status register.**

proportionally spaced: Said of a font whose characters vary in width, so the amount of horizontal space needed for each character is proportional to its width. Compare **monospaced.**

pull-down menu: A set of choices for actions that appears near the top of the display screen in a desktop application, usually overlaying the present contents of the screen without disrupting them. Dragging through the menu and releasing the mouse button while a command is highlighted chooses that command.

purge: To temporarily deallocate a memory block. The Memory Manager purges a block by setting its master pointer to NIL (0). All handles to the pointer are still valid, so the block can be reconstructed quickly. Compare **dispose.**

purge level: A memory block attribute, indicating that the Memory Manager may purge the block if it needs additional memory space. Purgeable blocks have different purge levels, or priorities for purging; these levels are set by Memory Manager calls.

queue: A list in which entries are added at one end and removed at the other, causing entries to be removed in first-in, first-out (FIFO) order. Compare **stack.**

QuickDraw II: The Apple IIGS tool set that controls the graphics environment and draws simple objects and text. Other tools call QuickDraw II to draw such things as windows.

QuickDraw II Auxiliary: An Apple IIGS tool set that provides extensions to the capabilities of QuickDraw II.

quit: To terminate execution in an orderly manner. Apple IIGS applications quit by making a ProDOS 16 QUIT call or the equivalent.

quit return stack: A table, maintained in memory by ProDOS 16, that contains the User ID's of programs that want to be reexecuted after the current program quits.

radio button: A common type of **control** in dialog boxes. Radio buttons are small circles organized into families—clicking any button on turns off all the others in the family, like the buttons on a car radio.

RAM: See **random-access memory.**

random-access memory (RAM): Memory in which information can be referred to in an arbitrary or random order. Programs and other data in RAM are lost when the computer is turned off. (Technically, the *read-only memory* (ROM) is also random access, and what's called RAM should correctly be termed *read-write memory.*) Compare **read-only memory.**

read-only memory (ROM): Memory whose contents can be read, but not changed; used for storing **firmware.** Information is placed into ROM once, during manufacture; it then remains there permanently, even when the computer's power is turned off. Compare **random-access memory.**

rectangle: One of the fundamental shapes drawn by QuickDraw II. Rectangles are completely defined by two points—their upper-left and lower-right corners on the coordinate plane. The upper-left corner of any rectangle is its **origin.**

reentrant: Said of a routine that is able to accept a call while one or more previous calls to it are pending, without invalidating the previous calls. Under certain conditions, the Apple IIGS **Scheduler** manages execution of routines that are not reentrant.

region: An arbitrary area or set of areas on the QuickDraw coordinate plane. The outline of a region must be one or more closed loops.

RELOAD segment: A segment that is always reloaded from disk when a program is executed, even if the program is in a **dormant** state in computer memory. Some programs require RELOAD segments in order to be **restartable.**

relocatable: Characteristic of a load segment or other OMF program code that includes no references to specific address, and so can be loaded at any memory address. A relocatable segment consists of a code image followed by a relocation dictionary. Compare **absolute.**

relocation: The act of modifying a program in memory so that its address operands correctly reflect its location and the locations of other segments in memory. Relocation is performed by the System Loader when a relocatable segment is first loaded into memory.

relocation dictionary: In object module format, a portion of a load segment that contains relocation information necessary to modify the memory image portion of the segment. See **relocation.**

resource: A type of organization for certain components of Macintosh files. Resources provide a convenient means for manipulating the fixed (unchanging) parts of a program file.

resource editor: A program for editing resources, especially data in a program, without having to recompile the program.

Resource Manager: The Macintosh toolbox component that retrieves, manipulates, and disposes of **resources.**

restart: To reactivate a **dormant** program in the computer's memory. The System Loader can restart dormant programs if all their static segments are still in memory. If any critical part of a dormant program has been purged by the Memory Manager, the program must be reloaded from disk instead of restarted.

restartable: Said of a program that reinitializes its variables and makes no assumptions about machine state each time it gains control. Only restartable programs can be resurrected from a **dormant** state in memory.

RGB: Abbreviation for *red-green-blue,* a method of displaying color video by transmitting the three primary colors as three separate signals. There are two ways of using RGB with computers: **TTL RGB,** which allows the color signals to take on only discrete values; and **analog RGB,** which allows the color signals to take on any values between their upper and lower limits.

ROM: See **read-only memory.**

routine: A part of a program that accomplishes some task subordinate to the overall task of the program.

RTI: Return from Interrupt, a 65816 assembly-language instruction.

RTL: Return from Subroutine (Long), a 65816 assembly-language instruction.

RTS: Return from Subroutine, a 65816 assembly-language instruction.

SANE: See **Standard Apple Numeric Environment.**

SANE Tool Set: The Apple IIGS tool set that performs high-precision floating-point calculations, following SANE standards.

scaled font: A font that is created by the Font Manager by calculation from a real font of a different size.

scan line: A single horizontal line of pixels on the screen. It corresponds to a single sweep of the electron gun in the video display tube.

scanline control byte (SCB): A byte in memory that controls certain properties, such as available colors and number of pixels, for a scan line on the Apple IIGS. Each scan line has its own SCB.

Scheduler: The Apple IIGS tool set that manages requests to execute *interrupted* software that is not reentrant. If, for example, an interrupt handler needs to make system software calls, it must do so through the Scheduler because ProDOS 16 is not reentrant. Applications normally need not use the Scheduler because ProDOS 16 is not in an interrupted state when it processes applications' system calls.

Scrap Manager: The Apple IIGS tool set that supports the **desk scrap,** which allows data to be copied from one application to another (or from one place to another within an application).

scroll: To move an image of a document or directory in its window so that a different part of it becomes visible.

scroll bar: A rectangular bar that may be along the right side or bottom of a window. Clicking or dragging in the scroll bar causes the view of the document to change.

segment: A component of an OMF file, consisting of a header and a body. In object files, each segment incorporates one or more subroutines. In load files, each segment incorporates one or more object segments.

segment kind: A numerical designation used to classify a segment in object module format.

self-booting: Said of a program that executes automatically when the computer is turned on or reset.

sequence: A series of commands that tells the computer what notes to play and when.

serial interface: A standard method, such as RS-232, for transmitting data serially (as a sequence of bits).

serial port: The connector for a peripheral device that uses a **serial interface.**

Shaston: The Apple IIGS system font.

shell: A program that provides an operating environment for other programs, and that is not removed from memory when those programs are running. For example, the APW Shell provides a command processor interface between the user and the other components of APW, and remains in memory when APW utility programs are running. A shell is one type of **controlling program.**

shell application: A type of program that is launched from a shell and runs under its control. Shell applications are ProDOS 16 file type $B5. In APW, compilers and certain Shell commands are shell applications that are launched from the APW Shell.

shell call: A request from a program to the APW Shell to perform a specific function.

shut down: To remove from memory or otherwise make unavailable, as a tool set that is no longer needed or an application that has quit.

size box: A small square in the lower-right corner of some windows, with which the user can resize the window. The size box corresponds to the **grow region.**

65C816: The version of the 65816 microprocessor used in the Apple IIGS. The 65C816 is a CMOS device.

65816: A general term for the type of microprocessor used in the Apple IIGS. The 65816 is related to, but more advanced than, the 6502 microprocessor. It has a 16-bit data bus and a 24-bit address bus.

65816 assembly language: A low-level programming language written for the 65816 family of microprocessors.

6502: The microprocessor used in the Apple II, in the Apple II Plus, and in early models of the Apple IIe. The 6502 is an NMOS device with an 8-bit data bus and a 16-bit address bus.

640 mode: An Apple IIGS video display mode, 640 pixels horizontally by 200 pixels vertically.

slot: A narrow socket inside the computer where the user can install peripheral cards. Also called an *expansion slot.*

SmartPort: A set of firmware routines supporting multiple devices connected to the Apple IIGS disk port.

software: A collective term for programs, the instructions that tell the computer what to do. Software is usually stored on disks. Compare **firmware, hardware.**

Sound Tool Set: The Apple IIGS tool set that provides low-level access to the sound hardware.

source: See **source location.**

source file: An ASCII file consisting of instructions written in a particular language, such as Pascal or assembly language. An assembler or compiler converts source files into **object files.**

source location: The location (memory buffer or portion of the QuickDraw II coordinate plane) *from* which data such as text or graphics are copied. Compare **destination location.** See also **source rectangle.**

source rectangle: The rectangle (on the QuickDraw II coordinate plane) from which text or graphics are taken when transferred somewhere else. Compare **destination rectangle.**

special memory: On an Apple IIGS, all of banks $00 and $01, and all display memory in banks $E0 and $E1 . It is the memory directly accessed by **standard-Apple II** programs running on the Apple IIGS.

spool printing: A two-step printing method used to print graphics on the ImageWriter. In the first step, it writes out (spools) a representation of your document's printed image to a disk file or to memory. In the second step, this information is converted into a bit image and printed. Compare **draft printing.**

S register: See **stack register.**

stack: A list in which entries are added (pushed) and removed (pulled) at one end only (the top of the stack), causing them to be removed in last-in, first-out (LIFO) order. *The stack* usually refers to the particular stack pointed to by the 65816's **stack register.** Compare **queue.**

stack pointer: See **stack register.**

stack register: A register in the 65816 processor that indicates the next available memory address in the **stack.**

Standard Apple Numerics Environment (SANE): The set of methods that provides the basis for floating-point calculations in Apple computers. SANE meets all requirements for extended-precision, floating-point arithmetic as prescribed by IEEE Standard 754 and ensures that all floating-point operations are performed consistently and return the most accurate results possible.

standard Apple II: Any computer in the Apple II family except the Apple IIGS. That includes the Apple II, the Apple II Plus, the Apple IIe, and the Apple IIc.

Standard File Operations Tool Set: The Apple IIGS tool set that creates a standard user interface for opening and closing files.

standard linker (APW): One aspect of the linker supplied with APW. The operation of the standard linker is automatic. Compare **advanced linker.**

standard window parts: The window features that allow the user to scroll through the data in the window, change the window's shape, or close the window. They also provide information about the document currently displayed in the window.

START: The name of the program in the SYSTEM/subdirectory of the startup disk that is launched automatically when the system is booted. START is typically a finder or program launcher.

start up: To get the system or application program running.

static segment: A program segment that must be loaded when the program is started, and cannot be removed from memory until execution terminates. Compare **dynamic segment.**

static text: Text on the screen that cannot be altered by the user.

status register: A register in the 65816 microprocessor that contains flags reflecting the various aspects of machine state and operation results.

string: A sequence of characters. See **C string, Pascal string.**

structure region: An entire window; its **content region** plus its **frame region.**

Style dialog box: A dialog box that allows the user to specify formatting information, page size, and printer options.

styled variation: An italicized, boldfaced, underlined, or otherwise altered version of a **plain-styled** character or font.

subdirectory: A file that contains information about other files. In a hierarchical file system, files are accessed through the subdirectories that reference them.

subroutine: A part of a program that can be executed on request from another point in the program and that, upon completion, returns control to the point of the request.

Super Hi-Res: Either of two high-resolution Apple IIGS display modes. 320 mode consists of an array of pixels 320 wide by 200 high, with 16 available colors; 640 mode is an array 640 wide by 200 high, with 16 available colors (with restrictions).

switcher: A **controlling program** that rapidly transfers execution among several applications.

switch event: An event type reserved for future use, such as in conjunction with a switcher.

symbolic reference: A name or label, such as the name of a subroutine, that is used to refer to a location in a program. When a program is linked, all symbolic references are *resolved;* when the program is loaded, actual memory addresses are **patched** into the program to replace the symbolic references. (This process is called **relocation.**)

synthesizer: (1) A hardware device capable of creating sound digitally and converting it into an analog waveform that you can hear. (2) By analogy, any sound-making entity, such as the **Note Synthesizer** tool set.

system disk: A disk that contains the operating system and other system software needed to run applications.

system event mask: A set of flags that control which event types get posted into the event queue by the Event Manager.

system failure: The unintentional termination of program execution due to a severe software error.

System Failure Manager: A part of the Miscellaneous Tool Set that processes fatal errors by displaying a message on the screen and halting execution.

system file level: A number between $00 and $FF associated with each open ProDOS 16 file. Every time a file is opened, the current value of the system file level is assigned to it. If the system file level is changed (by a SET_LEVEL call), all subsequently opened files will have the new level assigned to them. By manipulating the system file level, a controlling program can easily close or flush files opened by its subprograms.

system folder: The SYSTEM/subdirectory on a ProDOS 16 system disk.

system library prefix: ProDOS 16 **prefix** number 2/. It specifies the directory containing library files used by system software.

System Loader: The program that manages the loading and relocation of load segments (programs) into the Apple IIGS memory. The System Loader works closely with ProDOS 16 and the Memory Manager.

system menu bar: The menu bar that always appears at the top of the screen in desktop applications. It contains all of the commonly used functions, in menus such as File, Edit, and so on.

system prefix (ProDOS 8): The one prefix maintained by ProDOS 8.

system software: The components of a computer system that support **application programs** by managing system resources such as memory and I/O devices.

system window: A window in which a desk accessory is displayed.

task code: A numeric value assigned to the result of each event handled by TaskMaster. Compare **event code.**

task mask: A parameter passed to TaskMaster, specifying which types of events TaskMaster is to respond to.

TaskMaster: A Window Manager routine that handles many typical events for an application. Applications may call TaskMaster instead of **GetNextEvent.**

template: A data structure or set of parameters that defines the characteristics of a desktop feature, such as a window or control. The NewWindow parameter list is a template that defines the appearance of a window to be opened by the NewWindow call.

text-based interface: An interface between computer and user in which all screen drawing (or other output) consists of characters. The form of each character is stored in ROM and can be involved with a single byte of data. Compare **graphic interface.**

text buffer: A 1-bit-per-pixel pixel image reserved for the private use of the QuickDraw II text-drawing call.

text file: A file consisting of the ASCII representation of characters.

text mode: One of 16 possible interactions between pixels in text being drawn to the screen and pixels on the screen that fall under characters being drawn. Compare **drawing mode.**

Text Tool Set: An Apple IIGS tool set that provides an interface between Apple II character device drivers and applications running in native mode.

320 mode: An Apple IIGS video display mode, 320 pixels horizontally by 200 pixels vertically.

tick count: The (approximate) number of 60th second intervals since system startup.

title bar: The horizontal bar at the top of a window that shows the name of the window's contents. The user can move the window by dragging the title bar.

tool: See **tool set.**

toolbox: The collection of built-in routines on the Apple IIGS that programs can call to perform many commonly needed functions. Functions within the toolbox are grouped into **tool sets.**

tool call: A call to a function within a tool set.

Tool Locator: The Apple IIGS tool set that dispatches tool calls. The tool locator knows and retrieves the appropriate routine when you make a tool call.

Tool Pointer Table (TPT): A table, maintained by the Tool Locator, that contains pointers to all active tool sets.

tool set: A group of related routines (usually in ROM) that perform necessary functions or provide programming convenience. They are available to applications and system software. The Memory Manager, the System Loader, and QuickDraw II are Apple IIGS tool sets.

tool table: A list of all needed tool sets and their minimum required versions. An application constructs this table in order to load its RAM-based tool sets with the LoadTools call.

track: (1) One of a series of concentric circles magnetically recorded on the surface of a disk when it is formatted. Each track is further divided into *sectors*. Each sector can hold several K of data. (2) A grouping of items in a musical sequence. The Note Sequencer supports multiple tracks to facilitate writing multi-instrument music.

transfer mode: A specification of which Boolean operation QuickDraw should perform when drawing. See, for example, **XOR.**

TRUE: Nonzero. The result of a Boolean operation. Opposite of **FALSE.**

TTL RGB: A type of color video consisting of separate red, green, and blue signals that can have only discrete values.

typeID: A subfield of the User ID. The User ID Manager assigns a typeID value based on the *type* of program (application, tool set, and so on) requesting the memory.

unhighlight: To restore to normal display. Selected controls, menu items, or other objects may be **highlighted** (usually displayed in inverse colors) while in use, and unhighlighted when not in use.

unload: To remove a load segment from memory. To unload a segment, the System Loader does not actually "unload" anything; it calls the Memory Manager to either **purge** or **dispose** of the memory block in which the code segment resides.

unlock: To permit the Memory Manager to move or purge a memory block if needed. Opposite of **lock.**

unmovable: See **fixed.**

unpack: To restore to normal format from a **packed** format.

unpurgeable: Having a **purge level** of zero. The Memory Manager is not permitted to purge memory blocks whose purge level is zero.

update: A type of window event, signifying that all or part of the window needs to be redrawn.

update event: An event posted by the Window Manager when all or part of a window needs to be redrawn.

update region: A description of the part of a window that needs to be redrawn. The Window Manager keeps track of each open window's update region.

User ID: An identification number that specifies the owner of every memory block allocated by the Memory Manager. User ID's are assigned by the User ID Manager.

User ID Manager: A part of the Miscellaneous Tool Set that is responsible for assigning User ID's to every block of memory allocated by the Memory Manager.

vector: A location that contains a value used to find the entry point address of a subroutine.

vertical blanking: The interval between successive screen drawings on a video display. It is the time between drawing the last pixel of the last scan line of one frame and the first pixel of the first scan line of the next frame.

visible region: The part of a window that's actually visible on the screen. The visible region is a **GrafPort** field manipulated by the Window Manager.

voice: Any one of 16 pairs of oscillators in the Ensoniq sound chip on the Apple IIGS.

volume name: The name of the volume directory.

wedge: A filled **arc,** one of the fundamental shapes drawn by QuickDraw II.

window: A rectangular area that displays information on a desktop. You view a document through a window. You can open or close a window, move it around on the desktop, and sometimes change its size, scroll through it, and edit its contents. The area inside the window's frame corresponds to the **port rectangle** of the window's **GrafPort.**

window frame: The outline of the entire window plus certain standard window controls.

Window Manager: The Apple IIGS tool set that updates and maintains windows.

window menu bar: A menu bar that appears at the top of the active window, below the **system menu bar.** Window menu bars can contain document titles, applications, and functions.

window record: The internal representation of a window, where the Window Manager stores all the information it needs for its operations on that window.

word: On the Apple IIGS, a 16-bit (2-byte) data type. Compare **long word.**

x flag: One of three flag bits in the 65816 processor that programs use to control the processor's operating modes. In **native mode,** the setting of the x flag determines whether the index registers are 8 bits wide or 16 bits wide. See also **e flag** and **m flag.**

XOR: Exclusive-OR. A Boolean operation in which the result is TRUE if, and only if, the two items being compared are unequal in value.

X register: One of the two index registers in the 65816 microprocessor.

Y register: One of the two index registers in the 65816 microprocessor.

zero page: The first page (256 bytes) of memory in a standard Apple II computer (or in the Apple IIGS when running a standard Apple II program). Because the high-order byte of any address in this part of memory is zero, only a single byte is needed to specify a zero-page address. Compare **direct page.**

zoom box: A small box with a smaller box enclosed in it, found on the right side of the **title bar** of some windows. Clicking the zoom box expands the window to its maximum size; clicking it again returns the window to its original size.

zoom area: The window subregion that corresponds to the **zoom box.**

Bibliography

Here are four categories of books that can help you learn more about desktop programming on the Apple IIGS. We list only a few titles in each category; many more books are available.

Several of the books listed below are part of the Apple IIGS technical suite. See "Introduction to the *Programmer's Introduction*" for other titles in the suite.

Apple IIGS technical manuals

In this category, the most important book for writing programs is the toolbox reference manual. You cannot write desktop applications without it.

Apple IIGS Firmware Reference. Reading, Mass.: Addison-Wesley, 1987.

Apple IIGS Hardware Reference. Reading, Mass.: Addison-Wesley, 1987.

Apple IIGS ProDOS 16 Reference. Reading, Mass.: Addison-Wesley, 1987.

Apple IIGS Toolbox Reference, Volumes 1 and 2. Reading, Mass.: Addison-Wesley, 1987.

Technical Introduction to the Apple IIGS. Reading, Mass.: Addison-Wesley, 1986.

Programming manuals

This category includes both books and development environments. APW (Apple IIGS Programmer's Workshop) is essential if you plan to compile and modify HodgePodge. The usefulness of the other books depends on which language(s) you are programming in. This list is by no means complete: additional books for these and other Apple IIGS programming languages are available.

❖ *APDA:* Books marked "[APDA]" are distributed through the Apple Programmer's and Developer's Association. See Chapter 9.

Apple IIGS Programmer's Workshop Assembler Reference. Cupertino, Calif.: Apple Computer, Inc., 1987. [APDA]

Apple IIGS Programmer's Workshop C Reference. Cupertino, Calif.: Apple Computer, Inc., 1987.* [APDA]

Apple IIGS Programmer's Workshop Reference. Cupertino, Calif.: Apple Computer, Inc., 1987.* [APDA]

Eyes, David, and Ron Lichty. *Programming the 65816, Including the 6502, 65C02, and 65802.* New York: Prentice Hall Press, 1986.

Jensen, Kathleen, and Niklaus Wirth. *Pascal User Manual and Report.* 3rd.ed. New York: Springer-Verlag, 1982.

Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language.* Engelwood Cliffs, N. J.: Prentice-Hall, 1978.

ORCA/ Pascal: A Pascal Compiler and Development System for the Apple IIGS. Albuquerque, N. M.: The Byte Works, Inc., 1987.*

TML Pascal for the Apple IIGS: User's Guide and Reference Manual (APW version). Jacksonville, Fla.: TML Systems, Inc., 1987.*

* Includes software.

All-Apple manuals

Here, note especially the Human Interface Guidelines book—it contains a wealth of information to help you design your program for maximum effectiveness and ease of use.

Apple Numerics Manual. Reading, Mass.: Addison-Wesley, 1987.

Human Interface Guidelines: The Apple Desktop Interface. Reading, Mass.: Addison-Wesley, 1987.

Macintosh programming manuals

These books are included because many desktop concepts, although developed originally for the Macintosh, are directly applicable to the Apple IIGS. Remember, though, that details of implementation are often quite different!

Chernicoff, Stephen. *Macintosh Revealed. Volume One: Unlocking the Toolbox.* Hasbrouck Heights, N. J.: Hayden Book, 1985.

Chernicoff, Stephen. *Macintosh Revealed. Volume Two: Programming With the Toolbox.* Hasbrouck Heights, N. J.: Hayden Book, 1985.

Inside Macintosh, Volumes I–V. Reading, Mass.: Addison-Wesley, 1987.

Programmer's Introduction to the Macintosh Family. Reading, Mass.: Addison-Wesley, 1988.

Index

B

C

D

E

F

G

H

M

Q

R

S

X

Y

Z

Programmer's Introduction to the Apple IIGS®

The Official Publication from Apple Computer, Inc.

The Apple IIGS® personal computer—with its high speed, expandable memory, super-high-resolution color graphics, and extensive Toolbox of programming routines—has created a powerful new programming environment. Written for programmers and software developers, *Programmer's Introduction to the Apple IIGS* explains essential concepts and provides tips and practical advice from the designers of the Apple IIGS Toolbox and the new ProDOS® 16 operating system.

To illustrate these concepts, *Programmer's Introduction to the Apple IIGS* includes three complete versions of a functioning sample program called HodgePodge—in 65816 assembly language, C, and Pascal. Using HodgePodge as an example, the book demonstrates:

- Event-driven programming techniques
- Programming with the Apple® Desktop user interface
- Effective use of the Apple IIGS Toolbox
- How to write segmented, relocatable code that will make programs run more efficiently
- File handling
- Memory management
- How to write specialized programs such as shells and desk accessories

Appendices include complete source code listings of HodgePodge in all three languages, as well as hints on converting Apple Macintosh® programs and earlier Apple II programs for the Apple IIGS.

Programmer's Introduction to the Apple IIGS contains a 3.5-inch disk that includes both source code and executable versions of HodgePodge.

030-3122-A
Printed in U.S.A.

Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, California 95014
(408) 996-1010
TLX 171-576

53295
9 780201 177459
ISBN 0-201-17745-5

Addison-Wesley Publishing Company, Inc.