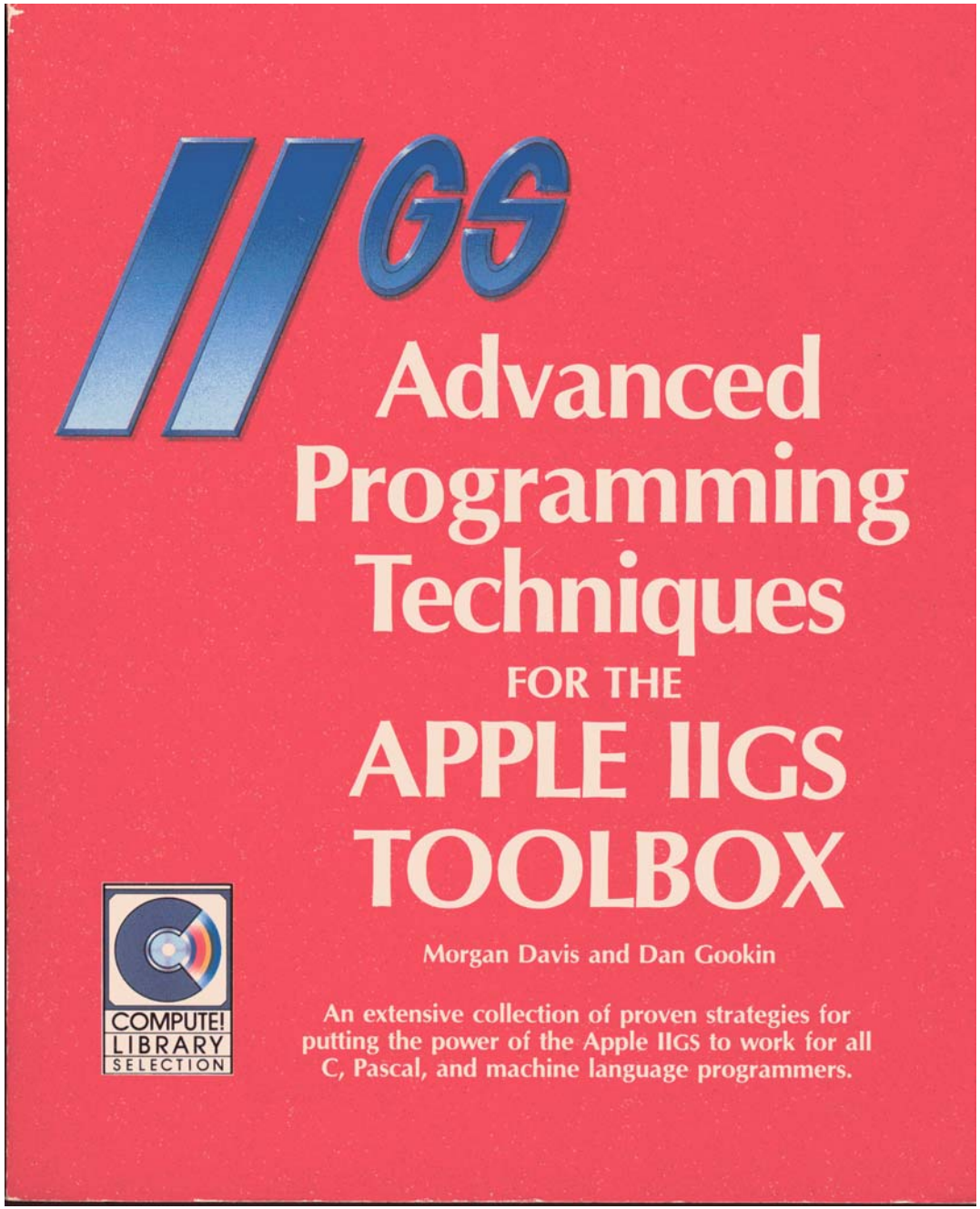


Advanced Programming Techniques for the APPLE IIGS TOOLBOX

Morgan Davis and Dan Gookin

COMPUTE! Publications, Inc.
A Capital Cities/ABC, Inc. Company
Greensboro, North Carolina

Cover design: Lee Noel, Jr.
Editor: Robert Bixby

Copyright 1988, COMPUTE! Publications, Inc. All rights reserved.

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

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-87455-130-7

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

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

COMPUTE! Publications, Inc., Post Office Box 5406, Greensboro, NC 27403, (919) 275-9809, is a Capital Cities/ABC, Inc. Company, and is not associated with any manufacturer of personal computers. Apple is a registered trademark and Apple IIGS is a trademark of Apple Computer, Inc.

APW C is available only to members through Apple Programmer's and Developer's Association, 290 SW 43rd St., Renton, WA 98055.

TML Pascal is a product of TML Systems, Inc., 4241 Baymeadows Rd., Suite 23, Jacksonville, FL 32217.

Contents

Foreword v

Chapters

1. Introduction 1

2. Programming Subtleties 9

3. How Programs Work 25

4. About the Toolbox 43

5. A Matter of Language 61

6. The DeskTop 77

7. Memory Management 95

8. Pull-Down Menus 113

9. Windows 145

10. Dialog Boxes 187

11. Controls 241

12. Interrupts 271

13. Desk Accessories 309

14. ProDOS 329

Appendices

A. Apple's Human Interface Guidelines 371

B. Tool Sets in the Apple IIGS Toolbox 384

C. Error Handling 391

D. Error Codes 396

E. Event and TaskMaster Codes 401

F. QuickDraw II Color Information 408

Index 411

Foreword

If you have a solid understanding of machine language, Pascal, or C, you'll find *Advanced Programming Techniques for the Apple IIGS Toolbox* invaluable in helping you to improve your Apple IIGS programming skills. This book examines in detail the structures and procedures necessary to make the Apple IIGS perform for you. Although the machine has been available for over a year and a half, the programming market for the Apple IIGS is still wide open. Programs that take full advantage of the machine's capabilities have only begun to appear.

"This book is not for the beginner," the authors warn early in the first chapter. But intermediate- to advanced-level programmers will find *Advanced Programming Techniques for the Apple IIGS Toolbox* packed with solid information on this fast-selling machine. This book delves into the intricacies of the powerful set of libraries known collectively as the Toolbox.

The program examples given here are ready to be merged with your own program code, giving your programs greater flexibility within the IIGS operating system. Mirroring the flexibility of the machine, this book provides a nonlinear approach that allows you to turn to your area of immediate interest, begin learning the things you need to know, and produce the program you're trying to write, in your choice of languages. Along the way, you'll learn about the other languages and the inner workings of the operating system.

Inside this book is information covering DeskTop applications, the mouse, pull-down menus, windows, dialog boxes, controls, special applications like interrupts, and the use of ProDOS 16. You'll also find a condensed version of Apple's Human Interface Guidelines as well appendices packed with technical information.

Advanced Programming Techniques for the Apple IIGS Toolbox is a treasure trove of inside information of all kinds. You'll value its insights into this exciting machine. This is the book every intermediate- to advanced-level programmer needs to enhance his or her programming skills.

Chapter 1

Introduction

To use this book, you should have an assembler, or you should have a Pascal or C language compiler. The software mentioned in this book are the *APW (Apple Programmer's Workshop)* C and machine language development kit and *TML Pascal*. These programs and their associated utilities

are available at the addresses given on the copyright page.

This book is intended to be a complement to *COMPUTE!'s Mastering the Apple IIGS Toolbox*, a tutorial on programming the Apple IIGS Toolbox. This book goes more deeply into the intricacies of using this powerful set of libraries to put a professional polish on applications. It's both a reference and a book of advice on designing and building solid programs in machine language, C, and Pascal.

It's assumed that you've read *COMPUTE!'s Mastering the Apple IIGS Toolbox*, or you're already a highly skilled programmer of the Apple IIGS. If so, you're ready to begin a challenging and enjoyable programming adventure. Keep in mind that this book is not for the beginner.

If you haven't read *COMPUTE!'s Mastering the Apple IIGS Toolbox*, or one of the other worthy introductory texts on this computer, you'd be wise to purchase one and read it before venturing further into this book.

This book also assumes you have an Apple IIGS handy to test the routines. Your computer should have at least 512K of RAM, with one or two disk drives. A color monitor is more interesting to look at, but it is not a necessity.

What This Book Is About

This book provides programming advice for the Apple IIGS in three different languages: machine language, Pascal, and C. A solid understanding of one or more of these programming languages is required to be able to grasp the concepts in this book. You can't program the IIGS without them.

Although the Apple IIGS has the same, decade-old, proven BASIC as its ancestors, Applesoft BASIC is not an appropriate language for writing application programs. In fact, the only way to access the advanced techniques of the Apple IIGS from BASIC is by using in-line machine language, a technique that is not recommended, even for the most venturesome programmers.

If you are a BASIC programmer, you might be interested to know that two new BASICs were announced for the Apple IIGS as this book was being prepared. One, from TML Systems of Jacksonville, Florida, and a second from Apple. (There may be more BASICs forthcoming from other developers.) These BASICs are in their "beta test" stage at this writing (which means they are not yet

ready for general release; they have too many bugs).

The Toolbox. The key to programming the Apple IIGS is its Toolbox. The Toolbox contains hundreds of routines and functions that provide the core of programming. Programming the Toolbox is a central part of this book. And examples for programming the Toolbox are provided in C, Pascal, and machine language.

This book is not intended to be a Toolbox tutorial. Instead, it was written to acquaint readers, programmers, and Apple enthusiasts with the finer aspects of programming the machine.

The scope of this book is limited to these general areas: the DeskTop, graphics, low-level tools, and other rarely discussed aspects of the Apple IIGS. These areas are well covered and offer an in-depth look at the inner workings of the computer.

Who This Book Is For

This book will benefit Apple IIGS owners who understand machine language, Pascal, or C. Any one of these languages will do, and, after reading a few chapters, you'll probably learn more about the others.

As mentioned above, you'll need an Apple IIGS with at least 512K. The IIGS is currently being sold with 256K. Even Apple admits this isn't enough. Producing a 256K machine was a decision made to keep the base unit as inexpensive as possible. Another decision based on economy was the choice of the 65816 microprocessor, which is only rated at 2.8 MHz. The engineers could have used a faster chip, but it would have added $100 to the price of the computer.

It's recommended that when you buy a memory card for your IIGS, you pack it full of memory. Memory is relatively inexpensive. For the cost of 16K of RAM in 1980 you can easily put over 1024K (one megabyte) of RAM into your IIGS.

Finally, this book is for anyone excited about the Apple IIGS. It's been over a year and a half since the machine was introduced. Exciting and interesting programs are only starting to appear. With the knowledge you'll gain from this book, you'll soon add your own programs to the growing list of applications for the Apple IIGS.

Unlike *COMPUTE!'s Mastering the Apple IIGS Toolbox*, this book doesn't rely upon complete programs to convey ideas. Instead, only

small program examples and snippets of code are used. It's assumed you'll be able to put the example pieces together in your own way when creating applications. The examples listed in this book all work and will function in any program you write.

Though you may be tempted to dive into programming without preparation, you'll gain more if you read the text dealing with each program example before cutting and pasting code. While it looks easy and simple, the Toolbox routines have interdependent relationships with each other. To understand how one tool relies upon another, read the text before and after an example. Then you should be able to adapt it successfully to your use.

This book is half reference and half tutorial. The "refertorial" approach makes this book modular. You can start reading at any point. For example, if you'd like to know how to put a custom icon in one of your dialog boxes, turn to the chapter on dialog boxes, and you'll find an example. Replace the graphic in the example with your own, and you'll have a custom icon. This entire book works that way.

On a larger scale, this book is divided into four major parts, each part concentrating on a specific area of programming the Apple IIGS and the Toolbox. Each part is further broken down into individual areas that cover specific topics. Within each area are individual examples and routines you can use to help you understand and program the Apple IIGS.

You can read any section that interests you, provided that you have taken the time to understand the fundamentals. If any information overlaps or is covered elsewhere, you'll be directed to the proper part and section. Most of the groundwork is covered in this, the first section, so naturally, you should take the time to read through the introductory material. When you're finished, you may proceed through the book at your own pace and in any order.

The book is divided as follows:

- The early chapters offer a general introduction. When you're finished reading them, you'll understand how information is presented in this book. For example, one chapter demonstrates how Toolbox routines are documented in this book for all three programming languages. You may skim that section and return to read it in detail if a concept in a later chapter confuses you. You'll also find a great deal of interesting trivia and general background information in this section.

- The middle portion of the book covers DeskTop applications and using the mouse. It details DeskTop programs, pull-down menus, windows, dialog boxes, and controls. This section covers many concepts unique to the IIGS. Don't be surprised if you find yourself referring to this part of the book often.
- The final chapters go into detail on special applications—those features of the Apple IIGS that don't have a category of their own. This part covers such advanced topics such as interrupts and desk accessories. At the end of this section is a chapter on ProDOS 16. Though unrelated to the Toolbox, ProDOS is as much a part of the Apple IIGS as anything mentioned so far. Loading and saving files from and to disk and other file-management techniques are mentioned in the ProDOS chapter.
- The Appendices provide a reference to the first part of the book. You'll find a version of Apple's Human Interface Guidelines in Appendix A. While not an exact duplicate, this version highlights the most important parts of the Human Interface Guidelines, ensuring that your programs will fall in line with Apple's recommendations for all DeskTop applications.

Interesting trivia surrounding the Apple IIGS is just now rising to the surface. Where appropriate, comments and insights have been included in the main body of the text, but when they are tangential or circumstantial to the topic at hand, they will be set aside in boxes. They were included to give a better understanding of Apple IIGS hardware and software construction.

Conventions Used in This Book

Every effort has been made to maintain notations and conventions used in *COMPUTE!'s Mastering the Apple IIGS Toolbox*. For example, the majority of the numbers you'll see in this book are in hexadecimal (base-16) notation. All hexadecimal numbers are preceded by a $ (dollar sign), and they contain the numbers 1–9 and the capital letters *A–F*, which stand for the values 10–15.

There are three types of hexadecimal numbers used in this book: bytes, words, and long words.

A byte value is a two-digit hexadecimal number ranging from $00 through $FF (0–255 decimal). A word value is a four-digit hexadecimal number ranging from $0000 through $FFFF (0–65535 decimal). Words are composed of two bytes, the most significant byte (MSB) and the least significant byte (LSB). In the word value $FACE, $FA is the MSB and $CE is the LSB.

Long words are new to the Apple II. A long word is an eight-digit hexadecimal number equivalent to two words or four bytes. It ranges in value from $00000000 through $FFFFFFFF (0 through 4,294,967,295 decimal). Long words are composed of two words—the high-order word and the low-order word. In the long-word value $00E100A8, $00E1 is the high-order word, and $00A8 is the low-order word. Long words are primarily used in the Apple IIGS to denote memory locations. Refer to the section on memory addressing in the next chapter for details.

Though not a type of number (or size), the Toolbox uses *logical*, or *Boolean*, values to represent the true or false result of certain operations. A true value is any value not equal to 0. Commonly, true is set to the hexadecimal word value of $8000. A false value is 0.

Logical True = $8000 or any nonzero value
Logical False = $0000

When the Toolbox returns a logical true or false value, the actual numbers returned are as listed above. As might be expected, there are times when the computer breaks this rule and returns 0 for true and a nonzero value for false. When this happens, a note will be provided to warn you about it.

One final convention concerns the program listings in this book. Line numbers are included with all program listings above a certain size. Unless specified, the line numbers are not to be entered (when you type in the examples) or considered as part of the source. The line numbers are intended for use as references from the text. Again, where there are exceptions, they will be noted.

Books Worthy of Note

At this writing, there are few books on the subject of programming the Apple IIGS. However, the books listed below are recommended for anyone interested in programming the Apple IIGS:

- *COMPUTE!'s Mastering the Apple IIGS Toolbox,* Dan Gookin and Morgan Davis (1987, COMPUTE! Publications, ISBN 0-87455-120-X).
- *COMPUTE!'s Apple IIGS Machine Language for Beginners,* Roger Wagner (1987, COMPUTE! Publications, ISBN 0-87455-097-1). Roger wrote the definitive machine language book years ago. This book carries on the tradition.
- *COMPUTE!'s Guide to Sound and Graphics on the Apple IIGS,* William B. Sanders (1987, COMPUTE! Publications, ISBN 0-87455-096-3). Though lacking extensive Toolbox programming examples, this book contains a wealth of information on fundamental Apple IIGS sound and graphics.
- *Apple IIGS Technical Reference,* Michael Fischer (1986, 1987; McGraw-Hill; ISBN 0-07-881009-4). One of the first books to appear on the market, this book is an excellent hardware and software reference to the Apple IIGS. Some of the material is outdated, but it's still worthy.
- *Programming the 65816,* David Eyes and Ron Lichty (1986, Prentice-Hall, ISBN 0-89303-789-3). The ultimate reference to the 65816, with programming examples and the best command reference of any microprocessor book.

Chapter 2

Programming Subtleties

The purpose of this chapter is to acquaint you with some things you should know before attempting to program the Apple IIGS. This information—background material, plus some interesting tidbits—was gathered over a long period of time during visits to the offices of

Apple Computers and through research in virtually every book available on this machine. The material listed here is the distillation of this research. (For more detailed explanations, refer to *COMPUTE!'s Mastering the Apple IIGS Toolbox*.)

How the Apple IIGS Is Different from Other Apples

The Apple II is an "ancient" and honored computer, with a respectable lineage dating back just a little over ten years. Generally speaking, the Apple IIGS is simply the latest incarnation of the Apple II. It has a faster and more powerful microprocessor, better graphics, and advanced sound capabilities, but it can run Apple II software and accommodate Apple II hardware. It also has a *tool set* of programming routines that allow it to mimic its distant cousin, the Macintosh.

In fact, the Apple IIGS is actually one step closer to the Macintosh computer than simply an improvement on the older Apple II design. While the computer is still compatible, the DeskTop extensions, the graphics, and the sound found in the Toolbox routines separate the Apple IIGS from the rest of the Apple II family.

The Apple IIGS is an evolutionary computer in terms of design and implementation. It's difficult to document. The machine's operation is different now from its operation a few months ago. This implies that a shortcut or trick you learn today might not work tomorrow.

Apple is constantly working on the IIGS. Internal modifications are being made, and the firmware and tool sets are constantly being upgraded and modified. Because of this, a warning is offered: Do not stray from the standard.

The Macintosh is another evolutionary machine. The first Macintosh, introduced in 1984, could not compare to the powerful machines Apple makes today. While the Apple IIGS probably won't have the same expensive upgrades the Mac had, it will share the technological advances of its distant relative. Apple has assured its developers that as long as they stick to the standards, their programs will run on all future releases of Apple II computers.

A good example of programmers not sticking to the standards is in the area of the super-hi-res graphics display. Apple has repeatedly warned against finding the screen's secret location in memory. Why? Because it may change in the future. The proper way to use the graphics screen is through the Toolbox. Yet, some

developers consider the Toolbox routines slow. For this reason, they prefer to access the screen directly so their programs will work faster. By doing so, they run the risk that in the future they may not work at all.

As long as you adhere to the techniques and programming examples used in this book, you can be assured that your applications will have a long and healthy life—as long as the Apple II series stands. According to Apple, it will last forever.

Here is an abridged history of the Apple computer: The first Apple computer, the Apple I, was actually a circuit-board kit that sold for $666.66 in July 1976.

The Apple II, which came in a case with a keyboard and power supply, was unveiled at the West Coast Computer Faire in April 1977. It came with its own BASIC, 4K of memory, color graphics, and game paddles. The Apple II was available for sale to the general public in June of 1977 for $1,298.

In June 1979, the Apple II+ was introduced. It had an improved ROM, could handle up to 48K of RAM, and retailed for $1,195. In October of that year, the software program *VisiCalc* became available.

The Apple IIe was presented in January 1983. It came with 64K, which could be upgraded to 128K. Also included was a lowercase keyboard option, as well as an 80-column screen. The IIe retailed for $1,395.

The Apple IIc portable was introduced in April 1984. A marketing genius came up with the slogan "Apple II Forever."

In September 1986, the Apple IIGS was introduced. Nine years after the first Apple, the IIGS was priced at $999, came standard with 256K of memory, a keyboard, a mouse, and a mountain of potential.

Graphics

The Apple IIGS contains all the graphics modes of its predecessors, plus a new high-resolution graphics mode. The *super-hi-res* screen is used for all the IIGS graphics and provides a Macintosh-like environment. The responsibility for producing these graphics is given to the Video Graphics Controller (VGC) chip.

The VGC has a big job. Not only does it control the super-hi-res graphics screen, it handles the older Apple II graphics modes,

as well as dealing with two different types of interrupts. The VGC allows the Apple IIGS with a color monitor to have a different text, background, and border colors. It also provides foreign language character sets and international video output (for European countries). It's a remarkable piece of engineering.

The following chart shows the Apple IIGS text and graphics screens and their resolutions. The Apple IIe and IIc are both represented by the IIe. The resolution is shown as horizontal pixels by vertical pixels.

Graphics Mode	Resolution	Colors	II+	IIe	Apple IIGS
Text screen	40 × 24	2 (16 for IIGS only)	*	*	*
Text screen	80 × 24	2 (16 for IIGS only)		*	*
Lo res	40 × 48	16	*	*	*
Double lo res	80 × 48	16		*	*
Hi res	280 × 192	6	*	*	*
Double hi res	560 × 192	1		*	*
Super hi res	320 × 200	16			*
Super hi res	640 × 200	4			*

The 80-column text screen was available to Apple II+ owners via a special 80-column card. However, with the introduction of the Apple IIe, and later the IIc, the 80-column text format became standard.

The lo-res mode displayed graphic "bricks" called pixels (though a pixel usually refers to a small dot). In the hi-res mode, the colors of the pixels and other graphics variations depended on a number of things, most of which are too specific to go into in this book. (A good book on the subject is *COMPUTE!'s Guide to Sound and Graphics on the Apple IIGS* by William B. Sanders.)

Super Hi Res

This book is concerned with the super-hi-res screen on the Apple IIGS. It has two modes: low and high resolution. The high-resolution mode has a pixel resolution of 640 × 200. This mode provides four colors. However, by using a process known as *dithering*, more colors can be produced on the screen. Also, by altering certain attributes of the screen, up to 256 different colors can be produced on one super-hi-res screen.

Questions almost every computer owner asks are "Where is the screen in memory? Is it bitmapped?"

As explained above, this knowledge will not come in handy. However, to be accommodating, a few secrets can be revealed.

At this writing (it will almost certainly change), the super-hi-res graphics screen is located in memory bank $E1, at offset $2000. (Refer to the section on memory management later in this chapter for further explanation of this memory reference.) To activate the screen from Applesoft BASIC, you can type the following (the bracket is the Applesoft prompt):

```
]CALL -151
```

That will put you in the monitor. Type the following to reference memory bank $E1 (the asterisk is the monitor's prompt):

```
*E1/0000
```

Now, activate the super-hi-res mode by putting the byte value $C1 into memory location $C029, the New-video register:

```
*C029:C1
```

That will activate the super-hi-res screen, which implies that from here on you'll be typing "in the dark." Text will be invisible. Sometimes a pretty pattern will appear on the screen. Other times, the data previously on the super-high-res screen can be seen.

Now, any value poked into memory locations $2000–$9CFF will appear on the screen as a pixel, series of pixels, pattern, or color. For example, putting the value $00 into memory location $60B0 might put a black dash near the middle of the screen:

```
*60B0:00
```

You can experiment with your own values (within the proper range of $2000–$9CFF). When you want to return to normal, you must poke a value of $01 back into memory location $C029:

```
*C029:1
```

Or you can type Control-T followed by the RETURN key.

Have fun, but remember the warnings.

The low-resolution mode of the super-hi-res screen has the same vertical resolution (200 pixels) but only half the horizontal resolution of the high-resolution mode. It does, however, have more colors—up to 16—chosen from over 4096 possibilities. By using dithering you can squeeze even more colors out of the low-resolution graphics mode.

You might hear the 640 mode of the super-hi-res screen referred to as "80 columns," and the 320 mode as "40 columns." While this is entirely inaccurate, it does express the appearance of the two modes. In fact, by displaying text on the graphics screen using different fonts, your actual text-screen size varies from 16 rows by 63 columns to 32 rows by 132 columns. (Text is displayed on this screen using a combination of the QuickDraw II and Font Manager tool sets. The size of the text is determined by the font chosen.)

The QuickDraw II tool set in the Apple IIGS Toolbox is responsible for all graphics appearing on the screen. Drawing lines, circles, boxes, arcs, and patterns is easy once you learn how to use the over-250 routines provided by QuickDraw II. By using QuickDraw, you save development time. It eliminates the necessity of writing graphics primitives. The basic code has been written for you. Also, sticking to the QuickDraw routines ensures that your graphics programs will work on and be compatible with all future releases of the Apple IIGS.

Sound

To make the Apple IIGS more competitive and attractive to the marketplace, something had to be done about sound. Sound was one thing the Apple II series of computers barely provided.

For years, Apple II programmers created sound by *bit twiddling*. The speaker has a memory location—$C030. By peeking this location from Applesoft BASIC or by examining this location using assembly language, the speaker could be made to tick (see following box). A rapid succession of ticks produced a tone. By varying the number of ticks and their duration, a chorus of tones could be created. This complicated-yet-simplistic method of producing sound got the job done, yet there had to be a better way.

To tick the speaker in Applesoft BASIC, the PEEK statement is used. PEEK returns the byte value of a specific memory location, in this case $C030, which is 49200 decimal:

```
A = PEEK (49200)
```

The actual value of A can be discarded. By repeatedly reading memory location 49200, as well as varying the interval between PEEKs, the speaker can produce a variety of tones. Note that PEEK's counterpart, POKE, has no audible effect on memory location 49200.

The following program shows how the PEEK statement in Applesoft BASIC can be used to tick the speaker:

```
10 FOR X = 1 TO 10
20 A = PEEK (49200)
30 FOR T = 1 TO 10 : NEXT T
40 A = PEEK (49200)
50 NEXT X
```

The two PEEK statements in lines 20 and 40 tick the speaker. Line 30 contains a delay that produces the pitch of the tone: Increase the delay, and the pitch deepens; decrease the delay, and a higher pitch is produced. The main FOR-NEXT loop between lines 10 and 50 sets the duration of the tone.

The better way turned out to be the Ensoniq 5503 Digital Oscillator Chip (DOC) included with the Apple IIGS. This is the same chip that appears in many of Ensoniq's synthesizers and MIDI (Musical Instrument Digital Interface) equipment.

The DOC contains 32 oscillators. These are paired to form 15 voices, each capable of producing its own sound (like 15 separate instruments in a band). The sixteenth voice is used internally for timing purposes.

Also included with the DOC is 64K of RAM referred to as *sound memory*, or *sound RAM*. Into this special area of memory can be placed various waveform patterns or even digitized samples of analog sounds such as a human voice.

The DOC can be programmed on two levels. Low-level programming involves reading and writing to the DOC's sound RAM and altering its registers directly. This method is complex, yet it's proven. In fact, the majority of the Apple IIGS sound applications available use this technique. The second way to program the DOC is using the Apple IIGS Toolbox. This is the preferred way. The advantage of Toolbox routines becomes clear when you consider that three lines of code are required to play a note using the Toolbox and 30 or more lines of code and data statements would be required to play the same note using low-level routines. But there is a problem: The Toolbox sound routines aren't finished.

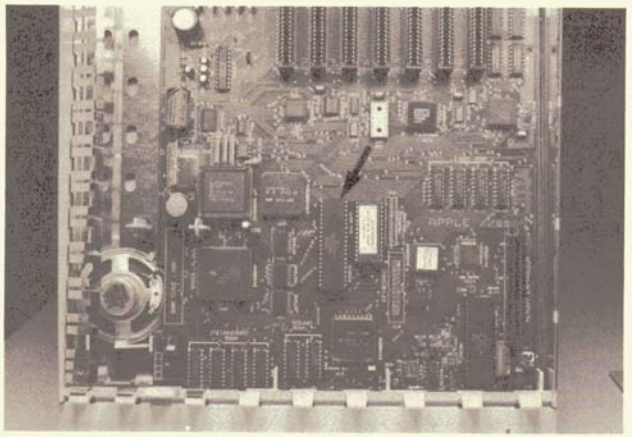
Soon, you'll be able to choose from a variety of sounds and tones as easily as opening a window. Apple is fast at work completing the sound routines. Unfortunately, they won't be finished in time for inclusion in this book.

> The sound lab in the Advanced Technologies section of Apple Computer is impressive. The goal of the researchers is to create a sound environment for computers that is as advanced as the computer's graphics capabilities. While graphics have continually progressed, and programming the graphics has become easier, sound continues to be an orphan.
>
> In the lab, they're concentrating on not only making sound easier to program, but on how to tailor sound toward specific applications. According to one of the researchers, "Any computer can go 'beep.' What about other sounds? How can they enhance the performance of a piece of software? How can sound help a user better interact with a program?"
>
> Sadly, all this technological magic is being worked out on a Macintosh II, not an Apple IIGS. The researchers want you to know, however, that all information discovered will be shared with the IIGS development team. You may see interesting and exciting sound advancements on this computer in the near future.

The 65816 Processor

The actual brain of the Apple IIGS is the 65816 microprocessor. It's the latest generation of the 6502 series of processors. This family began with the 6502 microprocessor used in the first Apple computer. Since that time, the chip has been improved upon. It became faster and able to address megabytes of memory and handle 16-bit-wide operations.

Figure 2-1. Apple IIGS Motherboard with 65816 Pointed Out

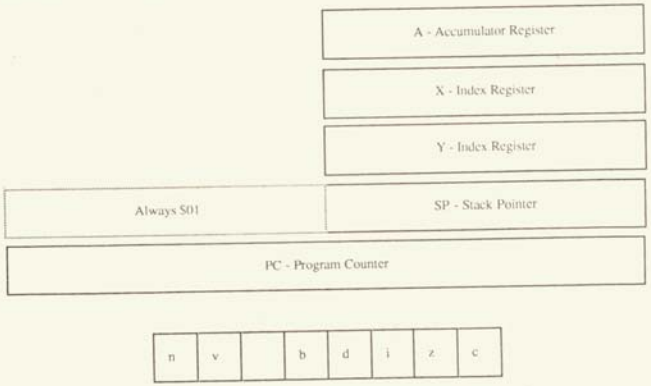


The 65816 is the brain of the Apple IIGS.

To maintain compatibility with the older 6502 chips (and the software that ran on them), the 65816 can emulate a 6502. In the emulation mode, it behaves exactly as a 6502 would, with very few exceptions. While emulating its ancestor, the Apple IIGS works on eight bits of data at a time and can access only 64K of memory.

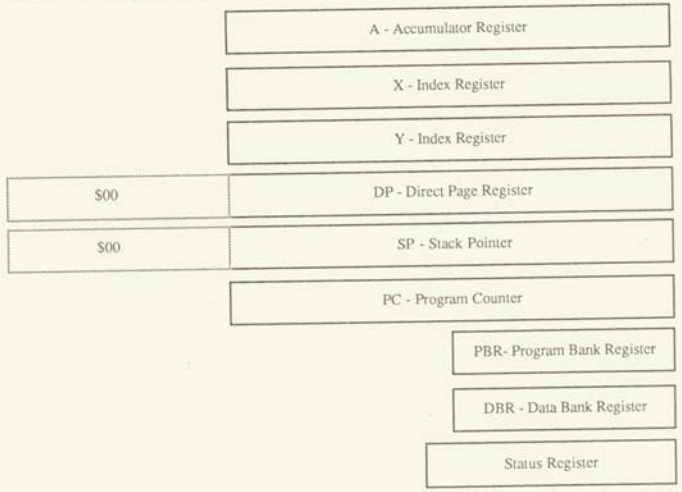
Note that while the 65816 is capable of emulating the 6502, older machines using the 6502 cannot run 65816 machine language programs. In fact, most of the 65816 machine language instructions are not defined for the 6502. Running a 65816 program on one of those machines (which would be hard to do in the first place) would crash the computer.

Figure 2-2. Diagram of 6502



The 6502 chip used in older Apple II machines can only handle eight-bit operations.

Figure 2-3. Diagram of 65816



The 65816 can handle 16-bit operations as well as emulate the 6502.

When programming, it's possible to switch emulation on and off, as well as configure the A, X, and Y registers to either 8 or 16 bits. In machine language, this is done manually by setting the 65816 to emulation or native mode and by setting or clearing the register configuration bits. If you use the *APW* assembler, special assembler directives must be used to ensure that all following code is interpreted properly for the emulation mode. (See the *APW* manual for details.)

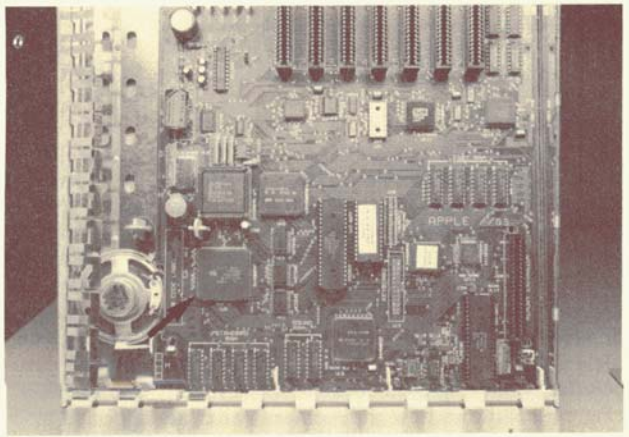
When programming in Pascal or C, emulation or native mode selection is taken care of automatically, either by default or through certain directives, depending on the software used. It's not necessary to ensure the processor is in one mode or the other when programming in Pascal or C.

To access the Toolbox, the 65816 must be in its native mode and all registers must be configured to 16 bits.

Apple IIe Emulation

One of the smartest things Apple Computer has done is to ensure that the software used on older Apple computers will work on new machines. Lack of compatibility has killed more than one microcomputer.

Figure 2-4. Apple IIGS Motherboard with Mega II Pointed Out



The Mega II: An Apple IIe all on one chip.

Programs that ran on the Apple II can run on the Apple II+. Apple II+ programs run on the Apple IIe and IIc. And the majority of those programs (about 90 percent) still run on a brand-new Apple IIGS. The reason for this is that the Apple IIGS contains a custom chip called the Mega II. The Mega II is an Apple IIe all on one chip.

Operation of the Mega II is transparent as far as programming the machine goes. While running older Apple II software, the Mega II takes charge and causes the machine to be an Apple IIe. When running Apple IIGS software, the Mega II does handle some operations. For the most part, however, its purpose is to emulate an Apple IIe and provide compatibility for older applications.

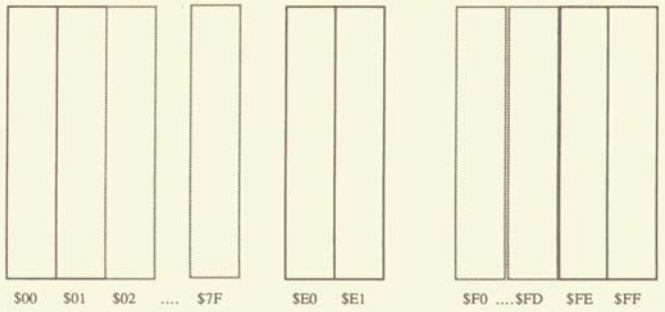
> The Mega II has an interesting history. Apparently, the Mega II took the Apple IIGS design team by surprise. People "upstairs" requested that the Mega II (supposedly designed for some other project) be used in the Apple IIGS. Because of this addition relatively late in the IIGS design, the Mega II chip contains many features made redundant by the VGC video chip.
>
> A slightly less-than-delighted design team did successfully incorporate the Mega II into the Apple IIGS, and it does perform its job very well. One question remains: What was the original purpose of the Mega II? A one-chip IIe or IIc, perhaps? Only time will tell.

Memory Addressing

The Apple IIGS has an alluring ability to address a tremendous amount of memory. This will be particularly attractive to programmers weaned on 64K (or even 128K) computers. Technically, the 65816 is capable of addressing 16 megabytes. The way the Apple IIGS is currently designed, only 8 megabytes of memory can be used for RAM, but that is still more than you're ever likely to need.

The eight megabytes of memory are divided into 128 separate banks of 64K each. The full 16 megabytes represents a total of 256 banks. Several of those banks are dedicated to the computer's ROM, possible ROM upgrades, and the Mega II chip. The memory map in Figure 2-5 shows how the memory banks are allocated in the Apple IIGS.

Figure 2-5. Memory Banks in the Apple IIGS



Each memory address (location) in the computer's RAM is represented by a bank number and an offset within that bank. For example, address $000200 indicates memory location $0200 in bank $00, the first bank of memory. Memory location $00A8 in bank $E1 is expressed as $E100A8. The first byte value represents the bank number; the second word value indicates an offset within that bank.

It's assumed that a leading $00 precedes all memory addresses. Because $E100A8 is not a true long-word value, the actual address is $00E100A8. However, because the MSB of the high-order word is always $00, it's usually left off (or assumed).

Allocating and controlling all this memory is the job of a very special tool set called the Memory Manager. One of the most important tool sets in the Toolbox, the Memory Manager is responsible for divvying up and setting priorities to blocks of memory. It's so well implemented that you need not know the exact location of a memory block. The Memory Manager takes care of all that for you. Blocks of memory can be moved, deactivated, or purged all via a call to the Memory Manager.

More details about memory and the Memory Manager can be found in Chapter 7.

Because the Apple IIGS currently comes only with 256K on the motherboard, you'll need to upgrade your machine's memory (as has been previously recommended). When you upgrade, you'll probably purchase a RAM card that allows you to use 256K RAM chips. Eight of these chips are equal to 256K of memory. The Apple IIGS considers 256K to be four banks.

As you add memory, the IIGS automatically assigns that memory to banks, beginning with bank $02. (Remember that you already have four banks of memory. Banks $00 and $01 are built-in FPI RAM, and the Mega II RAM and I/O are located in banks $E0 and $E1.) The typical memory card comes with at least four blocks which can each hold 256K of memory, making it capable of holding up to one megabyte of memory—16 banks of IIGS memory.

Memory cards with more than four blocks of 256K may cause some problems with future releases of the Apple IIGS. According to its designers, a memory card should have a maximum of four blocks of 256K. But certain hardware developers thought they could put more on a memory card. While the memory upgrade cards will still function, and the IIGS will be able to make use of the extra memory, some problems may result.

The best way to avoid problems when using a memory card with more than four blocks of 256K is to assign the extra memory as a ramdisk. This can be done using the Control Panel's ramdisk.

For example, the development systems this book was tested on contained RAM cards with 1.75 megabytes of RAM on them (six blocks of 256K). With an 800K ramdisk selected (the same size as the IIGS disk drive), the rest of the memory fit easily into the four-block maximum, and there were no problems.

Operating Environment

The operating system for the Apple IIGS is ProDOS 16, a custom operating system for the IIGS based on Apple's ProDOS 8 (which used to be just ProDOS). ProDOS 16 is very similar to ProDOS 8. In fact, updating is as easy as copying the ProDOS 16 files onto your old ProDOS 8 disks or hard drive.

ProDOS 16 controls disks and manages the file system. It uses the same file structure as ProDOS 8 and will even recognize, load, and run ProDOS 8 files, such as *AppleWorks*. However, ProDOS 8 cannot run the ProDOS 16 files. (And ProDOS 16 will not run on an Apple IIe, IIc, or II+.)

Incidentally, ProDOS 16 serves as a file-management system and isn't a true operating system in the sense that UNIX, MS-DOS, or OS/2 are operating systems. In fact, in the old days, all programming tasks were taken care of by the Apple's built-in BASIC interpreter. A program was run by typing its name at the BASIC command prompt, prefixed by a hyphen:

```
]-APLWORKS.SYSTEM
```

The above BASIC command would run the *AppleWorks* program, provided an *AppleWorks* disk was in the disk drive. (Another method to run *AppleWorks* was to place the *AppleWorks* disk into the primary disk drive and reboot the computer.)

The Apple IIGS provides a better way to interact with your programs.

Since late 1987, Apple introduced a Finder program, similar to the operating environment of the Macintosh. In fact, if you're familiar with the Mac, the IIGS Finder looks like a color version of the Mac's. Programs, data files, and file folders (which contain subdirectories) all appear as graphics images on the screen. The Finder allows files and programs to be manipulated with relative ease as compared to the older, slower ProDOS utilities. And, not only can ProDOS 16 and native Apple IIGS applications be run using the Finder, but because Apple also included a copy of ProDOS 8 on the Finder disk, older Apple II applications can be run as well.

Unlike the Macintosh's Finder, however, the Apple II Finder does have some limitations. Most notable among them is that not all Apple II applications are based on the graphic DeskTop environment. Older applications—and even some new ones—still use the Apple's text screen. Most of the newer applications, including examples in this book, will use the graphic environment of the DeskTop.

Chapter 3

How Programs Work

One trait most avid computer programmers share is a love of solving puzzles. Most great programmers are also great puzzle solvers. The self-taught computer wizard can unravel mysteries and evoke programming incantations that make a machine perform magical feats.

These programmers are not satisfied with just getting the job done. They want to make code tighter, faster, more ingenious. This chapter is directed to them.

This chapter explains how programs work on the Apple IIGS. Of course, if the subject doesn't make any sense, please read on. Whether you're a programming wizard or just an apprentice, this chapter contains interesting background information on how the IIGS works, how programs are loaded, what happens when they start, and where they go when they die. It's a chapter full of secrets revealed and undercover skullduggery—ideal for the potential programming prodigy.

Anticipation

Before you can begin serious programming on the Apple IIGS, you will need at least one disk drive and a system disk. The programming tips in this book were tested on one of the original computers using system disk version 3.1, as well as one of the later ROM 01 machines, so it should be applicable to your machine.

Of course, by the time you read this, ROM version 09 and system disk version 86 might be available. Things change that quickly. But don't worry. The information in this book is still good and all of it applies.

When you turn on your Apple IIGS, it looks for the *startup slot*. This is one of the slots on the motherboard into which a disk drive should be plugged. A specific startup slot can be specified in the Control Panel, or you can set it to *scan*. When set to scan, the Apple IIGS will scan all slots for the appropriate startup device.

When scan is selected, the system begins looking for an I/O device starting with slot 7 and continues searching down to slot 1. For example, if you have a hard disk drive connected to slot 7, that will be the startup device. Otherwise, the scan continues with slot 6 (the old floppy drive slot), slot 5 (the 3½-inch drive slot), and so on.

If you have selected a specific slot from the Control Panel, your IIGS will look for a startup device in that slot only. This way, if you had a disk controller card in slot 6 and you wanted the computer to startup from that device, it would do so, regardless of what was in the other slots or what devices were plugged into the IIGS ports (on the back panel).

> The connectors on the back panel of the computer are really considered devices plugged into slots. In fact, if you run an old Apple IIe diagnostic program, it assumes you have every slot in the computer filled with specific devices, even though your IIGS may be totally empty inside.

Once the computer is turned on, its primary job is to find a disk drive. Once the disk drive is found, the computer checks to see whether a disk is in that I/O device. If not, or if the disk is of alien origin, the following message is displayed along with the Apple character bouncing back and forth across the screen:

```
Check startup device!
```

If a disk is found, the computer checks to see whether it's a boot disk, specifically, a ProDOS disk. If it's either a ProDOS 8 or 16 system disk, the system continues to load ProDOS from disk. If the disk is just a data disk (meaning there's no operating system present) the following is displayed:

```
*** UNABLE TO LOAD PRODOS ***
```

If this or the previous message is displayed, you should insert a ProDOS system disk into your disk drive and try again.

If you do have a bona fide ProDOS disk in the drive, your IIGS will attempt to load ProDOS into memory. For ProDOS 8, this is a very simple operation. For ProDOS 16, things are a little more complex.

> The original disk operating system for the Apple II computer was simply called DOS, for Disk Operating System. It went through various iterations until its final version, DOS 3.3, was replaced by ProDOS in early 1983.
>
> ProDOS was modeled after the SOS operating system Apple developed for the late Apple III computer. SOS stood for Sophisticated Operating System.
>
> SOS introduced the hierarchical file system of volumes and prefixes now used by ProDOS. In fact, SOS files and ProDOS 16 files have identical structures to a certain extent. And because the Apple III Pascal used a file system similar to SOS, ProDOS 16 can read Apple III Pascal files as well.

Booting ProDOS 8

Because the Apple IIGS is Apple IIe compatible (for about 90 percent of the programs, according to the literature), it can load and run a ProDOS 8 program just as if it were a IIe. Due to this compatibility, it's logical to assume that both ProDOS 8 and ProDOS 16 are initially loaded from disk in a similar manner.

The program (actually ROM code) that loads ProDOS into memory is called Boot ROM. These instructions are located on the disk's controller card. The actual memory location of the Boot ROM is in memory bank $00, at location $C000 plus $100 times the slot number. So, if slot 6 contains the disk's controller card, the Boot ROM is at memory location $C000 plus $100 × 6, or $C600. (All memory locations from here on are in bank $00 unless otherwise specified.)

Boot ROM has only one job: to read in the first one or two blocks of the disk (or hard disk) into memory. The contents of these blocks are copied to memory location $800. With its dying breath, the Boot ROM's last job is to perform a JMP instruction to the machine language routines (loaded from disk) at the address $801.

The routine loaded from disk is $200 bytes long and occupies memory locations $800 through $9FF. If the disk being booted is formatted for ProDOS (either version), the information loaded from disk is called the ProDOS Boot Loader. This code will read in the rest of block 0, as well as the entire contents of block 1 of the disk. However, the information on block 1 is used primarily by the Apple III computer as a means of booting into the SOS operating system.

> A block, the smallest unit of storage on a ProDOS disk, consists of 512 bytes of information. A sector, the smallest accessible unit of a DOS 3.3 formatted disk, holds only 256 bytes.

The Boot Loader's job, like the Boot ROM, is to load more information from disk—in this case, the rest of ProDOS. The ProDOS Loader searches the disk's *volume*, or *root*, directory for the file called PRODOS, which contains the ProDOS Relocator. If this file cannot be found, the following message is displayed:

```
*** UNABLE TO LOAD PRODOS ***
```

It's not unusual to see this when booting data disks. They're formatted for use with ProDOS, but aren't meant to be booted.

If the PRODOS file is found, it's loaded into memory locations $2000–$5BFF. And, like the Boot ROM, with the Loader's dying breath, it jumps to the machine language routines at address $2000 which make up the ProDOS Kernel Relocator.

The ProDOS Relocator is the program that prints the ProDOS version number and copyright on the screen. It does a number of other interesting things: evaluating RAM, determining the type of Apple computer you have, and so on. But its biggest job is to copy the ProDOS Kernel, the actual operating-system part of ProDOS, to high memory. It also sets up the System Global Page. Incidentally, when the Relocator is copying the Kernel image to high memory, it makes a grating sound on the computer's speaker.

Once the relocated Kernel is running, ProDOS 8 scans the volume directory of the disk for a system file with a .SYSTEM suffix. If a .SYSTEM file is found—BASIC.SYSTEM, for example—it's loaded into memory at location $2000, and then a JMP instruction is performed to that address.

If the .SYSTEM program is in fact BASIC.SYSTEM, the BASIC interpreter looks for a BASIC program named STARTUP in the volume directory. If found, that program is loaded into memory, and its instructions are executed.

This may seem like a very complex way of loading in something as simple as a BASIC program. Yet, nearly all microcomputers operate this way: First, a small bit of the disk is read, then a larger piece, and then, finally, the operating system is loaded into memory. It would probably be much more efficient to directly load the entire operating system when a computer starts, but not as flexible. Imagine all your data disks needing a 30-block boot sector simply to display the message, ******UNABLE TO LOAD PRODOS******.

> Actually, a better justification for loading ProDOS in pieces is to allow the system to run more than one operating system. For example, using this method, an alien operating system could have its own Boot Loader on the first two sectors of a disk. This custom Boot Loader could then look for a special Loader file on disk—something other than PRODOS. The new Loader file could then load itself into memory and the Apple IIGS would run a new operating system, such as the old Apple Pascal.

You'll really appreciate the speed with which ProDOS 8 loads, especially after you have encountered the apparently sluggish ProDOS 16.

Booting ProDOS 16

As they're started, ProDOS 8 and ProDOS 16 are remarkably similar. They have to be similar, so they are compatible and use the same disk structure. But bear in mind that although the Apple IIGS can boot ProDOS 8 disks and run ProDOS 8 applications, older Apple IIs cannot run ProDOS 16 nor can they run Apple IIGS applications.

> Actually, as far as the computer is concerned, it doesn't matter whether the operating system is on disk or not. All it's looking for are the first two sectors, which it copies from disk into memory beginning at location $800. It then executes the instructions beginning at location $801, whether they mean something or not.

As with ProDOS 8, the first thing the Boot ROM does is load disk blocks 0 and 1 into memory location $800 in bank $00. The next step is also similar. In starting a ProDOS 16 disk, the program at $800 (the boot code) looks for a file named PRODOS in the volume directory—the same name as the ProDOS 8 Relocator. If the PRODOS file is not found, the ******UNABLE TO LOAD PRODOS****** message appears.

These similarities are not remarkable coincidences. This is because a disk formatted for ProDOS 16 will contain exactly the same boot code on blocks 0 and 1 as does a ProDOS 8 disk.

Once the jump is made to memory location $2000 (the PRODOS program), the two operating systems behave quite differently. The PRODOS program under ProDOS 8 is the Relocator and Kernel—the actual operating system. Under ProDOS 16, the PRODOS file loaded at memory location $2000 is just another link in a long chain of commands.

> If you try to boot ProDOS 16 on an Apple II other than a IIGS, the following is displayed:
>
> ```
> PRODOS 16 REQUIRES APPLE IIGS HARDWARE
> ```

The primary duty of the PRODOS file is to pass execution to the Apple IIGS System Loader. But before it does that, it sets up the ProDOS 16 quit code by transferring that part of itself to memory location $D000 in bank-switched memory. This code, referred to as PQUIT, stays in memory permanently and is used when a program quits. (The actual memory location is one of those pieces of information that you don't really need to know. There is no practical purpose for knowing that the code is loaded into the $D000 location, except to impress your friends.)

The Apple IIGS System Loader file is named P16. It's found in the SYSTEM subdirectory on the boot disk. The System Loader works closely with ProDOS as well as the Memory Manager to allocate, relocate, load, and save information between the disk drives and memory. As the System Loader (P16) is started, it displays a name and version number on the screen, just as ProDOS 8 does. See Figure 3-1.

Figure 3-1. System Loader Display

```
                    APPLE II
PRODOS 16 V1.3                          29-JUN-87

                  LOADER V1.3

        COPYRIGHT APPLE COMPUTER, INC., 1983-87
              ALL RIGHTS RESERVED.
```

The ProDOS version number appears after booting a system disk. V1.3 is the version and release number of ProDOS 16 (P16), as well as the System Loader (PRODOS) file. In the above example, both numbers are the same, though that may not always be the case.

Once ProDOS 16 is in memory, the PRODOS Loader (still in memory at $2000) continues its job. It looks in the SYSTEM/SYSTEM.SETUP subdirectory. All files in this directory are executed, starting with the file named TOOL.SETUP.

TOOL.SETUP patches or modifies any of the ROM tool sets (ID numbers $01–$0D). This file must be in the SYSTEM/SYSTEM.SETUP directory, and it is executed ahead of any other files in the subdirectory.

The SYSTEM.SETUP directory contains any file or program that needs to be loaded or initialized when the system is started. Primarily, two types of files can be included in SYSTEM.SETUP,

along with TOOL.SETUP: Permanent Initialization files and Temporary Initialization files.

Permanent Initialization files. Permanent Initialization files have a file type of $B6. They're referred to as STR (STaRtup) files. These files are loaded and executed but not shut down like standard applications. They're actually more like subroutines because they're always in memory and end with an RTL instruction rather than calling the ProDOS Quit command. Permanent Initialization files must also be loaded into nonspecial memory and cannot allocate any stack or direct-page space.

An example of a Permanent Initialization file is the TOOL.SETUP program that patches the ROM-based tool sets. TOOL.SETUP contains adjustments and modifications to the ROM tool sets. It's actually an extension of the ROM code. When Apple learns of new bugs in the ROM tool sets, they release a new TOOL.SETUP file rather than new ROM chips. TOOL.SETUP must always be in memory, therefore it's a Permanent Initialization file and not a Temporary Initialization file.

Temporary Initialization files. Temporary Initialization files have a file type of $B7. They're referred to as TSF (Temporary Startup File) files. These files are similar to Permanent Initialization files, except they are shut down when completed, and their memory space is released. But, like Permanent Initialization files, they also end with an RTL instruction rather than calling the Quit function.

An example of a Temporary Initialization file is the BEEP.SETUP program listed later in this book. BEEP.SETUP replaces the normal system beep sound with a more pleasant noise. Once BEEP.SETUP completes its task, it's removed from memory (see Chapter 12 for more information on BEEP.SETUP).

After the SYSTEM.SETUP directory is scoured, and the STR and TSF programs are run, ProDOS looks in the directory SYSTEM/DESK.ACCS to load any desk accessories found there. Classic desk accessories (CDAs), with a file type of $B8, are placed into memory and can be accessed via the Control Panel. New desk accessories, with a file type of $B9, can only be used by DeskTop applications. All desk accessories in the SYSTEM/DESK.ACCS directory are loaded at this time.

You don't need to keep all your desk accessories in the SYSTEM/DESK.ACCS subdirectory—only those you want to load. Other desk accessories can be kept in a backup directory and then transferred to SYSTEM/DESK.ACCS for use when the system is rebooted.

After the desk accessories are loaded, ProDOS looks for a file named START in the SYSTEM directory. The file could be an applications file, or it could be the Finder or Launcher (discussed later). If a START file isn't found, ProDOS looks in the volume directory for a file with a suffix of either .SYS16 or .SYSTEM. The .SYS16 suffix indicates a ProDOS 16 file, and that program is loaded and executed. The .SYSTEM suffix is for a ProDOS 8 program.

If the .SYS16 program is found first, ProDOS calls its own quit code with the name of the .SYS16 file and executes it. If a .SYSTEM (ProDOS 8) file is found first, ProDOS calls a modified ProDOS 8 Quit call and executes the .SYSTEM file. However, in order to do this, the ProDOS 8 operating system file, P8 must be the SYSTEM directory.

If a SYSTEM/START file—or a .SYSTEM or .SYS16 file in the volume directory—does not exist, a fatal error occurs.

All this is done simply to boot the ProDOS 16 disk, so it's easy to see how ProDOS 16 can be accused of booting slowly when compared with ProDOS 8. However, given the power of this operating system and all the things it enables a programmer to do, it is well worth the extra wait.

ProDOS 16 Disk Contents

There are so many files on the ProDOS 16 boot disk that, even in the minimum configuration, all of them wouldn't fit on one of the old-style 140K disks. Most of these files and their duties were discussed in the previous section, but for review (and as a handy reference), they are touched on briefly here. The following programs (in alphabetic order) are on a sample system disk named /A/. Remember that throughout this section the volume name /A/ is used only for reference. Your system disk may have a different name.

/A/BASIC.SYSTEM The ProDOS 8 version of the BASIC interpreter. It contains the disk extensions to Applesoft BASIC in ROM.

/A/PRODOS The System Loader that is responsible for setting up the operating system, Toolbox, desk accessories, and generally getting the Apple IIGS running. Remember that both ProDOS 8 and 16 use the name PRODOS for their System Loader. One way to tell the difference is by looking at the file's size. ProDOS 8 is approximately 32 blocks in size, whereas ProDOS 16 is significantly larger at approximately 42 blocks. The sizes may vary depending on the release version, but ProDOS 16 will always be larger.

/A/SYSTEM/ The directory containing important files and folders (other directories).

/A/SYSTEM/DESK.ACCS Contains new and classic desk accessories to be loaded when ProDOS 16 boots. Other desk accessories can be included on your boot disk, but they will be loaded only if they are in this directory.

/A/SYSTEM/LIBS A directory holding system libraries. It appeared on the original System Disk, but not on the current (3.1) version. Apple may include it on future versions if an application needs library files.

/A/SYSTEM/P8 The ProDOS 8 operating system. If this file is renamed PRODOS and copied to the volume directory, the disk will boot as a ProDOS 8 disk.

/A/SYSTEM/P16 The ProDOS 16 operating system and Apple IIGS System Loader.

/A/SYSTEM/START A program to be run after ProDOS has finished loading (the startup program). It may be an actual application or a loader file to launch an application.

/A/SYSTEM/SYSTEM.SETUP A directory containing initialization files to be run at boot time.

/A/SYSTEM/SYSTEM.SETUP/TOOL.SETUP A required file used to patch tool sets in ROM.

/A/SYSTEM/TOOLS A directory containing all the disk-based tools for the Toolbox. The tool sets appear with the name TOOL followed by the three-digit decimal number of the tool set. So the Window Manager, tool set ID# $0E, appears in this directory as TOOL014.

The above are all the files of a typical ProDOS 16 system disk. Of course, more files exist depending on the application and version of the system disk. Besides DESK.ACCS and SYSTEM.SETUP in the SYSTEM directory, the following folders might also be found:

/A/SYSTEM/DRIVERS A directory containing control files for printers, *AppleTalk*, modems, and a variety of devices.

/A/SYSTEM/FONTS A directory containing a variety of fonts to be taken advantage of by programs that use them. The files are named after the font they describe, followed by a dot and the point size of the font. So, COURIER.10 is a ten-point Courier font, and TIMES.12 is a 12-point Times Roman font.

Two other files you might find on your system disk are these:

/A/SYSTEM/FINDER A program, run by the /A/SYSTEM/START program, that contains a DeskTop environment similar to the one found on the Macintosh.

/A/SYSTEM/LAUNCHER A simple program launcher, run by the /A/SYSTEM/START program. This program was around when the original Apple IIGS arrived and the Finder was not yet completed.

The Finder uses a number of other files on disk, most notably icon files containing the graphic images it uses as icons. The two icon files used by the Finder are DIALOG.ICONS in the volume directory and FINDER.ICONS in the directory /A/ICONS. (It's permissible to move the DIALOG.ICONS file into the ICONS subdirectory to keep your volume directory clean, by the way.)

If you're writing applications for distribution, you'll have to find a way to get the following files and programs on your ProDOS 16 disk:

/A/PRODOS
/A/SYSTEM
/A/SYSTEM/P16
/A/SYSTEM/SYSTEM.SETUP
/A/SYSTEM/SYSTEM.SETUP/TOOL.SETUP

These files are required by ProDOS 16 in order to boot successfully. However, the startup application will probably require tool sets and other support files.

For example, DeskTop programs may need /A/SYSTEM/FONTS/ (and the fonts) or /A/SYSTEM/START or the .SYS16 file in the volume directory. BASIC programs will need BASIC.SYSTEM and P8. And, if your program uses a disk-based tool set, you'll need to include it in the /A/SYSTEM/TOOLS directory.

If you plan to write and distribute your applications on a ProDOS 16 disk, you should know that your system disk and its contents contain software copyrighted by Apple. Only very wealthy companies can afford to pay the fees required to distribute ProDOS with their programs. For you, as a software wizard, it's best to put your applications on a data disk and then provide instructions for copying your software to a ProDOS disk or to have the user copy ProDOS and the Finder to your disk.

Contact Apple Computer for more information on licensing.

Launching Applications

Launching a program on the Apple IIGS is different from running programs on older Apples. The Apple IIGS offers a very diverse environment and, as usual, there are always a few more things going on than meets the eye. Of course, programmers will love to take advantage of the new features of ProDOS 16.

Because this book is about the Apple IIGS, ProDOS 8 is beyond its scope. There are many worthy texts already available on the subject to which the reader is referred. The concentration here will be on launching (or running) applications under ProDOS 16.

The first and most bizarre feature of ProDOS 16 is that programs start with a call to the ProDOS Quit function. A program starts by quitting.

To launch a ProDOS 16 application, the program can be one of three types:

- The program named START in the SYSTEM directory
- Any program with an S16 ($B3) file type
- Any program with a SYS ($FF) file type

The SYS file type is a ProDOS 8 application. Even so, the ProDOS 16 Loader will recognize this and, as part of the application's startup, ProDOS 8 will be loaded and executed, allowing you to run your ProDOS 8 program.

Programs can also be launched via the Finder or the Launcher. Whichever method is used, the program is loaded into memory and control is transferred to that program.

But there's considerably more to the story than that. If you have purchased this book and have read this far, you're probably interested in knowing the real information on program launching.

Launching. Your programs are actually loaded via the ProDOS 16 Quit call. When one application quits and performs the obligatory call to ProDOS notifying the operating system that it is finished, the program has the option of immediately running another program.

If a second program is not specified, the ProDOS 16 Quit call allows any previously launched programs to be rerun, either by reloading them from disk or restarting them from memory.

When the ProDOS 16 Quit function is called, the program making the call is basically finished. It can, however, tell ProDOS the following:

- Which program to run next
- Whether it can be used again after the next program quits

If the program doesn't specify the next program, ProDOS checks to see whether it can return to any other programs previously run, and if not, it executes the special quit code, PQUIT.

The ProDOS 16 Quit Function

Programming for ProDOS 16 is different than programming for the Toolbox, yet very similar to ProDOS 8. More information on using ProDOS is presented in Chapter 14. The ProDOS 16 Quit function is number $29. It has two parameters:

- The pathname of an optional program to run
- The quit-parameter word

To call ProDOS on the Apple IIGS, a long jump is made to the ProDOS vector in memory bank $E1, offset $A8:

```
jsl   $E100A8     ;Call the ProDOS vector
```

The JSL instruction is followed by two values. The first value is the function number, and the second is the long address of the parameter list:

Value	Size
Function number	Word
Parameter address	Long word

The function number is a word-sized value, and the parameter address is the memory location of a list of parameters required by the call. A sample Quit call in machine language would be

```
jsl     $E100A8         ;ProDOS vector
dc      i2'$29'         ;Function number $29, Quit
dc      i4'Params'      ;Address of Parameters
```

Or, if using macros (discussed in Chapter 4):

```
_QUIT   Params          ;see above
```

The information at the address indicated by the label Params contains the address of a pathname of a program to run, plus the quit parameter word. For example:

```
Params  Anop            ;Memory Address of parameters
        dc      i4'0'   ;A long word of zero, no pathname
        dc      i2'0'   ;quit parameter word of zero
```

This example would be used if a program were just quitting and not running another program. Using a long word of 0 for the pathname tells ProDOS to quit without running another program.

If the program were quitting and running another program, the following parameters might be used:

```
Params  Anop
        dc      i4'Prog'    ;The address of Prog's pathname
        dc      i2'0'       ;quit parameter word of zero
```

The label Prog, in this case, is the address of the pathname of a program to run next:

```
Prog    dc      i1'14'              ;must start with a count byte
        dc      '/GAMES/MONSTER'    ;pathname to run
```

In ProDOS, pathnames are always preceded by a count byte denoting the length of the path, which follows immediately. If a program were to quit with the above Params, the program MONSTER on the GAMES volume would be run.

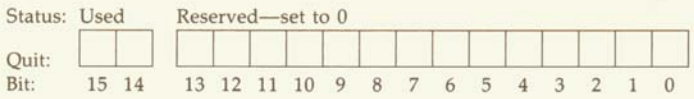
This is how one program can run another and how the Finder, Launcher, *APW* shell, *TML Pascal* environment, and a plethora of other shells and operating systems will load and execute programs. They'll all do it via the ProDOS 16 Quit call.

The Quit-Parameter Word

If you don't want to run another program, or if you want to run another program and then have control come back to the original program, that is where you need the quit-parameter word.

The quit-parameter word is part of the ProDOS 16 Quit function's parameter list (see above). Out of the 16 bits of this word, only two are used. The rest are labeled *forbidden* by Apple:

Figure 3-2. The Quit-Parameter Word



Bit 15. Bit 15 of the quit parameter controls the quitting program's User ID (discussed later in this book) and whether or not the program will restart after a second program quits. (Each program has its own, unique ID number.) Bit 14 determines if the program quitting can be restarted from memory or should be reloaded from disk.

To stop one program, start another, and then return to the original program requires some fancy footwork. To assist in this ballet, ProDOS maintains something called a *Quit Return Stack*. As each program quits, it has the option of placing its User ID (uniquely identifying that program) onto the Quit Return Stack.

Likewise, when a program quits, ProDOS checks the Quit Return Stack for a User ID. If found, the program identified by the User ID is run again. It's like magic.

If Bit 15 of the quit parameter is set to 1, the quitting program's ID number is pushed to the ProDOS 16 Quit Return Stack. This means that, once a second program is done, control will return to the original program.

This is how programs like the Finder and Launcher work. When you select a program to run, the Finder sets bit 15 of the quit-parameter word and calls the ProDOS Quit function to run that program. Because this bit is set, the Finder or Launcher's User ID is saved on the ProDOS 16 Quit Return Stack. When the program you've selected is finished, ProDOS checks the Quit Return Stack, removes previous program's ID number, and returns to that program.

If Bit 15 were not set when the first program quits, then whatever program belongs to the User ID pulled from the Quit Return Stack is run. If the Quit Return Stack is empty, control returns to the PQUIT code established by PRODOS when the machine was booted.

Bit 14. Bit 14 of the quit-parameter word determines whether or not the program making the Quit call can be restarted from memory or should be reloaded from disk. If bit 14 is set to 1, the program can be restarted from where it sits in memory. If it is reset to 0, the program must be reloaded into memory by the System Loader. (This is all done by ProDOS. All you do is set or reset the bit.)

So, launching a program on the Apple IIGS starts with a Quit call. Quitting programs can specify the name of another program to run, as well as determine whether control returns to the original program after the second is run.

Programs may crash when run through a debugger because of the way the ProDOS 16 Quit function works in conjunction with the Quit Return Stack: When your program makes a ProDOS Quit call, the operating system becomes confused because the debug program is still running. This causes the system to crash. When using the trace mode in DEBUG, place a breakpoint before your code to make the ProDOS 16 Quit function call.

Computer States at Runtime

When ProDOS passes control to a program via the Quit call, the System Loader determines whether the new program is relocatable, or must reside at a specific location in memory. When this determination is done, the program is allocated its own space, given its own zero page, and enough memory to operate. A number of other things can happen, depending on the program and how it was loaded.

Only file types $B3–$BE can be loaded by the System Loader, and only file types $B3 and $B5 can be run as programs (and specified by a Quit call). If a file of an unusual type is specified, the System Loader reports error $5C, *Not an executable file.*

Table 3-1. ProDOS 16 Load File Types

Type	Hex	Dec	Description
S16	B3	179	ProDOS 16 system application file
RTL	B4	180	*APW* runtime library file
EXE	B5	181	ProDOS 16 shell application file
STR	B6	182	ProDOS 16 Permanent Initialization File
TSF	B7	183	ProDOS 16 Temporary Initialization File
NDA	B8	184	New desk accessory
CDA	B9	185	Classic desk accessory
TOL	BA	186	ProDOS 16 tool set file
DRV	BB	187	ProDOS 16 driver file
...	BC	188	System use
...	BD	189	System use
...	BE	190	System use

Unlike older ProDOS 8 applications, there is no way to be certain exactly where a program running under ProDOS 16 will be put in memory. (ProDOS 8 programs were always loaded at memory location $2000 in bank $00.) However, there are a few guarantees made by Apple regarding the state of the system when your program takes control.

As with the Boot ROM, once the Loader places your program into memory, control of the machine passes to the first instruction of your program. Because the Apple IIGS is a single-tasking computer, meaning it's capable of doing only one thing at a time, your program has complete control when it starts. The computer states listed in Table 3-2 will be set at the time your application is launched.

Table 3-2. The 65816 Registers Set at Launch

Register	Type	Value
A	Accumulator	The application's User ID
X	Index	$0000
Y	Index	$0000
S	Stack pointer	The top of stack space
D	Direct page	The bottom of stack space
P	Processor status	All zero, native 65816 mode
PBR	Program bank	Determined by the Loader
DBR	Data bank	Determined by the Loader
PC	Program counter	Determined by the Loader

The addresses pointed to by the S and D registers are in bank $00. (The stack and direct page must always be in bank $00.) For example, the S register might point to $1BFF, and the D register might point to $1800, defining the stack and direct-page space to that $400 byte block. Note, however, that tool sets must request their own direct-page space from the Memory Manager (see the next chapter).

The values of the program and data bank registers, as well as the program counter will be determined by the Loader and what your application requires. There is no guarantee that the program-bank and data-bank registers will be pointing to the same bank of memory.

Other aspects of the system are set as follows:

- The standard input and output devices used by the Text tool set are both set to the Pascal 80-column video screen. These can be changed by using the Text tool set commands to specify new input or output devices. However, at startup, both are set to the Pascal 80-column device, also loosely referred to as *the screen*.
- *Memory shadowing* is set on for the language card, I/O spaces, and text pages, and is set off for the graphics pages. Unless you are truly an expert, it is not recommended that you alter memory shadowing.

Chapter 4

About the Toolbox

The Toolbox is crucial to programming the Apple IIGS. All the routines necessary for programming the Apple IIGS are kept in the Toolbox. But the Toolbox is more than a simple set of programming routines: It's the secret to writing programs and developing DeskTop

applications for the Apple IIGS. Know the Toolbox, and you can master the machine.

This chapter introduces the Apple IIGS Toolbox. The Toolbox contains about 1000 unique routines (called *functions*) that take much of the effort out of programming the Apple IIGS. Though the name *Toolbox* is accurate when it describes these routines and functions as tools, it might be more fitting to refer to the Toolbox as a treasure chest of programming features.

This chapter won't detail the operation of the Toolbox, but it does show how to use the Toolbox to your best advantage. For detailed information about the Toolbox, including a complete list of the Toolbox function numbers and parameters, refer to a comprehensive reference, such as that found in *COMPUTE!'s Mastering the Apple IIGS Toolbox*.

Toolbox Briefing

The Toolbox contains routines found in the computer's ROM as well as some routines that must be loaded from disk into RAM (called *disk-based* tools). The nearly 1000 unique functions in the Toolbox are grouped into 28 different categories called *tool sets*. For example, all of the functions related to the manipulation of windows are found in the Window Manager tool set, the pull-down menu functions are in the Menu Manager tool set, and so on. (See Table 4-1 for a complete listing.)

A tool set can contain as many as 255 different functions. At present, the QuickDraw II tool set, the largest by far, contains 206 unique routines.

Each tool set function is given a unique identification number. The number shows which tool set the function belongs to and gives the individual function number within that tool set. Together, these two numbers create a two byte (16-bit, or word-sized) number identifying the function. One byte gives the tool set; the other, the function number:

function number (1 byte) tool set number (1 byte)

The byte representing the function number comes first, followed by the tool set. It's backwards, but it's consistent. All of the functions in the Toolbox are identified this way. For example, the Miscellaneous tool set is tool set number $03. A function within

that tool set, SysBeep, is function number $2C. SysBeep is referred to as function $2C03 in the Toolbox:

function number ($2C), tool set number ($03)
SysBeep = $2C03

The tool set ID is the low-byte value of $03, and the function ID is the high-byte value of $2C. Any other function in the Miscellaneous tool set will also end with the low-byte value of $03, but it will have a different high-byte value.

Table 4-1 contains a complete list of tool sets, their names, and ID numbers. Note which ones are found in ROM and which ones are located on disk.

Table 4-1. Tool Set Chart

ID	Name	Where	Comments
$01	Tool Locator	ROM	
$02	Memory Manager	ROM	
$03	Miscellaneous tool set	ROM	
$04	QuickDraw II	ROM	$300 bytes direct-page space
$05	Desk Manager	ROM	
$06	Event Manager	ROM	$100 bytes direct-page space
$07	Scheduler	ROM	
$08	Sound Manager	ROM	$100 bytes direct-page space
$09	Apple DeskTop Bus	ROM	
$0A	SANE	ROM	$100 bytes direct-page space
$0B	Integer Math	ROM	
$0C	Text tool set	ROM	
$0D	RAM Disk	ROM	Internal use only
$0E	Window Manager	Disk	Uses Event Manager's direct page
$0F	Menu Manager	Disk	$100 bytes direct-page space
$10	Control Manager	Disk	$100 bytes direct-page space
$11	System Loader	Disk	
$12	QuickDraw II Auxiliary	Disk	Uses QuickDraw's direct pages
$13	Print Manager	Disk	$200 bytes direct-page space
$14	Line Edit	Disk	$100 bytes direct-page space
$15	Dialog Manager	Disk	Uses Control Manager's direct page
$16	Scrap Manager	Disk	
$17	Standard File	Disk	$100 bytes direct-page space
$18	Disk Utilities	Disk	(No information)
$19	Note Synthesizer	Disk	(No information)
$1A	Note Sequencer	Disk	(No information)
$1B	Font Manager	Disk	$100 bytes direct-page space
$1C	List Manager	Disk	

The tool set ID is the identification number used to reference the tool set during calls to functions. For the sake of convenience, and to be consistent with Apple's documentation, hexadecimal (base-16) notation is used. This also makes it easier to spot the tool set number when looking at only a two-byte Toolbox function value.

The names listed in the second column of Table 4-1 are the official tool set names. The purposes of most tool sets may be easily discerned from their names. The Miscellaneous tool set, number $03, contains a hodgepodge of important functions that don't fit comfortably under the rubric of any of the other tool sets.

The third column in Table 4-1 indicates whether a tool set is located in ROM (built into the IIGS) or whether it is loaded into RAM from disk.

Additional information is listed under Comments, such as how many direct pages are required by the tool set. The tool sets often need a certain amount of direct-page memory. Its use is similar to BASIC's use of zero page: as a scratch pad for temporary storage of data and pointers. The amount needed depends on the tool set, and its use is discussed in greater detail later in this chapter.

Opening the Toolbox

Before the Toolbox can be accessed, the microprocessor must be placed into native mode. That is, the computer must be running with Apple IIe (Mega II) emulation turned off. Additionally, all registers in the 65816 microprocessor must be set to 16-bit widths. The following code does this in machine language:

```
clc             ;clear the carry bit
xce             ;and the emulation bit
rep    #$30     ;use 16-bit memory and registers
```

Depending on where and how an application has been launched, the code above may not be necessary. If the *APW* or *ORCA/M* assembler is used, there's no need to establish the size of the registers and turn off emulation. However, with other assemblers and especially for BASIC programs using the Toolbox with machine language subroutines, you must perform the above operation. The Toolbox cannot be accessed when the 65816 microprocessor is in emulation mode.

With high-level-language compilers you don't need to worry about turning off emulation. All ProDOS 16 program launchers automatically set the 65816 into native (non-emulation) mode before your application starts.

Calling the Toolbox

To call the Toolbox using machine language, place the function ID (tool set number and function number) in the X register. Push onto the stack any parameters passed to the function. Finally, make a long jump to the subroutine (JSL) at address $E10000, the Toolbox dispatcher. Any parameters returned from the function should be pulled from the stack after returning from the function.

The first Toolbox commandment: Thou shalt not access a Toolbox function unless its tool set has been started up. Every function in the Toolbox is part of a specific tool set. And before that function can be used, its tool set must be started.

Each tool set has a special function to do this, called the StartUp function. This function is always function number $02. So, before you can use any function in the Miscellaneous tool set, you must call the MTStartUp function, ID number $0203 ($02 for the StartUp function and $03 for the Miscellaneous tool set). Once StartUp is called, other routines in the tool set can be accessed.

The specifics of calling the Toolbox, along with step-by-step analysis, is provided in *COMPUTE!'s Mastering the Apple IIGS Toolbox*. Refer to that text if these concepts are new to you.

To perform the MTStartUp function in machine language, the X register is loaded with the 16-bit function ID number, $0203 and then a JSL instruction is made to memory location $E10000. (JSL is Jump to Subroutine Long, and memory address $E10000 is the memory location of the Toolbox.) This is the door through which you get to the Toolbox.

So in order to start this tool set, a machine language program would use the following code:

```
ldx     #$0203      ;MTStartUp
jsl     $E10000     ;start the Miscellaneous tool set
```

The short form of this call is

```
_MTStartUp          ;Start the Miscellaneous tool set
```


This is an *APW* assembler macro call defined in the M16.MISCTOOL macro file (macros and macro files are discussed in the next chapter). Throughout the remainder of this book, both the long and short (macro) forms of making Toolbox calls in the assembler will be used.

In C, calling the StartUp function is as easy as typing the function name. For example, to start up the Miscellaneous tool set, the following is used:

```
MTStartUp( );
```

And it's done. The information needed by the compiler to perform the Toolbox call is contained in an include file. Just use the Toolbox function name in your source code, and the function is called automatically. Remember to place the following at the top of your C source code listing:

```
#include <misctool.h>
```

With Pascal, making a Toolbox call is just as easy. Using *TML Pascal*, the Miscellaneous tool set is started as follows:

```
MTStartUp;
```

As with C this is simply a statement in your Pascal source code. The information is built into the *TML Pascal* unit file called MISCTOOLS.USYM. In the USES portion of your Pascal program, you would include this file in the following manner:

```
USES    MiscTools;
```

Once the tool set has been started up, an application can use its features. For example, the SysBeep function, which beeps the speaker, is function number $2C03 of the Miscellaneous tool set. To call the system beep procedure in machine language, use the following:

```
ldx     #$2C03      ;the SysBeep function ID
jsl     $E10000     ;call the Toolbox
```

or, if using macros:

```
_SysBeep            ;call SysBeep
```

Remember, the StartUp function has already been called. For C, the source would be

```
SysBeep( );
```

and in Pascal, simply

```
SysBeep;
```

As mentioned previously, in Pascal each Toolbox function call contains its definition in a support file. These files can be included, used, or copied into your source file, depending on which language your program speaks. For example, with the *APW* assembler, the MCOPY command is used to copy macro definitions from external macro libraries into your program. In the C language, the #include directive causes the compiler to include a *header file* defining the Toolbox calls, as if it were an extension of your source code. Similarly, *TML Pascal* incorporates unit symbol files which are brought into the compilation step with the USES statement.

These techniques of including, using, or copying are all covered in the next chapter.

Tool Set Interdependencies

Many tool sets in the Toolbox call upon other tool sets to perform a special operation. This collaboration requires that interdependent tool sets—those that rely upon others—must be active and available.

While your program may only deal directly with the Menu Manager, the process of drawing menus relies upon the graphics wizardry of QuickDraw II. So your application must start up both the Menu Manager and QuickDraw II. To further complicate matters, the order in which the tool sets are started is equally important.

Fortunately, the following table presents a list of the interdependent tool sets, the tool sets they need, and the order in which they should be started:

ID	Tool Set	Tool Sets Required (by Tool Set ID)
$01	Tool Locator	None
$02	Memory Manager	$01
$09	DeskTop Bus	$01
$0B	Integer Math	$01
$0C	Text tool set	$01
$0A	SANE	$01, $02
$16	Scrap Manager	$01, $02

ID	Tool Set	Tool Sets Required (by Tool Set ID)
$03	Miscellaneous tool set	$01, $02, $0B
$04	QuickDraw II	$01–$03
$07	Scheduler	$01–$03
$08	Sound Manager	$01–$03
$19	Note Synthesizer	$01, $02, $08
$11	System Loader	$01–$03
$12	QuickDraw Auxiliary	$01–$04
$06	Event Manager	$01–$05, $09
$0E	Window Manager	$01–$06, $10, $0F
$14	Line Edit	$01–$04, $06, $16
$10	Control Manager	$01–$04, $06, $0E, $0F
$0F	Menu Manager	$01–$04, $06, $0E, $10
$1C	List Manager	$01–$04, $06, $0E, $10, $0F
$15	Dialog Manager	$01–$04, $06, $0E, $10, $0F, $14
$05	Desk Manager	$01–$04, $06, $0E, $10, $0F, $14, $15, $16
$17	Standard File	$01–$04, $06, $0E, $10, $0F, $14, $15
$1B	Font Manager	$01–$04, $0B, $0E, $10, $0F, $1C, $14, $15
$13	Print Manager	$01–$04, $12, $06, $0E, $10, $0F, $14, $15, $1C, $1B

For example, if your program uses any functions in the Line Edit tool set, it must start up in the following order: tool sets $01–$04 (Tool Locator, Miscellaneous tool set, Memory Manager, QuickDraw II), tool set $06 (Event Manager), and tool set $16 (Scrap Manager).

The First Six Functions

Consistency has never been highly regarded in the computer programming world. But the Apple IIGS programmer will be delighted to know that the first six function calls in each tool set follow a standard format. These functions are housekeeping, or tool set management routines, and every tool set has them.

As shown in the previous section, the StartUp function must be called before other functions in a tool set can be used. StartUp is just one of the first six functions.

Apple Computer has reserved tool set functions $07 and $08 for future enhancements. Until they are placed on the duty roster, the next usable function in each tool set is $09.

The first six function calls in each of the first six tool sets are as follows:

ID	Function	Description
$01	BootInit	Initializes the tool set for the first time
$02	StartUp	Starts up the tool set for application usage
$03	ShutDown	Shuts down the tool set when no longer needed
$04	Version	Returns the version number of the tool set
$05	Reset	Initializes the tool set after a system reset
$06	Status	Determines whether the tool set is active or not

These function names are unique to each tool set since they are always prefixed by a short name. For example, the Memory Manager uses the letters *MM* before each of these function names: MMStartUp, MMVersion, and so on. The Tool Locator tool set uses TL: TLBootInit, TLStartUp, and so on.

- BootInit must never be called by an application. If the tool set is ROM-based, this function is performed when the computer starts up. If the tool set is RAM-based (loaded from disk), BootInit is called after it is first loaded into memory.
- StartUp: As stated in the previous section, applications must call StartUp so that the tool set's functions become available. Some tool sets require input parameters for use with the StartUp function. Passing parameters to a Toolbox function is discussed in the next section.
- ShutDown must be called before exiting to the operating system when an application is finished with a tool set. The tool set would then free up any memory it had allocated and, in general, would clean up after itself.
- Version: An application can determine the version number of a tool set by calling this function. It returns a word (16-bit integer) result. The high-order byte of the result consists of the major version number. The low-order byte contains the minor version. If the tool set is a prototype, bit 15 of the version number result will be set. (In this text when a bit is said to be *set*, it is made equal to 1. A *reset* or *cleared* bit is one made equal to 0.)
- Reset occurs when you press Control-Reset or make a DeskTop Bus reset call from software. The computer performs the Reset function in each of the active tool sets.
- Status: A program can find out if a tool set has been started by making a call to it's Status function. If not active, it returns an integer value of 0, otherwise it returns a nonzero value.

Passing and Receiving Arguments from the Toolbox

The majority of the Toolbox functions require an argument (a value or parameter) to be sent to the Toolbox, or they return an argument, or a combination of both. The Toolbox works with three types of parameters: bytes, words (two bytes), and long words (two words).

If any arguments are required by a function, they are pushed onto the processor's stack before the Toolbox call is made. Arguments returned from a function are then pulled from the stack after the call. This is demonstrated in the following portion of code which obtains the version number of the Miscellaneous tool set:

```
pha             ;push space for the result
ldx   #$0403    ;the MTVersion function
jsl   $E10000   ;call the Toolbox
pla             ;retrieve version information
```

The values returned from the stack must have space reserved for them before the call is made. This is done by pushing arbitrary values onto the stack. These values are replaced with useful information, pulled from the stack, after the call is made.

The above function is handled as follows in C:

```
Version = MTVersion( );
```

Note that Version must be declared beforehand as a word value, an unsigned integer. After the call, the Version variable contains the version number of the Miscellaneous tool set.

In Pascal, the function call is similar:

```
Version = MTVersion;
```

Remember to declare the variable, *Version*, as an integer. In both Pascal and C, the code for stack manipulation is provided by the compiler.

When starting up the Memory Manager, a word-sized value is pushed onto the stack before the call to MMStartUp is made. This provides the result space for an ID number:

```
pha             ;push space for the result
ldx   #$0202    ;MMStartUp
jsl   $E10000
pla             ;pull the user ID
sta   UserID    ;save it in a safe place
```

When MMStartUp is called, not only does it allow access to other Memory Manager functions, it also assigns your application a unique identification number. You should store the value pulled from the stack as your program's User ID. You'll need it later on.

The Memory Manager is covered in detail in Chapter 7.

Direct Pages

Many tools need only a call to their StartUp function to get them going. Others require additional information, such as timing information, graphics modes, the User ID returned by the Memory Manager's StartUp function, or a combination of these.

A few tool sets require a small block of RAM to use as scratch space for their functions. This memory buffer is called a direct page, and it consists of one page (256 bytes) of RAM. The direct-page memory must exist in the first 64K bank of memory.

Space for the direct page is allocated using a function in the Memory Manager. This function is called NewHandle. Since a program may use many tool sets and require a large quantity of direct-page space, it's common to allocate one large block of memory for use by each of the tool sets requiring direct pages. Therefore, you should calculate the total amount of direct-page memory needed before using the NewHandle function. See Table 4-1 for the amount of direct-page space each tool set requires.

Once the direct page is established (by some sleight-of-hand programming you'll be reading about later), portions of it are divided among the tool sets which require them.

Tools on Disk

Some tool sets are stored in the SYSTEM/TOOLS subdirectory on the ProDOS 16 disk your computer is booted with. Tools on disk cannot be accessed until they have been loaded into memory. This is accomplished with the LoadTools function of the Tool Locator tool set.

LoadTools uses a list of tool set numbers in memory to load corresponding files from disk. It accesses the disk and copies the tools into memory.

When calling LoadTools, an application first pushes a four-byte address of the tool list to the stack. For example:

```
pea   Toolist|-16     ;push long word address of list
pea   Toolist
ldx   #$0E01          ;LoadTools
jsl   $E10000
```


Toolist (above) points to the memory location of the list of tool sets to be loaded from disk. The structure of the list of tool sets begins with a count word (two bytes) which tells LoadTools how many entries there are in the list.

The count word is followed by several four-byte entries that describe the tools to be loaded. The first two bytes constitute a word that contains the tool set's ID number. For example, $0003 would indicate the Miscellaneous tool set. The second two bytes are a word that specifies the minimum version of the tool. If a program requires version 1.3 or later of a tool set, $0103 is specified. By using a minimum version number of $0000, any version on disk will be loaded.

The following is a sample table showing three tool sets to be loaded from disk: the Window Manager (tool set $0E), the Menu Manager (tool set $0F), and the Control Manager (tool set $10).

```
Toolist  dc    i'3'               ;count word (3 tool sets)
         dc    i'$0E',i'0000'     ;Window Manager 0.0 or newer
         dc    i'$0F',i'0000'     ;Menu Manager 0.0 or newer
         dc    i'$10',i'0000'     ;Control Manager 0.0 or newer
```

After the LoadTools call is complete, the program can proceed by starting up each of the loaded tool sets as needed.

In C, the method of loading tools from disk starts by globally declaring an array of tool sets as a group of unsigned word-length integers:

```
Word  Toolist[ ] =  {3,        /* Tool count */
                    14, 0,     /* Window Manager */
                    15, 0,     /* Menu Manager */
                    16, 0};    /* Control Manager */
```

From within a function in your application, the LoadTools( ) function is called in this manner:

```
LoadTools(Toolist);
```

If you're using Pascal, the procedure is almost the same, except Toolist is defined in the VAR section of the program as a ToolTable type, a special record which follows the structure of the tool list:

```
Toolist: ToolTable;
```

Unfortunately, Pascal forces the values in the Toolist array to be assigned at run time within a procedure. This results in longer code. Example:

```
Toolist.NumTools := 3;              { Tool count }
Toolist.Tools[1].TSNum := 14;       { Window Manager }
Toolist.Tools[1].MinVersion := 0;
Toolist.Tools[2].TSNum := 15;       { Menu Manager }
Toolist.Tools[2].MinVersion := 0;
Toolist.Tools[3].TSNum := 16;       { Control Manager }
Toolist.Tools[3].MinVersion := 0;

LoadTools(Toolist);
```

However, the LoadTools function call is identical in syntax to the call in C.

When Errors Occur

Calling some Toolbox functions can result in errors. Errors can occur under a variety of circumstances. Not all of them are fatal.

The way to tell whether there was an error during your Toolbox call is to test the carry flag after the function returns. If the carry flag is set, an error occurred, and your program can take appropriate action. If the carry flag is clear, no error occurred, and the program can continue.

If an error does occur, the Toolbox places a special error code in the A register. This two-byte value describes the error that occurred and the tool set called. Unlike the Toolbox function numbers, the tool set number in an error code is in the upper byte. The error number is in the lower byte. For example, if the error returns $0110 in the A register, the upper byte ($01) indicates that the error occurred with tool set $01, the Tool Locator. The error code ($10) is in the lower byte. Error code $10 of the Tool Locator is *Minimum Version Not Found*. (All error codes are documented along with the Toolbox functions in *COMPUTE!'s Mastering the Apple IIGS Toolbox*.) This error might occur when the LoadTools function is called to load tool sets from disk into RAM. If the minimum version specified is not found on disk, this error is returned after the LoadTools function is called.

Note that only some of the functions in the Toolbox result in actual errors. Some are unable to produce errors, yet may return with the carry flag set. An application should only test for errors after making Toolbox calls capable of producing errors.

Trapping for Toolbox errors in a C program is done by testing an external variable called _toolErr (note the underscore). This variable is declared as type extern in the types.h header file, which should be the first file included by any C program that uses the Toolbox. If _toolErr is a nonzero value, it means that the most recent Toolbox function resulted in an error. The value in _toolErr is the error code.

Here is a sample error-handling statement in C:

```
if (_toolErr) SysFailMgr(_toolErr, nil);
```

Care should be taken when handling errors in C by referencing the _toolErr variable. Since this variable is changed after each function call, your program should make a copy of _toolErr before using any other Toolbox functions.

TML Pascal programmers handle errors in a similar fashion. To see if an error has occurred, the value of a predefined variable called IsToolError is tested. The error code is stored in another predefined variable called ToolErrorNum.

Here is a sample error-handling statement in *TML Pascal:*

```
IF IsToolError THEN
        SysFailMgr(ToolErrorNum, 'Fatal system error —> $');
```

All the examples for handling errors, shown here, take the easy way out. The Miscellaneous tool set includes a function called SysFailMgr which brings up the familiar sliding Apple error message screen. (You see it when you try to boot the Apple IIGS without a disk in the drive).

SysFailMgr is adequate for testing purposes, but it shouldn't be used when errors occur in end-user or commercial applications. There are elegant (and user-friendly) ways of handling errors. It just takes a little extra effort to incorporate them into your programs.

Closing the Toolbox

When an application is finished using a particular tool set, it should shut it down. This is done by calling the tool set's ShutDown function, number $03. For example, to shut down the Menu Manager, the MenuShutDown call is made:

```
ldx   #$030F      ;MenuShutDown
jsl   $E10000
```

Since an application uses many tool functions throughout the running of the program, tool sets are usually shut down all at once before the program quits.

As a rule, tool sets should be shut down in the reverse order that they were started up. If, for example, the Miscellaneous tool set was shut down before other tool sets, it would cause the application to crash.

The Memory Manager is one of the last two tool sets to be shut down just before a program ends. Before the MMShutDown call is made, all allocated memory handles associated with an application should be disposed (that is, those requested for direct-page space). The easiest way to do this is with the DisposeAll function:

```
lda   MemID      ;Identify the blocks
pha              ;. . . by their ID numbers
ldx   #$1102     ;DisposeAll (memory handles)
jsl   $E10000
```

This disposes of all memory handles allocated by the application (identified by the MemID value). DisposeAll should never be used with the UserID value that was returned by MMStartUp.

Handles, doled out by the Memory Manager's NewHandle function, can be disposed of one at a time. This example demonstrates how easily a handle can be removed from C or Pascal:

```
DisposeHandle(MyHandle);
```

When memory handles are disposed, the space they occupied is freed and is made available to other applications. More details on memory management are discussed in Chapter 7.

Once all the memory handles allocated by your program are disposed, the MMShutDown function can be called.

Chapter Summary

The following Toolbox functions were referenced in this chapter:

Function: $0E01
Name: LoadTools
Loads a list of tools from disk into RAM
Push: Tool List Address (L)
Pull: nothing
Errors: $0110 Version Error; possible ProDOS errors
Comments: The list of tools starts with a count word.

Function: $0302
Name: MMShutDown
Shuts down the Memory Manager
Push: User ID (W)
Pull: nothing
Errors: none
Comments: The User ID is obtained when MMStartUp is first called.

Function: $1002
Name: DisposeHandle
Disposes of a handle and the memory block it references
Push: The Handle (L)
Pull: nothing
Errors: $0206 (invalid handle)

Function: $1102
Name: DisposeAll
Disposes of all memory handles associated with an ID
Push: User ID (W)
Pull: nothing
Errors: $0207 (invalid User ID)
Comments: Do not use with the program's master User ID.

Function: $0203
Name: MTStartUp
Starts up the Miscellaneous tool set
Push: nothing
Pull: nothing
Errors: none
Comments: This call must be made before any Miscellaneous tools can be used.

Function: $0403
Name: MTVersion
Returns the version number of the Miscellaneous tool set
Push: Result Space (W)
Pull: Version (W)
Errors: none
Comments: MSB is major release; LSB is minor release.

Function: $1503
Name: SysFailMgr
Displays an error message and halts the program
Push: Error Code (W); C-String Address (L)
Pull: nothing
Errors: none
Comments: A standard message is displayed if the string address parameter is 0.

Function: $2C03
Name: SysBeep
Beeps the Apple IIGS speaker
Push: nothing
Pull: nothing
Errors: none

Function: $030F
Name: MenuShutDown
Shuts down the Menu Manager
Push: nothing
Pull: nothing
Errors: none

Chapter 5

A Matter of Language

As stated earlier, this book assumes that you have a strong background in programming languages, either machine language, Pascal, or C. This isn't a tutorial on programming.

Yet there's more to using a programming language and developing software than just

knowing the meaning of such terms as *ASL, printf,* or *begin.* There is a wealth of programming information to learn once you understand the basics. This information will make you a better programmer. The purpose of this chapter is to fill you in on some of the finer points of programming the IIGS, no matter which language you use.

This chapter offers programming hints and tips for the three languages covered in this book. On the following pages, you will find helpful information and suggestions for making programming and developing applications for the Apple IIGS computer much easier.

Take Life a Little Easier

Because this chapter tries to cover three very different programming environments, extra care was taken to ensure that everything was presented properly. To do that, this chapter is divided into two sections. The first section covers support files for all three languages, and the second deals with each language individually.

Support files, though they may be referenced by each language differently, are common to all three programming environments. Most amateurs avoid using support files because they don't understand them, which is a big mistake. By taking advantage of support files, you can save time and massive headaches. You should take the time to learn about support files.

The second part of this chapter concentrates on each programming language individually: The *APW* Assembler, *TML Pascal,* and C are each given a separate section. The purpose of the second half of this chapter is to help you use the language you have chosen to its full potential. After reading about Support Files, skip to the section on the language that interests you.

Of course, the adventurous reader will want to read everything: If you are only fluent in one or two languages, you may be surprised to find out what you are missing.

Support Files

To smooth the process of writing applications, the makers of *APW* and *TML Pascal* have created scores of utility and support files. These files typically contain defined routines, macros, or subroutine libraries. By taking advantage of support files, you can decrease

development time and, at the same time, make your program easier to read and better looking.

In machine language, support files contain common routines and functions written as macros. For example, to avoid the redundancy of making Toolbox calls by loading the X register and performing a long jump to the subroutine at $E10000 each time a call is made, a Toolbox macro support file can be used instead. This support file already contains the defined Toolbox calls. All your source needs to do is reference the specific support file.

TML Pascal support routines, which include Apple IIGS Toolbox calls and other Pascal-oriented functions, are stored in symbolic unit files. These files end with a .USYM extension on disk. The USES keyword tells the compiler to use the unit file that corresponds to functions used in your program.

The #include directive is used to insert a source file into the compilation step when compiling a C program. Support files for C, called header files, end with a .h extension on disk. Since files used with #include can contain any instructions at all, they are far more flexible than Pascal's compile-time unit files.

The following tables illustrate how your source code could take advantage of predefined QuickDraw II functions. The following are QuickDraw II support files, each of which can be referenced by your code.

Language	Directive	Support Filename
APW Assembler	MCOPY	M16.QUICKDRAW
TML Pascal	USES	QDIntf
APW C	#include	quickdraw.h

In your source code, the above directives might take on the following syntax:

Language	Syntax
APW Assembler	MCOPY 2/AINCLUDE/M16.QUICKDRAW
TML Pascal	USES QDIntf;
APW C	#include <quickdraw.h>

After these statements, your source code could then use the QuickDraw II functions defined in the appropriate support file. (This will be explained in greater detail below, under each language's category.)

When programming high-level languages such as C and Pascal, these support files must be included in the compilation phase

of your program to use them. Otherwise, you'll receive an *undefined function call* error message. It's best not to argue with the compiler if you want your code to run.

Macros are not required in order to make machine language Toolbox calls. The programmer can use the corresponding 65816 instructions if desired. However, using the macros defined in the *APW* Toolbox support files is accepted and a more common practice than writing out the necessary code.

The most common use for support files is to define Toolbox calls. Each tool set in the Toolbox has an associated support file. There are several other specialty and utility files, depending on your language, which can also be used to simplify writing applications.

Table 5-1 shows the support files that belong to each tool set for machine language, C, and Pascal.

Table 5-1. Tool Set Support Files

Tool Set Name	*APW* Assembler (MCOPY)	*APW C* (#include)	*TML Pascal* (USES)
Tool Locator	M16.LOCATOR	locator.h	GSIntf
Memory Manager	M16.MEMORY	memory.h	GSIntf
Miscellaneous Tools	M16.MISCTOOL	misctool.h	MiscTools
QuickDraw II	M16.QUICDRAW	quickdraw.h	QDIntf
Desk Manager	M16.DESK	desk.h	GSIntf
Event Manager	M16.EVENT	event.h	GSIntf
Scheduler	M16.SCHEDULER	scheduler.h	Scheduler
Sound Manager	M16.SOUND	sound.h	Sound
DeskTop Bus	M16.ADB		
SANE	M16.SANE	sane.h	SANE
Integer Math	M16.INTMATH	intmath.h	IntMath
Text Tool Set	M16.TEXTTOOL	texttool.h	TextTools
Window Manager	M16.WINDOW	window.h	GSIntf
Menu Manager	M16.MENU	menu.h	GSIntf
Control Manager	M16.CONTROL	control.h	GSIntf
System Loader	M16.LOADER	loader.h	Loader
QuickDraw II Aux.	M16.QDAUX	qdaux.h	QDIntf
Print Manager	M16.PRINT	print.h	PrintMgr
LineEdit	M16.LINEEDIT	lineedit.h	GSIntf
Dialog Manager	M16.DIALOG	dialog.h	GSIntf
Scrap Manager	M16.SCRAP	scrap.h	GSIntf
Standard File	M16.STDFILE	stdfile.h	GSIntf
Disk Utilities			

Tool Set Name	*APW* Assembler (MCOPY)	*APW C* (#include)	*TML Pascal* (USES)
Note Synthesizer	M16.NOTESYN	notesyn.h	NoteSyn
Note Sequencer	M16.NOTESEQ		
Font Manager	M16.FONT	font.h	GSIntf
List Manager	M16.LIST	list.h	ListMgr

Depending on the language you're using, there might be additional support files for working with ProDOS or a shell environment. Check your language's reference manual for more details.

At the time of this writing, some of the tool sets do not have support files, most notably those still being worked on by Apple Computer.

Although *TML Pascal*'s unit symbol files end with a .USYM extension on disk, do not include the extensions in the USES statements in your program.

In addition to the above assembler macro files, the *APW* assembler can also take advantage of *equate* files. These, like macro files, are text files that contain some of the constants and symbols listed in the Toolbox reference. For example, wAmBooli is a flag used by one of the tool sets. If your source code were using wAmBooli, as in

```
PEA   #wAmBooli
```

and if the equate file for that tool set were referenced by your source code with the COPY directive, then the assembler would replace wAmBooli with the proper value.

Table 5-2 lists the support files for equates to be used with *APW* source code. Like the macro files, they are found in the LIBRARIES/AINCLUDE subdirectory.

Table 5-2. Assembler Equate Files

Tool Set Name	Equate File
Tool Locator	E16.LOCATOR
Memory Manager	E16.MEMORY
Miscellaneous Tools	E16.MISCTOOL
QuickDraw II	E16.QUICDRAW
Desk Manager	E16.DESK
Event Manager	E16.EVENT
Scheduler	E16.SCHEDULER
Sound Manager	E16.SOUND
DeskTop Bus	E16.ADB
SANE	E16.SANE

Tool Set Name	Equate File
Integer Math	E16.INTMATH
Text Tool Set	E16.TEXTTOOL
Window Manager	E16.WINDOW
Menu Manager	E16.MENU
Control Manager	E16.CONTROL
System Loader	E16.LOADER
QuickDraw II Aux.	E16.QDAUX
Print Manager	E16.PRINT
LineEdit	E16.LINEEDIT
Dialog Manager	E16.DIALOG
Scrap Manager	E16.SCRAP
Standard File	E16.STDFILE
Disk Utilities	
Note Synthesizer	E16.NOTESYN
Note Sequencer	
Font Manager	E16.FONT
List Manager	E16.LIST

Note: The Disk Utilities and Note Sequencer equate files were not included in version 1.0 of the *APW* assembler.

Individual Languages

The way each language takes advantage of its support files is discussed in the following sections.

The C Language Environment

C is an elegant language, but don't let its elegance fool you. It's a nuts-and-bolts programming language. C has the detail of machine language, while retaining some of the conveniences of the high-level languages. Anyone trained in BASIC and then forced into machine language because of BASIC's crudity and slowness will enjoy C.

The road from your first *Hello World* C program to a complete application on the Apple IIGS should be smooth. Even though the *APW C* development system isn't as flashy as other programming environments, it can be used to develop large and complex applications. In fact, most of the new IIGS programs that originated on other computers are written in C, simply because the original source code can be moved to the IIGS with only minor modifications, in most cases.

Support files for *APW C* are kept in the LIBRARIES/CINCLUDE area on your *APW* program development disk. They all end with .h extensions because they are known as header files. This means that they should be included in your source code, with the #include directive, at the top (or at the head) of your program.

The following is an example of how to use a support file in a C program:

```
/* Including Header Files in C—Kinda Boring */
#include <locator.h>   /* Include the Tool Locator header file */
main( )
{
     TLStartUp( );     /* Start the Tool Locator */
     TLShutDown( );    /* Shut it down ASAP */
}
```

All the definitions for the Tool Locator functions are kept in the locator.h header file. By including this header file, the TLStartUp, TLShutDown, and other Tool Locator routines can be accessed by the C program. The same is true for any other tool set that your program uses. Include the header file for each tool set you intend to use.

```
#include <locator.h>
#include <memory.h>
#include <misctool.h>
```

The MODEL.C program, introduced in Chapter 6, has some real-life examples of support files in use.

The Pascal Environment

Pascal (not to be confused with *UCSD Pascal*, an early Apple operating system) is famous because of its structure. In fact, most educational institutions prefer to teach programming with Pascal because it forces the student to think logically and to break a problem down into smaller, easier-to-solve tasks.

Currently, the only Pascal compiler for the Apple IIGS is the one from TML Systems of Jacksonville, Florida. It's more than just a compiler. In fact, *TML Pascal* is a complete and powerful program-development system.

Support files for the *APW* version of *TML Pascal* are kept in the TOOLINTF area on your *APW* disk. The regular *TML Pascal* allows you to define where the unit files are stored. They all end

with .USYM extensions because they are known unit symbol files. Rather than being included as source code as is done in C, unit symbol files are USED in *TML Pascal*. Here's an example:

```
{ Using Unit Symbol Files in Pascal }
PROGRAM Yawn;
USES QDIntF, GSIntF, MiscTools;
BEGIN
      TLStartUp;
      TLShutDown;
END.
```

The USES section of the Pascal program tells the compiler to use the unit symbol files included in the list. The corresponding functions for each tool set then become available for your program to work with.

TML doesn't intend to stop with Pascal. At this writing, they are about to release a BASIC compiler for the Apple IIGS.

The Machine Language Environment

If you're doing machine language development, you're probably using *APW*, the *Apple Programmer's Workshop*. So far, it's the most popular machine language development environment for the Apple IIGS.

To use the *APW* Assembler effectively, you'll need at least two disk drives, or one 3½-inch disk drive and a very large ramdisk of about 800K. The *APW* programs should be on one disk with your source code and any other files you need on the other. However, the best setup for any serious programming involves a hard disk with at least ten megabytes of storage. When this is the case, *APW* and all its files should be put in their own subdirectory.

The latest version of *APW* requires at least 768K of RAM on your computer, which is 512K more than the 256K that comes with the Apple IIGS .

When developing programs, it's best not to put all of your code into one, huge, cumbersome file. In fact, the best way to program is to keep your source code in small, separate modules. Not only will this help you keep track of updates (by checking the date column in a catalog listing), but it will reduce the time it takes to patch code.

The rest of the machine language examples in this book will, where applicable, use the modular concept to add pieces to the

MODEL.ASM program demonstrated in the next chapter. You can make decisions about how many modules to make, and what size to make them, on your own.

Modules are added, or chained, to one another by use of the COPY directive. For example, if the MODEL.ASM program references two other modules, DISKIO.ASM and WINDOW.ASM, the following directives should be placed at the end of the source code:

```
COPY    DISKIO.ASM
COPY    WINDOW.ASM
```

This will copy the source code from those two files to create the final program.

It is helpful to know that you're not chained to the *APW* Editor, considered by many to be a simple-minded text editor. By the time you read this, there should be several good public domain or shareware text editors on the market, any one of which could be used to edit *APW* source files.

Using *APW* is similar to using MS-DOS or UNIX. However, programs created under *APW* are not directly executable by the Finder or Launcher. You must change their filetype from an EXE ($B5) to a S16 ($B3) file type. This is done with the FILETYPE command at the *APW* system prompt. For example,

```
FILETYPE MODELA S16
```

changes the file type of the MODELA program from EXE to S16, allowing the program to be run directly from the Finder or Launcher.

Other than that, *APW* is straightforward and easy to use, considering that you're writing machine language. However, there is one more detail about the *APW:* Machine language programmers should pay special attention to the way the *APW* assembler uses macros.

Macros. *Macro* is a an abbreviation of *macroinstruction.* A macro is used to represent a number of other statements, like an abbreviation. Some complex macros can even make decisions and perform evaluations. Yet, you only write the macro instruction once. Then, from that point on, you use only the name of the macro to reference it.

You probably won't find any machine language examples in any books that don't use macros (other than *Mastering the Apple IIGS Toolbox*, where macros were not used in order to better explain

certain concepts). Because of this fact, all the machine language source code in this book uses macros. You'll find that macros make programs easier to read and easier to write. For this reason, the rest of this chapter is devoted to *APW* machine language macros and how to use them.

Macro etiquette. Macros are most commonly used to make Toolbox calls. With the *APW* assembler, the convention for Toolbox macros is to start them with the underscore character as in the following example that invokes the MoveTo Toolbox call:

```
_MoveTo
```

Case is unimportant as far as macros are concerned, unless you've specifically told the *APW* assembler to pay attention to case by using the CASE ON directive. Each of the following lines will invoke the MoveTo macro (assuming you have not used the CASE ON directive):

```
_moveto
_MOVETO
_MoVeTo
```

All Toolbox calls have a macro, as defined in the support files in the LIBRARIES/AINCLUDE subdirectory. And all Toolbox macros carry the same name as their Toolbox function, with each preceded by an underscore.

Aside from the Toolbox calls, several other *APW* assembler macro types are popular. The ones most often seen are these:

Macro	Action
PushLong	Push a long-word value onto the stack
PushWord	Push a word value onto the stack
PullLong	Pull a long-word value from the stack
PullWord	Pull a word value from the stack
Str	Create a Pascal string

There are some distinct advantages to using the PushLong and PushWord macros over the PEA instructions. The most common is the error that occurs when a memory location rather than a value is pushed to the stack (PEA $1234 instead of PEA #$1234). If you use the PushLong and PushWord macros, there will be no question about which type is being pushed.

Also, the Str macro eliminates some of the tedious labeling that occurs when defining a Pascal string. (Pascal strings start with

a count byte to tell the program how many characters to expect.) Because Pascal strings are used frequently in the Toolbox, the Str macro is very handy.

Macros at work. When the assembler sees your macro, it expands the macro into the code it stands for. This is one of the most confusing aspects of using macros.

Macros make the source code easier to read and easier to debug. They help the programmer avoid redundancy by eliminating the need to type the same code repeatedly. A beginning machine language programmer might assume that using macros tightens up code. That's only half true: Macros make your source code tighter, but your object code will be just as long as if you didn't use macros.

When your source code is assembled into object code, the macros you use are expanded out into their raw form. So for each _MMStartUp the assembler sees, it replaces it with the appropriate code:

```
ldx   #$0202
jsl   #E10000
```

Macros can be simple (as above) or complex. For example, a macro can look rather innocent in the middle of your source code:

```
PushLong   #1234
```

The PushLong macro is much more complex than _MMStartUp. PushLong will be translated by the assembler into the codes defined in the macro. Because there can be a number of arguments for PushLong (a value, a memory location, or a zero-page location plus an offset, the stack plus an offset, and so on), the PushLong macro must make a few decisions.

The actual definition for the PushLong macro is quite complex:

```
        MACRO
&lab    pushlong &addr,&offset
&lab    ANOP
        LCLC    &C
        LCLC    &REST
&C      AMID    &addr,1,1
        AIF     &C=#,.immediate
        AIF     &C=[,.zeropage
        AIF     C:&offset=0,.nooffset
        AIF     &offset=s,.stack
```

```
        pushword    &addr+2,&offset
        pushword    &addr,&offset
        MEXIT
.nooffset
        pushword    &addr+2
        pushword    &addr
        MEXIT
.immediate
&REST   AMID        &addr,2,L:&addr-1
        dc          I1'$F4',I2'(&REST)|-16'
        dc          I1'$F4',I2'&REST'
        MEXIT
.stack
        pushword    &addr+2,s
        pushword    &addr+2,s
        MEXIT
.zeropage
        ldy         #&offset+2
        pushword    &addr,y
        ldy         #&offset
        pushword    &addr,y
        MEND
```

Inside PushLong's definition are conditional branches and evaluations to determine exactly what type of long-word value is being pushed on the stack. The assembler, when it replaces the macro PushLong with the above instructions, will make certain evaluations and then use only those instructions to push the proper long value onto the stack.

For example, if

```
PushLong    #1234
```

is specified in your source, the assembler will use the following instructions from the PushLong macro to push #1234 onto the stack:

```
+           ANOP
+           LCLC    &C
+           LCLC    &REST
+&C         AMID    #1234,1,1
+.immediate
+&REST      AMID    #1234,2,L:&addr-1
+           dc      I1'$F4',I2'(1234)|-16'
+           dc      I1'$F4',I2'1234'
```


That seems like a very complex procedure to go through just to push a long word on the stack, yet some complex decision making is occurring. For PushLong to be a versatile macro, capable of pushing a variety of values onto the stack, it has to be complex. Fortunately, the logic and debugging of the PushLong macro has been taken care of for you. You need only specify it in your source and let the assembler do the rest.

To see how a macro expands, the TRACE ON directive can be listed at the top of your assembler source. By adding the LIST ON directive, you'll be able to see your source code as it's assembled and, with TRACE ON set, see the macros expanded as well.

Using macros in your source code. With the *APW* Assembler, macros exist in an external file and are referenced in your source code by the MCOPY directive:

```
MCOPY   [pathname]
```

The pathname is the name of a path or file that contains all your program's macro definitions. For example:

```
MCOPY   MYMACROS
```

The above instruction directs the assembler to look for any macro references in the file MYMACROS. Make sure you have this statement at the top of your source code. The macros used by your source cannot be accessed until the assembler has encountered the MCOPY command.

So, if your source code makes extensive use of QuickDraw II Toolbox calls, and you want to use the QuickDraw II macros supplied with *APW*, you could place the following at the top of your source code:

```
MCOPY   /APW/LIBRARIES/AINCLUDE/M16.QUICKDRAW
```

Because *APW* takes advantage of ProDOS 16 prefix numbers, you can substitute the number 2 for /APW/LIBRARIES above. (No matter what the configuration of your drive, using 2 will work. See the section in the *APW* manual about the LOGIN file for more information.)

```
MCOPY   2/AINCLUDE/M16.QUICKDRAW
```

After using this instruction, any QuickDraw II macros referenced in your source code will be replaced by the definitions in the M16.QUICKDRAW support file.

This sounds like a powerful feature when you're reading the *APW* Assembler manual and might cause you to think you could just MCOPY *all* the predefined macros in the AINCLUDE subdirectory into your source code. While that sounds logical, and it would make things easier, it's just not the case.

The MCOPY command only allows four macro files to be in use at one time. The manual seems to suggest that you can juggle these four macro files using the MLOAD and MDROP directives throughout your code. However, this is a fallacy. Rather than toss about MCOPY, MDROP, and MLOAD directives, it's much faster and easier to create a custom macro file for your source code files. This is done with the MACGEN program from the *APW* shell:

```
MACGEN [source.code] [macro.file] [macro.libraries . . .]
```

MACGEN creates a custom macro file for your source code. It's one of *APW*'s better utilities.

First, MACGEN reads in your entire source file. Then, it scans a specified list of macro support files, pulls out only the macros referenced by your source code, and finally creates a custom macro file containing only the macros referred to by your source.

For example, consider the program MODEL.ASM in the next chapter. MODEL makes extensive use of macros. Most of those macros are defined by the M16 files in the AINCLUDE prefix. To build a custom macro file containing only the macros referenced by MODEL.ASM, the following MACGEN command was typed at the *APW* system prompt:

```
MACGEN model.asm model.macros 2/ainclude/m16.=
```

This reads: From the source code model.asm, generate a macro file named model.macros using all the files that start with m16. in the subdirectory AINCLUDE. (The equal sign is a wildcard specifying all files starting with M16.)

MACGEN reads in the source code and then reads through all the files M16.= for any matching macros. It then places the macros it finds into the file MODEL.MACROS. To take advantage of them, the following is placed at the start of MODEL.ASM:

```
MCOPY    MODEL.MACROS
```

If your program has more than one module, you should use the MACGEN command on the main module. As long as the main module has COPY or APPEND directives, the other related source

file modules will also be scanned for macro references.

Building a custom macro file with the aid of MACGEN is the best way to provide the macros your program needs. If you update your source listing with new macro calls, you can run MACGEN a second time to create a new custom macro file. Also, any unique macros you create can be typed into the macro file using the *APW* editor.

Summary

Macro files, header files, unit symbol files, nearly all the information covered in this chapter can be found elsewhere. However, many people who consider themselves old hands at programming have never taken advantage of support files. With the Apple IIGS Toolbox at your disposal, using support files and paying attention to the tips offered in this chapter can make you a better programmer.

Chapter 6

The DeskTop

Applications for the Apple IIGS fall into two categories: DeskTop and non-DeskTop. This chapter introduces some new and exciting things happening in the DeskTop world of programming. It begins with a description of a DeskTop program and provides a sample program, in three languages, that you can run on your computer.

The DeskTop

A DeskTop program is one that takes advantage of the 16-bit processing power of the Apple IIGS and uses its built-in tools to manipulate pull-down menus, windows, dialog boxes, icons, the mouse, and so on. This interface has proven to be highly intuitive to the user and is popular on a variety computers. DeskTop programs written for the Apple IIGS will not run on the Apple IIe or IIc.

A non-DeskTop program is one written for the eight-bit personality of the Apple IIGS. This half of the computer, also called the Mega II, emulates an Apple IIe with 128K of RAM and a 65C02 processor. Programs in that environment rarely use the powerful tools that reside in the Apple IIGS Toolbox ROM. They are required to provide their own memory-management schemes and custom interfaces. This entails a lot of work for the programmer. However, these programs can run on the Apple IIGS as well as on the Apple IIe and IIc.

Having a "canned interface" inside the computer provides many advantages. Users feel at home with DeskTop programs because the interface is consistent from one program to the next. Programmers can concentrate on the tasks of their software and are spared the details of interacting with the user. Since most of the code for the interface resides in ROM, programs require only a few calls to drive the entire DeskTop.

The DeskTop interface, remarkably similar to that found in Apple's Macintosh computer, is the most exciting aspect of the Apple IIGS.

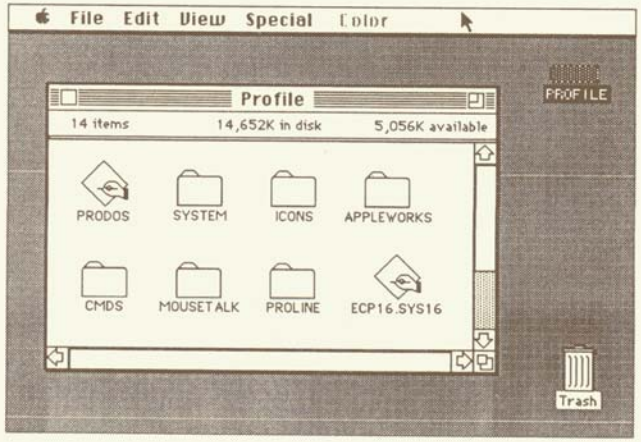
Managers

Here's a quick description of the Apple IIGS DeskTop and how the various managers built into the IIGS are responsible for maintaining it.

When a DeskTop program is first launched, a blank background pattern is displayed across the entire Apple IIGS super-hires graphics screen. Traditionally, the background pattern is a solid shade of light blue, though the programmer can choose any color supported by computer.

Inevitably, the DeskTop will have a menu bar at the top of the screen which contains the titles of one or more pull-down menus. These menus contain all of the program's commands and functions available to the user.

Figure 6-1. Figure of DeskTop with Menus, Dialog Boxes, and Windows



It is the responsibility of the Event Manager to track the location of the mouse and update the mouse pointer on the screen. The mouse pointer, an arrow shape, marks the position on the screen where the mouse is located on the DeskTop. Moving the physical mouse device will cause the mouse pointer to move accordingly on the screen. This function is completely transparent to the DeskTop application because it relies on the interrupt feature of the Apple IIGS microprocessor.

The mouse is used to select items on the DeskTop. For example, the user moves the mouse pointer over a title on the menu bar and presses the mouse button. This causes a pull-down menu to be displayed, showing a list of available selections. By holding down the mouse button and moving the pointer (an action called *dragging*) the user chooses a menu item from the menu. A selection is made when the mouse button is released.

The programmer organizes what is to be placed into the menus and passes that information along to the Menu Manager. The job of drawing pull-down menus and interacting with the user while a selection is made is handled completely by the Menu Manager. To do this, of course, it relies on other tool sets, especially QuickDraw II.

Some menu items are selected with the keyboard instead of the mouse. This is done by pressing the Open Apple key in conjunction with another key that corresponds to a menu item. The user determines if a menu item has a keyboard equivalent by examining the list of items in a pull-down menu. Menu items with keyboard equivalents have apple symbols, followed by the command character, after their name in the menu.

For the programmer, all of the work involved in getting the user's selections via the mouse or keyboard equivalents is handled by the routines in the Toolbox. It's the job of the Window Manager's TaskMaster function to manage these details.

After a menu item is selected, any number of events might occur. As an example, a dialog box could be displayed asking the user to supply input for the application. Appropriately named, dialog boxes let the user communicate with the DeskTop program by filling in blank entries with text, turning switches on or off, pressing buttons, or by using other controls.

Using software, the programmer builds the dialog box to the required specifications. Buttons and other controls can be installed on the box. The functions in the Dialog Manager and Control Manager allow the user to manipulate the controls and report to the application which buttons have been pressed.

The function of a typical DeskTop program is as simple as making a selection from a vending machine. The user makes selections from the menu bar and interacts with a few dialog boxes, and the computer performs its assigned task.

The Apple IIGS has more managers than a small baseball league. The programmer is well assisted in driving the DeskTop.

Parts of a DeskTop Program

At the software level, DeskTop applications consist of three main parts:

Startup	Before a program can begin to interact with the user, it must complete the startup phase. This involves starting a host of tool sets, allocating memory, and setting up the DeskTop environment with pull-down menus and so forth.

Event handling — Once everything is initialized, a DeskTop program basically sits idle, waiting for the user to make selections from the pull-down menus. When a menu item is selected, a corresponding function for that item is dispatched and carried out.

Shutdown — Eventually, the user will be finished with the program and will want to quit. As part of the shutdown process, the application will take care of unfinished business, such as saving changes to disk. It shuts down the tool sets it started up, deallocates reserved memory, and exits to the operating system.

These three steps provide the basic framework of practically every DeskTop program written. The nice thing about this is that once you've created the *overhead code* (the basic code that performs these three functions), it can be used over and over again for new programs.

The Tower of Babel

The following sample program—shown here in *APW* machine language source code, *APW C*, and *TML Pascal*—demonstrates how a typical DeskTop program starts up, handles events, and shuts down. It doesn't do anything spectacular. But it sets the stage for some very exciting programming ventures using the powerful abilities of the Apple IIGS Toolbox.

Referring to these programs as models, the next few chapters will describe the important details in creating DeskTop programs. Study closely the program listing written in the language you're most interested in.

Program 6-1. MODEL.ASM

```
*-------------------------------------------------------*
*                    MODEL.ASM                          *
*  Sample Desktop Application in APW Assembler (1.0)    *
*-------------------------------------------------------*

: To create the ModelMacs macro file, use this APW shell command:
: # macgen model.asm modelmacs 2/ainclude/m=

        ABSADDR ON
        KEEP    ModelA
        MCOPY   ModelMacs

*------------------------------*
*      Global Equates          *
*------------------------------*
```



```
Toolbox gequ    $e10000         ;Primary tool dispatcher
TRUE    gequ    $8000           ;True value
FALSE   gequ    $0000           ;False value
Page    gequ    $100            ;The size of a page (256 bytes)

ModelA  START
        phk                     ;Make the data bank...
        plb                     ;...the current code bank
        brl     Main            ;branch over functions to Main

*------------------------------*
*    Handle Toolbox Errors     *
*------------------------------*

ErrChk  bcs     Die             ;Carry set if error

        rts                     ;Else, return

Die     pha                     ;Toolbox returns error in A
        pushlong #0             ;Use standard system death message
        _SysFailMgr             ;Get ready to slide apples back and forth

*------------------------------*
*  Manage Direct Page Buffers  *
*------------------------------*
; Returns address of next free Direct Page. (Modifies Y register)
; The GetDPs entry point requires byte count in A register.

GetDP   lda     #Page           ;Ask for one 256 byte DP block
GetDPs  clc                     ;Alternate entry: A = Number of bytes
        ldy     DPBase          ;Get base value (we return this)
        adc     ,DPBase,DPBase  ;Add A to our last DP buffer address
        tya                     ;Return entry value
        rts

*------------------------------*
*       Start Up Tools         *
*------------------------------*

DPSpace equ     $000600         ;Memory needed for direct pages
HndlRef equ     $00             ;direct page handle deref pointer

StartUpTools    anop
        _TLStartUp              ;----Start the Tool Locator

        pha                     ;Result Space for User ID
        _MMStartUp              ;----Start the Memory Manager
        jsr     ErrChk          ;Check for errors
        pullword UserID         ;Get our User ID and save it
        ora     #%100000000     ;munge an auxiliary ID...
        sta     MemID           ;used for Memory Manager handle usage

        _MTStartUp              ;----Start the Misc Toolset

; Get direct page space for other tools

        pha                     ;Long result space...
        pha                     ;...for returned handle
        pushlong #DPSpace       ;Long value: size of memory block
        pushword MemID          ;Use the special ID for handle allocation
```

```
        pushword #$c005         ;Fixed, Page-aligned, Locked, Unpurgable
        pushlong #$000000       ;Where our block resides ($00/0000)
        _NewHandle
        jsr     ErrChk          ;Check for errors
        pullong HndlRef         ;Get handle of new block
        lda     [HndlRef]       ;Get address of the storage area
        sta     DPBase          ;Save it for GetDP utilities

        lda     #3*Page         ;QuickDraw requires 3 direct pages
        jsr     GetDPs          ;Get address for them...
        pha                     ;...and push it
        pushword #$0080         ;Screen Mode (use $0000 for 320 mode)
        pushword #$00a0         ;Pixel Map Size (use $0050 for 320 mode)
        pushword UserID         ;Push our program's ID
        _QDStartUp              ;----Start QuickDraw II
        jsr     ErrChk

        jsr     GetDP           ;Requires a Direct Page
        pha                     ;Push the DP address
        pushword #20            ;Event queue size
        pushword #0             ;Min X clamp
        pushword #640           ;Max X clamp (640 mode)
        pushword #0             ;Min Y clamp
        pushword #200           ;Max y clamp = 200 (bottom of screen)
        pushword UserID         ;Push program's User ID
        _EMStartUp              ;----Start the Event Manager
        jsr     ErrChk
        pushword #0             ;Standard background color
        _SetBackColor

        pushword #3             ;Standard foreground color
        _SetForeColor

        pushword #260           ;X position of message
        pushword #85            ;Y position of message
        _MoveTo                 ;Move pen to X,Y

        pushlong #Moment        ;Point to a message string
        _DrawCString            ;Print "One moment..."

        pushlong #Toollist      ;Point to a list of tools
        _LoadTools              ;Read tools from disk into RAM
        jsr     ErrChk

        pushword UserID         ;Requires User ID
        _WindStartUp            ;----Start the Window Manager
        jsr     ErrChk

        pushword UserID         ;Requires User ID...
        jsr     GetDP           ;...and a Direct Page
        pha
        _CtlStartUp             ;Start the Control Manager
        jsr     ErrChk

        pushword UserID         ;Requires User ID...
        jsr     GetDP           ;...and a Direct Page
        pha
        _MenuStartUp            ;----Start the Menu Manager
        jsr     ErrChk

        _DeskStartUp            ;----Start the Desk Manager
        rts
```



```
*------------------------------*
*   Prepare Desktop and Menus  *
*------------------------------*

PrepDeskTop     anop
        pushlong #0             ;Draw entire desktop using default values
        _RefreshDesktop
        _InitCursor

NxtMenu pha                     ;Result Space (Long) for...
        pha                     ;...the menu's handle
        lda     MenuTbl         ;Get menu count
        asl     A               ;x 2
        tax                     ;Make it an index into word values
        lda     MenuTbl,x       ;Get address of menu structure
        phb                     ;Push program bank twice
        phb                     ;(PHB pushes only a byte)
        pha                     ;Push address of menu structure
        _NewMenu                ;the menu handle is now on the stack
        pushword #0             ;Insert menu at left, shifting right
        _InsertMenu

        dec     MenuTbl         ;More menus to install?
        bne     NxtMenu         ;Yes

        pushword #1             ;Put Desk Accessories's in Apple Menu
        _FixAppleMenu

        pha                     ;Result space
        _FixMenuBar             ;Calculate menu bar's height
        pla                     ;Discard height for now

        _DrawMenuBar            ;Display the menu bar
        rts

*------------------------------*
* Apple Menu: About            *
*------------------------------*

About   rts                     ;Does nothing (for now)

*------------------------------*
* File Menu: Quit              *
*------------------------------*

Quit    dec     QFlag           ;User wants to quit (QFlag = $ffff)
        rts

*------------------------------*
*      Do Menu Selection       *
*------------------------------*

DoMenu  lda     TaskData        ;Get TaskData Item ID number
        and     #$00ff          ;Discard upper 8-bits
        asl     A               ;Double the value
        tax
        jsr     (MTable,x)      ;Dispatch the proper menu item handler

        pushword #FALSE         ;We need to unHilite the menu title now
        pushword TaskData+2     ;Get TaskData Menu number
```

```
        _HiliteMenu             ;Unhilite menu title
        rts

*-----------------------------*
*      Shutdown Toolsets      *
*-----------------------------*

ShutDownTools   anop
        lda     Toolist         ;Get # of toolsets started up
        asl     A               ;x2
        asl     A               ;x2 (to create index over longwords)
        tax
        lda     Toolist-2,x     ;Get toolset ID from list
        cmp     #$0002          ;Memory Manager?
        bne     Shut1           ;No, so shut this down right now
        pushword MemID          ;Dispose all handles allocated
        _DisposeAll
        pushword UserID         ;Shut down this program's memory
        lda     #$0002          ;MMShutDown
Shut1   ora     #$0300          ;Make it a shutdown call
        tax                     ;Set X to call number
        jsl     Toolbox         ;Shut it down!
        dec     Toolist         ;Shutdown another toolset?
        bne     ShutDownTools   ;Yes

        rts

*-----------------------------*
*           Main              *
*-----------------------------*

Main    jsr     StartUpTools    ;Start toolsets
        jsr     PrepDeskTop     ;Prepare desktop and menus

Scan    pha                     ;Result Space
        pushword #$ffff         ;Event Mask
        pushlong #EventRec      ;Point to Event Record
        _TaskMaster
        pla                     ;Get HandleEvent flag
        beq     Scan            ;If nothing, continue looping

        cmp     #$11            ;A menu event? ($11=wInMenuBar)
        bne     Scan            ;Nope, just keep scanning

        jsr     DoMenu          ;Do menu item dispatch
        bit     QFlag           ;Time to quit?
        bpl     Scan            ;No, keep scanning for events

        jsr     ShutDownTools   ;Shut down all tools started

        _QUIT   Qparms          ;Exit this program through ProDOS 16

*-----------------------------*
*     Variable Storage        *
*-----------------------------*

UserID  ds      2               ;Our User ID
MemID   ds      2               ;Memory User ID (made from User ID)
DPBase  ds      2               ;Used by DP buffer manager
QFlag   dc      i'FALSE'        ;Boolean: Quit flag (starts out as false)
```



```
*-----------------------------*
*  StartUp/Shutdown Tool List *
*-----------------------------*

Toolist dc      i'(ToolstE-Toolist-1)/4'        ;Tool count
        dc      i'1,0'          ; Tool Locator
        dc      i'2,0'          ; Memory Manager
        dc      i'3,0'          ; Misc Tools
        dc      i'4,0'          ; QuickDraw II
        dc      i'6,0'          ; Event Manager
        dc      i'14,0'         ; Window Manager
        dc      i'16,0'         ; Control Manager
        dc      i'15,0'         ; Menu Manager
        dc      i'5,0'          ; Desk Manager
ToolstE         anop
*-----------------------------*
*   Pull Down Menu Structures *
*-----------------------------*

MenuTbl dc      i'(MenTblE-MenuTbl-1)/2'        ;Menu count
        dc      i'Menu1'        ;Apple
        dc      i'Menu2'        ;File
        dc      i'Menu3'        ;Edit
MenTblE anop

Menu1   dc      c'>>@\XN1',i1'0'                ;Apple
        dc      c'--About This Program...\N256',i1'0'
        dc      c'---\D',i1'0'
        dc      c'>'

Menu2   dc      c'>>  File  \N2',i1'0'          ;File
        dc      c'--Quit\N257*Qq',i1'0'
        dc      c'>'

Menu3   dc      c'>>  Edit  \N3D',i1'0'         ;Edit
        dc      c'--Undo\N250V*Zz',i1'0'
        dc      c'--Cut\N251*Xx',i1'0'
        dc      c'--Copy\N252*Cc',i1'0'
        dc      c'--Paste\N253V*Vv',i1'0'
        dc      c'--Clear\N254',i1'0'
        dc      c'>'

*-----------------------------*
* Menu Item Dispatch Addresses *
*-----------------------------*

MTable  dc      i'About'        ;256/About      (Apple Menu)
        dc      i'Quit'         ;257/Quit       (File Menu)

*-----------------------------*
*      The Event Record       *
*-----------------------------*

EventRec        anop                    ;Event Record used by TaskMaster
EWhat           ds      2               ;What
EMsg            ds      4               ;Message
EWhen           ds      4               ;When
EWhere          ds      4               ;Where
EMods           ds      2               ;Modifiers
TaskData        ds      4               ;Task Data
TaskMask        dc      i4'$1fff'       ;Task Mask
```

```
*------------------------------*
*      Miscellaneous Data      *
*------------------------------*

Moment  dc      c'One Moment...',i1'0'

GParms  dc      i4'0'           :ProDOS 16 Quit Code parameters
        dc      i'$0000'

        END
```

The sample program written in *APW* machine language will create a four-block object file on disk. Of that, one block (512 bytes) of header information is used for the System Loader. The last three blocks contain the actual machine language program.

Program 6-2. MODEL.C

```
/*---------------------------------------------*
 *                  MODEL.C                    *
 * Sample Desktop Application in APW C (1.0)   *
 *---------------------------------------------*/

#include <types.h>
#include <stdio.h>
#include <locator.h>
#include <memory.h>
#include <misctool.h>
#include <quickdraw.h>
#include <event.h>
#include <window.h>
#include <menu.h>
#include <control.h>
#include <desk.h>

/*------------------------------*
 *      Global Variables        *
 *------------------------------*/

WmTaskRec   EventRec;           /* Event Record Structure */

Word        Event,              /* Event code */
            UserID,             /* Our User ID */
            MemID,              /* Memory Management ID */
            QFlag;              /* Boolean: Quit flag */

Word        Toolist[] = {
                3,              /* Tool count */
                14, 0,          /* Window Manager */
                15, 0,          /* Menu Manager */
                16, 0           /* Control Manager */
};

char        *DPBase;            /* Direct Page base pointer */

/*------------------------------*
 *    Handle Toolbox Errors     *
 *------------------------------*/
```



```
ErrChk()                        /* Check for error, die if so */
{
    if (_toolErr) SysFailMgr(_toolErr, nil);
}

/*------------------------------*
 *  Manage Direct Page Buffers  *
 *------------------------------*/

char    *GetDP(bytes)
Word    bytes;
{
    char *OldDP = DPBase;
    DPBase += bytes;            /* Update base level pointer */
    return (OldDP);             /* Return old DPBase pointer */
}

/*------------------------------*
 *        Start Up Tools        *
 *------------------------------*/

StartUpTools()
{
    Word    GetDP();        /* Force words from GetDP */

    TLStartUp();
    UserID = MMStartUp();                                       ErrChk();
    MemID = UserID | 256;
    MTStartUp();
    DPBase = *(NewHandle(0x600L, MemID, 0xc005, nil));          ErrChk();
    QDStartUp(GetDP(0x300), 0x80, 0xa0, UserID);                ErrChk();
    EMStartUp(GetDP(0x100), 0x14, 0, 0x280, 0, 0xc8, UserID);   ErrChk();

    SetBackColor(0);                /* Show Intro Screen */
    SetForeColor(3);
    MoveTo(0x104, 0x55);
    DrawCString("One Moment...");

    LoadTools(Toolist); ErrChk();   /* Load & Startup tools */

    WindStartUp(UserID);                ErrChk();
    CtlStartUp(UserID, GetDP(0x100));   ErrChk();
    MenuStartUp(UserID, GetDP(0x100));  ErrChk();
    DeskStartUp();
}

/*------------------------------*
 *  Prepare Desktop and Menus   *
 *------------------------------*/

PrepDeskTop()
{

    static char *AppleMenu[] = {
        ">>@\\XN1",
        "--About This Program...\\N256",
        "---\\D",
        ">"
    };
```

```
    static char *FileMenu[] = {
        ">>  File  \\N2",
        "--Quit\\N257*Qq",
        ">"
    };

    static char *EditMenu[] = {
        ">>  Edit  \\N3D",
        "--Undo\\N250V*Zz",
        "--Cut\\N251*Xx",
        "--Copy\\N252*Cc",
        "--Paste\\N253V*Vv",
        "--Clear\\N254",
        ">"
    };

    RefreshDesktop(nil);                    /* Display Desktop */
    InitCursor();                           /* Show mouse cursor */

    InsertMenu(NewMenu(EditMenu[0]), 0);    /* Install menus */
    InsertMenu(NewMenu(FileMenu[0]), 0);
    InsertMenu(NewMenu(AppleMenu[0]), 0);

    FixAppleMenu(1);                        /* Display menu bar */
    FixMenuBar();
    DrawMenuBar();
}

/*-----------------------------*
 *  Apple Menu: About          *
 *-----------------------------*/

About()
{
        /* Does nothing (for now) */
}

/*-----------------------------*
 *      Do Menu Selection      *
 *-----------------------------*/

DoMenu()
{
    switch(EventRec.wmTaskData) {
    case 256:   About;              /* Apple Menu: About */
                break;
    case 257:   QFlag = TRUE;       /* File Menu: Quit */
                break;
    }

    HiliteMenu(FALSE, EventRec.wmTaskData>>16);
}

/*-----------------------------*
 *      Shutdown Toolsets      *
 *-----------------------------*/
```



```
ShutDownTools()
{
    DeskShutDown();
    MenuShutDown();
    CtlShutDown();
    WindShutDown();
    EMShutDown();
    QDShutDown();
    MTShutDown();
    DisposeAll(MemID);
    MMShutDown(UserID);
    TLShutDown();
}

/*-----------------------------*
 *          Main               *
 *-----------------------------*/

main()
{
    StartUpTools();             /* Start toolsets */
    PrepDeskTop();              /* Prepare desktop and menus */

    QFlag = FALSE;
    EventRec.wmTaskMask = 0x00001fff;

    while (!QFlag) {            /* Wait for a menu event */
        do {
            Event = TaskMaster(0xffff, &EventRec);
        } while (!Event);
        if (Event == wInMenuBar) DoMenu();
    }

    ShutDownTools();            /* Shutdown all tools started */
    exit(0);
}
```

The sample program written in *APW C* compiles into a 16-block object file. However, a compiled C program containing no instructions at all produces a 12-block file. This means that, like the assembly program, about 4 blocks contain the actual code, while the other 12 consist mostly of overhead from the standard C library and System Loader.

Program 6-3. MODEL.PAS

```
{ *-------------------------------------------------*
  *                  MODEL.PAS                      *
  * Sample Desktop Application in TML Pascal (v1.01) *
  *-------------------------------------------------* }

PROGRAM ModelP;

USES    QDIntF,
        GSIntF,
        MiscTools;
```

```
{ *------------------------------*
  *       Global Variables       *
  *------------------------------* }

VAR EventRec:   EventRecord;    { Taskmaster Structure }
    Event:      Integer;        { Event code }
    UserID:     Integer;        { Our User ID }
    MemID:      Integer;        { Memory allocation ID }
    DPBase:     Integer;        { Direct Page base pointer }
    QFlag:      Boolean;        { Boolean: Quit flag }

    AppleMenu:  String;         { Pull down menu strings }
    FileMenu:   String;
    EditMenu:   String;

{ *------------------------------*
  *    Handle Toolbox Errors     *
  *------------------------------* }

PROCEDURE ErrChk;               { Check for error, die if so }
BEGIN
    IF IsToolError THEN
        SysFailMgr(ToolErrorNum, 'Tool error -> $');
END;

{ *------------------------------*
  *  Manage Direct Page Buffers  *
  *------------------------------* }

FUNCTION GetDP(bytes: Integer): Integer;
BEGIN
    GetDP := DPBase;            { Return current DPBase }
    DPBase := DPBase + bytes;   { Update base level pointer }
END;

{ *------------------------------*
  *       Start Up Tools         *
  *------------------------------* }

PROCEDURE StartUpTools;
VAR
    Toolist:    ToolTable;      { Disk-based tool list }
    Height:     Integer;        { Menu bar heigth (unused) }

BEGIN
    Toolist.NumTools := 3;                  { Tool count }
    Toolist.Tools[1].TSNum := 14;           { Window Manager }
    Toolist.Tools[1].MinVersion := 0;
    Toolist.Tools[2].TSNum := 15;           { Menu Menager }
    Toolist.Tools[2].MinVersion := 0;
    Toolist.Tools[3].TSNum := 16;           { Control Manager }
    Toolist.Tools[3].MinVersion := 0;

    TLStartUp;
    UserID := MMStartUp;                                        ErrChk;
    MemID := UserID + 256;
    MTStartUp;
    DPBase := LoWord(NewHandle($600, MemID, $c005, Ptr(0))^);   ErrChk;
    QDStartUp(GetDP($300), $80, $a0, UserID);                   ErrChk;
    EMStartUp(GetDP($100), $14, 0, $280, $0, $c8, UserID);      ErrChk;
```



```
    SetBackColor(0);                { Show Intro Screen }
    SetForeColor(3);
    MoveTo($104, $55);
    DrawString('One Moment...');

    LoadTools(Toolist); ErrChk;     { Load & Startup tools }

    WindStartUp(UserID);                ErrChk;
    CtlStartUp(UserID, GetDP($100));    ErrChk;
    MenuStartUp(UserID, GetDP($100));   ErrChk;
    DeskStartup;
END;

{ *------------------------------*
  *  Prepare Desktop and Menus   *
  *------------------------------* }

PROCEDURE PrepDeskTop;
VAR
    Height:     Integer;

BEGIN
    AppleMenu := CONCAT('>>@\XN1\0',
                        '--About This Program...\N256\0',
                        '---\D\0',
                        '>');

    FileMenu := CONCAT('>>  File  \N2\0',
                       '--Quit\N257*Qq\0',
                       '>');

    EditMenu := CONCAT('>>  Edit  \N3D\0',
                       '--Undo\N250V*Vz\0',
                       '--Cut\N251*Xx\0',
                       '--Copy\N252*Cc\0',
                       '--Paste\N253V*Vv\0',
                       '--Clear\N254\0',
                       '>');

    Refresh(Nil);                       { Display Desktop }
    InitCursor;                         { Show mouse cursor }

    InsertMenu(NewMenu(@EditMenu[1]), 0);   { Install menus }
    InsertMenu(NewMenu(@FileMenu[1]), 0);
    InsertMenu(NewMenu(@AppleMenu[1]), 0);

    FixAppleMenu(1);                    { Display menu bar }
    Height := FixMenuBar;
    DrawMenuBar;
END;

{ *------------------------------*
  *  Apple Menu: About           *
  *------------------------------* }

PROCEDURE About;
BEGIN
      { Does nothing (for now) }
END;
```

```
{ *------------------------------*
  *       Do Menu Selection      *
  *------------------------------* }

PROCEDURE DoMenu;
BEGIN
    CASE LoWord(EventRec.TaskData) OF
        256:    About;                  { Apple Menu: About }
        257:    QFlag := TRUE;          { File Menu: Quit }
    END;

    HiliteMenu(FALSE, HiWord(EventRec.TaskData));
END;

{ *------------------------------*
  *      Shutdown Toolsets       *
  *------------------------------* }

PROCEDURE ShutDownTools;
BEGIN
    DeskShutDown;
    MenuShutDown;
    CtlShutDown;
    WindShutDown;
    EMShutDown;
    QDShutDown;
    MTShutDown;
    DisposeAll(MemID);
    MMShutDown(UserID);
    TLShutDown;
END;

{ *------------------------------*
  *            Main              *
  *------------------------------* }

BEGIN
    StartUpTools;               { Start toolsets }
    PrepDeskTop;                { Prepare desktop and menus }

    QFlag := FALSE;
    EventRec.TaskMask := $00001fff;

    REPEAT                      { Wait for a menu event }
        REPEAT
            Event := TaskMaster($ffff, EventRec);
        UNTIL Event <> 0;
        IF Event = wInMenuBar THEN DoMenu;
    UNTIL QFlag;

    ShutDownTools               { Shutdown all tools started }
END.
```

Surprisingly, the *TML Pascal* example compiles into an eight-block runtime file, half the size of the C program.

These sample programs are written so they can be compared to each other easily. This does not necessarily mean that they have been written in the best format for the language used. For example,

since Pascal is relatively inflexible, the machine language and C programs have a very Pascal-like "bottom-up" format in order to keep the functions parallel.

Since all three programs make extensive use of routines in ROM, they run at roughly the same speed. They should be studied carefully and used as models for more complex DeskTop applications.

If you diligently typed in one of the model program listings, successfully compiled it, and were mildly impressed with the results, don't give up now. These model programs were intentionally created as skeletons. They form the basic parts of all DeskTop applications.

The chapters that follow discuss key portions of the sample programs in greater detail. They show how to add more pull-down menus, custom windows, dialog and alert boxes, and special dialog controls. With a little imagination and this book at your side, you're on your way toward a rewarding programming experience.

Chapter 7

Memory Management

Many Apple IIGS programmers have their roots in earlier Apple II computers. Perhaps you're one of them. If so, you are well aware of the anarchy that prevailed in the 64K RAM Apple II. Apple had cleaned up the neighborhood when the memory-management system was created

for the Apple IIGS. It was something that had to be done.

This chapter is about memory management. It may sound like a dry subject, but it really isn't. In fact, compared to the jungle-gym memory management of earlier Apple II computers, the designers of the Apple IIGS have blessed the programmer with a memory-management system that's reliable, easy to manage, and easy to program.

New Rules

Imagine an Apple II with eight megabytes of RAM (128 times more memory than a 64K Apple II) and no sensible way of managing it all. Programs would overwrite each other, and there would be no way to locate lost data, which might be intact but as irretrievable as a needle in a hayfield. Before long, memory would be as packed with as much useless information as a poorly managed bookstore. A horrific thought. But thanks to the Memory Manager built into the IIGS, programs can coexist in peace for the first time in Apple II history.

This is a radical departure from the programming environment of older Apple IIs. If you're moving up to an Apple IIGS from a IIe (or IIc or II+), you're in for a surprise. Gone are the days when a program grabbed a hunk of memory for its own purposes.

With the Memory Manager in charge, memory blocks are allocated to applications that request them. Memory blocks can be any size, and they can contain any type of information. But a program must specifically ask for a block of memory or risk the complete destruction of any space it arbitrarily claims.

A memory block may be located anywhere in RAM. It is very rare for a program to ask for a block of memory that always resides at a fixed address in the machine. In fact, it's considered sloppy programming if your application cannot deal with memory blocks that move around in the Apple IIGS. The memory blocks that the Memory Manager hands out will not always live at the same address in the computer, and there's a good reason for this.

As more and more applications reside inside the computer at the same time, their impact on memory usage will vary. Some programs might require a small portion of RAM for temporary usage and then throw it away when it's no longer needed. Other programs might require memory blocks that could be considered permanently reserved. And still other programs may require great amounts of memory. Managing that memory without the assistance

of the Memory Manager would be a big headache.

So, imagine having hundreds of small memory blocks scattered throughout your computer's memory. Then imagine that your application needs a large, contiguous piece of RAM, but an unused area of memory that size doesn't exist. If you couldn't move the smaller blocks, rearranging them to make room for the one large block, the program would crash. With this kind of demand on RAM, things can get messy fast if memory blocks are not allowed to be moved.

Figure 7-1. Memory Blocks Distributed All Over Memory in a Random Dispersal

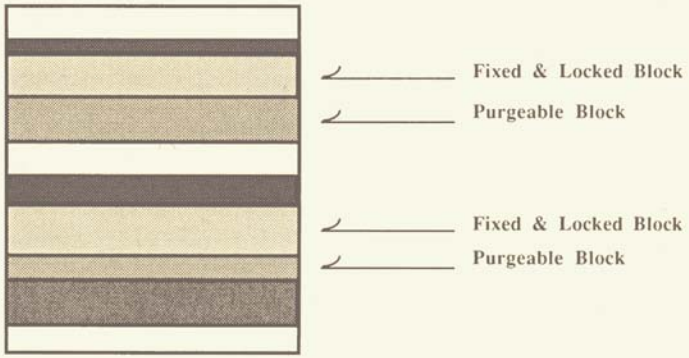
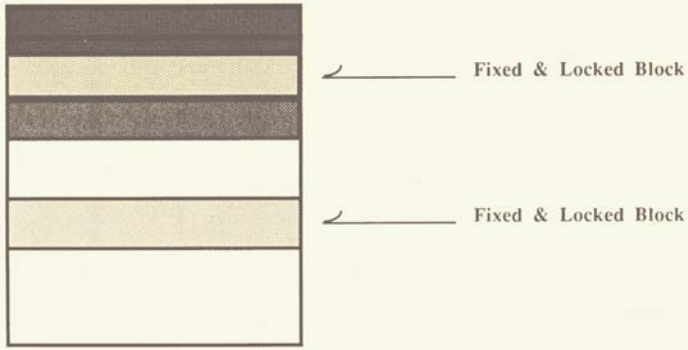


Figure 7-2. Memory Blocks After Reorganization by the Memory Manager



Fortunately, part of the Memory Manager's job is reorganizing memory blocks. It shuffles movable blocks around in an efficient and resourceful manner. This is done by removing blocks that are flagged as unused and then sliding blocks around in order to fill any gaps. The effect is that the landscape inside the computer is kept neat and orderly.

Don't let this worry you. It's possible for an application to request a block of memory that will reside at a fixed location, if you want it. However, the odds of the Memory Manager denying your request are higher because that space might already be reserved by another program.

Of course, if blocks of memory can be allowed to move about, seemingly at will, there must be a way to keep track of where they are.

Getting a Handle on Memory Blocks

Since a memory block can move around inside the computer, it is referenced by a handle. You'll see handles used with anything that moves about or that doesn't have a specific, given, or constant location, such as memory blocks, records, structures, and so on. Handles are simply long-word pointers to an address stored in memory, and they're used frequently in programming the Apple IIGS.

In the case of memory blocks, a handle points to a location in memory that contains a list of items. This list is also referred to as the *memory-block record*. For example, the first item in the list is a long-word address containing the actual location of the memory block's data in memory. The other items will be discussed later in this chapter.

Recall that a memory handle is a pointer. It points to a list of items. The first item in the list is an address which points to the location in memory where the memory block lives. This can be confusing.

If the memory block is moved, the only thing that changes is the address in the memory-block record. The handle still points to the same structure. Your program won't need to adjust anything if it's working with the handle correctly to begin with.

Suppose that you have a friend whose name is Kitty. On a page in your address book, you have recorded her name, address, birth date, and dozens of other pieces of information about her.

Kitty has a problem: She is always being evicted from her

apartment. Whatever other information you have about Kitty is always the same: She never changes her hair color, her birth date, her parents, or her telephone number. Only her address. The line in your address book where her address is written is the only thing you need to change in order to keep up to date on Kitty. That much-erased line in your address book is analogous to the memory-block record.

Starting the Memory Manager

Just as its name implies, the Memory Manager is responsible for keeping the computer's RAM neatly organized. This is done by tagging with an identification number each chunk of memory owned by an application. Whenever the Memory Manager is called upon for moving, purging, or manipulating a block of memory, your program must identify its piece of RAM. This is done by passing along an identification value when calling the Memory Manager.

Even the space that your program occupies is branded with its own identification number. The ID of your program is obtained when the Memory Manager is started.

The following examples show how a program obtains its own ID. This is typically one of the first calls an application should make.

In machine language:

```
pha                     ;word result space
_MMStartUp              ;start the Memory Manager
pla                     ;pull Master User ID
sta             UserID  ;and save it
```

In Pascal:

```
UserID := MMStartUp;
```

In C:

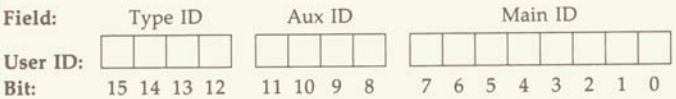
```
UserID = MMStartUp( );
```

These samples demonstrate the steps involved in starting the Memory Manager in 65816 machine language, Pascal, and C. The UserID, declared as a 16-bit unsigned integer, is a unique identifier that belongs to your application. It should always be saved for later use.

User ID Numbers

The ID value returned by MMStartUp is your program's master User ID. It references the space your application takes up in memory. The User ID is used when shutting down both your program and the Memory Manager.

The ID value consists of 16 bits, grouped into three parts, or fields. Bit positions within the word value represent the different fields:



The Type ID field occupies bits 12–15. Type ID identifies the class of software the User ID belongs to. It may be one of 11 values:

Value	Class of Software
$0	Memory Manager
$1	Application
$2	Control program
$3	ProDOS
$4	Tool set
$5	Desk accessory
$6	Runtime libraries
$7	System Loader
$8	Firmware
$9	Tool Locator
$A	Setup file
$B–$F	Undefined

The Auxiliary ID field occupies bits 8–11. This field is initially set to 0, but you can manipulate it to create up to 16 different sub-ID's for your program. For example, to set bit 8 (the least significant bit of the Aux ID field), the following can be done.

In machine language:

```
lda   UserID            ;Get the User ID . . .
ora   #%100000000       ;. . . and set bit 8
sta   MemID             ;Save the new ID
```

In Pascal:

```
MemID := UserID + 256;
```

In C:

```
MemID = UserID | 256;
```

This should be done before an application requests memory from the Memory Manager. An auxiliary ID value is used rather than the program's User ID. The memory allocated can then be categorized by your program as an example of using this field. Otherwise, if you don't want to get that detailed, you can ignore the Aux ID field. But it's there if you need it.

The Main ID field occupies the lower eight bits of the User ID returned from the Memory Manager. This is a unique number assigned to your program's User ID by the Memory Manager. Your User ID is what makes your program special. It makes your program different from any others that are running in the machine. The lower eight bits of your User ID should never be altered. To do so would be like changing your own fingerprints.

Asking for Memory

When a ProDOS 16 application is launched, it is given its own 64K bank of memory to live in. It also has its own direct page and stack. If the program requires memory outside its code space for storage, it must call the Memory Manager's NewHandle function to request a block of memory.

This 65816 code segment calls the NewHandle function in order to request a 256-byte buffer in bank $00 of the computer:

```
pha                     ;long word result space
pha                     ;
pushlong    #$100       ;push size of requested block (one page)
pushword    MemID       ;push a Memory ID (made from the User ID)
pushword    $C005       ;Attribute bits (discussed later)
pushlong #0             ;Location of block in memory (not used)

_NewHandle              ;call the NewHandle function
```

This call requires four input parameters (and result space when called from machine language) in order to work:

Value	Parameter Description
Long word	Size of the memory block needed
Word	An ID value to assign to this block
Word	Attributes (discussed later)
Long word	Location of the block (if applicable)

Value	Parameter Description
Size	The size of a memory block can be anything from zero bytes to whatever free memory space there is left in the machine.
ID	As described earlier in this chapter, each memory block allocated to a program must be identified by an ID number.
Attributes	The attributes of a memory block determine certain characteristics about it (where it can reside, if it can be moved, and so on). Attributes are very important. They will be discussed in detail later in this chapter.
Location	If the memory is to reside at a fixed location in memory, this long-word value determines the address requested.

NewHandle doesn't return a pointer to the location of the requested block of memory. Rather, it returns a handle to that block. The handle will be waiting to be pulled from the stack after this call is made from machine language—which is an important detail to remember.

The handle references the memory-block structure just allocated. Within that structure is the actual address of the memory block. That address is obtained in the following manner:

```
pla                     ;get low-order word of the handle
plx                     ;get high-order word of the handle
sta  0                  ;build a long pointer at location $00
stx  2
sta  TheHandle          ;and save the address for later
stx  TheHandle+2        ;(might be used for disposal)
```

The handle has been pulled from the stack and stored in four bytes from memory locations $00–$03. A copy has also been stored in TheHandle, a four-byte storage area within the program. By putting the value returned by NewHandle at location $00, a long-address pointer is created. This can be referenced indirectly in order to fetch values in the memory block's record.

```
lda  [0]                ;get 16-bit address of the memory block
sta  BlockAddr          ;. . . and save it
ldy  #2                 ;index passed the first word . . .
lda  [0],y              ;. . . then get the bank of the memory block
sta  BlockAddr+2        ;and save it
```

The address contained in the four-byte storage area named BlockAddr is the location of the 256-byte page of memory that was allocated with NewHandle. In fact, due to the location and

attributes of this memory block, it could be used as direct-page space by a tool set.

Since direct pages reside in bank $00 only, the high word of the address need not be retrieved from the memory handle record. The most significant word of the address is assumed to be 0. Example:

```
lda   [0]           ;get the direct page address
sta   DPageAddr     ;. . . and save it
```

DPageAddr would simply be a two-byte storage area in your program.

Using NewHandle in Pascal and C is far easier. In Pascal, the following is used to obtain memory for direct-page space:

```
TheHandle := NewHandle($100, MemID, $C005, Ptr(0));
DPageAddr := LoWord(TheHandle^);
```

These statements are identical in operation to the machine language example listed earlier. The four parameters in the above example that constitute the NewHandle requirements are block size requested ($100), an ID (MemID), attributes ($C005), and the block's address (0).

The following illustrates grabbing a memory handle using C:

```
TheHandle = NewHandle(0x100L, MemID, 0xC005, nil);
DPageAddr = (int) *(TheHandle);
```

These statements are identical to the machine language and Pascal examples.

The Memory-Block Record

The memory-block record is one of those things you really don't need to know about in order to program the Apple IIGS. The structure and manipulation of memory handle records is not part of the regular programmer's repertoire. In fact, the only time you would examine a record is to locate a memory block's true location in memory. And the purpose of having a Memory Manager is to avoid that.

Each block of memory allocated by the Memory Manager has a corresponding record. (Recall that the record is what the handle points to.) The structure of this record consists of six fields that give the memory block's address in memory and provide additional information about the block.

The long-word value (handle) returned by NewHandle is the address in memory where the memory block's record is stored. This record is 20 bytes long, and it contains the following information:

Size	Contents
Long word	Address of the block
Word	Attributes
Word	Owner's User ID
Long word	Size of the block
Long word	Pointer to the next handle record
Long word	Pointer to the previous handle record

The first four fields are copies of the parameters used when the NewHandle call was first made. See the previous section for details.

The last two items require further explanation.

In order to keep track of these handle records, the Memory Manager uses a set of *next* and *previous* record pointers to create a linked list. The first long-word pointer points to the next 20-byte memory handle record, while the second pointer points to the previous record. This allows handle records to reside in any order throughout the computer's memory, yet they can be referenced in order due to their link fields.

Block Attributes

When NewHandle is used to allocate a block of memory, you must decide how that block should be treated by the Memory Manager. For example, should the block be allowed to move around? Does it have to be aligned on a 256-byte page boundary (which speeds up some processes)? Can it reside in special memory banks? You'll have to consider these points, and more, when allocating a new handle. Time to think.

Block attributes are assigned by the programmer before the NewHandle function is called. The attributes parameter is a word value and consists of 16 bits of information:

Bit	Meaning If Set (Made Equal to 1)
0	Block must reside in a particular memory bank
1	Block must reside at a particular address
2	Block must be page-aligned
3	Block can reside in special memory banks
4	Block cannot cross a bank boundary

Bit	Meaning If Set (Made Equal to 1)
5	Reserved
6	Reserved
7	Reserved
8	Purge level (low bit)
9	Purge level (high bit)
10	Not used (0)
11	Not used (0)
12	Not used (0)
13	Not used (0)
14	Block is fixed (cannot move)
15	Block is locked (fixed and unpurgeable)

Each bit position represents a specific attribute describing the memory block to be allocated. Setting a bit asserts that attribute.

Bit 0 — Specifies whether the block should reside in a particular bank of memory in the computer. For example, if your application required a memory block that must reside in bank $05, you would set this bit.

Bit 1 — Specifies that the block must live at a particular address in memory. Memory blocks that reside at fixed addresses should also have bit 14 set (which means they cannot be moved).

Bit 2 — Causes the block of memory to reside on a page boundary. A page is 256, or $100, bytes of RAM. The first page boundary is at location $0000 in a bank. The next page is at location $0100. The next page would be at location $0200, followed by $0300, and so on, all the way up to $FF00, the last page boundary in a bank.

Bit 3 — Determines if a block can reside in the special memory banks $00, $01, and $E0 and in bank $E1. These banks are used by the Mega II (Apple IIe–emulation) mode of the Apple IIGS when an application runs under the 8-bit version of ProDOS. If you create a memory-resident application for the Apple IIGS, such as a desk accessory, it cannot reside in special memory. In native (16-bit) mode, bank $00 is used mainly by DeskTop applications for direct-page space.

Bit 4 — Tells the Memory Manager if the block can cross from one bank to the next in the computer. For example, a $2000-byte block, living at location $03FE00 could cross over into bank $04 if bit 4 was not set.

Bits 5–7 — Reserved and should not be set.

Bits 8 and 9 — Classify the purge level of a memory block. Because there are just two bits, four unique settings can be assigned ($2 \times 2 = 4$):

Value	Meaning
0	The block cannot be purged
1	Very low purge level
2	Moderate purge level
3	Very susceptible to purging

Blocks are purged when the Memory Manager is called to compact memory and clean house. As you can see, blocks with the highest nonzero purge levels are purged first.

Bits 10–13 — No use at this time.

Bit 14 — Fixes a block in memory so it cannot be moved.

Bit 15 — Used to lock a memory block. Locking causes the block to become immovable and unable to be purged, regardless of the settings of bits 8, 9, and 14.

The Memory Manager tool set provides functions for changing the attributes of a block after it has been allocated with NewHandle. They are the following:

Function	Description
HLock	Locks a memory block referenced by its handle
HLockAll	Locks all memory blocks referenced by a User ID
HUnLock	Unlocks a memory block referenced by its handle
HUnLockAll	Unlocks all memory blocks with a certain User ID
SetPurge	Sets the purge level of a block referenced by handle
SetPurgeAll	Sets the purge level for all blocks with the same ID

The summary at the end of this chapter lists the parameters for these functions. Note that *COMPUTE!'s Mastering the Apple IIGS Toolbox* provides parameter descriptions for the entire Apple IIGS Toolbox.

Removing Memory

Memory is removed by eliminating memory handles (the same handles that were obtained by the NewHandle function). When a handle is removed, the Memory Manager is allowed to make available the space that its memory block took up in the computer.

Any handles allocated by your application should be removed as soon as they are no longer needed. This will make the memory

they occupy free for use by other applications. Handles can be removed in a number of ways using the Memory Manager tool set.

The most straightforward method of removing a memory block is to use the DisposeHandle function. Your application pushes the handle's value onto the stack and then DisposeHandle is called. The memory block and its allocated handle are removed from the system instantly.

In machine language:

```
pushlong TheHandle     ;push the handle on the stack
_DisposeHandle         ;and now dispose of it
```

The same example in C or Pascal:

```
DisposeHandle(TheHandle);
```

If you have allocated multiple handles with a single identification value, your application can take a shortcut by using the DisposeAll function. DisposeAll will remove all handles associated with a particular ID number. For example, in C or Pascal:

```
DisposeAll(MemID);
```

Your programs should never call DisposeAll with the master User ID returned by MMStartUp. This would cause the memory space that a program occupies to be freed, which might result in a system crash.

Another approach to freeing a block is to set its purge level to the highest setting (3). This would cause the Memory Manager to dispose of your block the next time it was called to compact memory (CompactMem). Note, however, that the handle remains allocated and will have to be removed eventually.

When a handle is purged, the block allocated to this handle is freed, but the handle is kept alive. The address of the block in the memory block record is set to $0000000 (a long word of 0). This tells the Memory Manager that the handle is valid, but does not have a block allocated to it. This would allow you to reallocate (ReAllocHandle) a memory block at a later time without having to use NewHandle to create a brand new one. It is understood that if purging does not dispose of the handle, your application will still need to do so before quitting.

Chapter Summary

The following Toolbox functions were discussed in this chapter. Also included are a few of the popular Memory Manager functions.

Function: $0202
Name: MMStartUp
Starts the Memory Manager
Push: Result Space (W)
Pull: UserID (W)
Errors: $0207
Comments: One of the first calls made by an application.

Function: $0302
Name: MMShutDown
Shuts down the Memory Manager
Push: UserID (W)
Pull: nothing
Errors: none
Comments: Make this call when your application is finished.

Function: $0902
Name: NewHandle
Makes a block of memory available to your program
Push: Result Space (L); Block Size (L); UserID (W); Attributes (W); Address of Block (L)
Pull: Block's Handle (L)
Errors: $0201, $0204, $0207

Function: $0A02
Name: ReAllocHandle
Reallocates a purged block with new parameters
Push: Block Size (L); UserID (W); Attributes (W); Address of Block (L); Old Block's Handle (L)
Pull: nothing
Errors: $0201, $0203, $0204, $0206, $0207

Function: $0B02
Name: RestoreHandle
Reallocates a purged block using original parameters
Push: Old Block's Handle (L)
Pull: nothing
Errors: $0201, $0203, $0206, $0208
Comments: Uses same parameters of original block (unlike function $0A which allows the parameters to be reset).

Function: $1002
Name: DisposeHandle
Deallocates a block and releases its memory
Push: Block's Handle (L)
Pull: nothing
Errors: $0206
Comments: The block is deleted regardless of its locked status or purge level.

Function: $1102
Name: DisposeAll
Releases all blocks associated with a UserID
Push: UserID (W)
Pull: nothing
Errors: $0207
Comments: Ruthless.

Function: $1202
Name: PurgeHandle
Purges a block of memory
Push: Block's Handle (L)
Pull: nothing
Errors: $0204, $0205, $0206
Comments: The block must be purgeable and unlocked. The block's handle is not deallocated by this call.

Function: $1302
Name: PurgeAll
Purges all blocks associated with a UserID
Push: UserID (W)
Pull: nothing
Errors: $0204, $0205, $0207
Comments: The blocks must all be purgeable and unlocked.

Function: $1B02
Name: FreeMem
Returns memory available for programs
Push: Result Space (L)
Pull: Integer Value (L)
Errors: none
Comments: Returns the total number of bytes in memory, not counting ramdisks or other allocated blocks.

Function: $1C02
Name: MaxBlock
Returns memory available to programs
Push: Result Space (L)
Pull: Integer Value (L)
Errors: none
Comments: Returns the largest free block in memory.

Function: $1D02
Name: TotalMem
Returns total RAM in the System
Push: Result Space (L)
Pull: Integer Value (L)
Errors: none
Comments: Returns all RAM in your Apple IIGS, including the basic 256K, any ramdisks, and so on.

Function: $1F02
Name: CompactMem
Compacts memory
Push: nothing
Pull: nothing
Errors: none
Comments: Performs memory garbage collection, purging purgeable blocks and reorganizing memory. Don't do this during an interrupt.

Function: $2002
Name: HLock
Locks and sets a specific handle to a purge level of 0
Push: Handle (L)
Pull: nothing
Errors: $0206

Function: $2102
Name: HLockAll
Locks and sets all handles associated with a specific UserID to a purge level of 0.
Push: UserID (W)
Pull: nothing
Errors: $0207

Function: $2202
Name: HUnLock
Unlocks a block of memory
Push: Handle (L)
Pull: nothing
Errors: $0206

Function: $2302
Name: HUnLockAll
Unlocks all blocks of memory associated with a specific UserID
Push: UserID (W)
Pull: nothing
Errors: $0207

Function: $2402
Name: SetPurge
Sets the purge level of a given block
Push: New Purge Level (W); Handle (L)
Pull: nothing
Errors: $0206
Comments: Only the lower two bits of the word pushed are significant.

Function: $2502
Name: SetPurgeAll
Sets the purge level for all blocks associated with a given UserID
Push: New Purge Level (W); UserID (W)
Pull: nothing
Errors: $0207

Function: $2B02
Name: BlockMove
Copies a block of memory from one address to another
Push: Source Address (L); Destination Address (L); Length (L)
Pull: nothing
Errors: none

Memory Manager Tool Set Error Codes

$0201 Unable to allocate block
$0202 Illegal operation on an empty handle
$0203 Empty handle expected for this operation
$0204 Illegal operation on a locked or immovable block
$0205 Attempt to purge an unpurgeable block
$0206 Invalid handle given
$0207 Invalid User ID given
$0208 Operation illegal on block-specified attributes

Chapter 8

Pull-Down Menus

In menu-driven programs not too many years ago, the computer's monochrome screen would clear and a long list of menu items, usually numbered, marched down the display:

```
MAIN MENU OPTIONS:

1. GO TO MENU 2
2. GO TO MENU 3, SUB MENU C
3. GO TO MENU 5 AND STAY THERE
4. DO MAIN MENU OPTION 7
5. PRETEND TO GO TO MENU 7 BUT GO TO MENU 6 INSTEAD
6. GIVE ME THE BREAKFAST MENU
7. DO MAIN MENU OPTION 4
8. JUST GET ME THE CHECK

ENTER YOUR SELECTION (1-8):
```

Pressing a number would erase the old menu and would likely unravel yet another screenful of menu items. Sometimes this would go on through several levels, before anything could get done. Mobility in this environment was like jogging blindfolded—with shackled legs.

With a DeskTop environment however, the user can see all the possible menus at once. Their titles are positioned horizontally across the top of the screen. Navigating through these menus requires little instruction. They are intuitive and are becoming commonplace in the computer world. Just about everyone has had exposure to them.

The Two Managers

Programmers who have written interactive software know that when life is made easier for the user, it usually means the opposite for the programmer. Creating user-friendly software requires hard work. While this is generally true for applications in other environments, things couldn't be sweeter for the Apple IIGS programmer. All the credit goes to the Menu and Window Managers.

As its name implies, the Menu Manager is responsible for maintaining the lists of numerous commands and functions a program may contain. It takes care of shuffling menus around, drawing them on the screen, and interacting with the user while selections are made with the mouse or keyboard.

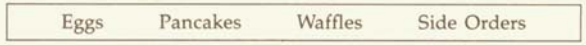
What does the Window Manager have to do with menus? A vital part of the Window Manager is the TaskMaster. The purpose of the TaskMaster is to watch for menu events that occur on the DeskTop and to handle them appropriately. It relieves the programmer of those bothersome details. However, if an application requires custom event handling, the TaskMaster can be bypassed altogether.

Organizing Menu Items

Organization of functions and subroutines is an essential step in creating any new program. The same applies to creating pull-down menu items. For example, a coffee-shop menu is grouped into sections such as Eggs, Pancakes, Waffles, and Side Orders. This makes it easier for the diner to locate a particular item.

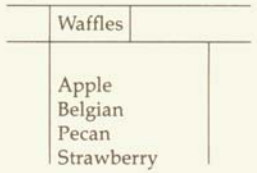
In a DeskTop program, the Main Menu of yesteryear's application is replaced by the System Menu Bar at the top of the screen, as shown in Figure 8-1.

Figure 8-1. Breakfast Menu Bar



Within each menu are menu items. For example, the third menu, Waffles, might include four items, shown in Figure 8-2.

Figure 8-2. Waffle Menu



For the user's sake, items in a pull-down menu should be related to the title of the menu. This falls into the department of Apple's Human Interface Guidelines (see Appendix A). The guidelines were created to help the programmer decide where certain commands should go, how they should be named, and so on.

As an example, commands that open and close files, save changes to disk, create new files, and interact with the printer, are found in the File menu on the System Menu Bar. The command to quit a program is also in the File menu. Practically all DeskTop programs have a File menu so long as a means exists for quitting the application.

Once an application's commands are organized into menus, the programmer must decide which, if any, should have keyboard equivalents. Keyboard equivalents are awarded to commands used most often. Applications relying heavily on keyboard input, such as word processors, ought to provide the user with as many key equivalents as possible. On the other hand, people tend to make

menu selections using the mouse while working with drawing or painting programs, which makes it less important that graphics program menu items have keyboard-equivalent commands.

According to the Human Interface Guidelines, created by Apple's Bruce Tognazzini (lovingly known as "Saint" Tognazzini), some keyboard command characters should be reserved for certain functions in order to maintain consistency from one DeskTop application to the next.

Table 8-1. Command Key Equivalents

Key	Command
C	Copy
O	Open
Q	Quit
S	Save
V	Paste
X	Cut
Z	Undo

The letters listed in Table 8-1 are commonly reserved for the listed functions.

Keyboard equivalents are shown to the right of a menu item, preceded by the Open Apple symbol. On the Macintosh, they are preceded by the clover-leaf (Command key) symbol.

The placement of items on the menu bar is also discussed in the Human Interface Guidelines. The menus are positioned on the menu bar starting with the Apple menu (also called the New Desk Accessory menu) at the left side. Following that comes the File menu. And if the application manipulates text or graphics, usually an Edit menu follows. Consult Appendix A for other reserved menu titles suggested by the guidelines.

Designing a Menu

The Menu Manager works with strings of characters in order to build a menu and create its contents. A list of these strings is passed to the Menu Manager via the NewMenu Toolbox function. In machine language, C, or Pascal, the data for a menu list can be created by defining text-string constants.

These strings must remain in memory for as long as the menu bar is present. Machine language programmers should not reuse the space occupied by these strings, and C programmers should define the strings as global, static text.

A menu list consists of three parts:

- A title
- The menu items
- The end of menu marker

Additionally, each item of a menu, and its title, are tagged by a unique identification number. The ID number of a menu title is useful only to the Menu Manager. The ID number of a menu item is used by the application when the user selects a menu item.

Figure 8-3 shows a sample menu list.

Figure 8-3. Menu List

```
>> Waffle \N3          ← Menu Title
--Apple \N256
--Belgian \N257        ← Menu Items
--Pecan \N258
--Strawberry \N259
>                      ← End of Menu
```

Each line in the list begins with two unique characters. The exception is the last line which requires just one character.

The first line in the list describes the title of the menu. It begins with two letters, numbers, or symbols. Following these characters is the menu title. Incidentally, the title is usually surrounded by one or more spaces to provide padding between the other titles on the menu bar. The backslash character (\ ) signals the end of the title and the beginning of the special characters. Therefore, a backslash cannot be part of the menu's name. The special characters further describe the menu item.

The commercial *at* symbol (@) is used to produce the colored Apple logo used for the Desk Accessory menu. It must be the only character in the title, with no surrounding spaces.

You can also specify the @ sign for any other menus you may have. However, only the Apple logo will appear as long as there is no other text along with it.

The strings that follow the title line make up the list of items in this menu. Each line starts with the same two characters, which can be any characters, except the two that begin the menu's title line. The name of the menu item follows, and then finally, a backslash signals the start of the special characters.

A special menu item, called a dividing line, can be placed into the menu by using a single hyphen as a menu item. Its purpose is to divide members in an item list. Dividing line items should be dimmed (see below) so that they cannot be chosen as a legal menu item.

The very last line of the menu list consists of a single character. This character must be different from the characters that start the menu item lines. However, it can be the same character that begins the title line, as shown in Figure 8-3 above.

Each line in the list, except for the very last, ends with a carriage return ($0D) or a null character ($00). This tells the Menu Manager that the end of the line has been encountered and it is allowed to proceed to the next line.

The special characters that follow the backslash have the following functions:

Character	Does This
*	Defines the command key equivalents
B	Draws the menu item's text in boldface
C	Places a character in front of the item name
D	Dims and disables the menu or menu item
H	Indicates that a two-byte hexadecimal ID number follows
I	Draws the menu item's text in italic style
N	Indicates that a decimal ASCII ID number follows
U	Underlines the menu item's text
V	Places a dividing line between this item and the next
X	Activates color replace for highlighting

These characters can be upper- or lowercase. Two characters must follow the * for keyboard equivalents. They are used to specify the case sensitivity of the command letter. For example:

*Bb Both *B* and *b* are accepted
*BB Only uppercase *B* is accepted
*bb Only lowercase *b* is accepted

Similarly, using *?/ would allow the user to press the Open Apple key and either the slash or question mark (shift-slash), for example, to execute a Help command.

The key equivalent will be displayed on the menu after an Apple symbol. Also, only the first letter after the * is displayed on the menu, though both keys will work.

The *B*, *I*, and *U* special characters, which stand for boldface, italic, and underline type styles, respectively, are used to enhance the display of text items. They may or may not be available for use depending on the system font.

The letter *C* places a character before the item's text. Typically, this is used to mark the item with a special character, such as the following:

Character	ASCII Value
Check mark	18 ($12)
Diamond	19 ($13)
Open Apple	17 ($11)
Solid Apple	20 ($14)

For example, to place a check mark before a menu item, the following string would be defined in a machine language source code file:

```
dc    c'--Checked Item\C',i1'18',c'N256V',i1'0'
```

More information on creating the menu strings from assembly language is covered in the next section.

D is used to dim and disable an item. The item appears in a dimmed state and cannot be chosen. If the menu itself is disabled, every item in that menu will be dimmed and disabled.

H and *N* allow a menu or item to be assigned an ID number. When *H* is used, it's followed by a two-byte hexadecimal value in low-byte/high-byte order. If *N* is used, it is followed by a string of decimal characters. Not every menu item requires an ID, and only certain IDs are used as shown in the following chart:

Menu IDs	Description
0	Used internally
1–65534	Used by an application's menus
65535	Used internally

Item IDs	Description
0	Used internally
1–249	Used by desk accessory items
250	Undo
251	Cut
252	Copy
253	Paste
254	Clear
255	Close
256–65534	Used by an application's menu items
65535	Used internally

Identification numbers don't have to be sequential or defined in any order. They have to be unique, but only if they are enabled. Machine language programmers will want to assign menu item IDs starting with 256 and work upwards, not skipping over any values. (The reason for this is discussed later.)

V is used to draw a dividing line between two items, across the entire width of the menu. It does not take up a line in the list of items, as the hyphen character does. (See above.)

X uses color-replace mode that affects the way a menu is highlighted when it is chosen. When a colored menu is selected with color replace activated, the colors will remain the same. For example, in the Apple menu, the X option should be specified. If not, the Apple character will appear gray on a black background, rather than colored on a black background. For ordinary menus, the X option need not be specified.

Creating Menu Strings

When using the *APW* assembler, string constants are defined using the DC (Define Constant) directive:

```
Menu3   dc    c'>> Waffle \N3',i1'0'
        dc    c'--Apple\N256*Aa',i1'0'
        dc    c'--Belgian\N257*Bb',i1'0'
        dc    c'--Pecan\N258*Pp',i1'0'
        dc    c'--Strawberry\N259*Ss',i1'0'
        dc    c'>'
```

Each line ends with a single zero byte. ID numbers are assigned using the special character *N*. However, the *H* character could have been used:

```
dc    c'--Apple\H',i1'256',c'*Aa',i1'0'
```


Why use *H* when *N* will do? Because it saves a byte and is easier for the Menu Manager to parse. Unfortunately, it makes the source code look messy.

Things are done a bit differently using the C language. The Waffle menu could be defined using static text strings as follows:

```
char *Menu3[] = { ">> Waffle \\N3",
                  "--Apple\\N256*Aa",
                  "--Belgian\\N257*Bb",
                  "--Pecan\\N258*Pp",
                  "--Strawberry\\N259*Ss",
                  ">" };
```

Since the C compiler uses the backslash character for various purposes, it must be entered twice in a row in order to insert one backslash into a string of text. And since C strings by definition end in a null byte, the end-of-line terminator will be inserted automatically at compile time.

> An alternative method to define text strings is to use C's in-line assembly feature to define the strings with 65816 instructions. Or you could write an external program in machine language that is linked with the C code later on.

For programmers using *TML Pascal*, menu strings must be defined as global string types. They are built at runtime using the CONCAT function:

```
Menu3 := CONCAT('>> Waffle \N3D\0',
                '--Apple\N256*Aa\0',
                '--Belgian\N257*Bb\0',
                '--Pecan\N258*Pp\0',
                '--Strawberry\N259*Ss\0',
                '>');
```

Notice how each line is terminated by the null, \0, escape sequence. Unlike C, Pascal strings are not automatically terminated by nulls, and, therefore, the programmer must provide them.

Installing a Menu

Before any Menu Manager functions can be called, the Menu Manager must be started. The Menu Manager, like a few other tool sets, also requires its own direct page. If you're not sure how to start up

a tool set, or how to get direct page space, see Chapter 4 in this book, "About the Toolbox," and Chapter 7, "Memory Management."

Placing a menu into the System Menu Bar is a two-step process. First, all menu strings must be passed to the NewMenu function. NewMenu uses them to create an internal menu record. Once completed, NewMenu returns a handle to the menu record.

The second step involves inserting the menu record into the System Menu Bar by using the InsertMenu function. This is done by passing the menu handle, returned by NewMenu, to InsertMenu. It then places the menu at the desired position.

To accomplish this process from a machine language program, the following can be used:

```
    pha                     ;long-word result space
    pha                     ;long-word result space
    pushlong    #Menu3      ;point to Menu 3's strings
    _NewMenu                ;create the menu record . . .
;                           ;. . . whose handle is now on the stack
    pushword    #0          ;insert it before all other menus
    _InsertMenu
```

InsertMenu's two input parameters are the handle of the menu record and a position value that determines where on the menu bar the menu title will be inserted. If the position is 0, the menu will be the leftmost menu. Note how the menu record handle is kept on the stack for the call to _InsertMenu.

> The position argument, if 0, will insert the menu at the leftmost side of the menu, pushing any existing menus to the right. But if the position value is a Menu ID number, it instructs the Menu Manager to insert the menu after the menu referenced by that ID.

Creating and inserting a menu in C or Pascal is practically effortless when compared to machine language.

With C:

```
InsertMenu(NewMenu(Menu3[0]), 0);
```

With *TML Pascal*:

```
InsertMenu(NewMenu(@Menu3[1]), 0);
```


The two tasks can be taken care of with just one statement by embedding the NewMenu function within the InsertMenu function. This is a very common programming technique.

TML Pascal requires the *at* symbol in front of the Menu3 variable in order to reference its address in memory. Also, the data in Pascal strings starts with element 1, because element 0 is a count byte.

Drawing the Menu Bar

Even though a menu has been inserted into the menu bar, it does not appear on the screen. To cause the menu to appear, call the FixMenuBar function.

In machine language:

```
pha                 ;word result space
_FixMenuBar
pla                 ;returns the bar's height in pixels
sta   Height        ;(optional—you don't need to save it)
```

Or in C:

```
Height = FixMenuBar( );
```

The Height assignment is optional. A simple FixMenuBar( ) alone can be used.

In Pascal,

```
Height := FixMenuBar;
```

does the same, but the variable assignment (Height) is required.

FixMenuBar calculates the height of the System Menu Bar and vertical placement of menu items. This depends on the type of system font in use. If this function is not called, all the menu items will appear on top of each other, and the program will look peculiar.

Finally, when the menu records have been created, inserted, and fixed, the System Menu Bar can be displayed on the screen using the DrawMenuBar function.

With C, use

```
DrawMenuBar( );
```

Or with Pascal, use

```
DrawMenuBar;
```

To perform the equivalent with a machine language macro call, use

```
_DrawMenuBar
```

This function displays the titles of all your pull-down menus on the menu bar.

Using the TaskMaster

The easiest way to manage your menus is to let the TaskMaster do all the work. The TaskMaster takes over whenever the user does something to affect the menu-bar area. As an example, if the user clicks the mouse over a menu title, the TaskMaster calls the functions in the Menu Manager that draws the menu on the screen.

If the user begins to drag the mouse pointer down through a menu, TaskMaster calls the appropriate Menu Manager functions that allow the user to make a selection. TaskMaster also recognizes keyboard equivalents of menu items and treats them as if menu selections were made with the mouse.

Before TaskMaster is used, your application must provide an event record where TaskMaster places information. The event record consists of seven fields, structured in this manner:

```
EventRec    anop            ;Event Record used by TaskMaster
What        ds    2         ;word
Message     ds    4         ;long word
When        ds    4         ;long word
Where       ds    4         ;long word
Modifiers   ds    2         ;word
TaskData    ds    4         ;long word
TaskMask    dc    14'$1fff' ;long word
```

The address of this record is passed to TaskMaster as one of its arguments.

Calling TaskMaster with machine language:

```
pha                         ;Result Space
pushword    #$FFFF          ;Event Mask (screen all event types)
pushlong    #EventRec       ;Point to Event Record
_TaskMaster
pla                         ;Get Event code
```

Calling TaskMaster with C:

```
Event = TaskMaster(0xffff, &EventRec);
```


After calling TaskMaster, a code is returned to your application. If its value is not 0, an event is pending. The application can continue to call TaskMaster until a nonzero code is reported. This is demonstrated by the following loop in Pascal:

```
REPEAT
    Event := TaskMaster($ffff, EventRec);
UNTIL Event <> 0;
```

When the user eventually makes a menu selection, TaskMaster returns control to your application, informing it that an event has occurred. If a menu item (other than a desk accessory) has been selected, TaskMaster returns an extended event code of $0011 (17 decimal). This is usually equated to the constant called wInMenuBar, as shown in this Pascal statement:

```
IF Event = wInMenuBar THEN DoMenu;
```

The lowercase *w* in *wInMenuBar* identifies it as a Window Manager constant. In *TML Pascal*, this constant is already defined as 17 for your application's use.

When menu event $11 has occurred, the menu number and menu item ID of the item selected can be obtained from the TaskData field in the Event Record.

Table 8-2 shows the contents of the TaskData field and how each word is referenced from machine language, C, and Pascal. Some real-life examples follow.

Table 8-2. TaskData Field

Language	Low-Order Word Menu Item ID	High-Order Word Menu Number
Machine language	TaskData	TaskData+2
C	EventRec.wmTaskData	EventRec.wmTaskData<<16
Pascal	LoWord(EventRec.TaskData)	HiWord(EventRec.TaskData)

To retrieve the menu selection in machine language:

```
lda   TaskData+2
sta   MenuSelected
```

To retrieve the menu selection in Pascal:

```
MenuSelected := HiWord(Eventrec.TaskData);
```

To retrieve the menu selection in C:

```
MenuSelected = EventRec.wmTaskData<<16;
```

The contents of the high- and low-order words of the TaskData field break down as follows:

Low-order word	The low-order word of TaskData holds the Menu Item ID of the selected item. For example, if the Pecan item in the Waffle menu were selected, the low-order word of TaskData would contain 258 (see Figure 8-3).
High-order word	The high-order word of TaskData contains the Menu Number. Again, if the Pecan item were selected, the high-order word of TaskData would contain 3.

Dispatching Item Handlers

Once the Item ID is known, as obtained from TaskData, the appropriate action can be taken by the program. Suppose that when the user has selected the Pecan item from the Waffle menu, you want the program to execute the PecanWaffle routine. C and Pascal programmers can use the SWITCH and CASE statements to do this.

The CASE statement example in Pascal:

```
CASE LoWord(EventRec.TaskData) OF
     256 :   AppleWaffle;
     257 :   BelgianWaffle;
     258 :   PecanWaffle;
     259 :   StrawberryWaffle;
END;
```

PecanWaffle is a previously declared procedure. It fulfills the user's request, perhaps by bringing up a dialog box asking if whipped cream is desired on the pecan waffle.

Dispatching the corresponding routine in machine language is done in one of two ways. The brute force method is to compare the item ID with an immediate value. If the two numbers match, a branch is made to the appropriate subroutine. Otherwise, the program continues to compare the ID with other immediate values.

A more elegant method, common among experienced machine language programmers, is to use the lower eight bits of the Item ID as an index into a table of pointers that point to the corresponding routines. It sounds more complex than it is. For example:

```
lda   TaskData      ;Get TaskData Item ID number
and   #$00FF        ;Discard upper 8 bits
asl   A             ;Double the value
tax                 ;Transfer to X as an index
jsr   (MTable,X)    ;Dispatch the proper menu item handler
```


If Pecan (item 258, the third menu item) were selected, the AND #$00FF instruction results in $02. This is multiplied by 2 using the ASL instruction which produces $04. That value is transferred to the X register to be used as an index.

> Using an index into a table of subroutines is one example of how useful numbering your menu items sequentially can be. The drawback is that during the cycle of development, you'll often move, reassign, insert, or change your menus as the program evolves. When this happens, renumbering menu items to keep them sequential can turn into a headache.

```
MTable  dc    i'AppleWaffle'        ;Item 256 (X = 0)
        dc    i'BelgianWaffle'      ;Item 257 (X = 2)
        dc    i'PecanWaffle'        ;Item 258 (X = 4)
        dc    i'StrawberryWaffle'   ;Item 259 (X = 6)
```

The JSR (MTable,X) instruction is known as an indexed, indirect jump to subroutine. The processor jumps to the two-byte address in MTable plus the value in the X register. Since X is 4, the subroutine PecanWaffle in the above table would be executed.

Unhighlighting the Menu's Title

During the dispatch of a menu item, the menu's title remains highlighted on the menu bar. This reminds the user that a menu item is being handled. When the service routine is finished, the menu's title should be inversed (unhighlighted). This is done with the HiliteMenu function.

In machine language:

```
pushword     #FALSE          ;Unhilite the menu title now
pushword     TaskData+2      ;Push TaskData Menu number
_HiliteMenu
```

With *TML Pascal*:

```
HiliteMenu(FALSE, HiWord(EventRec.TaskData));
```

And in C:

```
HiliteMenu(FALSE, EventRec.wmTaskData>>16);
```

The integer constant, FALSE, is defined as 0.

Keep in mind that unhighlighting a menu item is not automatic. You must do it manually after each menu item's function is completed.

Changing Menu Items

Not only are menu items an excellent way to initiate subroutines in an application, but they can also be used to toggle certain states (flags) in your program.

In a drawing program, for example, a check mark may appear next to the Ruler Guides item in the Tools menu. This would indicate that the rulers are in use. Should the user wish not to have rulers while painting (perhaps the artist is an impressionist), the Ruler Guides item could be selected from the Tools menu, which toggles the rulers off; the check mark would then disappear. But that doesn't happen by magic.

Assume that Ruler Guides has a Menu Item ID of 268. To place a check mark to the left of its name in the menu, the CheckMItem function is used:

```
pea   TRUE        ;TRUE: yes, check the item
pea   268         ;Item 268 (Ruler Guides)
ldx   #$320F      ;CheckMItem
jsl   $E10000
```

In C or Pascal:

```
CheckMItem(TRUE, 268);
```

Conversely, to remove a check mark or to make sure that there isn't one there, the same code can be used but with a FALSE value pushed to the stack instead of TRUE.

If your program has many menu items with check marks, it's best to create one procedure responsible for updating the check-mark state of all the items. An example in C:

```
UpdateCheckMarks( )
{
     CheckMItem(Rulers, 268);
     CheckMItem(BigBits, 270);
     CheckMItem(Clamps, 271);
     CheckMItem(WindowLocks, 273);
     CheckMItem(ColorMode, 281);
}
```


The integer (or Boolean) variables Rulers, BigBits, Clamps, WindowLocks, and ColorMode contain values representing true or false for the states of those items. If, in this drawing program, Rulers are turned on, but in order to use them Clamps and WindowLocks must be turned off, this C function would handle the correct toggling of Rulers:

```
ToggleRulers( )
{
     Rulers = !Rulers;                    /* Logical NOT toggle */
     if (Rulers)
          Clamps = WindowLocks = FALSE;
     UpdateCheckMarks( );
}
```

This is how the routine works:

- Toggle the current Rulers state to its opposite.
- If Rulers are now turned on (true), then make sure that Clamps and WindowLocks are turned off (false).
- Finally, update all the check marks according to the new states.

Another example of this technique, though not exactly similar to placing and removing the check mark, is the dimming of menu items, disabling them so that they cannot be selected. To disable a menu item, the DisableMItem function is used.

In machine language:

```
pushword  #256
_DisableMItem
```

In C or Pascal:

```
DisableMItem(256);
```

DisableMItem requires a menu item ID number as its argument. After the call is made, that menu item will be dimmed and not available for selection. To enable the item once again, the EnableMItem is used in a similar fashion:

In machine language:

```
pushword  #256
_EnableMItem
```

In C or Pascal:

```
EnableMItem(256);
```

Disabled menu items show up in a dimmed font in the pull-down menu, and the user of your program will not be able to select that item until it is enabled again.

Setting Menu Flags

Even though the Menu Manager has tools dedicated to one particular task, the SetMenuFlag function can perform the duties of three functions in one. SetMenuFlag works on an entire menu and affects all of its items. The following examples show what a typical call looks like.

In machine language:

```
PushWord    #MenuFlag    ;New menu flag value
PushWord    #MenuID      ;Menu ID number
_SetMenuFlag
```

In C and Pascal:

```
SetMenuFlag(MenuFlag, MenuID);
```

The values and attributes for the MenuFlag argument are expressed in Table 8-3. For example, using SetMenuFlag with $FFDF to invoke color-replace mode is the same as putting the special letter X in that menu's definition string.

Table 8-3. Values and Attributes of the MenuFlag Argument

MenuFlag	Description	Action
$FF7F	Enable	Menu becomes undimmed and its items selectable.
$0080	Disable	Menu becomes dimmed and its items not available.
$FFDF	Color Replace	Highlighting uses the color-replace method.
$0020	XOR Highlight	Highlighting uses the color XOR method.
$FFEF	Standard	Defines the menu as a standard type.
$0010	Custom	Defines the menu as a custom type.

Setting Item Flags

While SetMenuFlag (discussed in the previous section) reigns over entire menus, the SetMItemFlag function allows the attributes of a single menu item to be modified.

Table 8-4 provides a reference to the values that may be placed in the ItemFlag argument and their results.

Table 8-4. Values and Attributes of the ItemFlag Argument

ItemFlag	Description	Action
$FF7F	Enable	The item is enabled, selectable, and not dimmed.
$0080	Disable	The item is dimmed and disabled.
$FFDF	Color Replace	Highlighting uses the color-replace method.
$0020	XOR Highlight	Highlighting uses the color XOR method.
$0040	Underline	The item is drawn with an underline.
$FFBF	No Underline	The item is not underlined.

This is how a machine language routine that places a value in an Item Flag would look:

```
PushWord    #ItemFlag    ;New item flag value
PushWord    #ItemID      ;Item ID number
_SetMenuFlag
```

In C and Pascal:

```
SetMenuFlag(ItemFlag, ItemID);
```

Of course, the EnableMItem and DisableMItem functions should be used for enabling and disabling menu items just to keep your code looking clean and logical.

Menu Miscellany

The rest of the chapter deals with some of the minor details of working with the Menu Manager. Everything from changing the blink rate of a selected menu item to removing an entire menu is discussed in this section. This is where the fun starts.

Changing the Text Style

Menu items can appear in the standard text face or in special styles set by using the SetMItemStyle function. The normal system font can be displayed only in a bold style. However, the Toolbox has provisions for italic, underline, outline, and shadow styles when used with compatible fonts.

This brief table describes the effect of entering various values in the Style Word:

Style Bits	Style
0	Bold
1	Italic
2	Underline

Style Bits	Style
3	Outline
4	Shadow
5–15	Reserved

If a bit is set in the Style Word, it asserts that attribute. The following examples will set a bold style on the text of a menu item.

In machine language:

```
PushWord    #1      ;Bold
PushWord    #262    ;Item ID
_SetMItemStyle
```

In C and Pascal:

```
SetMItemStyle(1, 262);
```

To modify only one style bit without changing the others, use the GetMItemStyle function to return the current style, manipulate the appropriate bits, and then update the item with SetMItemStyle. This C language example sets a bold style to item #262 without changing any of its other style attributes:

```
Word Style;                     /* Style is an unsigned integer */

Style = GetMItemStyle(262);     /* Get the current style */
Style = Style | 1;              /* Logically OR with 1 */
SetMItemStyle(Style, 262);      /* Set the new style */
```

Or, the most compact form could be used:

```
SetMItemStyle(GetMItemStyle(262) | 1, 262);
```

Renaming a Menu Item

It's common to change the name of a menu item. In most cases, renaming an item draws a close relationship to using a check mark to show a certain state. For example, say you've written a communications program in which one of the items on a menu is Text Editor. By choosing this item, a user of your application is taken out of terminal mode and is placed into an editor mode. This would be an opportune time to rename that menu item, since Text Editor is no longer a valid choice: The user is already in it. Instead, that item could be renamed to Terminal Mode. By selecting this, the user could leave the editor and return to the terminal mode.

Changing the name of a menu item is quick and easy, as shown in these examples.

In machine language:

```
            PushLong    #NewName    ;Point to the new title
            PushWord    #262        ;Specify the item ID
            _SetMItem               ;Change the item's name
            rts
NewName     dc      c'--Terminal Mode',11'0'
```

In Pascal:

```
PROCEDURE NameMItem;
VAR
    NewName : String;
BEGIN
    NewName := '--Terminal Mode\0';
    SetMItem(@NewName[1], 262);
END;
```

In C:

```
SetMItem("--Terminal Mode", 262);
```

The SetMItem function requires two arguments:

- The address of a menu item string
- An integer that represents the ID of the item to rename

The string containing the new name is formatted just like a menu item: It begins with two starting characters (used only by the Menu Manager), and it ends with a null character.

Recall that strings in C always end with a null character. Therefore, there is no need to explicitly add one to the initialization string in the C example above.

SetMItem changes only the name of the item. All previous attributes—such as style, enabled or disabled states, and so on—are preserved. Even if the new item string contains a backslash (\ ) followed by special characters, only the name will be replaced. You can change attributes by using other Menu Manager functions discussed throughout this chapter.

Pascal users will undoubtedly want to use SetMItem's cousin, SetMItemName which is similar in syntax. The difference is that SetMItemName accepts a pointer to a Pascal string. Remember that

strings in Pascal always start with a count byte. Here is an alternate Pascal example using SetMItemName:

```
SetMItemName('Terminal Mode', 262);
```

> SetMItem is used to change the Save menu item in most Apple IIGS programs. After a file is opened, the Save item reads Save DOCUMENT, where DOCUMENT is the name of the file the user has opened. Simple string concatenation functions can be used in conjunction with SetMItem to accomplish this.

Although you've renamed the menu item, you're not done just yet.

When an item is renamed, the menu in which the item resides must adjust itself to the new width of the item, especially if it is longer than any of the others. This is done by using the CalcMenuSize function as demonstrated below.

In machine language:

```
PushWord    #0    ;New Width (0 = auto adjust)
PushWord    #0    ;New Height (0 = auto adjust)
PushWord    #2    ;The Menu's ID (not Item ID!)
_CalcMenuSize
```

C and Pascal:

```
CalcMenuSize(0, 0, 2);
```

CalcMenuSize, when used with nonzero arguments, can be used to set a menu's explicit height and width in pixels. If 0's are used, the Menu Manager will scan through the menu strings and automatically calculate the size of the menu, with room for checkmarks and Apple key equivalents. CalcMenuSize requires the ID of a menu as its third argument.

If the menu width is not resized, long menu item names will bleed right off the edge of the menu and into the DeskTop, which looks messy.

Renaming a Menu

It is far less common to change the title of a pull-down menu, but the Menu Manager will let you do it. The procedure is similar to changing a menu item's name.

Study this machine language routine:

```
            PushLong   #NewName     ;Address of title
            PushWord   #2           ;Menu ID number
            _SetMenuTitle           ;Change the title
            _DrawMenuBar            ;show the change
            rts
NewName     str    ' Modem '   ;Pascal-style string
```

The same routine in Pascal:

```
SetMenuTitle(' Modem ', 2);
DrawMenuBar;
```

The same routine in C:

```
SetMenuTitle("\p Modem ", 2);
DrawMenuBar( );
```

SetMenuTitle requires two arguments:

- The address of a Pascal string
- A Menu ID number

Since a Pascal string is needed, the only language that doesn't have to do anything unusual with the string is, of course, Pascal. The machine language example uses the Str macro, while the C example uses the \p string escape in order to put a count byte before the new menu title string.

After the title is changed, use DrawMenuBar to show off your handiwork.

Now You See It . . .

Another rarely used feature of the Menu Manager is the ability to insert both menus and menu items into an existing menu structure. This is accomplished with the InsertMenu and InsertMItem functions, respectively. InsertMenu was discussed in detail earlier in this chapter.

To insert a menu item, InsertMItem is used in the following machine language example:

```
            PushLong   #NewItem     ;address of item string
            PushWord   #$FFFF       ;make it the last item
            PushWord   #2           ;ID of the Menu to use
            _InsertMItem            ;insert it
            rts
NewItem     dc     c'--New Item\N281D',i1'0'
```

In Pascal:

```
PROCEDURE InsertNewItem;
VAR
    NewItem : String;
BEGIN
    NewItem := '--New Item\N281D\0';
    InsertMItem(@NewItem[1], $ffff, 2);
END;
```

In C:

```
InsertMItem("--New Item\N281D", 0xffff, 2);
```

InsertMItem's three arguments are

- The address to a complete menu item string
- The position where the item should be inserted
- The ID number of the menu to use

Values for the position (second) argument are

Position	Description
$0000	Insert into the menu before all other items
$FFFF	Insert into the menu after all other items
ItemID	Insert after the specified Menu Item ID

As described earlier, CalcMenuSize should be called after inserting a new menu item.

. . . Now You Don't

If the Toolbox allows you to insert menus and menu items, there must also be a way to delete them. DeleteMenu and DeleteMItem are practically identical in syntax. They both require a single input parameter:

- A menu ID number for DeleteMenu
- An item ID for DeleteMItem

To delete an entire menu in machine language, use

```
PushWord  #MenuID    ;the ID of the menu to delete
_DeleteMenu          ;it's gone!
```

Using C and Pascal:

```
DeleteMenu(MenuID);
```


To delete a menu item in machine language use

```
PushWord  #ItemID    ;the ID of the item to delete
_DeleteMItem         ;poof!
```

In C and Pascal:

```
DeleteMItem(ItemID);
```

Change Blink Rate

After a menu item is selected, the item winks at you a few times before your choice is acted upon. The number of times the item blinks is determined by the blink rate. Usually, this value is set to 3 upon starting the Menu Manager. But you can spring the following routine on some unsuspecting user.

In machine language:

```
PushWord  #50      ;blink 50 times!
_SetMItemBlink
```

In C and Pascal:

```
SetMItemBlink(50);
```

When SetMItemBlink is used to change the blink rate to 50, a selected menu item will flash on and off 50 times before the item is handled.

Menu Bar Colors

If you're enthralled by the myriad of colors your Apple IIGS can produce, you'll be happy to know that even the menu bar can show its true colors. The text, background, and outlining can be set to any of 16 different colors in 320 mode, and 4 colors in 640 mode. Even though the colors can be changed in 640 mode, it's hardly worth the trouble because so few colors are available. But in 320 mode, the effects can be quite interesting.

How about a blue background, yellow text, and red outlines? Use the MODEL program from Chapter 6 and insert the following code just before the DrawMenuBar function is called.

In machine language:

```
PushWord  #$49     ;Background and text colors
PushWord  #$94     ;Background & text for color replace
PushWord  #$70     ;Outline color
_SetBarColors
```

In C:

```
SetBarColors(0x49, 0x94, 0x70);
```

In Pascal:

```
SetBarColors($49, $94, $70);
```

> To get blue, yellow, and red menu bar colors, the QuickDraw tool set will have to be started up for 320 mode. To do this, use a MasterSCB (screen mode) value of $00. Also, make sure to specify a maximum X clamp of 320 pixels when starting the Event Manager.

SetBarColors uses three input values:

	Colors per Mode		
Value Name	**320**	**640**	**Description**
NewBarColor	16	4	Background (bits 4–7), text (bits 0–3)
NewInvertColor	16	4	Color-replace values for background/text bits
NewOutColor	16	4	Outline color (bits 4–7)

All unused bits are 0, except for bit 15, the most significant bit. This bit is used to cancel the effects of a value. In other words, your program could establish a new outline color, but leave the text and background colors as they were by setting bit 15 on the NewBarColor and NewInvertColor arguments.

When the modified MODEL program runs with new menu colors, the menu bar will be dark blue with yellow text. The outline around the menus, dividing lines, and underlines will be red. But selected menus and items will appear in light blue with orange text.

The Apple menu will retain its colorful logo, but on a yellow background. Why? Recall that the Apple menu uses the special character X in its menu string. This denotes a color-replace mode when selected. All other menus and their items use an XOR (eXclusive OR) method of highlighting when selected.

It's clear to see why this occurs by examining Table 8-5. The color number for dark blue is 4. When XORed with its complement (EOR #$FF), the result is 11, which corresponds to light blue. Likewise, yellow (9) XORed with its complement results in orange (6).

Table 8-5. Standard Colors in 320 Mode

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

The second argument, NewInvertColor, is applicable only to color-replace items. So in order to cause selected items to appear in blue text with a yellow background, the opposite of their backgrounds when not selected, the special X character would have to be placed in each item's menu string.

With a little creativity, you could create a menu where only selected items would show up in a color, indicating a warning or other message to the user, based on the color.

There are accepted guidelines governing the use of color in programs. See Appendix A, "Human Interface Guidelines," for more details.

Chapter Summary

The following tool set functions were referenced in this chapter:

Function: $010F
Name: MenuBootInit
Initializes the Menu Manager
Push: nothing
Pull: nothing
Errors: none
Comments: Do not make this call.

Function: $020F
Name: MenuStartUp
Starts the Menu Manager
Push: User ID (W); Direct Page (W)
Pull: nothing
Errors: none

Function: $030F
Name: MenuShutDown
Shuts down the Menu Manager
Push: nothing
Pull: nothing
Errors: none
Comments: Make this call when your application is finished.

Function: $0D0F
Name: InsertMenu
Inserts a menu into the menu bar
Push: Menu Handle (L); Insert After (W)
Pull: nothing
Errors: none
Comments: If Insert After is 0, the menu becomes the first in the menu bar.

Function: $0E0F
Name: DeleteMenu
Removes a menu from the menu list
Push: Menu Number (W)
Pull: nothing
Errors: none
Comments: Use DrawMenuBar to update the screen. The menu is not fully disposed, just deleted from the list.

Function: $0F0F
Name: InsertMItem
Inserts a menu item into a menu
Push: Item (L); Insert After (W); Menu Number (W)
Pull: nothing
Errors: none
Comments: If Insert After is 0, the item becomes the first in the menu.

Function: $100F
Name: DeleteMItem
Removes an item from a menu
Push: Item (W)
Pull: nothing
Errors: none
Comments: Use CalcMenuSize after making this call.

Function: $130F
Name: FixMenuBar
Standardizes the menu bar's sizes and returns its height
Push: Result Space (W)
Pull: Height (W)
Errors: none
Comments: The returned height is in pixels and is usually 13.

Function: $170F
Name: SetBarColors
Specifies the colors of the menu bar
Push: Normal Color (W); Selected Color (W); Outline Color (W)
Pull: nothing
Errors: none

Function: $1C0F
Name: CalcMenuSize
Calculates the new dimensions of a menu
Push: Width (W); Height (W); Menu Number (W)
Pull: nothing
Errors: none

Function: $1F0F
Name: SetMenuFlag
Specifies the attributes of a menu
Push: Attributes (W); Menu Number (W)
Pull: nothing
Errors: none
Comments: Attributes are: $FF7F = Enable, $0080 = Disable; $FFDF = Color Replace; $0020 = XOR Highlight; $FFEF = Standard; $0010 = Custom.

Function: $210F
Name: SetMenuTitle
Selects the title for a menu
Push: Title (L); Menu Number (W)
Pull: nothing
Errors: none

Function: $240F
Name: SetMItem
Selects the name for an item
Push: Name (L); Item Number (W)
Pull: nothing
Errors: none

Function: $260F
Name: SetMItemFlag
Sets the attributes of a menu item such as being underlined, enabled, and so on
Push: Attributes (W); Item Number (W)
Pull: nothing
Errors: none
Comments: Attributes are: $0040 = Underline, $FFBF = No Underline, $0020 = XOR Highlight; $FFDF = Redraw Highlight; $FF7F = Enable; $0080 = Disable.

Function: $280F
Name: SetMItemBlink
Sets the blink rate for selected items
Push: Blink Count (W)
Pull: nothing
Errors: none

Function: $2A0F
Name: DrawMenuBar
Draws the menu bar and its titles
Push: nothing
Pull: nothing
Errors: none

Function: $2C0F
Name: HiliteMenu
Determines if a menu title is highlighted
Push: Hilite Flag (W); Menu Number (W)
Pull: nothing
Errors: none
Comments: If Hilite Flag is nonzero, the title is highlighted; otherwise, it's unhighlighted.

Function: $2D0F
Name: NewMenu
Creates a new menu
Push: Result Space (L); Menu Structure (L)
Pull: nothing
Errors: none
Comments: This creates the menu internally and does not display or insert it into a menu bar.

Function: $300F
Name: EnableMItem
Enables a disabled menu item
Push: Item (W)
Pull: nothing
Errors: none

Function: $310F
Name: DisableMItem
Disables a menu item, making it dimmed
Push: Item (W)
Pull: nothing
Errors: none
Comments: The item will no longer be available for selection.

Function: $320F
Name: CheckMItem
Manages check marks for a menu item
Push: Check Flag (W); Item (W)
Pull: nothing
Errors: none
Comments: An item will be marked with a check if Check Flag is true; a check will be removed if false.

Function: $330F
Name: SetMItemMark
Sets the marking character (or none) for an item
Push: Mark Character (W); Item Number (W)
Pull: nothing
Errors: none
Comments: Use 0 for no mark.

Function: $350F
Name: SetMItemStyle
Sets the text style of a menu item
Push: Text Style (W); Item Number (W)
Pull: nothing
Errors: none

Function: $3A0F
Name: SetMItemName
Selects a name for a menu item
Push: Name (L); Item Number (W)
Pull: nothing
Errors: none
Comments: Name is a Pascal-type string.

Chapter 9

Windows

Next to pull-down menus, windows are the most important part of the desktop environment. A window is a region of the screen inside of which information and/or graphics can be displayed. The Toolbox's Window Manager provides the functions for creating a window and placing various objects into it.

This chapter covers programming, creating, and using Apple IIGS windows. Unfortunately, not everything about windows can be covered here. It would require a gargantuan book to demonstrate everything the Window Manager can do. And, of course, it would take a trilogy of these books to present program examples in three languages as is being done here. You'll find enough routines and examples in this chapter to start experimenting. If you practice, you'll be writing useful window applications of your own in short order.

A Frame to Build On

Windows are controlled by a joint cooperation between the Window Manager and the Control Manager. The Window Manager is what actually manages the windows (as you may have guessed). It also takes care of certain functions that occur behind the scenes. The Control Manager is responsible for all the controls on a window. Controls are the buttons, boxes, scroll bars, and other items that allow you to manipulate a window. Therefore, both managers share the responsibility of windows on the desktop.

To use windows in your Apple IIGS program, you'll need to have already started the Tool Locator, Memory Manager, and Miscellaneous tool sets (the "big three"), as well as QuickDraw II and the Event Manager. After that, you should start the Window Manager and then the Control Manager.

The Window Record

Once all your tool sets have been started, placing a window on the screen isn't a difficult task. In fact, you simply pass information about the window to a Window Manager Toolbox function. The window's information is kept in one of the longest record structures used by the Toolbox, the window record. The window record stores all sorts of information about the window: its size, contents, color, types of controls, movability, ability to zoom, and large quantities of additional information.

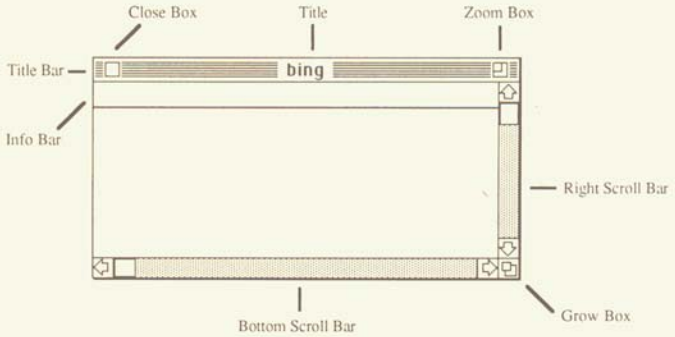
Unlike the Menu Manager, which uses several intervening steps between creating the menu and having it appear on the screen, the NewWindow function displays a window immediately. All you have to do is point to your window record so NewWindow can find it.

NewWindow returns a long pointer to your window's *port* in memory after a successful Toolbox call. Use this pointer to reference the window. For example, to close the window, the port address for that window is pushed onto the stack and a Toolbox call is made to the CloseWindow function. All other Window Manager calls use the port pointer, and there is a port for each window on your desktop.

Things in a Window

When you're creating a window, you should be familiar with all the controls it uses, and with what each control does. These controls are summarized in Figure 9-1.

Figure 9-1. Diagram of Window with Controls



The controls and items inside a window are explained below. All of these items are optional: A window need not contain any of them.

Bottom scroll bar. The bottom scroll bar moves the contents of the window right or left.

Close box. The close box is used to remove the window from the desktop (to make it disappear). This is not a direct function of this control. Your program actually makes the window close. Closing a window is covered in detail later in the chapter. The close box control is located in the title bar.

Grow box. The grow box is used to resize the window. The grow box can be grabbed with the mouse and moved to change the horizontal and vertical dimensions of the window.

Info bar. The info bar appears just below the title bar and is used to display additional information about the window. The Apple IIGS Finder program makes extensive use of window info bars to let you know how many files are present in each window, and so on.

Right scroll bar. The right scroll bar moves, or scrolls, the contents of the window either up or down. Only if a window has contents larger than can be seen through the window does it need a scroll bar.

Title. The title is the name of the window, centered in the title bar.

Title bar. The title bar shows the title of the window. The title bar also contains the optional close box, or *go-away button*, and the zoom button. The title bar is used to drag the window around the desktop. Because of this it's also referred to as the *drag region* of the window.

Zoom box. The zoom box can be used to make the window expand to fill the entire screen. Clicking the zoom a second time restores the window to its previous size. Both sizes, original and zoomed, are determined by the Window Record at the time the window is created.

When you're creating a window, all these items are specified in the window record. Depending on what type of data is in the window and how you want it displayed, any number of these options can be specified.

The TaskMaster

No discussion of windows and controls would be complete without mention of the TaskMaster. TaskMaster is a Window Manager function that acts as an extension of the Event Manager. It's especially handy when you're dealing with windows. Though the TaskMaster is discussed elsewhere in this book, it's important to know the window-related event codes returned by TaskMaster.

The following table shows the extended event codes and regular event codes returned by the TaskMaster function. Note that extended events 2–12 concern themselves with windows.

Table 9-1. Event and Extended Event Codes Returned by TaskMaster

Event Code	Extended Code	Description
16	0	Mouse is in desk
17	1	A Menu item was selected
18	2	Mouse is in the system window
19	3	Mouse is in the content of a window
20	4	Mouse is in drag region
21	5	Mouse is in grow
22	6	Mouse is in go-away
23	7	Mouse is in zoom
24	8	Mouse is in info bar
25	9	Mouse is in vertical scroll
26	10	Mouse is in horizontal scroll
27	11	Mouse is in frame
28	12	Mouse is in drop

When one of these events takes place, the event code is returned by TaskMaster. The window associated with the event can be determined by examining the TaskData field of the event record. For example, if your desktop had many windows on it and you clicked the go-away button in one of them, that window's pointer would be placed in TaskData. The window can be further manipulated by Window Manager functions that use the window pointer. (A good example of this is in the MONDO program listed at the end of this chapter.)

The important thing to remember about TaskMaster is that it assists in the trapping of window-related events. It also automatically updates the contents of a window as you scroll them around.

Opening a Window

Putting a window on the screen is a trivial task. A simple call to the Toolbox is all that is required. The complexity of the window lies in the window record—a group of values, ranges, and pointers that actually define the window.

For example, suppose you wanted to display a typical Apple IIGS window. To do this you need two things:

- A call to the Window Manager's NewWindow function
- The window record describing the window

In machine language, the call looks something like this:

```
pha                             ;Long result space
pha
pushlong        #WindowRec      ;Address of window record
_NewWindow                      ;the new window call
jsr             ErrorH          ;Remember to do error checking
pullong         WindowPtr       ;A pointer to the window
```

First, a long word of result space is pushed to the stack, followed by the address (long) of the window record. Then the call is made to NewWindow. After the call, the Toolbox returns a pointer to the window's record. All further reference to the window is made through this pointer, so it should be saved in memory. (The above routine saves the pointer at the label *WindowPtr*.)

The only possible errors at this point are $0E01 and $0E02. Error $0E01 is produced if the window record is of an unusual length (meaning you left something out or the pointer was inaccurate). Error $0E02 is a memory error and probably would only happen if your computer didn't have a memory upgrade or if you had too many windows already open.

A typical error in working with structures in machine language, especially if you're using macros, is to reference the address of a structure incorrectly. For example, the following pushlong macro is in error:

```
pushlong    WindowRec     ;This is wrong
```

Because the # in front of *WindowRec* is left off, the program attempts to push the long value that resides at WindowRec. This is akin to leaving off the ampersand (&) before a variable in a C program.

What is intended is that the address of WindowRec (its location in memory) be pushed onto the stack. The address of any object is always referenced as an immediate value. Thus, the following is the correct way to push the long address of a structure or label in memory:

```
pushlong    #WindowRec    ;This is the correct way
```

In C, the following routine can be used to summon up a new window:

```
WindowPtr  =  NewWindow(&WindowRec);
```

And in Pascal:

```
WindowPtr := NewWindow(WindowRec);
```

As was mentioned earlier, the hard part (if you want to call it that) is creating the window record. It contains a wealth of information about the window and is perhaps the most detailed record used by the Toolbox. The window record is covered later in this chapter.

Closing a Window

All that's needed to close a window is the pointer to the window record and, of course, a call to the Window Manager's CloseWindow function. After CloseWindow is called, the window is removed from the screen and all the data contained in the window is gone. Using the pointers returned from the NewWindow call in the previous section, the following code examples are used to close a window referenced by WindowPtr.

In machine language:

```
pushlong        WindowPtr    ;Saved when the window was opened
_CloseWindow                 ;(No errors are possible here)
```

In C and Pascal:

```
CloseWindow(WindowPtr);
```

After CloseWindow, the window disappears from the screen, it is removed from the current list of windows, and any data contained in the window is lost. A window doesn't have to be on top of all the other windows in the desktop, nor does it have to be active in order to be removed.

It's important to note that clicking in a window's close box does not automatically close the window. Nor does selecting a *close window* option from a pull-down menu. Closing down a window has to be done by the code in your program.

To detect when the close box has been clicked, you must use the Window Manager's TaskMaster function. The extended event

codes returned by the TaskMaster call are your clues as to what is going on in a window. Normally, most of the operations (scrolling, growing, moving, zooming, and so on) are taken care of automatically by the operating system. But your program will have to monitor the close box.

Extended event code 6, or regular event code 22, is returned by the TaskMaster call when the mouse is clicked in the close box (see Table 9-1). When extended event code 6 is returned, the corresponding window's pointer is found in the TaskData field of the event record. To close the window, the following machine language code can be used:

```
pushlong        TaskData     ;get window's pointer from TaskData.
_CloseWindow
```

The window record, placed in TaskData by the TaskMaster, is pushed to the stack for the CloseWindow call. This is the same as a regular close, except the window pointer is snatched from TaskData.

As usual, the examples for C and Pascal are a little more straightforward.

In C:

```
CloseWindow(EventRec.wmTaskData);
```

In Pascal:

```
CloseWindow(WindowPtr(EventRec.TaskData));
```

This method of using TaskData works even when there are a number of windows present on the desktop.

The Window Record

The window record defines the window, determines what the window can do, and establishes which controls (zoom, grow box, title bar, and so on) the window will have. The window record is huge—24 parameters determine what type of window is created.

In the following table, the parameter name is the word used by Apple in all documentation to refer to that particular parameter of the window record. Later on, when a sample window record is created, a few of the parameters will be combined into one to make the list easier to manage.

Table 9-2. The Window Record's Parameter List

Parameter Name	Type	Description
paramlength	Word	Size of this table
wFrame	Word	Bit pattern describing the frame
wTitle	Long	Window's title
wRefCon	Long	User-defined value, usually 0
wZoom	Rectangle	Size of window when zoomed
wColor	Long	Window's color table location
wYOrigin	Point	Window content's origin, Y position
wXOrigin	Point	Window content's origin, X position
wDataH	Word	Height of document
wDataW	Word	Width of document
wMaxH	Word	Maximum height for grow window
wMaxW	Word	Maximum width for grow window
wScrollV	Word	Number of Y pixels to scroll
wScrollH	Word	Number of X pixels to scroll
wPageVer	Word	Number of Y pixels to page
wPageHor	Word	Number of X pixels to page
wInfoRefCon	Long	Used by info-bar draw routine
wInfoHeight	Word	Height of info-bar
wFrameDefProc	Long	Window definition procedure
wInfoDefProc	Long	Info-bar drawing routine
wContDefProc	Long	Content drawing procedure
wPosition	Rectangle	Window's starting coordinates
wPlane	Long	Position, front to back
wStorage	Long	Memory for window record

Incidentally, the tiny *w* at the front of a parameter name is an instant tip-off that the parameter belongs to the Window Manager.

Each of the parameters is discussed in detail in *COMPUTE!'s Mastering the Apple IIGS Toolbox*. However, the following is a brief rundown of each of them, along with explanations and expansions where necessary.

paramlength. The parameter paramlength (word value) is the length of the entire window record. It's used by the Memory Manager in moving these parameters to the internal window record. It also serves as a form of error checking: If the paramlength is inaccurate, the Window Manager returns an error code of $0E01 after the NewWindow call.

wFrame. The parameter wFrame (word value) describes the frame of the window. Each bit in the word wFrame signals the presence or absence of one of the window controls. A window with everything on it has the following bit pattern:

```
1101111110100000
```

which is $DFA0 in hex. The individual significance of each of the bits is shown in Table 9-3.

Table 9-3. wFrame Values

Bit	If set, means
0	The window is highlighted (initially always 0)
1	Window is zoomed when first drawn
2	Internal use (determines window record allocation)
3	Window's controls can be active when the window is inactive
4	The window has an info bar
5	The window is visible
6	An inactive window is made active if the mouse is clicked in it
7	The window can be moved (bit 15 should also be set)
8	The window has a zoom box (bit 15 should also be set)
9	The size of the window is flexible (grow and zoom will not change the origin of the window's data)
10	The window has a grow box (bit 11, bit 12, or both should also be set)
11	The window has an up- and down-scroll bar (right side)
12	The window has a left- and right-scroll bar (bottom)
13	The window has a double frame, like an alert dialog box
14	The window has a go-away button (bit 15 should also be set)
15	The window has a title bar

The bits above that are set to define a window "with the works" are 5, 7, 8, 9, 10, 11, 12, 14, and 15. Bit 13 is used by the Dialog Manager when it creates a window. Bits 4, 8, 9, 10, 11, 12, 14, and 15 must be reset to 0 if this bit is set.

wTitle. wTitle (long pointer) points to the memory location containing the window's title. The title is a Pascal string, and it's a good idea to pad it with spaces. (This keeps the title from appearing too tight in the title bar. More on this in a while.) If a long word of 0 is specified, the window has no title.

wRefCon. wRefCon (long value) is a user-defined value, though typically a long word of 0 is specified. A few of the Window Manager's functions can return or set this value, but its meaning is up to you. For example, in the sample program, MONDO, wRefCon is used to number each window for later reference in the program.

wZoom. wZoom (rectangle) indicates the size of the window when zoomed. The four word values are listed in the order MinY, MinX, MaxY, MaxX. If four words of 0 are specified, the entire screen is filled with the window. It's also suggested your window have a zoom box (bit 8 of wFrame above).

wColor. wColor (long pointer) points to a table controlling the color of the window, title bar, and frame. If a long word of 0 is specified, the system default colors are used. (See the section on color later in this chapter for additional information.)

wYOrigin and wXOrigin. wYOrigin (point) and wXOrigin (point) set the Y and X origins of the window's data. Both Y and X are word values, expressed in global coordinates (with 0, 0 as the upper left corner of the screen). In this book, both wYOrigin and wXOrigin together are referred to as the point value wOrigin.

wDataH and wDataW. wDataH (word) and wDataW (word) designate the height and width of the data inside the window. If the data cannot be scrolled (meaning the window doesn't have scroll bars), two words of 0 are used. wMaxH (word) and wMaxW (word) specify the maximum height and width of the window. The size of the window is manipulated by the window's grow box and is measured in pixels.

wScrollV and wScrollH. wScrollV (word) and wScrollH (word) define the number of Y and X pixels, respectively, that a window may scroll when the arrows are clicked in either the up/down or left/right scroll bars.

wPageVer and wPageHor. wPageVer (word) and wPageHor (word) define the number of Y and X pixels that a window is paged. Paging occurs when the mouse is clicked inside a scroll bar. (This should be a proportionally larger value than for wScrollV and wScrollH, above.)

wInfoRefCon. wInfoRefCon (long pointer) points to a string to be placed in the window's information bar. If there is no string, it points to a long word of 0. (The window should have an information bar for this value to take effect—bit 4 of wFrame above.)

wInfoHeight. wInfoHeight (word) defines the height, in pixels, of the window's information bar. As with wInfoRefCon, the window should contain an information bar for this value to have any meaning.

wFrameDefProc. wFrameDefProc (long pointer) points to a window definition routine or procedure. It is normally set to a long word of 0 to use the default routines.

wInfoDefProc. wInfoDefProc (long pointer) points to a routine that draws the window's information bar, or it points to a long word of 0 if no info bar is present in the window.

wContDefProc. wContDefProc (long pointer) contains the address of a routine that draws the contents of a window. An example of such a routine is listed in the section "Window Contents" later in this chapter. If no routine is used, a long word of 0 is specified. Also, if you don't supply a redraw routine, your window shouldn't have scroll bars.

wPosition. wPosition (rectangle) defines the starting position and size of the window. The four word values are listed in the order MinY, MinX, MaxY, MaxX, and are in global coordinates.

wPlane. wPlane (long value) indicates this window's precedence—in other words, how many windows are stacked on top of it. A long word of 0 places the new window behind every other window on the desktop. A long word of $FFFFFFFF, which is also −1, places the new window on top of all the others.

wStorage. wStorage (long pointer) represents the address of additional storage for the window record. This value is always set to 0 because Apple has not officially said what other values will mean in the future.

The following are examples of window records. Each corresponds to the NewWindow toolbox calls demonstrated earlier in this chapter. To add a window to your program, simply add the NewWindow call as shown above and have it reference a window record with the data you desire. The following window records are simple, standard window records. Later in this chapter, more exciting, splashy, and mind-boggling window records are used.

In machine language:

```
WindowRec   anop
            dc      i'WRecEnd-WindowRec'    ;size of parameter list
            dc      i'%1101111110100000'    ;frame type
            dc      i4'Wtitle'              ;Title string pointer
            dc      i4'0'                   ;Reserved
            dc      i2'0,0,0,0'             ;Position When Zoomed (0=def)
            dc      i4'0'                   ;Pointer to color table
            dc      i2'0,0'                 ;Contents Vert/Horz Origin
            dc      i2'200,640'             ;Height/Width of document
            dc      i2'200,640'             ;Height width for grow window
            dc      i2'4,16'                ;Vert/horz pixels for scroll
            dc      i2'40,160'              ;vert/horz pixels scroll page
            dc      i4'0'                   ;Value passed to information
            dc      i2'0'                   ;Height of info bar
            dc      i4'0'                   ;Window Definition
            dc      i4'0'                   ;Draw info bar routine
            dc      i4'0'                   ;Draw Interior
            dc      i2'40,100,159,540'      ;Starting position and size
            dc      i4'$FFFFFFFF'           ;Starting plane
            dc      i4'0'                   ;window record
WRecEnd     anop
```

A title string (called Wtitle in the program example) also needs to be defined. The title string is a Pascal string, which means it's preceded by a count byte. Below, the macro *str* is used to define the title for this window:

```
Wtitle    str" Mr. Mondo "
```

Note that the title is padded with spaces. If the spaces were removed, the title would appear jammed into the title bar.

In C, global record structure can be used to create a window record as follows:

```
ParamList   WindowRec = {
            sizeof(WindowRec),      /* size of parameter list */
            0xdfa0,                 /* frame type */
            "\p Mr. Mondo ",        /* Pascal title string */
            NULL,                   /* refcon */
            0, 0, 0, 0,             /* Position When Zoomed (0=def) */
            NULL,                   /* Pointer to color table */
            0, 0,                   /* Contents Vert/Horz Origin */
            200, 640,               /* Height/Width of document */
            200, 640,               /* height/width for grow window */
```



```
            4, 16,                  /* vert/horz pixels for scroll */
            40, 160,                /* vert/horz pixels scroll page */
            NULL,                   /* information bar string */
            0,                      /* Height of info bar */
            NULL,                   /* Window Definition routine */
            NULL,                   /* Draw info bar routine */
            NULL,                   /* Draw content routine */
            40, 100, 159, 540,      /* Starting position and size */
            -1L,                    /* starting plane */
            NULL                    /* window record address */
};
```

In Pascal, your window record and its title string must first be declared in the VAR section of your program:

```
VAR
      WindowRec:   NewWindowParamBlk;
      TheTitle:    String;
```

The structure is then loaded with data at runtime (within a function or procedure) with the desired values:

```
TheTitle := ' Mr. Mondo ';

WITH WindowRec DO BEGIN
   param_length     := sizeof(NewWindowParamBlk);
   wFrame           := $dfa0;           { frame type }
   wTitle           := @TheTitle;       { pointer to title string }
   wRefCon          := nil;             { refcon }
   SetRect          (wZoom, 0, 0, 0, 0);
   wColor           := nil;             { color table pointer }
   wYOrigin         := 0;               { content origin }
   wXOrigin         := 0;
   wDataH           := 200;             { document size }
   wDataW           := 640;
   wMaxH            := 200;             { grow window size }
   wMaxW            := 640;
   wScrollVer       := 4;               { scroll range }
   wScrollHor       := 16;
   wPageVer         := 40;              { page range }
   wPageHor         := 160;
   wInfoRefCon      := LongInt(nil);    { Draw info bar routine }
   wInfoHeight      := 0;               { Height of info bar }
   wFrameDefProc    := nil;             { Window Definition routine }
   wInfoDefProc     := nil;             { Draw info bar routine }
   wContDefProc     := nil;             { Draw content routine }
```

```
   SetRect          (wPosition, 100, 40, 540, 159);
   wPlane           := -1;              { starting plane }
   wStorage         := nil;             { window record address }
END;
```

Once you've defined an acceptable window record, you can use it as a template for any other window applications you write. Customizing an existing window record is easier than building a new one each time from the ground up. The remaining sections in this chapter augment certain parameters of the window record, and help make your windows more exciting.

Naming a Window

About the only important thing to do when naming a window is to place spaces on either side of the title. The spaces provide adequate breathing room between the title and the rest of the title bar.

The title of the window is specified in the window record, in the wTitle field:

```
wTitle    (long pointer)
```

wTitle points to the address of a Pascal string that contains the window's title. In the previous window record example, the title of the window was listed as follows (in machine language):

```
   dc   i4'wTitle'    ;title string pointer
```

The actual title is at the address wTitle:

```
wTitle    str' My Window '
```

The *str* macro is used to create a Pascal string with a leading count byte for *My Window.*

You can name a window anything. Or, if you use a long word of 0 for the wTitle field of the window record, the window won't have a title. (This also holds true if you haven't specified a title bar for the window.)

Most often, you'll want to use the name of the file you're working on as the title of the window. This involves a little byte-wise sleight of hand in machine language because of the way ProDOS stores a filename in memory. C and Pascal programmers can use standard string-handling functions which make this job a cinch.

A ProDOS filename is stored as a Pascal string, and can be from 1 to 15 characters long (not including a prefix). The characters after the count byte make up the actual name of the file. Since a filename buffer can take up as many as 16 characters (the count byte plus the 15 letters), your programs should provide at least that much space for that worst-case scenario. The string buffer should always be 16 characters long no matter how long the expected filename is.

Suppose you've used the Standard Files tool set to return the name of a file on disk—either a text, picture, or some other file. When the file's name is returned, it is a maximum of 16 characters long, the first of which is a count byte. The filename can be as many as 15 characters long; if there are fewer characters, the remaining characters are padded with nulls (zero bytes).

The object for you, the programmer, is to examine the filename string returned by the Standard Files tool set and make it suitable as the title of a window. Of course, you can't just use the filename returned by ProDOS. Instead, you must delicately extract the filename, being careful to add a space in front and a space behind for padding. And, don't forget to add 2 to the count byte. But the hard part is done for you, as explained below.

In machine language, the following routine moves a ProDOS filename from its storage buffer to a window title storage buffer (the ProDOS filename is stored at location Fname; the window's title, at location wTitle):

```
Pro2Win  LONGA   OFF          ;ProDOS-to-Window Title
         LONGI   OFF
         sep     $30          ;use eight-bit registers
         lda     Fname        ;get the name's length in A
         tax                  ;save a copy in X
         inc     A            ;increase the length by 2
         inc     A
         sta     wTitle       ;save it in the window's title
         lda     $20          ;add a space
         sta     wTitle+1     ;to the beginning of the title
         sta     wTitle+2,x   ;and one to the end
```

```
loop    lda     Fname,x      ;read a character from filename
        sta     wTitle+1,x   ;store filename in title buffer
        dex                  ;work backwards to start of name
        bne     loop         ;if not zero, keep looping
        rep     $30          ;back to 16-bit registers
        LONGA   ON
        LONGI   ON
        rts                  ;and you're done
```

This routine takes the filename from location Fname and moves it to the window's title location, wTitle, and adds one space on each end of the filename. The size of the wTitle buffer should be 18 characters, two more than the Fname buffer, so that it can hold the largest possible filename.

First the routine reads the length of the filename; then it adds 2 to that length (with two INC A instructions), one for each space. Then, before the file's name is transferred, a space is placed at the start and end of the window title.

In the main program loop, the characters are moved from Fname to wTitle. Each character is taken from the right side of Fname, indexed by X, and it works to the left. When X reaches 0, the last character has been moved. This backwards copying method saves a few instructions that would have been needed if you were copying in a forward direction.

In C, the logic follows that of machine language since C's string-handling functions aren't meant to be used with Pascal strings. The routine is as follows:

```
Pro2Win()
{
        char x = Fname[0];              /* x = length of filename */
        wTitle[0] = x + 2;              /* set wTitle's new length */
        wTitle[1] = wTitle[x + 2] = ' '; /* pad with spaces */
        for (; x; --x)                  /* while x is not zero, */
          wTitle[x + 1] = Fname[x];     /* copy the string */
}
```

Far simpler, because of the compatible string format, the following single statement can be used in Pascal:

```
wTitle := CONCAT(' ', Fname, ' ');
```

Feel free to include these routines in any of your programs that use a filename as the window's title.

Colorful Windows

Windows on the Apple IIGS come in several styles, from the plain, black-titled windows, to stylish and colorful windows that would make a Macintosh owner green with envy.

The color of the window is set by the wColor parameter found in the window record:

```
wColor    (long pointer)
```

wColor points to the address of a table containing the colors to be used in the window. If a long word of 0 is specified, the Window Manager creates a black and white window with a solid black title bar.

But you can change that. For example, the following could be included as the wColor parameter of a window record:

```
dc  I4'WColor'   ;address of color table
```

The color table consists of five word-sized values, each of which describes a different color attribute of the window. The actual colors are determined by the bit positions within each of the words. The five words are described in Table 9-4.

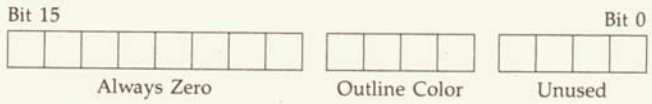
Table 9-4. Color Table

Color Word	Sets the Color For
FrameColor	Window outline
TitleColor	Title, zoom, close boxes
TBarColor	Title pattern and background
GrowColor	Grow box
InfoColor	Info bar

The bit positions are significant in each of these words. Generally speaking, each word is split into groups of four bits (one nibble). These four bits can represent 16 values from 0 (0000) – 15 (1111). Each value then represents one color from the current palette as set by QuickDraw II.

FrameColor (Figure 9-2) sets the color of the window's outline, including the outline of the title bar and info bar.

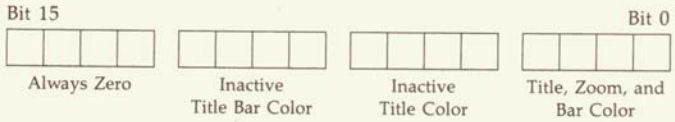
Figure 9-2. Meaning of Color Bits in FrameColor



The only bits of any significance here are at positions 4–7. Those values control the color of the window's frame. The other bits in this word should be set to 0.

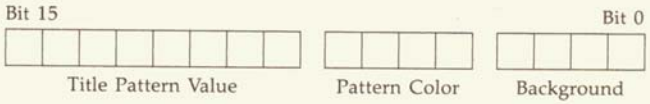
TitleColor (Figure 9-3) controls the color of the window's title (the name of the window), and the close and zoom buttons, as well as the colors of the title bar and title when the window is inactive.

Figure 9-3. Meaning of Color Bits in TitleColor



The inactive colors specified by TitleColor only appear when a window is inactive, or in the background. Otherwise, only bits 0–3 are used when a window is first displayed to color the title, zoom, and close boxes. The inactive title bar color and inactive title color are best used when both values are opposites, such as 0000 for the first and 1111 for the second. When both are the same, the title of an inactive window appears all one color.

Figure 9-4. Meaning of Color Bits in TBarColor

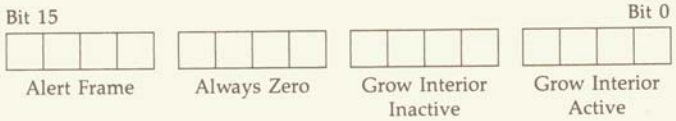


In the TBarColor slot (Figure 9-4), the title pattern value is one of three values: 0 (00000000) for solid, 1 (00000001) for dithered, or 2 (00000010) for barred—as on the Macintosh.

The pattern color and background (Figure 9-4) set the foreground and background colors for whatever type of pattern is selected. For the solid title bar, only Pattern color (bits 4–7) are used. For a dithered or barred pattern, both values are used.

In the GrowColor Slot (Figure 9-5), the alert frame, bits 12–15, are used when the type of window created is a dialog box and not an actual window. (Remember, the Window Manager is also responsible for creating dialog boxes.)

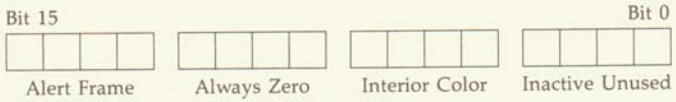
Figure 9-5. Meaning of Color Bits in GrowColor



A special type of dialog box, the alert box, has a few outlines. The alert frame parameter above colors the alert box's middle outline. The grow interior inactive parameter colors the inactive window's grow box. The grow interior active parameter colors the active window's grow box.

As with GrowColor, bits 12–15 of InfoColor (Figure 9-6) determine the color of an alert box. This time the parameter affects the inside outline's color.

Figure 9-6. Meaning of Color Bits in InfoColor



The only other significant bits are 4–7, which control the interior color of the window when it's inactive.

A sample color table would be as follows. In machine language, the percent sign is used to indicate a bit value description of the word. The following color table creates a typical Macintosh-style window using only black and white color values.

```
WColor  dc  i'%0000000000000000'  ;frame color
        dc  i'%0000111100000000'  ;Title Color
        dc  i'%0000001000001111'  ;Title Bar Color
        dc  i'%0000000011110000'  ;Grow Box Color
        dc  i'%0000000011110000'  ;Info Bar Color
```

In C, a sample color table declaration would be

```
WindColor  wColor = {
           0x0000,     /* frame color */
           0x0f00,     /* title color */
           0x020f,     /* title bar color */
           0x00f0,     /* grow box color */
           0x00f0 };   /* info bar color */
```

And in Pascal (wColor is defined as a WindowColorTbl type):

```
WITH wColor DO BEGIN
   FrameColor  := $0000;
   TitleColor  := $0F00;
   TBarColor   := $020F;
   GrowColor   := $00F0;
   InfoColor   := $00F0;
END;
```

Using these premanufactured structures, you'll find it easy to experiment until you create the right window for your needs.

Window Contents

What good is a window unless you can put something into it? Not much. Putting data into a window isn't that difficult; it just requires that you know which buttons to push.

The contents of a window are drawn by a routine indicated in the window record. The wContDefProc parameter contains the long address of a routine that draws the window's contents:

```
wContDefProc    (long pointer)
```

The routine, if written in machine language, should end in an RTL instruction. Functions and procedures in C and Pascal always end in RTLs. The wContDefProc routine draws the contents of the window, then exits. There are no input or output parameters, nor do you need to do any extensive graphics tweaking.

The wContDefProc routine is called by the TaskMaster to update the window's contents—for example, when the window is scrolled or its size is changed. When you're using graphics, the window's port will be the current GrafPort.

If wContDefProc is a long word of 0, then the window will be blank, and scrolling about in the window will erase the window's data. This is why windows without a wContDefProc routine should not have scroll bars.

The following are two examples of the wContDefProc parameter in a window record. The first is an empty field, meaning no procedure is defined. The second is the address of a procedure to draw the contents of a window.

In machine language:

```
dc  i4'0'           ;no update routine
```

or

```
dc  i4'WContent'    ;address of update routine
```

In C:

```
WindowRec.wContDefProc = NULL;        /* no update routine */
```

or

```
WindowRec.wContDefProc = WContent;    /* name of function */
```

In Pascal:

```
WindowRec.wContDefProc := nil;        { no update routine }
```

or

```
WindowRec.wContDefProc := @WContent;  { address of procedure }
```

See the MONDO program example in the next section for a wContDefProc routine that displays a string in a window. When designing your own update routines, remember that the actual size of the window's port is set by the window record when the window is created.

Also, for some reason, having a wContDefProc routine that is just an RTL instruction (or a null routine or procedure in C or Pascal) doesn't seem to work. It causes the machine to crash. Apparently some window access or graphics interaction is required by the routine. This could be because of the versions of tool sets that the IIGS uses at the time of this writing.

The MONDO Window Program

Below is the program MONDO as written in machine language, C, and Pascal. It uses the program MODEL (introduced earlier in this book) as a base upon which to work. To run the MONDO program, you'll need to copy and rename the MODEL program and merge in the following modifications. The program is only altered a little, so there need not be much retyping.

MONDO adds two windows to the MODEL program and performs some interesting trickery with pull-down menus. The windowing routines are used to open and close two separate windows. An extra routine has been added to hide or show the first window and to demonstrate how easy the Window Manager's functions are to use.

When you begin experimenting with this program, change the Show/Hide menu option to some other Window Manager function (SelectWindow, BringToFront, SendBehind, HiliteWindow, or a number of others.

MONDO also demonstrates some interesting Menu Manager functions. For example, when a window is open, its corresponding open menu item is dimmed. When the window is closed, its close menu item is dimmed. This is done with two Menu Manager calls, EnableMItem and DisableMItem. Additionally, when the first window is hidden, the Hide menu item changes to Show. This is accomplished with the SetMItemName function and can be seen in the code samples below.

Another interesting thing to note is how the first window makes use of a custom color table and the second window uses a default color table. Also, note how the wRefCon value is used to identify each window. Because wRefCon's value can be anything you want, MONDO uses it to identify which window is being closed in order to update (dim or enable) the associated menu item. This is done with the GetWRefCon function in the CloseW procedure in the following source code listings.

As usual, feel free to modify this program or use its routines in your own applications. As a special project, try to fix the error that occurs when an invisible window is closed and the Show menu item is not changed back to Hide.

Program 9-1. Machine Language Source for MONDO.ASM

```
*--------------------------------------------------*
*                                                  *
*                    MONDO.ASM                     *
*   Sample Desktop Application in APW Assembler (1.0)  *
*                 Windowing Routines               *
*--------------------------------------------------*

; To create the Mondo.Macros macro file, use this APW shell command:
; # macgen mondo.asm mondo.macros 2/ainclude/m16=

        ABSADDR ON
        KEEP    Mondo
        MCOPY   Mondo.Macros

*-----------------------------*
*       Global Equates        *
*-----------------------------*

Toolbox gequ    $e10000         ;Primary tool dispatcher
TRUE    gequ    $8000           ;True value
FALSE   gequ    $0000           ;False value
Page    gequ    $100            ;The size of a page (256 bytes)
mOpen1  gequ    257
mHide1  gequ    258
mClose1 gequ    259
mOpen2  gequ    260
mClose2 gequ    261

; *NOTE*     From this point on, copy the source from
; *NOTE*     the original MODEL.ASM APW program...

ModelA  START
        phk                     ;Make the data bank...
        plb                     ;...the current code bank
        brl     Main            ;branch over functions to Main

; *NOTE*     Et cetera, on down to the end of the "Main" routine
; *NOTE*     as follows:

        jsr     ShutDownTools   ;Shut down all tools started

        _QUIT   Qparms          ;Exit this program through ProDOS 16

; *NOTE*     Then add what follows after this point.
; *NOTE*     It augments and replaces the rest of the ModelA
; *NOTE*     source code:

*-----------------------------*
*      Window Routines        *
*-----------------------------*

Open1   pea $0000               ;long result space
        pea $0000
        pushlong #WindRec1      ;first window record
        _NewWindow              ;open it

        jsr ErrChk              ;check for errors
        pullLong WindPtr1       ;get pointer #1
```

```
        pea mOpen1              ;this menu item number
        _DisableMItem           ;dim it

        pea mHide1              ;enable these two
        _EnableMItem

        pea mClose1
        _EnableMItem
        rts

;-----------------------------

hidebit dc i2'0'                ;zero = window is visible

Hide1   lda hidebit
        beq HideIt

ShowIt  pushlong WindPtr1       ;show the window
        _ShowWindow

        pushlong #mHide         ;change the name back
        pea mHide1
        _SetMItemName

        lda #$0
        sta hidebit
        rts

HideIt  pushlong WindPtr1       ;hide this window
        _HideWindow             ;hide it, no errors

        pushlong #mShow         ;change menu item name
        pea mHide1              ;this item number
        _SetMItemName           ;change the name

        lda #$FFFF              ;change status byte
        sta hidebit
        rts

;-----------------------------

Open2   pea $0000               ;long result space
        pea $0000
        pushlong #WindRec2      ;second window record
        _NewWindow              ;open that window
        jsr ErrChk
        pullLong WindPtr2       ;get second window's pointer

        pea mOpen2              ;now, dim this menu
        _DisableMItem

        pea mClose2             ;and enable this item
        _EnableMItem
        rts

;-----------------------------

Close1  pushlong WindPtr1       ;close this window
        _CloseWindow
```



```
        pea mOpen1              ;this menu item number
        _EnableMItem            ;Enable this again

        pea mHide1              ;disable these two
        _DisableMItem

        pea mClose1
        _DisableMItem
        rts

;-----------------------------

Close2  pushlong WindPtr2       ;close window two
        _CloseWindow

        pea mOpen2              ;re-enable this item
        _EnableMItem

        pea mClose2
        _DisableMItem
        rts

;-----------------------------
; close whichever window was clicked

CloseW  pea $0000               ;long result space
        pea $0000
        pushlong TaskData       ;get the window's pointer from Taskdata..
        _GetWRefCon             ;get the window's wRefCon value

        pla                     ;get low order word into a
        plx                     ;toss away high order word in x
        cmp #1                  ;if 1, then it's window 1
        beq Close1              ;so close window 1
        bra Close2              ;else, close window 2

;-----------------------------
WContent pea $0028                      ;Horizontal pointer loc.
         pea $0020                      ;Vertical pointer loc.
         _MoveTo                        ; (QuickDraw)

         PushLong #String0
         _DrawCString

         rtl                            ;long return

String0  dc c"This is a string inside the window.",i1'0'

*-----------------------------*
*      Variable Storage       *
*-----------------------------*

UserID  ds      2               ;Our User ID
MemID   ds      2               ;Memory User ID (made from User ID)
DPBase  ds      2               ;Used by DP buffer manager
QFlag   dc      i'FALSE'        ;Boolean: Quit flag (starts out as false)

*-----------------------------*
*  StartUp/Shutdown Tool List *
*-----------------------------*
```

```
Toolist dc      i'(ToolstE-Toolist-1)/4'        ;Tool count
        dc      i'1,0'          ; Tool Locator
        dc      i'2,0'          ; Memory Manager
        dc      i'3,0'          ; Misc Tools
        dc      i'4,0'          ; QuickDraw II
        dc      i'6,0'          ; Event Manager
        dc      i'14,0'         ; Window Manager
        dc      i'16,0'         ; Control Manager
        dc      i'15,0'         ; Menu Manager
        dc      i'5,0'          ; Desk Manager
ToolstE anop

*-----------------------------*
*   Pull Down Menu Structures *
*-----------------------------*

MenuTbl dc      i'(MenTblE-MenuTbl-1)/2'        ;Menu count
        dc      i'Menu1'        ;Apple
        dc      i'Menu2'        ;Window
        dc      i'Menu3'        ;Quit
MenTblE anop

Menu1   dc      c'>>@\XN1',i1'0'                ;Apple
        dc      c'--About This Program...\N256',i1'0'
        dc      c'---\D',i1'0'
        dc      c'>'

Menu2   dc      c'>>  Window  \N2',i1'0'        ;Window
        dc      c'--Open Window1\N257',i1'0'
        dc      c'--Hide Window1\DN258',i1'0'
        dc      c'--Close Window1\DVN259',i1'0'
        dc      c'--Open Window2\N260',i1'0'
        dc      c'--Close Window2\DN261',i1'0'
        dc      c'>'

Menu3   dc      c'>>  Quit  \N3',i1'0'          ;Quit
        dc      c'--Quit\N262*Qq',i1'0'
        dc      c'>'

mShow   str 'Show Window1'                      ;changing menu items
mHide   str 'Hide Window1'

*-----------------------------*
* Menu Item Dispatch Addresses *
*-----------------------------*

MTable  dc      i'About'        ;256/About              (Apple Menu)
        dc      i'Open1'        ;257/Open window 1      (Window Menu)
        dc      i'Hide1'        ;258/Hide window 1
        dc      i'Close1'       ;259/Close window 1
        dc      i'Open2'        ;260/Open window 2
        dc      i'Close2'       ;261/Close window 2
        dc      i'Quit'         ;262/Quit               (File Menu)

*-----------------------------*
*       The Event Record      *
*-----------------------------*

EventRec        anop                    ;Event Record used by TaskMaster
EWhat           ds      2               ;What
```



```
EMsg            ds      4               ;Message
EWhen           ds      4               ;When
EWhere          ds      4               ;Where
EMods           ds      2               ;Modifiers
TaskData        ds      4               ;Task Data
TaskMask        dc      i4'$1fff'       ;Task Mask

*-----------------------------*
*      Window Data            *
*-----------------------------*

WindPtr1 ds 4                           ;Window pointer

W1title  str " Mr. Mondo One "

W1Ctable dc i'%0000000000000000'        ;frame color
         dc i'%0000111100000000'        ;Title Color
         dc i'%0000001000001111'        ;Title Bar Color
         dc i'%0000000011110000'        ;Grow Box Color
         dc i'%0000000011110000'        ;Info Bar Color
WindRec1 anop
         dc i'WR1end-WindRec1'          ;size of parameter list
         dc i'%1101111110100000'        ;frame type
         dc i4'W1title'                 ;title
         dc i4'1'                       ;Used for window number here
         dc i2'0,0,0,0'                 ;Position When Zoomed 0=def
         dc i4'W1Ctable'                ;Pointer to color table
         dc i2'0,0'                     ;Contents Vert/Horz Origin
         dc i2'180,640'                 ;Height/Width of document
         dc i2'180,640'                 ;height/width for grow window
         dc i2'4,16'                    ;vert/horz pixels for scroll
         dc i2'40,160'                  ;vert/horz pixels scroll page
         dc i4'0'                       ;Value passed to information draw
         dc i2'0'                       ;Height of info bar
         dc i4'0'                       ;Window Definition
         dc i4'0'                       ;Draw info bar routine
         dc i4'WContent'                ;Draw Interior
         dc i'40,100,159,540'           ;Starting position and size
         dc i4'$FFFFFFFF'               ;starting plane
         dc i4'0'                       ;Window Record
WR1end   anop

WindPtr2 ds 4                           ;Window pointer

W2title  str " Mr. Mondo Two "

WindRec2 anop
         dc i'WR2end-WindRec2'          ;size of parameter list
         dc i'%1101111110100000'        ;frame type
         dc i4'W2title'                 ;title
         dc i4'2'                       ;Used for window number here
         dc i2'0,0,0,0'                 ;Position When Zoomed 0=def
         dc i4'0'                       ;no color table
         dc i2'0,0'                     ;Contents Vert/Horz Origin
         dc i2'180,640'                 ;Height/Width of document
         dc i2'180,640'                 ;height/width for grow window
         dc i2'4,16'                    ;vert/horz pixels for scroll
         dc i2'40,160'                  ;vert/horz pixels scroll page
         dc i4'0'                       ;Value passed to information draw
         dc i2'0'                       ;Height of info bar
         dc i4'0'                       ;Window Definition
```

```
        dc i4'0'                    ;Draw info bar routine
        dc i4'0'                    ;Draw Interior (none)
        dc i'50,120,169,560'        ;Starting position and size
        dc i4'$FFFFFFFF'            ;starting plane
        dc i4'0'                    ;Window Record
WR2end  anop

*-----------------------------*
*     Miscellaneous Data      *
*-----------------------------*

Moment  dc      c'One Moment...',i1'0'

QParms  dc      i4'0'           ;ProDOS 16 Quit Code parameters
        dc      i'$0000'

        END
```

Program 9-2. C Language Source for MONDO.C

```
/*--------------------------------------------*
 *                 MONDO.C                    *
 * Sample Desktop Application in APW C (1.0)  *
 *--------------------------------------------*/

/* **NOTE** This is not a complete program.  Merge parts of
            this listing with the MODEL.C program from
            Chapter Six.  Insert portions from MODEL.C where
            indicated. */

/* #include directives -- insert from MODEL.C */

#define mAbout  256         /* Menu item IDs */
#define mOpen1  257
#define mHide1  258
#define mClose1 259
#define mOpen2  260
#define mClose2 261
#define mQuit   262

/*-----------------------------*
 *      Global Variables       *
 *-----------------------------*/

WmTaskRec   EventRec;           /* Event Record Structure */

Word        Event,              /* Event code */
            UserID,             /* Our User ID */
            MemID,              /* Memory Management ID */
            QFlag;              /* Boolean: Quit flag */

Word        ToolList[] = {
                3,              /* Tool count */
                14, 0,          /* Window Manager */
                15, 0,          /* Menu Manager */
                16, 0           /* Control Manager */
};

char        *DPBase;            /* Direct Page base pointer */
```



```
GrafPortPtr WindPtr1,           /* Window port pointers */
            WindPtr2;

WindColor   WiCtable = {
                0x0000,         /* frame color */
                0x0f00,         /* title color */
                0x020f,         /* title bar color */
                0x00f0,         /* grow box color */
                0x00f0          /* info bar color */
};

int WContent();
ParamList   WindRec1 = {
                sizeof(WindRec1),   /* size of parameter list */
                0xdfa0,             /* frame type */
                "\p Mr. Mondo One ",/* window title */
                1L,                 /* used for window number */
                0, 0, 0, 0,         /* position when zoomed */
                &WiCtable,          /* color table */
                0, 0,               /* content vert/horz origin */
                180, 640,           /* height/width of document */
                180, 640,           /* height/width for grow window */
                4, 16,              /* vert/horz pixels for scroll */
                40, 160,            /* vert/horz pixels for page */
                NULL, 0,            /* no info bar */
                NULL,               /* window definition */
                NULL,               /* draw info bar */
                WContent,           /* draw interior */
                40, 100, 159, 540,  /* position / size */
                -1L,                /* plane */
                NULL                /* window record address */
};

ParamList   WindRec2 = {
                sizeof(WindRec2),   /* size of parameter list */
                0xdfa0,             /* frame type */
                "\p Mr. Mondo Two ",/* window title */
                2L,                 /* used for window number */
                0, 0, 0, 0,         /* position when zoomed */
                NULL,               /* color table */
                0, 0,               /* content vert/horz origin */
                180, 640,           /* height/width of document */
                180, 640,           /* height/width for grow window */
                4, 16,              /* vert/horz pixels for scroll */
                40, 160,            /* vert/horz pixels for page */
                NULL, 0,            /* no info bar */
                NULL,               /* window definition */
                NULL,               /* draw info bar */
                NULL,               /* draw interior */
                50, 120, 169, 560,  /* position / size */
                -1L,                /* plane */
                NULL                /* window record address */
};

Boolean hidebit = FALSE;

/* ErrChk()         -- insert from MODEL.C */
/* GetDP()          -- insert from MODEL.C */
/* StartUpTools()   -- insert from MODEL.C */
```

```
/*-----------------------------*
 *  Prepare Desktop and Menus  *
 *-----------------------------*/
PrepDeskTop()
{
    static char *AppleMenu[] = {
        ">>@\\XN1",
        "--About This Program...\\N256",
        "---\\D",
        ">"
    };

    static char *WindowMenu[] = {
        ">>  Window  \\N2",
        "--Open Window1\\N257",
        "--Hide Window1\\DN258",
        "--Close Window1\\DVN259",
        "--Open Window2\\N260",
        "--Close Window2\\DN261",
        ">"
    };

    static char *QuitMenu[] = {
        ">>  Quit  \\N3",
        "--Quit\\N262*Qq",
        ">"
    };

    RefreshDesktop(nil);                    /* Display Desktop */
    InitCursor();                           /* Show mouse cursor */

    InsertMenu(NewMenu(QuitMenu[0]), 0);    /* Install menus */
    InsertMenu(NewMenu(WindowMenu[0]), 0);
    InsertMenu(NewMenu(AppleMenu[0]), 0);

    FixAppleMenu(1);                        /* Display menu bar */
    FixMenuBar();
    DrawMenuBar();
}

/*-----------------------------*
 *  Apple Menu: About          *
 *-----------------------------*/

About()
{
       /* Does nothing (for now) */
}

/*-----------------------------*
 *   Window Content Procedure  *
 *-----------------------------*/

WContent()
{
    MoveTo(0x28, 0x20);
    DrawCString("This is a string inside the window.");
}
```



```
/*-----------------------------*
 *  Window Menu: Open1         *
 *-----------------------------*/

Open1()
{
    WindPtr1 = NewWindow(&WindRec1);    /* open first window */
    DisableMItem(mOpen1);               /* disable open1 item */
    EnableMItem(mHide1);                /* enable these two... */
    EnableMItem(mClose1);
}

/*-----------------------------*
 *  Window Menu: Hide1         *
 *-----------------------------*/

Hide1()
{
    if (hidebit) {
        ShowWindow(WindPtr1);
        SetMItemName("\pHide Window1", mHide1);
        hidebit = FALSE;
    } else {
        HideWindow(WindPtr1);
        SetMItemName("\pShow Window1", mHide1);
        hidebit = TRUE;
    }
}

/*-----------------------------*
 *  Window Menu: Open2         *
 *-----------------------------*/

Open2()
{
    WindPtr2 = NewWindow(&WindRec2);    /* open second window */
    DisableMItem(mOpen2);               /* disable open2 item */
    EnableMItem(mClose2);               /* enable close2 item */
}

/*-----------------------------*
 *  Window Menu: Close1        *
 *-----------------------------*/

Close1()
{
    CloseWindow(WindPtr1);              /* close this window */
    EnableMItem(mOpen1);                /* enable open1 item */
    DisableMItem(mHide1);               /* disable these two... */
    DisableMItem(mClose1);
}

/*-----------------------------*
 *  Window Menu: Close2        *
 *-----------------------------*/

Close2()
{
    CloseWindow(WindPtr2);              /* close window 2 */
    EnableMItem(mOpen2);                /* enable open2 item */
```

```
    DisableMItem(mClose2);              /* disable close2 item */
}

CloseW()                        /* close whichever window was clicked */
{
    if (GetWRefCon(EventRec.wmTaskData) == 1)
        Close1();
    else
        Close2();
}

/*-----------------------------*
 *      Do Menu Selection      *
 *-----------------------------*/

DoMenu()
{
    switch (EventRec.wmTaskData) {
    case mAbout:    About();        break;
    case mOpen1:    Open1();        break;
    case mHide1:    Hide1();        break;
    case mClose1:   Close1();       break;
    case mOpen2:    Open2();        break;
    case mClose2:   Close2();       break;
    case mQuit:     QFlag = TRUE;   break;
    }

    HiliteMenu(FALSE, EventRec.wmTaskData>>16);
}

/* ShutDownTools()  -- insert from MODEL.C */

/*-----------------------------*
 *          Main               *
 *-----------------------------*/

main()
{
    StartUpTools();             /* Start toolsets */
    PrepDeskTop();              /* Prepare desktop and menus */

    QFlag = FALSE;
    EventRec.wmTaskMask = 0x00001fff;

    while (!QFlag) {
        Event = TaskMaster(0xffff, &EventRec);
        switch (Event) {
        case wInMenuBar:    DoMenu();   break;
        case wInGoAway:     CloseW();   break;
        }
    }

    ShutDownTools();            /* Shutdown all tools started */
    exit(0);
}
```


Program 9-3. Pascal Source for MONDO.PAS

```
{ *-----------------------------------------------*
  *                 MONDO.PAS                     *
  *   Desktop Application in TML Pascal (v1.01)   *
  *-----------------------------------------------* }

{ **NOTE**  This is not a complete program.  Merge sections
            from the MODEL.PAS program in Chapter Six
            where indicated. }

PROGRAM MondoP;

USES    QDIntF,
        GSIntF,
        MiscTools;

CONST   mAbout  = 256;          { Menu item IDs }
        mOpen1  = 257;
        mHide1  = 258;
        mClose1 = 259;
        mOpen2  = 260;
        mClose2 = 261;
        mQuit   = 262;

{ *-----------------------------*
  *      Global Variables       *
  *-----------------------------* }

VAR EventRec:   EventRecord;    { Taskmaster Structure }
    Event:      Integer;        { Event code }
    UserID:     Integer;        { Our User ID }
    MemID:      Integer;        { Memory allocation ID }
    DPBase:     Integer;        { Direct Page base pointer }
    QFlag:      Boolean;        { Boolean: Quit flag }

    AppleMenu:  String;         { Pull down menu strings }
    WindowMenu: String;
    QuitMenu:   String;

    WindPtr1:   WindowPtr;      { Window port pointers }
    WindPtr2:   WindowPtr;
    W1Title:    String;
    W2Title:    String;

    W1Ctable:   WindowColorTbl;

    WindRec1:   NewWindowParamBlk;
    WindRec2:   NewWindowParamBlk;

    hidebit:    Boolean;

{ PROCEDURE ErrChk -- insert from MODEL.PAS }
{ FUNCTION GetD -- insert from MODEL.PAS }
{ PROCEDURE StartUpTools -- insert from MODEL.PAS }

{ *-----------------------------*
  *  Prepare Desktop and Menus  *
  *-----------------------------* }
```

```
PROCEDURE PrepDeskTop;
VAR
    Height:     Integer;        ( Menu bar heigth (unused) )

BEGIN
    AppleMenu := CONCAT('>>9\XN1\0',
                        '--About This Program...\N256\0',
                        '---\D\0',
                        '>');

    WindowMenu := CONCAT('>>  Window  \N2\0',
                         '--Open Window1\N257\0',
                         '--Hide Window1\DN258\0',
                         '--Close Window1\DVN259\0',
                         '--Open Window2\N260\0',
                         '--Close Window2\DN261\0',
                         '>');

    QuitMenu := CONCAT('>>  Quit  \N3\0',
                       '--Quit\N262*Qq\0',
                       '>');

    Refresh(Nil);                           ( Display Desktop )
    InitCursor;                             ( Show mouse cursor )

    InsertMenu(NewMenu(@QuitMenu[1]), 0);   ( Install menus )
    InsertMenu(NewMenu(@WindowMenu[1]), 0);
    InsertMenu(NewMenu(@AppleMenu[1]), 0);

    FixAppleMenu(1);                        ( Display menu bar )
    Height := FixMenuBar;
    DrawMenuBar;
END;

( *-----------------------------*
  *  Apple Menu: About          *
  *-----------------------------* )

PROCEDURE About;
BEGIN
        ( Does nothing (for now) )
END;

( *-----------------------------*
  *   Window Content Procedure  *
  *-----------------------------* )

PROCEDURE WContent;
BEGIN
    MoveTo($28, $20);
    DrawString('This is a string inside the window.');
END;

( *-----------------------------*
  *  Window Menu: Open1         *
  *-----------------------------* )

PROCEDURE Open1;
BEGIN
    W1Title := ' Mr. Mondo One ';
```



```
WITH W1Ctable DO BEGIN
    FrameColor  := $0000;
    TitleColor  := $0f00;
    TBarColor   := $020f;
    GrowColor   := $00f0;
    InfoColor   := $00f0;
END;

WITH WindRec1 DO BEGIN
    param_length    := SIZEOF(NewWindowParamBlk);
    wFrame          := $dfa0;               ( frame type )
    wTitle          := @W1Title;            ( window title )
    wRefCon         := 1;                   ( window number )
    SetRect         (wZoom, 0, 0, 0, 0);    ( position when zoomed )
    wColor          := @W1Ctable;           ( color table )
    wYOrigin        := 0;                   ( content vert origin )
    wXOrigin        := 0;                   ( content horz origin )
    wDataH          := 180;                 ( height of document )
    wDataW          := 640;                 ( width of document )
    wMaxH           := 180;                 ( height of grow window )
    wMaxW           := 640;                 ( width of grow window )
    wScrollVer      := 4;                   ( vert pixels: scroll )
    wScrollHor      := 16;                  ( horz pixels: scroll )
    wPageVer        := 40;                  ( vert pixels for page )
    wPageHor        := 160;                 ( horz pixels for page )
    wInfoRefCon     := LongInt(nil);        ( no info bar )
    wInfoHeight     := 0;                   ( no info bar )
    wFrameDefProc   := nil;                 ( window definition )
    wInfoDefProc    := nil;                 ( draw info bar )
    wContDefProc    := @WContent;           ( draw interior )
    SetRect         (wPosition, 100, 40, 540, 159);
    wPlane          := -1;                  ( plane )
    wStorage        := nil;                 ( window record address )
END;

WindPtr1 := NewWindow(WindRec1);    ( open first window )
DisableMItem(mOpen1);               ( disable open1 item )
    EnableMItem(mHide1);                ( enable these two... )
    EnableMItem(mClose1);
END;

( *-----------------------------*
  *  Window Menu: Hide1         *
  *-----------------------------* )

PROCEDURE Hide1;
BEGIN
    IF hidebit THEN BEGIN
        ShowWindow(WindPtr1);
        SetMItemName('Hide Window1', mHide1);
        hidebit := FALSE;
    END
    ELSE BEGIN
        HideWindow(WindPtr1);
        SetMItemName('Show Window1', mHide1);
        hidebit := TRUE;
    END;
END;
```

```
( *------------------------------*
  *  Window Menu: Open2          *
  *------------------------------* )

PROCEDURE Open2;
BEGIN
    W2Title := ' Mr. Mondo Two ';

    WITH WindRec2 DO BEGIN
        param_length    := SIZEOF(NewWindowParamBlk);
        wFrame          := $dfa0;               ( frame type )
        wTitle          := @W2Title;            ( window title )
        wRefCon         := 2;                   ( window number )
        SetRect         (wZoom, 0, 0, 0, 0);    ( position when zoomed )
        wColor          := nil;                 ( color table )
        wYOrigin        := 0;                   ( content vert origin )
        wXOrigin        := 0;                   ( content horz origin )
        wDataH          := 180;                 ( height of document )
        wDataW          := 640;                 ( width of document )
        wMaxH           := 180;                 ( height of grow window )
        wMaxW           := 640;                 ( width of grow window )
        wScrollVer      := 4;                   ( vert pixels: scroll )
        wScrollHor      := 16;                  ( horz pixels: scroll )
        wPageVer        := 40;                  ( vert pixels for page )
        wPageHor        := 160;                 ( horz pixels for page )
        wInfoRefCon     := LongInt(nil);        ( no info bar )
        wInfoHeight     := 0;                   ( no info bar )
        wFrameDefProc   := nil;                 ( window definition )
        wInfoDefProc    := nil;                 ( draw info bar )
        wContDefProc    := nil;                 ( draw interior )
        SetRect         (wPosition, 120, 50, 560, 169);
        wPlane          := -1;                  ( plane )
        wStorage        := nil;                 ( window record address )
    END;

    WindPtr2 := NewWindow(WindRec2);    ( open second window )
    DisableMItem(mOpen2);               ( disable open2 item )
    EnableMItem(mClose2);               ( enable close2 item )
END;

( *------------------------------*
  *  Window Menu: Close1         *
  *------------------------------* )

PROCEDURE Close1;
BEGIN
    CloseWindow(WindPtr1);              ( close this window )
    EnableMItem(mOpen1);                ( enable open1 item )
    DisableMItem(mHide1);               ( disable these two... )
    DisableMItem(mClose1);
END;

( *------------------------------*
  *  Window Menu: Close2         *
  *------------------------------* )

PROCEDURE Close2;
BEGIN
    CloseWindow(WindPtr2);              ( close window 2 )
    EnableMItem(mOpen2);                ( enable open2 item )
    DisableMItem(mClose2);              ( disable close2 item )
END;
```



```
PROCEDURE CloseW;               ( close whichever window was clicked )
BEGIN
    IF GetWRefCon(WindowPtr(EventRec.TaskData)) = 1 THEN
        Close1
    ELSE
        Close2;
END;

( *------------------------------*
  *       Do Menu Selection      *
  *------------------------------* )

PROCEDURE DoMenu;
BEGIN
    CASE LoWord(EventRec.TaskData) OF
        mAbout:     About;
        mOpen1:     Open1;
        mHide1:     Hide1;
        mClose1:    Close1;
        mOpen2:     Open2;
        mClose2:    Close2;
        mQuit:      QFlag := TRUE;
    END;

    HiliteMenu(FALSE, HiWord(EventRec.TaskData));
END;

( PROCEDURE ShutDownTools -- insert from MODEL.PAS )

( *------------------------------*
  *            Main              *
  *------------------------------* )

BEGIN
    StartUpTools;               ( Start toolsets )
    PrepDeskTop;                ( Prepare desktop and menus )

    QFlag := FALSE;
    hidebit := FALSE;
    EventRec.TaskMask := $00001fff;

    REPEAT
        Event := TaskMaster(-1, EventRec);
        CASE Event OF
            wInMenuBar: DoMenu;
            wInGoAway:  CloseW;
        END;
    UNTIL QFlag;

    ShutDownTools               ( Shutdown all tools started )
END.
```

Chapter Summary

The following tool set functions were referenced in this chapter.

Function: $020E
Name: WindStartUp
Starts the Window Manager
Push: UserID (W)
Pull: Nothing
Errors: None

Function: $030E
Name: WindShutDown
Shuts down the Window Manager
Push: Nothing
Pull: Nothing
Errors: None

Function: $090E
Name: NewWindow
Creates a window on the DeskTop
Push: Result Space (L); Window Record (L)
Pull: Window Pointer (L)
Errors: $0E01, $0E02

Function: $0B0E
Name: CloseWindow
Closes a window, removing it from the DeskTop
Push: Window Pointer (L)
Pull: Nothing
Errors: None

Function: $120E
Name: HideWindow Hides a window, making it invisible
Push: Window Pointer (L)
Pull: Nothing
Errors: None

Function: $130E
Name: ShowWindow
Displays a previously hidden window
Push: Window Pointer (L)
Pull: Nothing
Errors: None

Function: $1D0E
Name: TaskMaster
Tracks mouse, menu, and window events
Push: Result Space (W); Event Mask (W); Event Record (L)
Pull: TaskCode (W)
Errors: $0E03

Function: $29OE
Name: GetWRefCon
Returns the value of a window wRefCon parameter
Push: Result Space (L); Window Pointer (L)
Pull: Window's wRefCon (L)
Errors: None

Menu Item Calls

Function: $300F
Name: EnableMItem
Enables a dimmed menu item
Push: Menu Item's ItemNum (W)
Pull: Nothing
Errors: None

Function: $310F
Name: DisableMItem
Dims, or disables, a menu item
Push: Menu Item's ItemNum (W)
Pull: Nothing
Errors: None

Function: $3A0F
Name: SetMItemName
Changes the name of a menu item
Push: Pascal String (L); Menu Item's ItemNum (W)
Pull: Nothing
Errors: None

QuickDraw II Calls

Function: $3A04
Name: MoveTo
Moves the graphics pen to a specific coordinate
Push: Horz Position (W); Vert Position (W)
Pull: Nothing
Errors: None