Toolbox Programming in GSoft BASIC

Partial Draft

by Eric Shepherd
Based on Original Work by Mike Westerfield

May 1999 1.0.0

Table of Contents

Lesson 0 – Before We Start

About This Course

This is a self-study course designed to help you learn to program the Apple IIGS toolbox in GSoft BASIC. This course is based on the Toolbox Programming in ORCA/C course that's been available for some time. The original course was written by Mike Westerfield; the GSoft BASIC conversion of the course and the corresponding materials was done by Eric Shepherd.

Before we get started, I want to give you the standard spiel about how you should use this course if you want to get anything out of it other than spending some time at the computer instead of the local bar. Of course, a lot of books you have read in the past started out with just this sort of advice, and by now you are probably practiced at skipping ahead. Before you do though, could you stop to consider one minor point? If I don't know how to help you learn to program the toolbox, why did you buy this book? And if you aren't going to take the best advice I can give you, why bother spending time here when you could be watching baseball at the local pub? In some ways, this is the most important part of the whole book!

Programming is a skill. Programming is not an abstract thought process, nor is it an inborn talent, nor is it something only kids born with video games in their crib can master. Early experience doesn't hurt, you understand, but it also isn't essential. Programming is a skill, like riding a bike, driving a car, talking, playing tennis and reading. Like all skilled activities, to be good at programming, you have to learn a few fundamentals and practice a lot. You have to practice while you learn the fundamentals, practice some more after learning the basics to firm up what you learned, practice still more to get good at programming, and keep on practicing to stay sharp. If you stop programming, even for a month, you will notice the difference when your fingers are again permitted to caress the keyboard.

This course is designed with this fundamental truth in mind. I've put all sorts of sample programs in this course. All of them are also on the disks that come with the course, so you don't have to type them in. You should run each and every one of these sample programs. I've also peppered the course with problems. Every lesson after this one has some real programming problems at the end of some of the sections. I'm not going to waste your time with the easy to grade make-work questions you find in a lot of fluff courses, like listing the twelve kinds of controls. Are there twelve? Who cares! Instead, the problems are real programming problems, problems you should be able to solve in the form of a working program after reading the lesson. If you can't, read the lesson again!

A few years ago (OK, maybe more than a few) I was at the University of Denver slugging away at a master's degree in physics. I took two courses there that are forever burned in my neural network. The first was a course that reviewed undergraduate classical mechanics and electricity and magnetism, two very tough one-year courses. The second was a year long graduate quantum mechanics course. In both courses, I had the good fortune to be taught by Dr. Tuttle, the hardest, toughest, most slave-driving professor I've ever had – and also the best. Dr. Tuttle assigned weekly homework problems which took us an average of 15 hours to finish. If we got stuck, which happened frequently, we were invited to stop by her office anytime we wanted and borrow a chunk of her chalk board. There we would struggle for minutes or even hours until we could work the problem. When we left, there was only one condition. The only

thing we took with us was chalk dust and a better trained neural network. Once we left, we scurried back to the lab to rework the problem before we forgot what to do. But do you know what? It worked. We *remembered* those lessons!

Sometimes you are just going to get stuck on a problem in this course. That's OK; it happens to everyone. As long as you're getting stuck once or twice a lesson, don't sweat it. You can find solutions to all of the problems on the disks. I hope your goal is to learn toolbox programming, though, not to learn how to read my solutions. Once you finish looking at the solution, close the file and start over. Write your own version of the solution. Sure, it's hard. Sure, you think you will remember what you read about. But remember what I said a moment ago? Programming is a skill. You really won't learn much by just reading my solutions. You will, however, learn quite a bit by writing a program yourself.

If you are good enough to work a problem without looking at my solution, good for you. Look at the solution anyway. This may seem like a waste of time, but it really isn't. Any programmer, no matter how much he knows or how experienced he gets, will learn something new by reading a program written by someone else. You should take advantage of all of that source code, reading it carefully to see what tricks you can pick up.

To summarize, here's what you should do:

1. Read each lesson, running the sample programs as you go.
2. Try working each problem on your own. If you don't know where to start, or can't quite get it to work, go back and reread the lesson. Try very hard to solve the problem on your own.
3. If you get stuck, look at my solution. Study it carefully. Then close the file, and without looking at it again, write your own solution to the problem.
4. If you solve a problem without looking at the solution, that's great. Look at the solution anyway to see what you can learn from it.

What You Should Already Know

This is a great first course in programming the toolbox, but programming the toolbox is not the first thing you should try to do on your computer. You probably know that, but I thought I should take a moment to tell you what I'm assuming you already know.

First, you should be pretty comfortable with your computer. You should already know how to do basic things like format disks, check for bad blocks, make backup copies, and so forth. If you don't you can find the information you need in the book that came with your computer.

You also need to know how to use desktop programs. I'm assuming you already know how to use a menu bar, how to load and save files, how to print, and how to manipulate windows and dialogs. Again, if you're new to the Apple IIGS, and you are trying to come up to speed in a hurry, the book Apple sent with your computer is a good place to start.

Finally, and most important, you need to know the GSoft BASIC language. Learning the toolbox is a big job. It's not insurmountable, but it is a big job. Learning GSoft BASIC is also a big job. Put these two big jobs together and try to learn GSoft BASIC and the toolbox at the same time, and you more than double the chance you will fail. If you don't know GSoft BASIC, stop. Put this book down. Try the course "Learn to Program With GSoft BASIC" first. When you finish, come back to this book. You can skimp on prior knowledge about the Apple IIGS

and its desktop interface, but you just can't use this course effectively unless you are already comfortable with GSoft BASIC.

Of course, you may wonder just how comfortable you really need to be. Basically, you should have a good working knowledge of the language, through records and pointers. The toolbox relies very heavily on records and pointers, so you'll see a lot of them. You should also be familiar with linked lists and recursion. You'll see both of these in this course, although you don't have to be an expert at either one.

What You Must Have

Other than patience, time and some prior knowledge, you need three things to work all of the problems in this course. These three things are the computer, this book, and the GSoft BASIC language.

Your Apple IIGS computer needs to have enough memory and disk space to handle the course. The exact amount of memory you need will depend a lot on what version of the operating system you are using; how many and what kind of drivers, desk accessories, and Inits you have installed; and how large a RAM disk you have allocated. As I write this, the latest operating system is 6.0.1. With system disk 6.0.1, 1.25 megabytes (M) of memory is just barely enough on a ROM 01 Apple IIGS, while 1.125M works if you have a ROM 3 machine. If you aren't sure which ROM version you have, boot your computer and look at the very first thing that appears on the screen – right at the bottom you will find the ROM version. If you have ROM 3, the computer comes with 1.125M of memory, so you're in great shape. If you have ROM 01, you need to check to make sure you have at least 1M on a memory expansion card.

You need to have at least two disk drives, one 3.5 inch floppy drive for GSoft BASIC and one other disk drive to save programs. That second disk drive can be anything, but if it isn't a hard disk, you might want to ask Santa for one real soon. A hard disk will make programming a whole lot easier. You should have enough blank 3.5 inch floppy disks to make a backup of GSoft BASIC, and several blank floppies that fit your second drive to hold your own programs.

This book is completely self-contained in the sense that it has all of the information you need to know about the Apple IIGS Toolbox to work all of the problems and understand all of the solutions. You'll also need your copy of GSoft BASIC and the manual that came with it.

Other Useful Things

Getting by with the basics from the last section is possible, but frankly it isn't easy. A hard disk is a very important addition to your computer. If you don't have one, please take a look at the prices for hard disks in the latest issue of some Apple II newsletter, or your favorite online resource. Good Apple IIGS-compatible hard disk drives can be had very inexpensively, and it's well worth the investment.

If you started with a keyboard in your crib or have spent the last ten years glued to a CRT screen you probably won't need a printer. Frankly, I don't use mine much at all. If you are more comfortable with a pen and paper than with keys and dots of colored phosphor, though, a printer is almost as nice as the hard disk.

On the software side, you might want to consider a programmer's CAD tool, like Design Master. You may not even know what that is yet. We'll get to it in time, and once we do get around to telling you what a programmer's CAD tool is, you may want to run right out and buy one. They cost about $60 to $100 from mail order houses, so start saving your pennies now. On the other hand, you don't actually have to have one to do desktop programming.

I've saved the most important thing for last. At last count, the Apple IIGS toolbox is a collection of 32 tools – well over 3000 different subroutine calls. This course teaches you how to use the tools, and gives you a good grasp on the most important tools and tool calls. There is no way I can teach you everything there is to know about each and every tool call, though. I can't even list all of them. The Apple IIGS toolbox reference manuals, which do list all of the tool calls, are a four volume set. For an Apple IIGS programmer, the toolbox reference manuals serve the same purpose that a good dictionary does for a writer. You can do a lot of writing without a dictionary, and you can do a lot of toolbox programming from the information you will find in this course. On the other hand, there's a lot more out there. Before you start writing your own toolbox programs from scratch, you really must get all four volumes of the toolbox reference manual. Please don't kid yourself into thinking you can get by without them. Appendix A of this course is a mini-reference manual that covers all of the tool calls we use in the course, and it does fine for the course itself – but there's a world of other calls out there waiting to be tapped. I would recommend that you try to get the toolbox reference manuals by about the middle of this course, when you will probably start to write your own programs that aren't given as samples.

The toolbox reference manuals I've talked about here, plus several more handy reference books, are described more completely in Appendix C. If you're not quite sure what the books are or where to get them, you can look there for details.

Lesson 1 – Current Events

Goals for This Lesson

This lesson shows you how to organize a desktop program. It covers starting the essential tools, setting up an event loop, and the basics of using an event loop to read the mouse and keyboard. Along the way, we'll start learning how to use the toolbox reference manuals. If you don't have the *Apple IIGS Toolbox Reference*, you can use Appendix A, which contains all of the tool calls from this course in pretty much the same format.

By the end of this lesson, you should understand how to start and shut down the tools, how to set up and use an event loop, and the basics of how to handle keyboard events and read the position of the mouse.

Starting the Tools

The Apple IIGS toolbox is really just a collection of subroutines designed to make it easy (or at least easier!) to write programs with windows, menu bars, and all the other jazzy accouterments of the standard Apple interface. It contains things like `SetSolidPenPat`, a subroutine that tells the graphics program what color to use when it draws things; and `NewWindow`, which opens a new window. These subroutines are collected in tools. QuickDraw II, for example, contains most of the low level graphics subroutines for drawing lines and rectangles, selecting pen patterns, and drawing text on the screen. `SetSolidPenPat` is a part of QuickDraw II. `NewWindow` is a part of the Window Manager, which has the subroutines you use to create windows, change things like the window title, resize windows, or bring a window from the back of the screen to the front.

The complete toolbox reference is made up of *Apple IIGS Toolbox Reference*, volumes 1 to 3, and *Programmer's Reference for System 6.0.1*. All together, these books weigh over 16 pounds. There are 3052 pages, covering 1226 different tool calls grouped into 32 different tools. Obviously, that's a lot of stuff. In fact, there's so much stuff that the Apple IIGS tools aren't actually all inside of your Apple IIGS. Apple crammed quite a few tools into the 128K ROM of the original Apple IIGS, and even more into the 256K ROM of the new Apple IIGS, but there are still quite a few tools on the system disk, tucked away in a folder called Tools. Apple makes mistakes, too, and they even fix them. Rather than sending out a new ROM each time they fix a minor bug, Apple creates tool patches, which are also in the system disk. These tool patches are in the Inits folder. To use the tools, you have to have these inits and RAM based tools available. The easiest, quickest, and most reliable way to make sure you have everything you need is to boot from Apple's system disk (or the one that comes with your language). If you are running from a hard disk, be sure to use Apple's installer to put the operating system on your disk. It's free, it works, and it makes real sure you have all of the files you need.

Each of the tools in the Apple IIGS toolbox needs a little work space for variables. The RAM based tools need to be loaded from disk. Some of the tools need to know just exactly how you want to start them; QuickDraw II, for example, wants to know right away if you want to use 320 mode graphics or 640 mode graphics. When you hear someone talk about "starting the tools,"

they are talking about loading the RAM based tools, finding direct page work space for the tools that need it, and making a call to each tool to tell it to wake up and start doing something useful.

There are a lot of ways to start the tools, but it truly is a bit of a pain. Later, we'll get into a lot of detail about just what you need to do to start the tools, but for now we're going to use a shortcut. The disks that come with this course include a user tool set, UserTool003, that provides some useful utility functions for GSoft BASIC programming. Among these is a subroutine called `StartDesk` that will start up most of the tools we will need in this course. You pass a single parameter, telling it whether you want to start in 640 mode or 320 mode, and it does all of the dirty little chores needed to get things going.

```
dim userID as Integer
LoadLibrary 3
userID = MMStartUp
UtilStartUp(userID)
StartDesk(640)
```

`LoadLibrary` asks GSoft BASIC to load up the user tool set number 3 (this course's Utility tool set). The `MMStartUp` function gets your application's user ID from the Apple IIGS Memory Manager. The Memory Manager keeps track of what parts of memory are being used by whom. The `UtilStartUp` subroutine starts up the UserTool003 "Utility" tool set. Once you've done these things, the `StartDesk` subroutine from the user tool set does the rest of the work.

I can still hear my mother's cheerful, chirping voice floating out the door from my idyllic youth. "You think this is a barn, kid?" she would holler. "Close that door behind you!" Dear old mom. Well, the same is true for the tools. If you open them, close them behind you when you're done. Using UserTool003, all of the tools started with `startdesk` can be closed by `EndDesk`:

```
EndDesk
UtilShutDown
UnloadLibrary 3
```

We also shut down the Utility user tool set by calling its `UtilShutDown` subroutine.

Just as dear old mom had a reason for gently reminding me to close the door, there's a good reason for closing the tools, too. Closing the tools tells each tool in turn to let loose of any memory it has allocated, so there's plenty left for the next program you run. Closing the tools also stops certain things called interrupts. If you don't know what they are, don't fret – but if you forget to shut down the tools, these things can cause your computer to crash. Crashing doesn't do any real harm, but you do have to reboot, which takes up valuable time that you could have used for mowing the lawn. There are also a lot of soft switches in the Apple IIGS that tell the computer to behave in certain ways, like to display a particular graphics screen. Shutting down the tools reverses the critical soft switches. In short, it's pretty important to remember to shut down the tools.

The Event Loop

When Apple released the Lisa back in 1983, they did something pretty radical. Apple said that programmers should work hard to give the end user a piece of software that was easy to use,

and could work the way the user wanted it to, instead of the traditional approach of making the user learn exactly how and in what order the program wanted him to do things. With the Apple interface, unless the program is doing something that takes a lot of time, you should always be able to press a key, move the mouse, click on a button, or pull down a menu. Always.

Well, programmers just weren't used to dealing with a program that gave the user so many options. How could all of this be done? The answer, once you know it, is pretty simple. It's called an event loop. The event loop is just an infinite loop that you drop into right after you initialize your tools and do any other chores you need to get done before you let the user start doing something with the program. In this infinite loop, the program checks to see if a key has been pressed, then checks to see if a menu has been pulled down, then to see if a desk accessory has been started, and so forth. The program is waiting for an event to happen. When something happens, the program goes off for a bit, taking whatever action is necessary, and then comes back to the even loop. Ignoring the particular computer we're on and just thinking about the idea, an event loop looks something like this:

```
done = False
do while not done
   if KeyPress then
     HandleKeyPress
   else if MouseDown then
     HandleMouseClick
   else if DAClosed then
     HandleDAClose
   else if MenuSelected then
     HandleMenu
   end if
loop
```

Rumors start in strange ways. Back when Apple first started asking programmers to write for the Lisa, a lot of folks rushed right over, trying to move their spaghetti coded programs to the jazzy new machine. The problem was that a lot of these programs were turned inside out compared to the way they needed to work. The program would check for a key press whenever and however it felt like it. Rather than do the job right, these adventurous souls would try to move their program over with as few changes as possible. It just didn't work very well. Asking for help, they would be told over and over that the program had to be written around an event loop, and they would struggle more and more trying to reshape the old program to the new ideas. Of course, programmers never make mistakes – just ask one – so the obvious conclusion resounded through the programming community: "Event loops are hard!"

Wrong. Thanks for playing at the game of programming. Better luck next time.

Event loops are easy. Like structured programming, event loops are something that makes so much sense that once you get used to them, you wouldn't write a program any other way, even on an IBM PC. What's hard is taking an old program that wasn't written with an event loop and trying to shoehorn it into a program that has to have an event loop. Starting from scratch on a new program, though, an event loop gives you a great start on organizing your program around a single, top-level controller.

The Event Record

Events and the event loop are so important on the Apple IIGS that an entire tool has been created that does nothing but handle events. Cleverly enough, this tool is called the Event Manager. To write an event loop, you create a loop that calls `GetNextEvent`. You pass `GetNextEvent` a structure, called an event record. `GetNextEvent` fills in the information in the event record, like what event occurred (if any) and where the mouse is. The current mouse position is always returned, even if the user is just jerking the mouse back and forth to pass time and watch the pretty arrow move. Since your program may have other things to do if the user isn't busy, `GetNextEvent` returns right away, even if nothing is happening. It returns a zero if there is nothing pressing for your program to do, and one if there is something you need to handle.

Putting this knowledge to work, we can write the outline of our first desktop program. The main part of the program looks like this:

```
GetNextEvent;
userID = MMStartUp
LoadLibrary 3
UtilStartUp(userID)
StartDesk(640)
done = FALSE;
do while not done
  if GetNextEvent($nnnn, myEvent) then
    HandleEvent
  end if
loop
EndDesk
UtilShutDown
UnloadLibrary 3
```

`HandleEvent` is some hypothetical subroutine that does all of the hard stuff; in a moment, we'll replace this hypothetical subroutine with something a bit more down to earth. `GetNextEvent` itself has to tell you if something happened, so Apple implemented it as a function. In addition to returning `TRUE` if something important happened, and `FALSE` if there's nothing for you to worry about, the Event Manager fills in the event record. The event record is packed with all sorts of information. In GSoft BASIC, an event record looks like this:

```
type eventRecord
  eventWhat as integer
  eventMessage as long
  eventWhen as long
  eventWhere as point
  eventModifiers as integer
end type
type eventRecPtr as pointer to eventRecord
```

Note If you actually look at the definition of an event record in the GSoftTools.int file, you will find some other fields after `eventModifiers`. Technically, those are part of a `TaskMaster` record.

We'll talk about those fields, and why they are in the event record, when we start using `TaskMaster`. □

Let's stop and take a close look at just what the Event Manager is telling us. The first item in the event record is `eventWhat`. The `eventWhat` field tells us what kind of event has occurred. Here's a list of the various events the Event Manager can return.

Name	Value	Description
`nullEvt`	0	Nothing happened. The mouse may have moved, but the mouse button hasn't been pressed, no key on the keyboard has been pressed, and the Event Manager didn't find any other sign that the person using the program is doing anything.
`mouseDownEvt`	1	The button on the mouse was pressed. This event occurs once, when the mouse button is originally pressed down. As long as the button is held down, you won't get another mouse down event.
`mouseUpEvt`	2	The button on the mouse was down, and has been released.
`keyDownEvt`	3	A key on the keyboard has been pressed.
`autoKeyEvt`	5	A key was pressed, and a `keyDownEvt` reported, but the key was held down. This event is reported periodically for as long as the key is held down.
`updateEvt`	6	A portion of a window needs to be redrawn (updated).
`activateEvt`	8	A window has been activated.
`switchEvt`	9	Switch events aren't used by the current system. Apple put them in just in case they ever wanted to generate a switch event, presumably for something like an application switcher or MultiFinder.
`deskAccEvt`	10	This event happens right before the CDA menu is brought up.
`driverEvt`	11	Device drivers can post these events. They really don't concern us.
`app1Evt`	12	You can define your own events. If you need one, this event code and the three that follow are for your use.
`app2Evt`	13	Application-defined event.
`app3Evt`	14	Application-defined event.
`app4Evt`	15	Application-defined event.

Table 1-1: Event Manager Events

Some of these probably don't make much sense, yet. We'll look at several of these events in more detail as we learn more about the toolbox.

The `message` field tells more about the event that occurred. For example, the `eventMessage` field for a `keyDownEvt` contains the ASCII character code for the key that was pressed. The `message` field is different for each event, so we'll wait to look at this field in detail until we start looking at the individual events.

When you start the Event Manager, your Apple IIGS starts a clock that counts off each 1/60th of a second. Each time the clock increments, it adds one to the tick count. The tick count is recorded in the `eventWhen` field. You can use this value to tell when a particular event occurred, or more commonly, how much time elapsed between two events. For example, when a program looks for a double-click, the tick count is used to see if the two clicks on the mouse button were close enough together to be considered a double click. The `eventWhen` field is set even if no event occurred, so you can use it as a timer inside your event loop.

Like the `eventWhen` field, the `eventWhere` field is always set, even if the event is a null event. The `where` field is a point; it gives the position of the mouse in global coordinates. A point is a record with two integer values, h and v. The value h is the horizontal position, starting from 0 and counting from the left of the screen. The vertical coordinate, v, counts from the top of the screen to the bottom. Global coordinates are the ones you would expect, with the 0,0 point at the top left of the screen. When we start to use windows, you'll learn about local coordinates. We'll take a closer look at global coordinates a little later in this lesson when we look at mouse events.

The `eventModifiers` field is a series of bit flags that give more information about the state of the computer. The `eventModifiers` field tells us if the mouse button is down, if the shift key is being pressed, and so forth. Figure 1-1 shows the modifier flags and what each bit is used for. It's worth taking a close look at figures like 1-1 – you find out some peculiar and interesting things about the toolbox when you do. For example, take a look at bits 6 and 7. Here we have two innocent bits that tell you when mouse button 0 and mouse button 1 are down, so you can tell which mouse button is being pressed. Well, that seems useful, but *my* mouse only has one button. Of course, Apple's desktop bus system that you use to connect your keyboard and mouse to the computer isn't limited to just the keyboard and mouse Apple sends with the computer. Apple's programmers put a lot of work into making the toolbox flexible enough to handle lots of options, many of which they don't intend to use themselves. That's a good design philosophy, but you need to keep alert for things like the two mouse buttons so you know what is available.

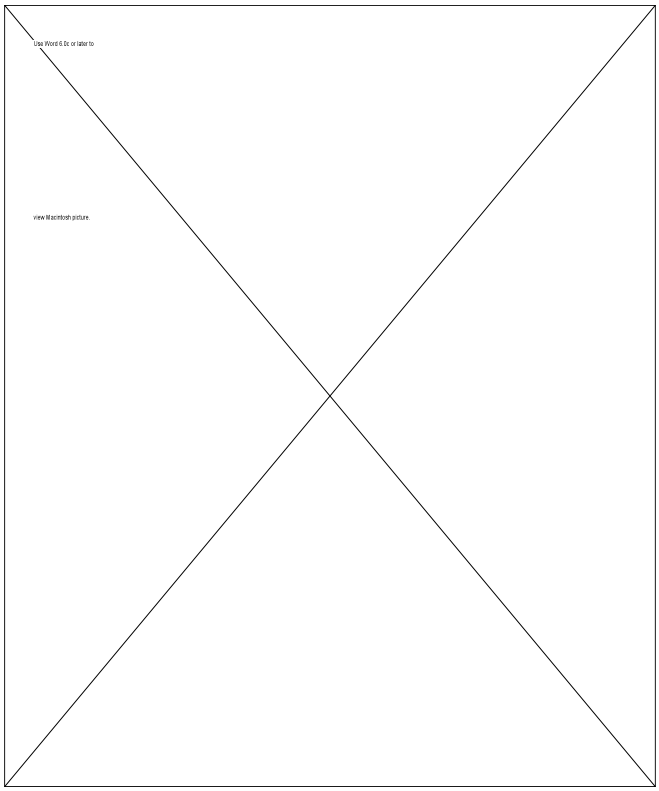

Figure 1-1: Modifier Flags

We've seen that `GetNextEvent` returns a value, which we have seen is `TRUE` if some event occurred, and `FALSE` if there was no event. We've seen how the event record is used to return lots of information about what event occurred – and some information, like the mouse position and modifiers, even when an event didn't occur. Looking back at `GetNextEvent`, though, there is one other parameter we haven't talked about. In some programs, you just may not need a particular event. The event mask parameter tells the Event Manager which events you want to see, and which events you want it to dump in the bit bucket. While this can be useful in some situations, it's really just as easy to ignore the events you don't want to handle in your event loop. For now, we'll use the value `everyEvent`, which tells the Event Manager to report each and every event it finds.

Our First Executable Program

Well, we can finally create a complete desktop program. This one doesn't to much – it just brings up the desktop and waits for us to press a key, but hey, it's a start. Even though the program doesn't do much, we've covered a lot of ground, and it's important to see some of this new knowledge put to use.

There are a couple of things in this program you may not have expected; we'll talk about those in a moment. The event loop itself, though, is exactly what we've been leading up to in the last few pages.

```
!
! Keypress
!
! Start the desktop environment and wait for a keypress.
!

dim done as Integer
dim myEvent as EventRecord
dim userID as Integer

!
! Main program
!

userID = MMStartUp
LoadLibrary 3
UtilStartUp(userID)
StartDesk(640)
if toolError <> 0 then
  print "Unable to start up desktop."
  end
end if

done = FALSE

do while not done
  if GetNextEvent(everyEvent, myEvent) then
    if myEvent.eventWhat = keyDownEvt then
      done = TRUE
    end if
  end if
loop

EndDesk
UtilShutDown
UnloadLibrary 3
end
```

Listing 1-1: Wait for a Key Press

You can either type the program in or load it from the disks that come with the course. (On the disk, the file is in a folder called Lesson.1. The source code is in the file Prog.1.1.bas.) Either way, take the time to run the program, either from the GSoft BASIC shell, or from the ORCA Shell (note: the book was written using the ORCA Shell because mixed-case source code is easier to read; however, all the programs will work in either environment). In either case, typing "RUN Prog.1.1.bas" will run the program.

Problem 1-1. Change the program so it stops when the mouse is released (not when it is pressed!).

 Note Like every other problem in this book, the solution is on the disks that come with the course. The source code is divided up into lesson folders that match the lesson you are reading; in this case, you want the folder called Lesson.1. It's on the disk labeled Disk 1. The

source itself is in a file called Prob.1.1.bas. The source code for Problem 1-2 will be in a file called Prob.1.2.bas, and so forth. □

Keyboard Events

Detecting a key press was pretty easy; now let's see how we find out which key was actually pressed. The ASCII character code for the key that was pressed is stored in the `eventMessage` field. Taking a look at the `eventMessage` field, you see that it's a long integer, but an ASCII character only uses 7 bits. The ASCII character is actually stuffed in the least significant byte of the long integer, and the other three bytes are undefined. The eighth bit of the byte that contains the character is set to 0.

Putting all of this together, we can grab the character from the `eventMessage` field like this:

```
dim c as Char
c = myEvent.eventMessage
```

Note that we assign the `eventMessage` to a variable of type Char (which is a single byte; look in GSoftTools.int and you'll find this definition). This eliminates the upper three bytes of the `eventMessage` value. This may seem like an unimportant technical detail, especially since they are actually zero. On the other hand, keep in mind that I said they were undefined, not that they were zero. The word undefined comes right from the toolbox reference manual. When you see something that's undefined in the toolbox, that means it isn't used now, but *it might be used for something at some point in the future!* To a careful programmer, there's a significant difference between something that's undefined but happens to be zero, and something that is defined as being zero. To a not so careful programmer, this means job security: they get to go back after each new release of the system disk and make changes to allow for things that used to be undefined and aren't any more.

Whenever the toolbox documentation says something is undefined, write your program so it doesn't matter what is in those bytes. If you do, you'll go through life wondering why I made such a big deal about this point. If you don't, you'll go through life changing your programs each time a new system disk is released. Like so many things in life, it's a choice you get to make for yourself.

The Event Manager can tell you a lot more about a `keyDownEvt` than just what key was pressed. The other information is in the `eventModifiers` field, which you saw a couple of pages back. You can tell whether the shift key or caps lock key is down, whether the apple or option key is being pressed, whether a "1" character came from the keyboard or the keypad, and whether the control key was being held down. For some applications, these differences are crucial, while for others you won't bother to look at the `eventModifiers` field.

I'm going to toss in one more piece of information that is just one of those things you pick up by constantly flipping through manuals. Something that isn't mentioned in the GSoft BASIC manual is the fact that when the super-hires graphics screen is active, you can use the `PRINT` command to draw text on the graphics screen. Here's a program that uses that fact to show what key is pressed and what the modifiers field is set to. It uses a few tool calls we haven't discussed yet; we'll take a look at them in a moment.

```
!
! Keypress
!
! Show keypresses on the desktop
!

dim done as Integer
dim myEvent as EventRecord
dim userID as Integer

!
! Main program
!

userID = MMStartUp
LoadLibrary 3
UtilStartUp(userID)
StartDesk(640)
if toolError <> 0 then
  print "Unable to start up desktop."
  end
end if

PenNormal
done = FALSE

do while not done
  if GetNextEvent(everyEvent, myEvent) then
    if myEvent.eventWhat = mouseUpEvt then
      done = TRUE
    else if myEvent.eventWhat = keyDownEvt then
      call HandleKeypress(myEvent)
    end if
  end if
loop

EndDesk
UtilShutDown
UnloadLibrary 3
end

!
! HandleKeypress
!
! Handle a key down event.
!
sub HandleKeypress(myEvent as EventRecord)
  dim r as Rect
  dim c as Char

  SetSolidPenPat(black)
  r.h1 = 0
  r.h2 = 100
  r.v1 = 0
  r.v2 = 40
  PaintRect(r)
  SetForeColor(white)
  SetBackColor(black)
```

```
    MoveTo(10,10)
    c = myEvent.eventMessage
    print chr$(c)
    print str$(myEvent.eventModifiers)
  end sub
```

Listing 1-2: Handling a Key Press Event

This lesson is about starting the tools and using the Event Manager, but the tools all have to be used together to create useful programs. This program uses a few calls from QuickDraw II, the graphics package for the Apple IIGS. We won't get a chance to study QuickDraw in detail for quite a while, but you will need to use a few calls here and there.

This program is writing text on the graphics screen. I mentioned that `PRINT` can write text, but it isn't quite as simple as writing text to the text screen. For one thing, the graphics screen doesn't scroll, so we have to clear some space (in case there were already some characters on the screen). QuickDraw's `PaintRect` call will paint all of the pixels in a rectangle; we'll use this to clear part of the screen. There are two pieces of information `PaintRect` needs to fill in the rectangle: the color to use, and the location of the rectangle. `PaintRect` uses something called the pen color to fill in the rectangle; `SetSolidPenPat` sets the pen color to one of the solid colors. You can use a number, like 0 for black or 3 for white, or you can use one of the names from GSoftTools.int. We'll see exactly how you can read the GSoftTools.int file later, but for now, you can trust me about `black` and `white` being properly defined there. Finally, any time you draw on the graphics screen, there are several parameters that effect the way the drawing is done. The `PenNormal` call you see right after the call to `StartDesk` sets up all of those parameters in a "normal" way. You will learn what the parameters are, and what "normal" means, when we go over QuickDraw in detail.

QuickDraw uses two main data structures to deal with locations on the screen, the point and the rectangle. In both cases 0,0 is at the top left corner. For a rectangle, you need to give a top, bottom, left and right side; the actual names used in a rectangle structure are `h1` (left), `h2` (right), `v1` (top) and `v2` (bottom).. In our program, the finished rectangle is passed to `PaintRect` as a parameter.

As with clearing a rectangle, QuickDraw needs to know two things to draw text, other than the text itself. The position of the text will be the current pen position, which we set using `MoveTo`. As with most QuickDraw calls, instead of passing a point as a structure, we pass the two individual values as parameters. The color of the text is set with the `SetForeColor` call, while the color of the background is set with `SetBackColor`.

Since the point of this program is to draw characters and modifiers on the screen, we don't want to use a `keyDownEvt` to stop the program. Instead, the program waits for the `mouseUpEvt` that occurs at the end of a mouse click.

Problem 1-2. Change the program so it reports auto-key events as well as the initial key press. Try the program, changing the repeat key speed from your control panel to see what effect this has.

Mouse Events

Mouse events are pretty simple: you get one event when the mouse button is pushed down, and another when it is released. Each time you call `GetNextEvent` you also get back the current position of the mouse in the event record, whether or not a mouse event has occurred.

The program in listing 1-3 puts this information to use to create a simple sketching program. It also uses one new QuickDraw call, `LineTo`. `LineTo` works pretty much like `MoveTo`, but instead of just moving the pen, it draws a line from the current location of the pen to the new location you pass as parameters to `LineTo`. The plan is to move around until the mouse is pressed, then draw lines until the mouse button is released, again.

Of course, there's the minor matter of where the mouse is at any given time. Desktop programs generally use the arrow symbol to show where the mouse is pointing; the QuickDraw call `InitCursor` sets up the arrow cursor and tells the toolbox to move it around for us. QuickDraw is also smart enough not to draw on top of the arrow. All of this work is done behind the scenes for us; all we have to do is remember to put that one, simple call to `InitCursor` in the program, right after the tools are initialized.

```
!
! Sketchpad
!
! Simple drawing program.
!

dim done as Boolean
dim mouseIsDown as Boolean
dim myEvent as EventRecord
dim userID as Integer

!
! Main program
!

userID = MMStartUp
LoadLibrary 3
UtilStartUp(userID)
StartDesk(640)
if toolError <> 0 then
  print "Unable to start up desktop."
  end
end if

PenNormal
call BlackScreen                        :! Paint the screen black
SetSolidPenPat(white)                   :! Draw in white
SetPenMode(modeCopy)                    :! Draw in copy mode
done = FALSE                            :! We aren't done yet
mouseIsDown = FALSE                     :! Mouse isn't down

do while not done
  if GetNextEvent(everyEvent, myEvent) then
    if myEvent.eventWhat = keyDownEvt then
      done = TRUE
    else if myEvent.eventWhat = mouseDownEvt then
      mouseIsDown = TRUE
    else if myEvent.eventWhat = mouseUpEvt then
      mouseIsDown = FALSE
    end if
  end if

  if mouseIsDown then
    LineTo(myEvent.eventWhere.h, myEvent.eventWhere.v)
  else
    MoveTo(myEvent.eventWhere.h, myEvent.eventWhere.v)
  end if
loop

EndDesk
UtilShutDown
UnloadLibrary 3
end

!
! BlackScreen
!
! Clear the screen to black.
```

```
!
sub BlackScreen
  dim r as Rect

  r.h1 = 0
  r.h2 = 640
  r.v1 = 0
  r.v2 = 200
  SetSolidPenPat(black)
  PaintRect(r)
end sub
```

Listing 1-3: Sketch program

Problem 1-3. Add color to the sketch program. To do this, when a key is pressed, check to see if it is a '0', '1', '2' or '3', and if so, set the pen color to the same value. For any other key, quit.

Color 3 will be pure white, and color 0 will be pure black, but colors 1 and 2 will vary a bit. This is caused by dithering, something you will learn to use to your benefit later in the course. For now, just file it away as an interesting phenomenon.

You can add even more color, handling 16 different colors, by using 320 mode graphics. The only changes you have to make to the program are to change the parameter of `startdesk` from 640 to 320, change the screen width in `BlackScreen` to 320 (this isn't strictly necessary, but is still good form) and allow more keys for the other 12 colors.

Problem 1-4. Write a program that prints the mouse position on the screen. Use this to explore the coordinate system used by the toolbox. (In other words, move the mouse around and make sure you understand what numbers to use to put the mouse in a particular place.)

Using Appendix A

You've learned a lot about the toolbox in this lesson, but it's fair to ask where some of this information comes from. How did I know what all of the flags were in the `eventModifiers` variable? How did I know that I could use `everyEvent` as a parameter to `GetNextEvent`? The rest of this lesson deals with just that issue.

A while back I gave a few statistics about the size of the toolbox reference manuals that the folks at Apple Computer wrote to describe the tools in the toolbox. Listening to the statistics about that four volume tome, you might have wondered how anyone could ever learn all there is to know about the toolbox. The answer, once you know it, is really pretty obvious: no one does. Just as a writer doesn't start by memorizing the Oxford English Dictionary, a toolbox programmer doesn't start by memorizing *Apple IIGS Toolbox Reference*. Instead, just like the writer, a toolbox programmer learns how to find things in the toolbox reference manuals, and learns the basics about how to use the toolbox. This course covers all of the basics about how the toolbox is organized and used, but you will still need the complete set of toolbox reference manuals to be any good at programming the toolbox.

On the other hand, you may not be ready to run right to your phone, call the Byte Works, and order the entire set of reference books for the Apple IIGS. Fair enough. Just keep in mind that someday you will need them, just as a writer knows he will need a good dictionary. Until that

time, you can get by very nicely with Appendix A, which is our abridged toolbox reference manual. Appendix A catalogs all of the tool calls used anywhere in this course. Like a pocket dictionary that covers the most common English words, Appendix A does not try to cover all of the toolbox, just the most commonly used parts. It also doesn't cover all aspects of the tools listed, but if the description of a tool call is incomplete, Appendix A says so very clearly.

I keep comparing the toolbox reference manuals to a dictionary, and I guess that's because the way the two are used is really quit a lot alike. Well, here goes again. One of the little tricks I've been taught for increasing my vocabulary is to keep a good dictionary handy when I read a book that is using words I'm not familiar with. Well, that's also the best way to use Appendix A. Each time you see a tool call you aren't familiar with, flip back to Appendix A and read the description. Among other things, you'll find a lot more information about most of the tool calls than we actually talk about in the text. You'll also get comfortable with how Appendix A is written and organized, plus the sort of information you can find there.

Using the Toolbox Reference Manuals

If you already have the toolbox reference manuals, everything I just said about Appendix A applies to them, too. About the only thing you have to keep in mind is that the toolbox reference manuals list the tool calls using C style parameters, while Appendix A and the GSoftTools.int file list the calls using GSoft BASIC style parameters. It shouldn't be any problem at all to figure out what the GSoft BASIC parameter would look like from what the reference manuals list.

Just in case you will be using the toolbox reference manuals, though, let's take a look at how they are organized. We'll do that by looking at the most complicated tool call we've made so far: `GetNextEvent`. There are two ways to see how to call `GetNextEvent` from the toolbox reference manuals. Since you may like one better than the other, while the next guy may like the other, I'll cover both.

The simplest and most direct way to see how a call is made is to look at the C declaration. Here's the information from page 7-39 of *Apple IIGS Toolbox Reference: Volume 1*.

```
extern pascal Boolean GetNextEvent(eventMask,eventPtr)
Word eventMask;
EventRecordPtr eventPtr;
```

Compare this to the declaration in the GSoftTools.int file:

```
Tool $06, $0A FUNCTION GetNextEvent (%, (eventRecord)) as boolean
```

Other than the fact that the declarations are for different languages, these are relatively similar. Whereas the K&R C style declaration lists the types of the parameters after the declaration itself, the GSoft BASIC declaration lists the types within the parameter list directly. In this case, `eventMask` is an Integer (remember that "%" indicates an integer variable), and `eventPtr` is an `eventRecord`, passed by reference.

The second major source of information is the stack diagram that you will find with each tool call. The stack diagram for `GetNextEvent` looks like this:

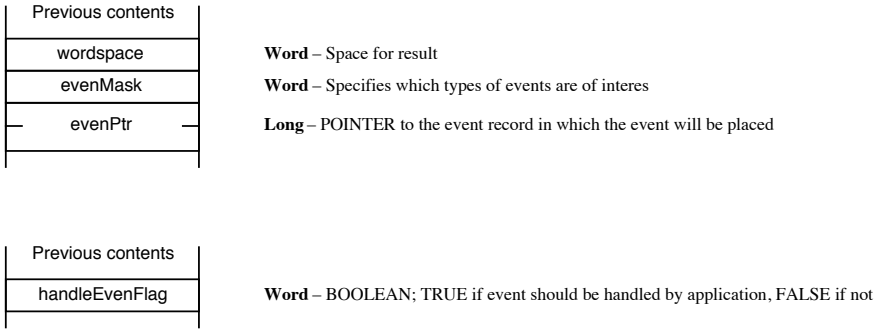

Figure 1-3: Stack Diagram for `GetNextEvent`

There's a wealth of information here, but it takes practice and an understanding of how GSoft BASIC parameters are passed to toolbox functions in order to get much out of the stack diagram. The top diagram shows the stack right before the call to `GetNextEvent`. `GetNextEvent` only has two parameters, but there are three things on the stack. That's because `GetNextEvent` returns a value (it's a function), which needs space reserved for it on the stack. This is handled automatically by GSoft BASIC. On return, this reserved space contains the return value, which is called `handleEventFlag` in the lower stack diagram.

The parameters themselves are read from top to bottom. The first parameter for `GetNextEvent` is the event mask, which is the one that appears first on the stack. The stack diagram only shows how big the parameter is, but usually the parameter's type is pretty obvious. For example, byte, integer, and boolean values all take two bytes on the stack when GSoft bASIC passes them as a parameter, but it's pretty obvious that the event mask isn't a boolean.

One of those subtle little points that can sneak up on you is the way structures are passed. When you call a function and pass the structure itself, GSoft BASIC actually pushes all of the values in the structure on the stack. However, the GSoft BASIC tool declarations in GSoftTools.int all establish these as being passed by reference, so pointers will be used automatically - you don't have to think about it. The only place in this course you'll see an actual structure passed as a value, rather than as a pointer, is in the Font Manager, and I'll be sure to mention that fact when we get there.

Problem 1-5. The Miscellaneous Toolbox has a call called `SysBeep` that makes the familiar error beep. QuickDraw II has a call called `PtInRect` that tests to see if a point is in a given rectangle. Use these calls to create a program that beeps continually any time the mouse is inside of a rectangle that extends from 50 to 150 vertically, and 160 to 480 horizontally. Set the program up so it quits when you press any key.

Summary

In this lesson, we've covered the basics of starting the tools and setting up an event driven program. You've learned how to use the GSoft utility tool set's `StartDesk` and `EndDesk` functions to start up and shut down the tools. You've learned how to use the Event Manager's `GetNextEvent` call to create an event driven program, and how to use the keyboard and mouse from a desktop program. Along the way, you've also started to learn to use other reference

sources, like Appendix A or the *Apple IIGS Toolbox Reference*, the GSoft BASIC reference manual, and the toolbox header files.

Tool calls used for the first time in this lesson:

```
GetNextEvent      InitCursor        LineTo            MoveTo
PaintRect         PenNormal         PtInRect          SetBackColor
SetForeColor      SetPenMode        SetSolidPenPat    SysBeep
```

Lesson 2 – What's on the Menu?

Goals for This Lesson

This lesson deals with creating and using menus. You'll learn how to create a system menu bar, and how to make use of the system menu bar in your programs. In the process, you will learn a new way to handle events, using `TaskMaster` to take care of a lot of detailed work for you, and you'll learn how to create a program that can be used with desk accessories.

Setting Up A Menu Bar

Menu bars are one of the most distinctive features of an Apple program. These days, menu bars have been copied by all sorts of systems, but the pull-down menu bar of the Apple interface is still the champ when it comes to completeness, flexibility, ease of use, and perhaps most important of all, consistency. That last point is one you'll see over and over in the course. Consistency is a valuable and often overlooked trait of the Apple desktop interface. When you can learn how to use a menu bar once, in one program, and instantly understand how to use any menu bar in any program on a Macintosh, Apple IIGS, or even the old Lisa computer... well, that's a big advantage over learning how to use a new user interface each time you pick up a new program, which is exactly what you had to do before Apple created their standard interface for all programs.

Getting back to the topic, this lesson is about menu bars – how to create them and how to use them. Menus are created and manipulated using the Menu Manager, a tool we haven't used up to this point. To understand how to create a menu bar, we'll start by looking at how menus are organized by the Menu Manager. Take a look at the menu bar in Figure 2.1.

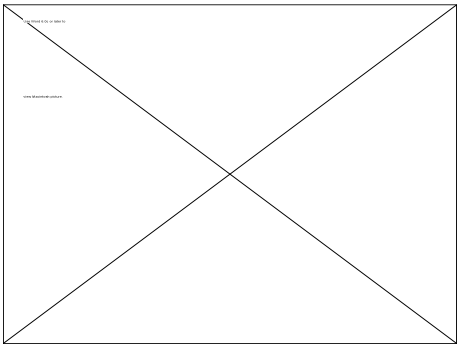

Figure 2-1: Typical Menu Bar

Looking at this menu bar, it's easy to understand how menus are organized. Just as with the physical menu bar you use in a program, the Menu Manager sets up the menu bar in a hierarchy. At the top level you have the menu bar itself. A menu bar is made up of several menus; these are the names and symbols you see written across the top of the menu bar. A menu is more than just the name at the top, though – it is also a list of menu items. At the bottom level, these menu

items make up the menu. So, to review, a typical menu bar consists of the menu bar itself, which contains several menus, each of which contain several menu items.

Later on we'll take a look at some variations on this theme, some of which are pretty easy to add to your programs and some of which take a lot of skill, and probably some assembly language. There is one point I'd like to bring up here, though: so far, we've said absolutely nothing about *where* the menu bar goes. When you think of a menu bar, you no doubt think of the white bar at the top of your screen when you run a desktop program. That's natural enough, since almost all menu bars you will ever see will be right there, at the top of the screen. That menu bar is called the system menu bar, and it's the one all complete desktop programs have. The Menu Manager isn't picky, though, and you can create other menu bars if you want to. You can create several different system menu bars, and switch between them, or you can even put a menu bar in a window. Most of the time the system menu bar is the only one you will use, though, and it's the only one we'll actually use in the programs in this course.

Now that you know how menu bars are organized, let's take a look at how they are created. When you start the Memory Manager, you get a system menu bar for free, whether you end up using it or not. In fact, you saw this in the last lesson. Even though the programs we wrote in Lesson 1 did not use menus, there was still a white menu bar at the top of the screen when your programs ran.

Filling in the menu bar is a multi-step process. You start with a series of calls to `NewMenu`, which takes a menu definition you create, and `InsertMenu`, which places the new menu in the system menu bar. After all of the menus have been created and inserted into the menu bar, you make a few more calls to tell the Menu Manager to do some housekeeping; this is when the Menu Manager figures out how big the menus are, and so forth. You also make a call to the Desk Manager to fill in any New Desk Accessories (NDAs) that the tools can find, and finally, you make a Menu Manager call to draw the new menu bar.

By far the most complicated of these steps is creating the initial record that is passed to `NewMenu` to create a new menu for the menu bar. This record is actually a multi-line character string, with one line for the menu name, and one additional line for each of the items that will be in the menu when you pull it down. Let's start by looking at the GSoft BASIC code to create one of these menus, and then we'll pull it apart piece by piece to find out how all of this works.

```
menuDef = ">>  Edit  \N3"+chr$(13)
menuDef = menuDef + "--Undo\N250"+chr$(13)
menuDef = menuDef + "--Cut\N251"+chr$(13)
menuDef = menuDef + "--Copy\N252"+chr$(13)
menuDef = menuDef + "--Paste\N253"+chr$(13)
menuDef = menuDef + "--Clear\N254"+chr$(13)
menuDef = menuDef + "."+chr$(13)
menuHand = NewMenu(@menuDef)
InsertMenu(menuHand, 0)
```

This example menu doesn't use any advanced features like keyboard equivalents or separators; we'll put those in later in the lesson. Other than those simplifications, though, it's a pretty typical Edit menu – more typical than you probably ever thought, as you'll find out later in this lesson. As you can see, the main part of the work involves creating a long string; we're doing that by successively concatenating each line of the menu string.

The menu string itself is made up of a series of lines. In this program, I used return characters to end each line. In GSoft BASIC, you can get a return character with the function call chr$(13). The Menu Manager also lets you use the null character to end a line, but because of the way GSoft BASIC handles strings internally, it's a little easier for us to use the return character.

Each menu has to have at least two lines, one for the menu itself, and one at the end of the line that just has a period. The Menu Manager looks for that last period to mark the end of the menu list, and if you forget it, the Menu Manager will happily romp and stomp through memory looking for the end of the list. As I'm sure you realize, that's a bad thing.

The line that defines the menu itself is `">>  Edit  \N3"`. This line actually has three separate parts. The first is a two-character sequence, `">>"`. You have to start the menu line with these two characters; the Menu Manager uses them both to mark the start of the menu, and later makes use of the two bytes of memory that the characters occupy for some internal housekeeping.

The second part of the menu string is the name of the menu itself, `"  Edit  "`. You can pick pretty much anything you want for a menu name, as long as it doesn't include a backslash character. It may seem a bit strange to include spaces in the name of the menu, but there's actually a very good reason for including them: when the Menu Manager creates the text for the menu bar itself, it crams all of the characters from all of the menu names together on the menu bar. If you don't put spaces around the name of the menu, all of the menu names get shoved together. The reason for putting the same number of spaces to the left and right of the menu name has to do with the way menu names are highlighted. When you pull down a menu, the Menu Manager highlights the name of the menu. The spaces are a part of that name, so it's a good idea to put the same number of spaces on the left and right sides of the menu name; if you don't, the highlighted menu names will look lopsided. As a general rule-of-thumb, you should use two spaces on each side of the name for 640-mode applications, and one space on each side for 320-mode applications.

The last part of the menu name string is a set of control characters, in this case `"\N3"`. The control character field always starts with the backslash character, which is why you can't use a backslash character as part of a menu name. The backslash character is followed by a series of control codes. We'll spend a lot of time looking at these control characters in detail later, since they let you do all sorts of fun things, like create menu items with dividers, bold text, and so forth. The one thing that is common to all of the menu items, though, is a menu number for the menu name string, and a menu item number for menu item strings. In each case, the number is the character N (to tell the Menu Manager that we're defining a number) followed by the number itself. That number is typed just like you would type an integer value in a GSoft BASIC program. An important note: the control characters are case sensitive, so you do have to use a capital N at the start of the number.

Menu items are defined pretty much the same way the menu name itself is defined. For example, the string for the Clear menu item is `"--Clear\N254"`. Like the menu name string, a menu item string starts off with two characters, but this time we use `"--"` instead of `">>"`.

Looking back, you can see that all of the menu items start off with the same two `"--"` characters. The menu item string also has a name part, but this time we don't need to put spaces around the name. Finally, there is a section for control characters, and again we need to set up a number, this time a menu item number.

When you pull down a menu, you want the menu items to appear in a certain order. It may seem pretty obvious, but just to be complete it's worth pointing out that the menu items in the finished menu will be listed from top to bottom in the same order you put the menu item strings in this master string. While this example shows menu item numbers that step up in a nice order, that isn't actually required; the menu item number is just a reference number, and has no effect whatsoever on the position of the menu item in the finished menu.

With the menu string complete, it's time to see what we do with it. The `NewMenu` tool call takes the menu string as a parameter and returns a menu handle. It turns out that we really don't need the handle very often. The menu handle is handy for tricks like adding and removing menus as the program runs, but for a normal, fixed menu bar, we just don't need it. For that reason, in simple programs, I just use the same local variable for the menu handle on each `NewMenu` call, and throw the value away when I leave the subroutine that sets up the menu bar.

The call to `NewMenu` itself is pretty simple. Just to keep you from flipping back a couple of pages, here it is again:

```
menuDef = ">>  Edit  \N3"+chr$(13)
menuDef = menuDef + "--Undo\N250"+chr$(13)
menuDef = menuDef + "--Cut\N251"+chr$(13)
menuDef = menuDef + "--Copy\N252"+chr$(13)
menuDef = menuDef + "--Paste\N253"+chr$(13)
menuDef = menuDef + "--Clear\N254"+chr$(13)
menuDef = menuDef + "."+chr$(13)
menuHand = NewMenu(@menuDef)
InsertMenu(menuHand, 0)
```

While the call is fairly simple, we really should stop and look at the `menuHand` variable.

```
dim menuHand as MenuHandle              :! For handling menus
```

The value returned by `NewMenu` is a menu handle, and the Menu Manager header file has a type declared for the menu handle. We'll talk about handles in detail later in the course, but for now just think of the menu handle as a value you use to tell the Menu Manager what menu you are talking about. Even after you learn what handles are, that's still a great way of thinking about a menu handle, because that's just what the menu handle is: a way of telling the Menu Manager which menu you mean when you make calls to remove or modify a menu.

Calling `NewMenu` creates a menu, but it doesn't do anything with it. The menu is in limbo, defined but not useable. The next step is to put the menu into the system menu bar with the call

```
InsertMenu(menuHand, 0)
```

One parameter is pretty obvious: `menuHand` is the handle of the menu, returned by `NewHandle`, and tells the Menu Manager what menu you want to put in the system menu bar. The other parameter tells the Menu Manager where to put the menu. This parameter is the number of some menu that is already in the menu bar; the new menu will appear right after the

menu with the menu number you pass. The number of the menu is something you set up; it's the menu number you declared in the menu string for the menu itself.

Hard coding menu numbers, like hard coding any other number, is something programmers learn to avoid whenever possible, since it means you have to be a lot more careful when you change a program. For that reason, most of the time you should pass a menu number of 0, which tells the Menu Manager to place the new menu before any other menu in the menu bar. When the Menu Manager draws the menu, it will draw the front menu at the left of the menu bar, the next menu right after it, and so on. Thinking about this for a moment, you can see that you will end up building your menus and inserting them in reverse order, putting the rightmost menu in the menu bar first, and the leftmost menu in the menu bar last.

After defining all of your menus, you need to make two more calls to the Menu Manager:

```
height = FixMenuBar                    :! Draw the completed menu bar
DrawMenuBar
```

The first of these calls tells the Menu Manager to scan the list of menus, doing some internal calculations to figure out how high the menu bar should be and how large each menu will be when it is pulled down. The height of the menu bar is something you might want to keep around if you are doing something on the screen that depends on exact placement, but in most programs you will use moveable, sizeable windows, and don't need to worry much about the height of the menu bar, or for that matter, how large the screen is. In most of our programs, we'll just throw this value away.

Finally, the last step is to draw the menu bar. This is when the text for the menus appears at the top of the screen.

Menu Events

When you sit down to use a desktop program, the first thing you are likely to do is move the cursor to the menu bar and press the mouse button. A menu pops up, and as you move the mouse down the menu, the various menu items are highlighted. If you let up on the mouse button while an item is selected the program does something.

The Menu Manager does an awful lot of the work to make all of this happen for you, but it doesn't do everything. The Menu Manager will handle all of the work once it knows that the mouse has been pressed on the menu bar, and give you back the number of the menu item that the user selected, or a zero if the user didn't end up selecting anything, but you have to figure out that the mouse was pressed in the menu bar and call the Menu Manager. Strangely enough, the easiest way to see if the mouse has been pressed in the menu bar is with a Window Manager call, `FindWindow`. When your program's event loop sees a mouse down event, it calls `FindWindow` to see if the user clicked on anything in particular, or just has a nervous twitch. Here's a typical call to `FindWindow`:

```
where = FindWindow(wPtr, myEvent.eventWhere.h, myEvent.eventWhere.v)
```

`FindWindow` checks to see if the location is in a window, setting `wPtr` to a pointer to the window if it is. Since our programs don't have any windows yet, `wPtr` will always be set to `NIL`. For our purposes, the value `FindWindow` returns is a lot more important; it is one of the following values:

number	name	description
0	`wNoHit`	Not in anything.
16	`wInDesk`	Somewhere on the desktop.
17	`wInMenuBar`	In the system menu bar; this is the one we want.
19	`wInContent`	In the content area of a window.
20	`wInDrag`	In the title bar of a window.
21	`wInGrow`	In the grow region of a window.
22	`wInGoAway`	In the close box of a window.
23	`wInZoom`	In the zoom box of a window.
24	`wInInfo`	In the information bar of a window.
25	`wInSpecial`	In a special menu item bar. These are the menus with numbers from 250 to 255.
26	`wInDeskItem`	In a desk accessory.
27	`wInFrame`	In the window, but not in any specific area defined in this table.
28	`wInactMenu`	In an inactive menu item.
<0	`wInSysWindow`	In a system window.

Table 2-1: `FindWindow` Return Codes

Of all of this list, the only values we care about at this point are `wInMenuBar` and `wInSpecial`, which tell us that the mouse click was in the system menu bar. (`wInMenuBar` is returned for menu items with a number of 256 or higher, while `wInSpecial` is returned for the reserved menu item numbers 250 to 255.) We'll learn about these other possibilities gradually, with most of them popping up when we learn to use windows.

Assuming `FindWindow` returns `wInMenuBar`, or `wInSpecial`, the next step is to call `MenuSelect`, which handles all of the hard work of actually highlighting the menu, drawing the menu items, tracking the mouse as the user moves over various menu items, and telling us what, if anything, finally got selected. The call to `MenuSelect` looks like this:

```
MenuSelect(myEvent, NIL)
```

The second parameter tells the Menu Manager what menu bar to use; `NIL` tells it to use the system menu bar. It's a little odd, but very efficient, for `MenuSelect` to take our task record as the first of the parameters. The reason this works so well is that `MenuSelect` needs to know where the mouse was clicked, and needs to tell us what menu item was selected. `MenuSelect` plucks the mouse location out of the event record, and sets a parameter in the event record to tell us what happened. Rather than changing one of the fields in the event record that is used by `GetNextEvent`, `MenuSelect` changes a field called `wmTaskData` to tell us what happened. In the abbreviated version of the event record we looked at in Lesson 1, I didn't even show the `taskData` field, but if you check the header files, it's there. We'll go into detail about this and the other fields in the event record a little later in this lesson.

In any case, `MenuSelect` sets the least significant word of `taskData` to the menu item number of the selected menu item, while the most significant word is set to the menu number.

There's one other detail about handling menu events that you need to know. When a command is being executed, your program is supposed to highlight the menu where the menu item is located. To help you out, `MenuSelect` leaves the menu highlighted, and it's up to you to unhighlight the menu after the command is finished. Here's how you unhighlight a menu:

```
HiliteMenu(FALSE, menuNum)              :! Unhighlight the menu
```

You could highlight a menu this way, too, by passing `TRUE` as the first parameter.

Sample Program – Quit

Well, finally, we know enough to write a program that actually creates and uses a menu bar! We'll start with a simple sample program that has one menu and one menu item: Quit. The traditional place for the Quit command is in the File menu, so our one menu will be called File. To handle the events, we'll use exactly the method described in the last section. Here's the complete sample program:

```
!
! Quit
!
! This program creates a menu bar with one command: Quit.
!

dim done as Boolean
dim myEvent as EventRecord
dim userID as Integer
dim where as Integer
dim wPtr as GrafPortPtr
dim menuNum as Integer
dim menuItemNum as Integer

!
! Main program
!

userID = MMStartUp
LoadLibrary 3
UtilStartUp(userID)
StartDesk(640)
if toolError <> 0 then
  print "Unable to start up desktop."
    end
end if

call InitMenus
done = False

do while not done
  if GetNextEvent(everyEvent, myEvent) then
    if myEvent.eventWhat = mouseDownEvt then
      where= FindWindow(wPtr, myEvent.eventWhere.h, myEvent.eventWhere.v)
      if where = wInMenuBar then
        MenuSelect(myEvent, NIL)
        menuNum = HiWord(myEvent.taskData)
        menuItemNum = LoWord(myEvent.taskData)
        if menuItemNum <> 0 then
          done = HandleMenu(menuNum, menuItemNum)
        end if
      end if
    end if
   end if
loop

EndDesk
UtilShutDown
UnloadLibrary 3
end

!
! InitMenus
!
! Initialize the menu bar and insert the menus.
!
sub InitMenus
  dim height as Integer
```

```
  dim menuHand as MenuHandle
  dim menuDef as String

  menuDef = ">>  File  \N1"+chr$(13)
  menuDef = menuDef + "--Quit\N256"+chr$(13)
  menuDef = menuDef + "."+chr$(13)

  menuHand = NewMenu(@menuDef)
  InsertMenu(menuHand, 0)
  height = FixMenuBar
  DrawMenuBar
end sub

!
! HandleMenu
!
! Handle menu selections.
!
function HandleMenu(menuNum as Integer, menuItemNum as Integer) as Boolean
  dim result as Boolean
  fileQuit = 256

  result = false
  select case menuItemNum
    case fileQuit
      result = True
  end select

  HiliteMenu(FALSE, menuNum)
  HandleMenu = result
end function
```

Listing 2-1: Quit Program

One of the tips I've picked up over the years is to type in a program like this. On the surface, that seems a little strange, since the program is on disk. Glancing through the program, everything looks pretty familiar, too. After all, we've spent several pages in an exhaustive analysis of how to create a program like this one. Strangely enough, though, you will get a lot from typing it in. The reason is that your mind works faster than your fingers, and while you type, you're really reading the program, and hopefully going over it in your mind. Besides, all good programmers love the feel of a set of keys gently giving way beneath their fingers, right? I'd suggest typing it in, but I suppose you can load it from disk. In any case, try the program out, and take the time to go through the code carefully.

Δ **Important** If you decide to use PRIZM's debugger on this program be sure to use the Step Through command to execute `InitMenus`. The debugger is a desktop program, too, and it is using the menu bar. One of the few real problems with this arrangement is that you can't safely step through code between an `InsertMenu` call and a `FixMenuBar` call, at least not if you switch between the debugger menu bar and your program's menu bar in the process.

Frankly, the simplest way to handle this problem is to set a break point (using the BREAK command) right after the call to `InitMenus`. That

way your program's menu bar already exists when you start debugging. The other alternative is to use ORCA/Debugger, which is a text-based Init debugger. It doesn't have its own menu bar, so there isn't a conflict. Δ

Problem 2-1. Add the Edit menu described at the start of the lesson to this program. The File menu should be on the left, and the Edit menu on the right.

Problem 2-2. Create a program with a File menu with the item Quit, just as in the sample program. Add a second menu called Beep, with three menu items, named One, Two, and Three. Use menu item numbers of 301, 302 and 303 for these three menu items, and use menu number 2 for the menu itself.

Set up your program so `SysBeep` is called once when menu item One is selected, twice when menu item Two is selected, and three times for Menu item Three.

Keyboard Equivalents

Pulling down a menu to tell a program what to do is easy to understand, and it's easy to quickly see what commands are available or to check the name of a command. It's also slow, especially in a program like a text editor, where your hands are generally on the keyboard. That's the biggest reason for keyboard equivalents, which let you hold down the Command key (A.K.A. the open-apple key) and type a key to execute a menu command.

When you set up a menu, there are two distinct parts to the process: creating the menu in the first place, and changing the event loop so your program recognized the menu command. The same is true with keyboard equivalents. The first step is to learn how to change the menu definition strings to add a keyboard equivalent. Once the keyboard equivalent is in the menu definition, we'll have to change the event loop to handle keystrokes that are really keyboard equivalents for menu commands.

Keyboard equivalents are defined when the menu item itself is defined. You tell the Menu Manager about the keyboard equivalent using a control code, just as you used to tell the Menu Manager what menu item ID number to assign to the menu item. The * character is used to start a keyboard equivalent; this is followed by two characters that will be used as the keyboard equivalents. As one example, it is customary to use Q as the keyboard equivalent for the Quit command. An uppercase Q is different from a lowercase q, though, so you need to tell the Menu Manager to support both the uppercase Q and lowercase q by listing both keys as keyboard equivalents. Adding this keyboard equivalent to our Quit menu item from the sample program in the last section, the line that defines the Quit menu looks like this:

```
"--Quit\N256*Qq "+chr$(13)
```

You can put in the control codes in any order you like. For example, putting the keyboard equivalent first and the menu item ID last, like this:

```
"--Quit\*QqN256\r"+chr$(13)
```

works perfectly well. The Menu Manager does the same thing either way.

You could actually use any two characters you like for the keyboard equivalents. Using "*Rq" would tell the Menu Manager to allow an uppercase R or a lowercase q as a keyboard equivalent. On the other hand, just because you can do something doesn't necessarily mean you should do it, as any parent of a teenager has undoubtedly pointed out on more than one occasion. The whole point of the Apple interface is to create programs that are easy to use and consistent between applications. Doing bizarre things like picking out strange keyboard equivalent key combinations doesn't exactly help the user keep track of what keys can be used. In general, the rule is to use the uppercase and lowercase versions of the same key. That way, it doesn't matter whether the shift key – in particular, the caps lock key – is down or up.

The Menu Manager draws the first of the key equivalents in the menu. Since the order of the keys does matter, it's important to follow the common conventions there, too. The rule is to put the uppercase letter first for alphabetic keys, and the unshifted letter first for non-alphabetic keys. The Quit menu item is a great example of an alphabetic keyboard equivalent. Non-alphabetic keyboard equivalents are more rare, but certainly not unheard of. The ORCA desktop environment (PRIZM) is an example of a program that uses several non-alphabetic keyboard equivalents. One of these is the [key, used for the Step command. The second key PRIZM uses for the Step command is the { key, so the keyboard equivalent is written as "*[{" in the menu item definition for the Step command.

When you want to use a keyboard equivalent in a program, you hold down the Command key and press the key. Inside the desktop program the key down event is reported just like any other keypress. Holding down the Command key while you press Q to quit a program does not change the key the Event Manager reports in the `eventMessage` field. The only way your program can tell that the Command key was held down is to look at the `eventModifiers` field of the event record to see if the appleKey bit is set, like this:

```
if BitwiseAnd(myEvent.eventModifiers, appleKey) then
  ! The command key (aka open-apple key) is down
end if
```

If the `appleKey` bit is set your program should make a `MenuKey` call. The `MenuKey` call does the same thing for key down events that the `MenuSelect` call did for mouse down events. In fact, the parameters are event the same. The `MenuKey` call looks like this:

```
MenuKey(myEvent, NIL);
```

Putting these ideas together, we can update the sample program from the last section to handle a keyboard equivalent of Q or q for the Quit command. The complete sample program is shown in Listing 2-2.

```
!
! Quit
!
! This program creates a menu bar with one command: Quit.
!

dim done as Boolean
dim myEvent as EventRecord
dim userID as Integer
dim where as Integer
dim wPtr as GrafPortPtr
dim menuNum as Integer
dim menuItemNum as Integer

!
! Main program
!

userID = MMStartUp
LoadLibrary 3
UtilStartUp(userID)
StartDesk(640)
if toolError <> 0 then
  print "Unable to start up desktop."
    end
end if

call InitMenus
done = False

do while not done
  if GetNextEvent(everyEvent, myEvent) then
    select case myEvent.eventWhat
      case mouseDownEvt
        where= FindWindow(wPtr, myEvent.eventWhere.h, myEvent.eventWhere.v)
        if where = wInMenuBar then
          MenuSelect(myEvent, NIL)
          menuNum = HiWord(myEvent.taskData)
          menuItemNum = LoWord(myEvent.taskData)
          if menuItemNum <> 0 then
            done = HandleMenu(menuNum, menuItemNum)
          end if
        end if
      case keyDownEvt, autoKeyEvt
        if BitwiseAnd(myEvent.eventModifiers, $0100) then
          MenuKey(myEvent, NIL)
          menuNum = HiWord(myEvent.taskData)
          menuItemNum = LoWord(myEvent.taskData)
          if menuItemNum <> 0 then
            done = HandleMenu(menuNum, menuItemNum)
          end if
        end if
    end select
    end if
loop

EndDesk
UtilShutDown
```

```
UnloadLibrary 3
end

!
! InitMenus
!
! Initialize the menu bar and insert the menus.
!
sub InitMenus
  dim height as Integer
  dim menuHand as MenuHandle
  dim menuDef as String

  menuDef = ">>  File  \N1"+chr$(13)
  menuDef = menuDef + "--Quit\N256*Qq"+chr$(13)
  menuDef = menuDef + "."+chr$(13)

  menuHand = NewMenu(@menuDef)
  InsertMenu(menuHand, 0)
  height = FixMenuBar
  DrawMenuBar
end sub

!
! HandleMenu
!
! Handle menu selections.
!
function HandleMenu(menuNum as Integer, menuItemNum as Integer) as Boolean
  dim result as Boolean
  fileQuit = 256

  result = false
  select case menuItemNum
    case fileQuit
      result = True
  end select

  HiliteMenu(FALSE, menuNum)
  HandleMenu = result
end function
```

Listing 2-2: Quit Program with Keyboard Equivalents

The above program uses BitwiseAnd, which is a function found in the utility tool set provided with this course. It performs a bitwise AND operation on two numbers. The integer result of an AND operation has 0 bits wherever either of the two operands have a 0 bit, and a 1 wherever both operands have a 1 bit. The utility tool set includes BitwiseOr and BitwiseXOR (exclusive or) functions as well.

Problem 2-3. In Problem 2-2 you created a program with a beep menu, with menu commands to beep one, two or three times. Add keyboard equivalents of 1, 2 or 3 for these three commands.

In general, a numeric shift is one case where it doesn't make a lot of sense to allow any key but the number. In a case like this, one way to set things up is to use the same key for both the shifted and unshifted keys; the "!" and "@" keys don't make sense as shifted versions of "1" and

"2". However, these keys can be used as keyboard equivalents (shifted and option-key combinations work, but they're not very user-friendly).

Standard Keyboard Equivalents

You've probably used enough desktop programs to be pretty used to Q as the keyboard equivalent of the Quit command. You may also be used to Z for Undo, X for Cut, C for Copy, and V for Paste. The reason you are so used to these keys isn't because they are the only good choices, it's because Apple laid out a set of standard keyboard equivalents long ago, and people who write programs using the Apple interface follow these guidelines rather closely. Standard keys for keyboard equivalents, along with other guidelines like what commonly used alert buttons should be called, how color should be used, and so forth are laid out in a book called *Human Interface Guidelines: The Apple Desktop Interface*. This is a book that every desktop programmer should have around as a reference book, and it is also a book I would recommend that you read at least twice, once right after you finish this course and again in about 6 months to a year. Creating programs that are consistent with Apple's guidelines is very important for a lot of reasons, but you can't create consistent programs if you don't know what the guidelines are!

Getting back to keyboard equivalents, Table 2-2 shows the keys Apple has set aside for specific purposes.

Menu	Item	Keyboard Equivalent
Apple	Help	?
File	New	N
File	Open	O
File	Save	S
File	Print	P
File	Quit	Q
Edit	Undo	Z
Edit	Cut	X
Edit	Copy	C
Edit	Paste	V
Style	Bold	B
Style	Italic	I
Style	Underline	U

Table 2-2: Standard Keyboard Equivalents

Two other keyboard equivalents are so common that they are de facto standards. ⌘W is generally used as the keyboard equivalent of the Close command in the File menu, while ⌘A is used for the Select All option in the Edit menu. Plain Text, often found in the Style menu of some applications, is typically given the keyboard equivalent ⌘T.

Note that the first and second editions of the *Human Interface Guidelines* indicate that ⌘P should be used for Plain Text, and doesn't suggest a key for the Print option. This has been changed since then (in a technical note, and in the most recent version of the Macintosh Interface Guidelines), so I list the current standard.

TaskMaster

If handling all of these various kinds of events, and figuring out whether they apply to the menu bar or not, is starting to seem a little complicated, just wait: it gets worse! As you add the various checks for your own windows, then pile checks for NDA windows (or system windows, in the parlance of the toolbox reference manuals), things get genuinely messy. And, on the Macintosh, that's just what you have to do.

Fortunately the folks who wrote the Apple IIGS toolbox realized that all of this was sort of a mess, and created an easier way of handling all of the routine tasks that almost any desktop program must handle. `TaskMaster` is a tool call that works a lot like the Event Manager's `GetNextEvent` call, but instead of handing back a raw event, `TaskMaster` does all of the standard work for you, and gives you back the results. For example, when `TaskMaster` sees that you have a mouse down event, it calls `FindWindow` automatically. If `FindWindow` tells `TaskMaster` that the mouse down event occurred in the system menu bar, `TaskMaster` calls `MenuSelect`, and returns `wInMenuBar` as the event. Since `MenuSelect` is already putting the information about the menu that was selected in the `taskData` field of the event record, you can still pluck out the menu item and menu number the same way you did when you handled the calls for yourself.

To see how this works, let's take a look at the Quit program's event loop again, this time using `TaskMaster`.

```
done = False
myEvent.taskMask = $001F7FFF

do while not done
   event = TaskMaster(everyEvent, myEvent)
   select case event
      case wInSpecial, wInMenuBar
         menuNum = HiWord(myEvent.taskData)
         menuItemNum = LoWord(myEvent.taskData)
         if menuItemNum <> 0 then
            done = HandleMenu(menuNum, menuItemNum)
         end if
   end select
loop
```

Let's face it – that's a lot easier than the event loop we wrote to handle the raw events for ourselves! And this event loop does exactly the same thing as the one we wrote for the Quit program.

While it is called just like `GetNextEvent`, `TaskMaster` needs to get a little more information from us, and needs to be able to pass back more information to handle some complicated events we haven't run across yet. All of this extra information is passed in an extended version of the event record. Once the `TaskMaster` fields are added, the event record looks like this:

```
type eventRecord
  eventWhat as integer
  eventMessage as long
  eventWhen as long
  eventWhere as point
  eventModifiers as integer
  ; TaskMaster fields
  taskData as long
  taskMask as long
  lastClickTick as long
  ClickCount as integer
  TaskData2 as long
  TaskData3 as long
  TaskData4 as long
  lastClickPt as point
end type
type eventRecPtr as pointer to eventRecord
type wmTaskRec as eventRecord
```

In fact, if you look in the GSoftTools.int file, this is the event record that is used throughout the toolbox, even for Event Manager calls like `GetNextEvent`.

For the most part, we'll ignore these extra fields for now, and talk about them in detail when we get to a point in the course where we actually need to use the field. Starting to talk about things like the result returned by the `defProc` of a control in a window just wouldn't make much sense until after we talk about windows and controls! There is one field of the task record that we need to learn about right away, though, and that's the `taskMask` field.

`TaskMaster` can do an awful lot of things for you, but that isn't always what you want. There are situations where you want to handle something for yourself so you can do something a little unusual.

The `taskMask` field of the extended event record is a bit mapped field. Each of the bits corresponds to one of the things you can have `TaskMaster` do for you. In most cases, if you set the bit to 1, `TaskMaster` will take care of the details of handling the situation for you, but if you set the bit to zero, `TaskMaster` will let you handle the details for yourself.

I used a qualifier there you should pay attention to: I said you set the bit to one to ask `TaskMaster` to do something *in most cases*. There are a few cases where you do just the opposite, clearing the bit when you want `TaskMaster` to perform the action, and setting the bit when `TaskMaster` should let you handle it. It probably seems odd to be so inconsistent, but this inconsistencies really did come about with the best of intentions. You have to realize that the Apple IIGS toolbox didn't just pop into being in its current form; it evolved over time. In the original version of the `TaskMaster` call, and even today, there were some fields that were not used. Apple specified from the very beginning that you had to set the unused bits to zero. That gave them a predictable value to count on when they added new features to `TaskMaster`, and in those cases, zero was used for the default case, and one when you wanted to ask `TaskMaster` to pick the unusual situation.

Table 2-3 lists the various bits in the `wmTaskMask` field. The description tells you what will happen if you *set* the bit. If you clear the bit (set it to zero) then the action described will *not* take place. In several cases, you have to set one bit before another can be used. For example, you have to set the `tmFindW` bit or `TaskMaster` will not need to even look at the value of `tmMenuSel`.

Only a few of the bits are likely to make sense to you at this point. Don't worry, we'll cover most of them in the course as you learn more about the Window Manager, controls, desk accessories, and so forth.

`tmMenuKey`	0	Call `MenuKey` to see if a keypress is a menu keyboard equivalent.
`tmUpdate`	1	For window update events, call the window's default draw routine.
`tmFindW`	2	Call `FindWindow` for mouse down events.
`tmMenuSel`	3	Call `MenuSelect` when `FindWindow` determines that a mouse down event occurred in a menu bar.
`tmOpenDA`	4	Open a desk accessory when `MenuSelect` determines that a menu event was a selection of an NDA.
`tmSysClick`	5	When `FindWindow` returns an event from a system window (an NDA window is a system window), call `SystemClick` to handle the event.
`tmDragW`	6	If `FindWindow` returns `wInDrag`, handle moving the window around on the screen.
`tmContent`	7	If `FindWindow` detects an event in a window's content region, and the window is not the active window, select the window.
`tmClose`	8	If `FindWindow` detects a mouse down in a close box, call `TrackGoAway`. If `TrackGoAway` returns `TRUE`, return `wInGoAway`; otherwise, return `nullEvt`.
`tmZoom`	9	If `FindWindow` detects a mouse down in a zoom box, call `TrackZoom`. If `TrackZoom` returns `TRUE`, call `ZoomWindow` to change the window's size.
`tmGrow`	10	If `FindWindow` detects a mouse down in the grow box of a window, call `GrowWindow` to allow the window's size to be changed, then `SizeWindow` to actually update the window's size.
`tmScroll`	11	Handle scroll bars.
`tmSpecial`	12	Handle special menu items.
`tmCRedraw`	13	When a window is activated or deactivated, redraw the controls in the correct state.
`tmInactive`	14	If an inactive menu item is selected, return `wInactMenu`. (This is generally used to put up a help dialog telling the user what the menu is for, or what has to happen before it is active.)
`tmInfo`	15	Don't activate inactive windows when a mouse down event occurs inside of the information bar of the window.
`tmContentControls`	16	If `FindWindow` returns `wInContent`, call `FindControl` and `TrackControl` to handle normal control actions.
`tmControlKey`	17	Pass key events to controls for control key equivalents.
`tmControlMenu`	18	Pass menu events to controls in the active window.
`tmMultiClick`	19	Check for multiple clicks and return information about them.

`tmIdleEvents`	20	Send idle events to the controls in the window.

Table 2-3: `TaskMaster` `taskMask` Codes

`taskMask` codes are one of the few places in this course where I won't use the constant labels, shown in the first column, instead of the number. In most of our programs, we'll set all of the bits except `tmInfo`, and it just takes too long and uses too much space to be clear if you add each and every one of these identifiers together to get your `taskMask` code. If you check back, the example event loop I gave set `taskMask` to $001F7FFF. This happens to be the value you would get by adding all of the constants except `tmInfo`. (The value in column two is the bit number, not the constant value of the label.) It is also the value we will use for most of our programs, so we'll be letting `TaskMaster` do all of the work for us that it can.

Problem 2-4. Modify the Quit program to use `TaskMaster`, rather than `GetNextEvent`, in the main event loop.

The Apple Menu

Looking at a menu bar from a fairly typical program, you see one major difference between the full-fledged program's menu bar and the one in the programs we have been creating in this lesson. The difference is the distinctive Apple menu on the left side of the menu bar. There are two reasons we haven't created an Apple menu for our programs yet. The first is that you need to know a couple of special tricks. The other reasons is even better, in a way: we haven't needed one, yet.

Traditionally, the Apple menu serves two purposes. At the top of most Apple menus is a menu item called About. The about menu generally brings up some sort of alert that tells you the name of the program you are running and its version number. This is also a great place for shareware authors to put those details about where to send money, as well as the traditional spot for any copyright notice.

Under the About menu you will generally find a list of NDAs, little desktop programs that run from almost any desktop program.

There are two pieces of information you need to know to create an Apple menu. The first is how to create the name of the menu itself. After all, the rainbow Apple isn't something you can type as a menu name! Apple Computer solves this problem by using the @ character for the apple character. When you use an @ character, with no other characters at all, as the name of a menu, the Menu Manager draws the rainbow Apple instead of an @ character. This is an exception to the normal rule of putting spaces on either side of the menu's name.

The Menu Manager inverts the area around the name of a menu when you pull it down. It reverses the area by flipping each pixel in the area to exactly the opposite color. For some peculiar reasons that are pretty hard to understand until you learn about color palettes in a later lesson, reversing the rainbow apple turns the entire apple character green or purple, depending on whether you are using 320 mode graphics or 640 mode graphics. Of course, most programs leave the rainbow apple in place when you pull down a menu, and only reverse the area around the apple, turning it black. To perform this trick, the Menu Manager uses a different mechanism to reverse the area on the menu bar. This technique, which the toolbox reference manuals call

color replace highlighting, is used to reverse the menu name when you put an X character in the menu name's control character sequence.

Putting all of this together, we can create an apple menu with a rainbow apple and an about item with this menu string:

```
menuDef = ">>@\XN1"+chr$(13)
menuDef = menuDef + "--About...\N257"+chr$(13)
menuDef = menuDef + "."+chr$(13)
menuHand = NewMenu(@menuDef)
InsertMenu(menuHand, 0)
```

Problem 2-5. Add an apple menu to the menu bar for the Quit program. (Use the version from your solution to Problem 2-4.) The apple menu should have a menu number of 1, which it will if you use the menu string shown above. The current version of the Quit program uses 1 for the menu number of the File menu. Two menus cannot have the same menu number, so be sure and change the menu number for the File menu to 2.

Supporting NDAs

Supporting New Desk Accessories turns out to be very easy – at least, it's easy if you use `TaskMaster`, like we are now doing. `TaskMaster` handles all sorts of messy details for us. `TaskMaster` keep track of when an NDA is selected from the apple menu, starting the desk accessory when one is selected. `TaskMaster` also checks to see if an event should be handled by a desk accessory that has already been opened, pulling it to front if the desk accessory window isn't the active window, passing menu commands to the desk accessory, and so forth. You get all of this almost for free. There are only four things you have to do to support desk accessories in your program:

1. Make a call to `FixAppleMenu` to add the desk accessories to your apple menu.
2. Use `TaskMaster`, and set the `taskMask` codes `tmMenuKey`, `tmUpdate`, `tmFindW`, `tmMenuSel`, `tmOpenDA`, `tmSysClick`, `tmDragW`, `tmContent`, `tmClose`, `tmZoom`, `tmGrow`, and `tmScroll`.
3. Create a few standard menus and menu items that desk accessories expect to find, using reserved menu item numbers.
4. Start up all of the standard tools that desk accessories assume are started.

Call FixAppleMenu

`FixAppleMenu` has single parameter, the menu number for the apple menu. You should call `FixAppleMenu` after you have created all of the menus in the menu bar, but right before you call `FixMenuBar` to calculate the size of each menu. The Menu Manager takes care of all of the details of looking for desk accessories, figuring out what name to use, and adding them to the apple menu.

Assuming the menu number of the apple menu is 1, here's the call to `FixAppleMenu`:

```
FixAppleMenu(1)
```

Actually, you can pass any menu number to `FixAppleMenu`, and it will add the desk accessories to the menu you specify. The convention is to place desk accessories in the apple menu, but if you have some peculiar requirement, you can add them to any menu you prefer.

Problem 2-6. Start with your solution to Problem 2-5. Call `FixAppleMenu` to add the desk accessories to your apple menu.

You can run the program, but don't select any desk accessories, yet. There's one more step you need to make before you can use desk accessories from your program.

Use `TaskMaster`

Hey, you're already doing this, right?

I did cheat a little. When I listed the `taskMask` bits that should be set for a desk accessory, I included a few that aren't strictly necessary, but that helps things out in the long run.

The Standard Menu Items

When you write an NDA you are allowed to assume that certain menu items exist. The Menu Manager identifies menu items by a menu item number, so these menus must have a fixed menu item number. The Menu Manager actually blocks off a whole series of menu item numbers for its own use – when you create a menu item, you must use a menu item number of 256 or higher (the maximum menu item number is 65534). The Menu Manager uses menu item numbers 0 and 65535 for its own internal bookkeeping, and blocks off menu item numbers 1 to 249 for numbering the NDA menu items.

The remaining menu item numbers, 250 to 255, must be defined. These are the ones NDAs assume exist. Table 2-4 shows a list of the menu item numbers, along with the traditional menu name and menu item name.

Menu ID	Menu Name	Menu Item Name
250	Edit	Undo
251	Edit	Cut
252	Edit	Copy
253	Edit	Paste
254	Edit	Clear
255	File	Close

Table 2-4: Required Menu Items

It's fair to ask what would happen if you didn't put these menus in your program, or if you switched the menu items around. If you leave the menu items out altogether you rob some desk accessories of important capabilities, since using these menu commands is probably the only way to get the desk accessory to do things like cutting text or reversing some action with Undo.

Using the wrong menu item numbers is even worse: the desk accessory could execute the wrong command, or a user might select a command and expect it to work, but the desk accessory might not recognize the command.

Problem 2-7. Starting with your solution to Problem 2-6, create a program with all of the required menu items.

The first menu should be an apple menu. It should include an About item, and the desk accessories should appear under the About item.

The second menu should be the File menu. It should have two items, Close and Quit, in that order. They should have key equivalents of W and Q, respectively.

The last menu is the Edit menu. The Edit menu should have four menu items; they are Undo, with a key equivalent of Z; Cut, with a key equivalent of X; Copy, with a key equivalent of C; Paste, with a key equivalent of V; and Clear. The menu items should appear in the order listed.

You can use your program to run NDAs as soon as all of these new menu items are installed.

The Minimal Tools

NDAs are allowed to assume that certain tools have already been started. The tools that must be started to support NDAs are shown in Table 2-5. All of these tools are started by `StartDesk`, which is what we have been using so far to start the tools.

Tool Locator
Memory Manager
Miscellaneous Tool Set
QuickDraw II
Event Manager
Window Manager
Menu Manager
Control Manager
LineEdit Tool Set
Dialog Manager
Scrap Manager
Desk Manager

Table 2-5: Minimum Tools for NDA Support

Customizing Your Menu Items

By now you've used three different control codes in your menus and menu items. These control codes let you set the menu number or menu item number, define keyboard equivalents, and use color replace to avoid green-apple menu sickness. There are several other control codes that we haven't used yet. It's time to get organized and list all of the control codes that are available to you.

Character	Description
*	The two characters that follow this one are used as keyboard equivalents. The first of the characters is shown in the menu when you pull the menu down. Most of the time, the first letter will be an uppercase letter or an unshifted non-alphabetic keyboard key, while the second letter will be the lowercase equivalent of the uppercase letter or the shifted equivalent of the unshifted key.
B	Use bold text for the menu item.
C	Mark the menu item with the character that follows this one. This is generally a check mark, which is `the byte $12`.
D	Dim (disable) the menu item.
H	This character is used before a two-byte sequence which defines the menu number or menu item number, as a raw hexadecimal value embedded in the text. From GSoft BASIC, it's generally best to use N instead (it's certainly easier).
I	Use italicized text for the menu item.
N	This character is followed by one or more decimal digits. The resulting number is used as the menu number or menu item number, depending on whether you are defining a menu or a menu item.
U	Underline the text for the menu item. (But see the note, below!)
V	Place a dividing line under the item.
X	Use color replace mode to avoid green-apple menu sickness.

Table 2-6: Menu String Control Characters

Let's briefly go over the new menu string control codes to see what they are for. At the end of this section, you will also get a chance to create a program that demonstrates all of these control codes.

Three of the new control codes are used to control the appearance of the text. B creates boldface text, I italicizes the text, and U underlines the characters. You can use these characters in combination, too, so you can create bold italicized text.

There is one minor problem with underlining: it doesn't usually work. The character set that is used by the Menu Manager to draw text is called the system font, and the default system font (a.k.a Shaston) doesn't have room to underline the text, so it doesn't let you try. Using the default system font, you can tell the Menu Manager to underline the text, but you'll be ignored. You could change the system font – but that causes more problems that it cures. Unless you're really desperate, it's best to ignore the underline option for the menus.

You've probably used some programs that placed a check mark beside a menu item when you selected some option. For example, many font menus put a check mark beside the current font and font size. The C option lets you create a menu item that starts off with a character to the left of the menu item name. While you generally use a check character for this character, you can actually put in any character you like using the C option.

There's a problem with the check mark character, of course. Careful hunting on your trusty keyboard will reveal that there is no such thing as a check mark character. As it turns out, many Apple IIGS fonts have more than the standard printable characters you see on your keyboard. Some of these are special foreign characters, like the German ß, while some are special

characters, like √. To get these from GSoft BASIC, you must use the CHR$() function to specify a numeric value for the character. As an example, the line

```
"--Check Me!\N300C"+chr$($12)+chr$(13)
```

defines a menu item with a check mark.

There are times when a particular menu command just doesn't make sense. If there aren't any open windows, it doesn't make much sense to use the Close command, for example. Any time a menu command can't be used, you can (and should!) disable it. A disabled menu item is dimmed, with some of the pixels turned off, and you can't select the menu item. The D control character lets you define a menu item that starts out dimmed.

Menus can get pretty long, and it helps to organize them a bit when this happens. As a general rule of thumb, menus are divided up based on groups of similar commands. A classic example is the apple menu, which has an about item as the first item in the menu, and has a list of desk accessories under the about item. These two separate classes of menu items are divided with a separator.

There are two ways to create a separator on the Apple IIGS. The V control code is one way; this option draws a line just below the menu item containing the V. Another way is to define a menu item with a title string of `'-'`. The Menu Manager treats this as a special case, drawing dashes across the entire menu bar instead of just writing one character. (You should dim this menu item, too, using the D character.)

Most if these control codes can only be used for menu items. Menus don't need keyboard equivalents, so the * control code can only be used for menu items. Menus also don't use dividing lines or check marks, so you can't use C or V, either. The three questionable calls are B, I and U, which control the appearance of the text. I guess Apple decided that giving programmers control over the style of text used in the menu bar was a bad idea, so you can't use those control codes with a menu, either.

Problem 2-8. This problem creates a menu sampler program that you can use to explore the various menu item options we've covered in this section.

Start with your solution to Problem 2-7. In the apple menu, put a separator under the About... command using `'-'` as the menu item title. Be sure and use a D character in the control character list to disable the menu item.

Add a new menu to the right of the Edit menu; this menu will show the various styles you can use for a menu item. Call this menu Style, and include these menu items: Bold, Underline, Italic, Bold Underline, Bold Italic, Underline Italic, and Bold Underline Italic. Put in the proper combination of control characters to show off each of these text styles. (Underline won't work, but you can see that for yourself.)

The last menu should be called Options. This menu should have three menu items: Separator, Dimmed, and Checked. Use the V control code with the first menu item to separate it from the other two. Dim the menu item Dimmed, and put a check character to the left of the menu item Checked.

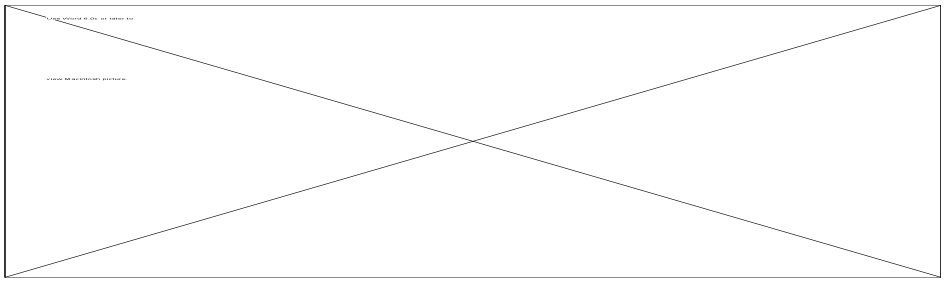

Figure 2-3: Menus Created in Problem 2-8

Changing Menu Items on the Fly

So far, we've looked at a lot of ways you can create menus. Once your program is running, though, you may need to make some changes in the menus to keep up with the changing needs in your program. We'll look at a variety of changes you can make, from something as simple as dimming a menu item (or an entire menu) through more complex changes, like adding or removing a menu.

The simplest and most common change to make in your menus is to dim or undim a menu item. When you dim a menu item, the Menu Manager erases half of the bits, causing a gray looking appearance like you see in Figure 2-4. The dimmed menu item also can't be selected, so there's no chance the menu item's item number will be returned as a menu command in your event loop. The reason this is such a common change is that it's a change the Apple Human Interface Guidelines actually suggest that you make as your program runs. When it doesn't make sense to use a command, you are supposed to dim the menu item so the command can't be selected. For example, if there are no open windows, it doesn't make sense to allow the user to print the contents of a window, so the Print command should be dimmed. Once the command is available again, the menu item should be changed back to normal.

All you need to know to dim a menu item is the menu item number; that's the number you assigned when you created the menu item. You dim the menu by calling `DisableMItem`, passing the menu item number as a parameter, like this:

```
DisableMItem(256)
```

The menu item will be dimmed when the menu bar is pulled down again, and the user won't be able to select the menu item from the menu. When you want to return the menu command to normal, so it is drawn normally and can be selected, use `EnableMItem`:

```
EnableMItem(256)
```

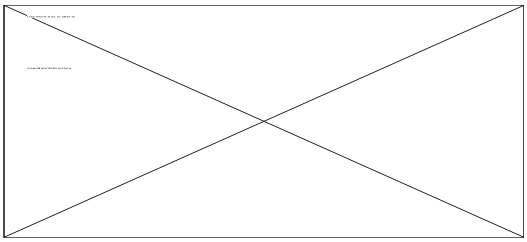

Figure 2-4: Dimmed Menu Compared to Normal Menu with Some Dimmed Items

Dimming a menu is almost as easy as dimming a menu item. Each of the menus has a set of flags that the menu manager uses to keep track of the menu. To dim an entire menu, you need to call `SetMenuFlag` to dim the menu; there's a predefined constant in Menu.h that makes this pretty easy:

```
SetMenuFlag(disableMenu, 2)                    :! disable menu 2
```

The same call with a different flag word enables the menu:

```
SetMenuFlag(enableMenu, 2)                     :! enable menu 2
```

`SetMenuFlag` changes the internal flags for the menu, but it does not draw the menu in the new state. To update the appearance of the menu bar, be sure to follow the call to `SetMenuFlag` with a call to `DrawMenuBar`.

A useful side effect of disabling a menu is that all of the menu items in the menu are automatically disabled, too. When you enable the menu, the menu items that were automatically disabled are enabled, although any that you specifically disabled stay disabled. That makes disabling an entire menu a great way to go when all of the options in the menu need to be turned off at once.

The second most common change you will make in your menus as your program runs is to check or uncheck various menu items. I'm sure you've used menus where options could be turned on or off, or you could select one of several options from a list of choices. In both of these cases it's common to check the menu item when it is the active choice. For one example of many, pull down the Font menu in the Teach application that comes with the system software. There is a check mark beside name of the current font. When you select a new font, the new one is checked, and the check mark is erased from the original choice.

Well, even though your programs need to be consistent, the folks who wrote the Menu Manager weren't. There's yet another way to handle checking and unchecking a menu item. Instead of two separate calls, like `DisableMItem` and `EnableMItem`, or even calls to get and set flags, this time everything is handled with a single call. `CheckMItem` takes two parameters, a boolean flag that tells the Menu Manager to put a check mark by the menu item if the flag is `TRUE`, and to erase any check mark that might be there if the flag is `FALSE`. The second parameter is the menu item number.

```
CheckMItem(TRUE, 256)                          :! check menu item 256
CheckMItem(FALSE, 256)                         :! uncheck menu item 256
```

Problem 2-9. This problem looks a little long, but that's mostly because I wanted to describe exactly what you are supposed to do. As you read along, check out Figure 2-5, which is a screen capture of the menu you are creating in this problem.

Start with our standard program that has a File menu with a Quit command and supports NDAs (the solution to Problem 2-7). Add a menu called Beeps. This menu will have three groups of commands. The first is called "Beep". This menu item tells the program to beep the speaker by calling `SysBeep`.

Right after "Beep" is a series of three menu items, named "1", "2" and "3". Start with "1" checked, and set up your program so selecting one of the other two numbers will remove the check mark from "1" and check the selected number. Of course, when the user selects a different number, the newly selected one will be checked. The beep command should beep one, two or three times, depending on which of these items is selected.

The last item is called "Silence". When you select silence, the beep command should be dimmed and disabled, but you should still be able to select the number of beeps. The "Silence" menu item should also be checked. Selecting it again returns the beep command to normal, and should turn off the check mark.

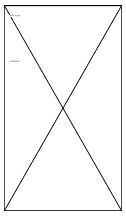

Figure 2-5: The Beeps Menu

Changing the Text for a Menu Item

Another important change you might want to make to a menu while your program is running is to change the name of a menu item, or even add a new one or completely get rid of a menu item that already exists. One of the most common reasons for changing a menu item is a variation on using a check mark next to a menu item. The idea is to show an option the user might pick, like "Hide Palette" in a paint program. When the user picks this option, the paint palette would disappear. Somehow, just checking or unchecking this option doesn't get across the full meaning of what's happened, so you might want to change the entire menu item from "Hide Palette" to "Show Palette." The `SetMItem` call is used to change the name of the menu, like this:

```
SetMItem("--Show Palette", 263)
```

In this particular case, we're changing the text for menu item 263, which reads "Hide Palette" before we make the call to the Menu Manager.

There's a few technical details I've swept under the rug by making the change the way it's shown. Let's go over those now.

The string you pass as a parameter to `SetMItem` looks a lot like the original menu item string that you've been using to create new menus at the start of your program. You still need a two

character sequence at the start of the menu name. As with the original menu item string, the new name has to end with either a null character or a return character. The string has to be in a permanent, fixed location in memory. As it turns out, string constants in GSoft BASIC happen to be in a fixed location in memory, and they also happen to be followed by a null character.

Thinking back for a moment, the original menu item string had some other stuff tacked onto the end of the string. The other characters were the control characters, starting with a backslash and continuing with things like the menu item ID number. When you use `SetMItem` to change a menu item, you don't use any of these control characters. The new name for the menu item inherits all of the control code information from the original menu item.

Putting all of this together, here's a subroutine you could call to change a menu item back and forth between "Hide Palette" and "Show Palette." This subroutine assumes that your main program has created a shared string variable called `paletteString`; this is the string number for the currently visible palette string, which will be 0 for the original string, "Hide Palette," and 1 for the alternate string, "Show Palette." Somewhere in your program's initialization section, this variable should be set to 0. Of course, the reason the variable is a shared variable is that the value of the variable has to survive between calls to `ChangePalette`, and local variables go away between subroutine calls. `ChangePalette` also assumes you've defined a shared constant called `optionsPalette`, which is the menu item number for this menu item.

```
!
! ChangePalette
!
! Change the menu item string for the palette choice.
!
sub ChangePalette
  shared paletteString, optionsPalette

  if paletteString = 0 then
    paletteString = 1
    SetMItem("--Show Palette", optionsPalette)
  else
    paletteString = 0
    SetMItem("--Hide Palette", optionsPalette)
  end if
end sub
```

Problem 2-10. Start with the solution to Problem 2-7 and add a new menu called Options. Use the subroutine we just developed to switch a menu item from "Hide Palette" to "Show Palette" and back again as the menu item is selected.

Other Things You Should Know About Menus

At this point, you know enough about the Menu Manager to create most of the menus you are likely to see in working desktop programs. You also know enough about manipulating those menus once they are created to do most of the common things you see done with menus, like dimming them or using check marks. There's a lot more to the Menu Manager than we've covered so far, though. Occasionally, as we write more advanced desktop programs, we'll come back to the Menu Manager and talk about some of these capabilities that weren't covered in this

section. By now, though, you're probably starting to see that the toolbox is truly a huge collection of subroutines with vast capabilities. There are some genuinely neat features of the Menu Manager that we just won't get to in this course due to lack of time and respect for America's forests.

While the Menu Manager is still fresh in your mind, I'd like to spend a moment and look at some of the capabilities that we've skipped. Like I said, some of these will pop up again later, and others won't. For the ones that aren't covered later in the course... well, that's why I've been encouraging you to look up toolbox calls in the toolbox reference manuals as you work through this course. If you do that, you'll get used to finding information in the toolbox manuals and using that information in your own programs. I've talked to a lot of people who have been scared off by the toolbox manuals, and I really can't blame those of you who have found them to be, shall we say, obtuse. Remember, though: the toolbox reference manuals were never written to teach you to use the toolbox. They were written for people who already know basically what the toolbox could do, and who wanted to find information quickly. This course will get you to the point that you know the concepts behind the toolbox, and hopefully you'll get used to the reference manuals as you go. In other words, even if you've found the toolbox reference manuals tough to use, once you finish this course, they should make a lot of sense, and should become a regularly used part of your programming reference library.

Getting back to the topic, you've seen how to create and modify menu items. You can also add menu items to a menu, or remove menu items that are already in a menu. There are a lot of reasons to do this; one common one is to create a menu that lists all of the open windows on the desktop. That way, you can use the menu to check to see what windows are available and bring them to front, even if the window you want is buried under a lot of other windows.

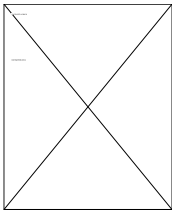

Figure 2-6: PRIZM's Windows Menu

You can also add and remove entire menus. That's not used nearly as often, but there are some really good reasons to do it on occasion. The most common reason to add and remove menus is when a program has context sensitive features, and it also has so many features that you can't show all of the menus at once. Just as one example, AppleWorks GS has several different applications that are all combined into a single program. The menus for the paint program and for the word processor won't fit on the menu bar at the same time, and you don't need all of them at once anyway, so AppleWorks GS deletes all of the word processor specific menus when you select a paint window, and draws all of the paint menus in their place.

We've used standard text menus in this lesson, but you've probably used programs that seem to do something very different with a menu. For example, Apple's Finder has a color picker menu that lets you pick colors for icons. Menus like the Finder's color picker menu are called custom menus. The Menu Manager doesn't even handle these. Instead of using Menu Manager calls to set up a menu and then letting the Menu Manager handle all of the details, you have to write the subroutines to handle all of the details yourself. The Menu Manager defines the

subroutines you need, along with all of the parameters, and then the Menu Manager calls your program to draw the menu, select various items in the menu, and depends on you to tell it which item (if any) was selected. That's a lot of work, but the end result is that you can do almost anything you want with a menu by rolling your own with custom menus. If you want to create custom menus in GSoft BASIC programs, you'll have to write a user tool set to do it.

There are also a host of minor calls you can use to do various things, ranging from actions that are almost silly to those that are useful, but just not used often enough to cover here. For example, you can change a menu item's style, like whether or not it is bold; you can flash the menu bar; you can change the color of the menu bar; and you can change the name of a menu, just like we changed the name of a menu item. Browsing through the toolbox reference manual is a great way to find out what capabilities are there, then you can go back and study the details if you ever need to use one of the features.

Summary

This lesson dealt mostly with the menu bar, but we also started using `TaskMaster` to control our event loops. By now, you should be able to create a program that will mimic the menu bar on almost any program you use that doesn't have a custom menu. Your programs should be supporting desk accessories, and you should be getting comfortable with the event loop and how you plan programs around an even loop.

Tool calls used for the first time in this lesson:

```
CheckMItem      DisableMItem    DrawMenuBar     EnableMItem
FindWindow      FixAppleMenu    FixMenuBar      HiliteMenu
InsertMenu      MenuKey         MenuSelect      NewMenu
SetMenuFlag     SetMItem        TaskMaster
```

Lesson 3 – Be Resourceful

Goals for This Lesson

This lesson introduces the concept of resources. We start by looking at what resources really are, and why they exist. Then we'll learn how to use Rez, the resource compiler, to create a resource fork. As a practical example of resources, we'll change a program from the last lesson to use resources for the menu bar.

You'll need to use the ORCA Shell to work with Rez; it's included with the course. See Appendix ?? for an overview of the ORCA Shell, and Appendix ?? for an overview of the Rez compiler.

What Are Resources?

Apple has always aimed high, and the folks who created resources for the Macintosh were no exception. Apple wanted to go after a global market, but there was a serious problem with that idea: it costs a lot to develop software, and very few companies could really afford to develop software in, say, Danish to support a relatively small market in Denmark. What Apple needed was a way for software developers to create a program so the Danish folks could convert the software to their own language on their own. One way to do that, of course, was to convince all of the world's software developers that they should give away source code with the programs. That, to say the least, would be an uphill fight. Instead, Apple's team of programmers invented resources.

Keep in mind that the whole goal, at least so far, is just to create some mechanism so that strings can be changed from one language to another by the end user. One way to do that would be to have the program read the strings from a file, using some sort of numbered index to find a string in case the length of some of the strings changed. Let's take a look at a simple example. Suppose you are trying to stuff the strings for our File menu into a file that can be changed by the end user of the program. The two strings are " File " and "Quit". Our goal is to put these strings into a file in such a way that we can find either of them, even if the lengths of the strings change or something else entirely is added to the file. One way to do that is to write a series of records to the file. Each record will consist of four parts:

length	A length word, which tells our program how far to skip ahead to find the start of the next record.
type	A number telling us what kind of information this record contains. If all we put in the file are strings we don't need this field, but we'll keep things general just in case we want to add more stuff later on.
ID	A number that uniquely identifies this particular string.
value	The values that make up the resource itself.

Now lets assume that we need to look for a string, and we've decided that we will use the number 0x8006 for the type of a string. Why pick such an odd number? More on that in a

moment. Furthermore, we'll assume that the ID number for the string "Quit" is 1. Then to find this string, we would open the file containing the records and scan through the list, using the length word to skip from one resource to the next. Any time the type of a resource is 0x8006, we stop and check the ID. When we find the ID 1, we stop and read the string from the file, building our menu from the string.

Let's build a short file like this to see how it might look. We'll put a word of 0 on the end to mark the end of the file. Decoded, we might type in the file as a series of numbers and strings, and use a program to build the actual file. Using a special language to create the resource file doesn't change what we are really doing, it just makes it a lot more convenient for us to read and change the resources. In our imagined resource file, the strings for our menus might look like this:

```
resource1.type = 0x8006;
resource1.ID = 1;
resource1.value = "Quit";

resource2.type = 0x8006;
resource2.ID = 2;
resource2.value = "  File  ";
```

Details like putting the zero at the end of the file and figuring out the length of each resource are the sorts of things we should expect a resource compiler to do for us. Once this file is processed, the binary file would look like this:

```
0x0000: 0B000680 01000551 7569740F 00068002  '       Quit     '
0x0010: 00082020 46696C65 20200000           '    File    '
```

It would take some work, but you can probably see how you could write a program that could read this file and create the File menu using the strings from the file, rather than hard coded strings in your program, like we've used in the first few lessons. Since your program would only need to know that the resource type was 0x8006 and the resource numbers for the strings were 1 and 2, you also would be creating a very flexible program. After all, you really don't care whether or not "Quit" and " Edit " are really the strings in the file, as long as the numbers used to identify the strings stay the same. Well, in a nutshell, that's exactly what Apple Computer did with resources. In fact, 0x8006 happens to be the resource type Apple Computer has assigned for p-strings, and resource IDs of 1 and 2 would work perfectly well.

There are some differences, of course. The biggest is where the resources are actually stored. If the resources were really put in a separate file, like we did in this thought experiment, you would run the risk of separating the file from the program, making things more complicated for the user of the program that we'd like to. On the other hand, if you stick the resources right into the program file, you run into some other problems. The biggest is wrapped up in the fact that resources turned out to be a very powerful idea, and folks started using them for a lot of things besides just creating string files so a program could be adapted for other languages. There are lots of times when it's nice to keep some of the information about a program in a separate spot that can be changed without changing the source code for a program. In fact, there are even cases when a resource is used with a data file, and not with an executable program at all. And that's the rub: if you stuff the resources into the executable file, which is certainly possible, you'd also have to stuff them into a data file, and that would cause all sorts of problems for

people trying to read the data from, say, a picture, and tripping over resources in the process. Apple solved this problem by creating a file format with two separate parts, called the resource fork and the data fork. The data fork is basically what you normally think of as the file, while the resource fork is a special part of the file that you don't normally see, and can't access with standard file access calls from a language like GSoft BASIC. The resource fork is hidden and out of the way, but still a part of the file. To read, change, or add resources, you make special calls to a tool called the Resource Manager. It's also the Resource Manager that handles little details like figuring out where a resource is in a file, and when it has hit the end of the list of resources. Does it use a length word and a zero terminator, like we did? Who cares? It gets the job done, and in fact the details might even change as Apple's engineers discover new ways to deal with resources.

Using Rez to Create a Menu Bar

When you set out to create a program, you have a choice of a lot of different tools. You could pick a BASIC interpreter, like the one we're using in this course, or a C compiler, or an assembler. In the end, though, you always end up with an executable program. There are also a lot of different ways to create a resource fork. The first one we'll look at is Rez, Apple's resource compiler. Like BASIC or C, Rez tries to make programming a little easier by giving you a language that lets you express your ideas without getting down to the byte level. Using Rez, you don't know or care about the exact structure of the resource fork, just as you don't know or care about 65816 assembly language or object module formats when you write a program in GSoft BASIC.

Rez is a whole new language, though, so there's a lot to learn. Fortunately, it's also patterned after the C language, so it's pretty easy to pick up (especially if you're a C programmer).

We'll learn about Rez and resources gradually, introducing new ideas as we go along. If you would like to know more about Rez, you can refer to any of the places where the Rez compiler is documented, including Appendix ?? in this book.

That also brings up one other point. This course includes a copy of the ORCA Shell and the Rez compiler. You'll have to use the ORCA Shell to use the Rez compiler; fortunately, there's a version of GSoft BASIC that can be used from the ORCA Shell, too, so this isn't much of a hardship.

Appendix D tells you how to install Rez, and talks about some of the issues you will face if you don't have a hard disk.

A Menu Bar Using Rez

When we looked at what resources really are at the start of this lesson, we really only talked about strings, but the Resource manager can actually stuff anything into a resource. The Menu Manager has quite a few calls that can get information directly from the resource fork. One of the most useful is `NewMenuBar2`, which creates a whole menu bar in one step from resources. We'll start with the Rez source file to create the menu bar, then step through the source file to

see how it's built. Later, we'll use Rez to compile the source file, creating a resource fork with our menu bar resources.

```
#include "types.rez"

resource rMenuBar (1) {                 /* the menu bar */
   {
      1,                                /* resource numbers for the menus */
      2,
      3
      };
   };

resource rMenu (1) {                    /* the Apple menu */
   1,                                   /* menu ID */
   refIsResource*menuTitleRefShift      /* flags */
      + refIsResource*itemRefShift
      + fAllowCache,
   1,                                   /* menu title resource ID */
   {257};                               /* menu item resource IDs */
   };

resource rMenu (2) {                    /* the File menu */
   2,                                   /* menu ID */
   refIsResource*menuTitleRefShift      /* flags */
      + refIsResource*itemRefShift
      + fAllowCache,
   2,                                   /* menu title resource ID */
   {255,256};                           /* menu item resource IDs */
   };

resource rMenu (3) {                    /* the Edit menu */
   3,                                   /* menu ID */
   refIsResource*menuTitleRefShift      /* flags */
      + refIsResource*itemRefShift
      + fAllowCache,
   3,                                   /* menu title resource ID */
   {                                    /* menu item resource IDs */
      250,
      251,
      252,
      253,
      254
      };
   };

resource rMenuItem (250) {              /* Undo menu item */
   250,                                 /* menu item ID */
   "Z","z",                             /* key equivalents */
   0,                                   /* check character */
   refIsResource*itemTitleRefShift      /* flags */
      + fDivider,
   250                                  /* menu item title resource ID */
   };

resource rMenuItem (251) {              /* Cut menu item */
   251,                                 /* menu item ID */
   "X","x",                             /* key equivalents */
   0,                                   /* check character */
   refIsResource*itemTitleRefShift,     /* flags */
   251                                  /* menu item title resource ID */
```

```
   };

resource rMenuItem (252) {                /* Copy menu item */
   252,                                   /* menu item ID */
   "C","c",                               /* key equivalents */
   0,                                     /* check character */
   refIsResource*itemTitleRefShift,       /* flags */
   252                                    /* menu item title resource ID */
   };

resource rMenuItem (253) {                /* Paste menu item */
   253,                                   /* menu item ID */
   "V","v",                               /* key equivalents */
   0,                                     /* check character */
   refIsResource*itemTitleRefShift,       /* flags */
   253                                    /* menu item title resource ID */
   };

resource rMenuItem (254) {                /* Clear menu item */
   254,                                   /* menu item ID */
   "","",                                 /* key equivalents */
   0,                                     /* check character */
   refIsResource*itemTitleRefShift,       /* flags */
   254                                    /* menu item title resource ID */
   };

resource rMenuItem (255) {                /* Close menu item */
   255,                                   /* menu item ID */
   "W","w",                               /* key equivalents */
   0,                                     /* check character */
   refIsResource*itemTitleRefShift        /* flags */
      + fDivider,
   255                                    /* menu item title resource ID */
   };

resource rMenuItem (256) {                /* Quit menu item */
   256,                                   /* menu item ID */
   "Q","q",                               /* key equivalents */
   0,                                     /* check character */
   refIsResource*itemTitleRefShift,       /* flags */
   256                                    /* menu item title resource ID */
   };

resource rMenuItem (257) {                /* About menu item */
   257,                                   /* menu item ID */
   "","",                                 /* key equivalents */
   0,                                     /* check character */
   refIsResource*itemTitleRefShift        /* flags */
      + fDivider,
   257                                    /* menu item title resource ID */
   };


                                          /* the various strings */
resource rPString (1, noCrossBank)        {"@"};
resource rPString (2, noCrossBank)        {"  File  "};
resource rPString (3, noCrossBank)        {"  Edit  "};
resource rPString (250, noCrossBank)      {"Undo"};
resource rPString (251, noCrossBank)      {"Cut"};
```

```
resource rPString (252, noCrossBank)    {"Copy"};
resource rPString (253, noCrossBank)    {"Paste"};
resource rPString (254, noCrossBank)    {"Clear"};
resource rPString (255, noCrossBank)    {"Close"};
resource rPString (256, noCrossBank)    {"Quit"};
resource rPString (257, noCrossBank)    {"About Frame..."};
```

Listing 3-1: A Menu Bar Resource Description File

How Rez Files are Typed

Comments are indicated by lines that start with two slashes, or are bracketed by "/*" and "*/", like this:

```
/* this is a comment */

// so is this
```

The { and } characters form blocks of lines — they're used to group lists of data, or to indicate data that's initializing a record. See Appendix ?? for more information on how this works.

If you know the C programming language, the Rez syntax will seem somewhat familiar, although there are differences. For example, the syntax of a `resource` statement is much like a cross between a function definition and initialization of a structure.

The first line of the resource description file is

```
#include "types.rez"
```

This includes the definitions from the file 13:RInclude:types.rez when compiling your project; any constants or resource types defined in types.rez will be available to your resource file. It contains a whole lot of type statements and constant declarations. It defines an interface to the current set of resources for the Apple IIGS toolbox, just like the Menu Manager's header file for C defines all of the tool calls and data structures in the Menu Manager.

Like any source code files, you can open and read the Rez header file. If you would like to look at the source for the header file, use the command

```
edit 13:RInclude:Types.rez
```

The first resource in our file creates a menu bar; the resource looks like this:

```
resource rMenuBar (1) {                 /* the menu bar */
   {
      1,                                /* resource numbers for the menus */
      2,
      3
      };
   };
```

Later on we'll come back and see how you could figure out the format for this resource from types.rez, but for now let's concentrate on the syntax you use to type the resource.

Each resource starts with the reserved word `resource`. Right after this is the name of the resource we want to create; this matches one of the type declarations in types.rez. In this case, we are defining a menu bar resource, which has a name of `rMenuBar`. The last part of the header for the resource is the resource ID, enclosed in parenthesis. This is a number you pick; it can be any number from 1 to 65535, as long as there are no other `rMenuBar` resources in your resource fork with the same resource ID. You can have other resources with a resource ID of 1; you just can't have another `rMenuBar` resource with a resource ID of 1.

The body of the resource fills in the actual information that the program will use. The body of a resource is enclosed in curly brackets, and is always followed by a semicolon.

This particular resource consists of an array of menu resource IDs. Arrays are variable length groups of values in a resource description file, not fixed length structures like they are in GSoft BASIC. The resource compiler figures out how long the array is by counting the number of things you put in the array. The array itself is enclosed in brackets, again, and followed by a semicolon.

The array itself has three values, separated by commas. These values are the resource IDs for three more resources, each of which is a menu resource. Here's the resource for the File menu, which is the menu with a resource ID of 2:

```
resource rMenu (2) {                    /* the File menu */
   2,                                   /* menu ID */
   refIsResource*menuTitleRefShift      /* flags */
      + refIsResource*itemRefShift
      + fAllowCache;
      2,                                /* menu title resource ID */
      {255,256};                        /* menu item resource IDs */
   };
```

The resource ID of 2 is easy to pick out of the header, but the same value is used twice more in the resource. The first time is for the menu ID; this is the number we use for Menu Manager calls, and the number `TaskMaster` sends back to us to tell us which menu has been highlighted. Later on, we use 2 again for the resource number for the menu title, which is yet another resource containing a string. All of this is just a handy convention we are using to keep all of the various resources straight in our head. While we need to know the menu ID inside of our program, as you already know, we can pick whatever number we want. We could also pick any number we want for the resource ID of this resource, as long as we used the same number in the `rMenuBar` resource array. And, of course, we can use any resource ID we want for the string resource that is the title for the menu. By using 2 for all three values, though, it's a little easier to scan the resource file for all of the resources that are used together.

The flags field in this resource takes the place of the flags characters we used to use in a menu title string. In a resource description file, the title for a menu is just the name of the menu itself, and all of the other information, like the menu ID and the various format flags, are in other parts of the `rMenu` resource. We also need to tell the Resource Manager that the title is actually a resource. We could also make it a pointer or a handle, but frankly, that's a lot of trouble and defeats the whole purpose of resources. We will always use a resource for the title of a menu it this course, so you will always see the value `refIsResource*menuTitleRefShift` in the flags field. This value just uses some predefined constants to set one bit in the flags value.

This menu has two items, “Close” and “Quit”. As you probably guessed by now, these are resources, too. The resource IDs are in the array at the end of the `rMenu` resource. We also need to set another flag to tell the Resource Manager that the menu items are in resources; that’s what the value `refIsResource*itemRefShift` in the flags word does. The last flag is `fAllowCache`; this is actually the only flag that is used by the Menu Manager for something that isn’t directly involved with resources. It tells the Menu Manager to use menu caching, a technique that uses a little extra memory to remember a picture of the menu when it is pulled down so the menu can be redrawn quicker.

The resource for the Quit menu item is a pretty typical example of the menu item resources:

```
resource rMenuItem (256) {                  /* Quit menu item */
   256,                                     /* menu item ID */
   "Q","q",                                 /* key equivalents */
   0,                                       /* check character */
   refIsResource*itemTitleRefShift;         /* flags */
   256                                      /* menu item title resource ID */
   };
```

In this resource, we’re using the convention of keeping all of the numbers the same, again. This time we use the value for the menu item ID, which is the same value we created in our older programs with “V256” in the options part of the menu item string. The resource ID, shown in parenthesis at the start of the resource, and the resource ID for the menu item title string are both 256.

The various items in the rest of the resource are once again replacements for the information we used to put in the menu item string. In this case, we’re telling the Menu Manager that the menu item has a keyboard equivalent of “Q”, with an alternate character of “q”. When we created this keyboard equivalent in the last chapter with the characters “*Qq” in the menu item string, the first character was the one that was actually drawn in the menu item when the menu was pulled down, and that’s still true. If we didn’t want any keyboard equivalents, fill in the entry with a pair of empty strings, like this:

```
   "","",                                   /* no key equivalents */
```

There’s also a special place for the check character that shows up to the left of the menu item; in this example, it’s a zero, which tells the Menu Manager that we don’t want a check mark. You can use an integer value here, since a single character is equivalent in every way to a number, just as it is in GSoft BASIC. Keep in mind that Rez expects a character here, not a string.

The flags field is used for all of the other options that used to go in the menu item string, as well as for another flag that tells the Menu Manager that the name of the menu item is a resource; that last flag is the only one you see here. We’ll look at the other flags a little later, after actually creating a program with this resource file.

The only kind of resource left in the resource description file is the string. The strings for all of the menu titles and menu item titles have been collected at the end of the file. Here’s a typical example:

```
resource rPString (256, noCrossBank) {"Quit"};
```

The `rPString` resource is used to define p-strings, which are strings with a leading length byte. The details of the internal format aren't really important, since the resource compiler knows what to do based on the declarations in the Types.rez header file. The only entry in the body of the resource is the string itself.

There's a new flag after the resource ID, though. The `noCrossBank` flag tells the Resource Manager not to load the resource in a position where it would span a 64K bank boundary in memory. The Menu Manager doesn't work right if the string does cross a bank boundary, and without this flag, the Resource Manager would be free to load the resource to any location in memory that was big enough.

Using Constants

I tried to keep things simple for your first look at a resource description file by keeping all of the values out in the open, where you could see them. In real life, though, using constants in a resource description file is a great idea for the same reasons you use constants in a program. By using constants, you can gather all of the "magic numbers" into one spot at the start of the program. That's a big help when you're looking for problems or making changes. Also, keep in mind that the resource ID numbers and menu ID numbers used in the resource description file have to match the numbers used in the C program for things to work correctly. By collecting these numbers as constants at the beginning of the program, you can change and check the values a lot easier.

Here's the same resource description file we just went over, recoded to use #define statements for constants:

```
#include "types.rez"

#define appleMenu     1
#define fileMenu      2
#define editMenu      3
#define editUndo      250
#define editCut       251
#define editCopy      252
#define editPaste     253
#define editClear     254
#define fileClose     255
#define fileQuit      256
#define appleAbout    257

resource rMenuBar (1) {                 /* the menu bar */
   {
      appleMenu,                        /* resource numbers for the menus */
      fileMenu,
      editMenu
      };
   };

resource rMenu (appleMenu) {            /* the Apple menu */
   appleMenu,                           /* menu ID */
   refIsResource*menuTitleRefShift      /* flags */
      + refIsResource*itemRefShift
      + fAllowCache,
   appleMenu,                           /* menu title resource ID */
   {appleAbout};                        /* menu item resource IDs */
   };

resource rMenu (fileMenu) {             /* the File menu */
   fileMenu,                            /* menu ID */
   refIsResource*menuTitleRefShift      /* flags */
      + refIsResource*itemRefShift
      + fAllowCache,
   fileMenu,                            /* menu title resource ID */
   {fileClose,fileQuit};                /* menu item resource IDs */
   };

resource rMenu (editMenu) {             /* the Edit menu */
   editMenu,                            /* menu ID */
   refIsResource*menuTitleRefShift      /* flags */
      + refIsResource*itemRefShift
      + fAllowCache,
   editMenu,                            /* menu title resource ID */
   {                                    /* menu item resource IDs */
      editUndo,
      editCut,
      editCopy,
      editPaste,
      editClear
      };
   };

resource rMenuItem (editUndo) {         /* Undo menu item */
   editUndo,                            /* menu item ID */
   "Z","z",                             /* key equivalents */
```

```
    0,                                  /* check character */
    refIsResource*itemTitleRefShift     /* flags */
       + fDivider,
    editUndo                            /* menu item title resource ID */
    };

 resource rMenuItem (editCut) {         /* Cut menu item */
    editCut,                            /* menu item ID */
    "X","x",                            /* key equivalents */
    0,                                  /* check character */
    refIsResource*itemTitleRefShift,    /* flags */
    editCut                             /* menu item title resource ID */
    };

 resource rMenuItem (editCopy) {        /* Copy menu item */
    editCopy,                           /* menu item ID */
    "C","c",                            /* key equivalents */
    0,                                  /* check character */
    refIsResource*itemTitleRefShift,    /* flags */
    editCopy                            /* menu item title resource ID */
    };

 resource rMenuItem (editPaste) {      /* Paste menu item */
    editPaste,                          /* menu item ID */
    "V","v",                            /* key equivalents */
    0,                                  /* check character */
    refIsResource*itemTitleRefShift,    /* flags */
    editPaste                           /* menu item title resource ID */
    };

 resource rMenuItem (editClear) {       /* Clear menu item */
    editClear,                          /* menu item ID */
    "","",                              /* key equivalents */
    0,                                  /* check character */
    refIsResource*itemTitleRefShift,    /* flags */
    editClear                           /* menu item title resource ID */
    };

 resource rMenuItem (fileClose) {       /* Close menu item */
    fileClose,                          /* menu item ID */
    "W","w",                            /* key equivalents */
    0,                                  /* check character */
    refIsResource*itemTitleRefShift     /* flags */
       + fDivider,
    fileClose                           /* menu item title resource ID */
    };

 resource rMenuItem (fileQuit) {        /* Quit menu item */
    fileQuit,                           /* menu item ID */
    "Q","q",                            /* key equivalents */
    0,                                  /* check character */
    refIsResource*itemTitleRefShift,    /* flags */
    fileQuit                            /* menu item title resource ID */
    };

 resource rMenuItem (appleAbout) {      /* About menu item */
    appleAbout,                         /* menu item ID */
    "","",                              /* key equivalents */
```

```
   0,                                    /* check character */
   refIsResource*itemTitleRefShift       /* flags */
      + fDivider,
   appleAbout                            /* menu item title resource ID */
   };

                                         /* the various strings */
resource rPString (appleMenu, noCrossBank)  {"@"};
resource rPString (fileMenu, noCrossBank)   {"  File  "};
resource rPString (editMenu, noCrossBank)   {"  Edit  "};
resource rPString (editUndo, noCrossBank)   {"Undo"};
resource rPString (editCut, noCrossBank)    {"Cut"};
resource rPString (editCopy, noCrossBank)   {"Copy"};
resource rPString (editPaste, noCrossBank)  {"Paste"};
resource rPString (editClear, noCrossBank)  {"Clear"};
resource rPString (fileClose, noCrossBank)  {"Close"};
resource rPString (fileQuit, noCrossBank)   {"Quit"};
resource rPString (appleAbout, noCrossBank) {"About Frame..."};
```

Listing 3-2: A Better Menu Bar Resource Description File

Compiling the Rez File

The ORCA family of languages is designed to handle more than one language at the same time, which is fortunate, since the Rez compiler is actually another language. If you load the resource description file from disk and check the language from the editor, you will find that the language stamp for the file is Rez instead of BASIC, CC or Shell, like you're used to. If you type in the file from scratch, you'll have to be sure to set the language yourself. Exactly how you do that depends on which editor you are using. If you need help, refer to the documentation for the editor in the reference manual. As a last resort, check out the CHANGE shell command.

You compile the file using the COMPILE command, like this:

```
compile Frame.rez keep=Frame
```

This will cause the compiled resources to be saved in the file Frame. If there's already a file Frame, the resource fork of the file is replaced with the new resources. Otherwise, a new file named Frame is created. If your GSoft BASIC program is named Frame, this will add the resources to your GSoft BASIC program.

Problem 3-1: Type in the resource description file from the last section or load it from disk. Compile it, using a keep name of Frame.bas. Once you finish, use the CATALOG command from the ORCA shell. You should see a + beside the file type for Frame.bas. The + is the shell's way of flagging an extended file. Extended file is the formal name for a file with a resource fork under the Apple IIGS operating system.

Using the Menu Bar Rez Created

Creating the resource description file involved a lot of new stuff, and ended up being pretty tough because of these new concepts. Well, it's payback time. Creating the menu bar was hard, but using it is actually pretty easy. Listing 3-3 shows the complete Frame program to use the menu bar we just created with Rez.

There really isn't much difference between this program and the one you had developed on your own by the end of the last lesson. In fact, other than changing some comments, all of the differences are in `InitMenus`, where the long series of calls we used to use to build up a menu string and create a series of menus have been replaced by these three calls:

```
menuBar = NewMenuBar2(refIsResource, 1, NIL)
SetSysBar(ctlRecHndl(menuBar))
SetMenuBar(NIL)
```

The first call creates the menu bar itself, using a Menu Manager call from the *Apple IIGS Toolbox Reference: Volume 3*. The first parameter, `refIsResource`, is a constant that tells the Menu Manager that the menu bar description is in a resource file, while the second parameter tells the Menu Manager which resource ID to look for. The last parameter tells the Menu Manager that we want to create a menu bar that is not inside of a window. Once we have the menu bar, we still need to install it as the system menu bar. The call to `SetSysBar` makes our new menu bar the system menu bar, and the call to `SetMenuBar` makes the system menu bar the current, or active, menu.

Like so many other parts of our basic program, these calls are generally figured out once, stuffed into a function like `InitMenus`, and moved from program to program as a block, without really worrying too much about the details. Frankly, if you were to ask me what calls have to be made, and in what order, to create a system menu bar from a resource when I wasn't working on something like this course, I probably couldn't tell you. About all I could be sure of is that I have a subroutine in my collection of libraries that does it right. When I write a new program, I just copy the working code from a working program, and don't worry about the details.

```
dim done as integer               : ! tells if the program should stop
dim userID as Integer             : ! our user ID
dim event as integer              : ! event #; returned by TaskMaster
dim lastEvent as eventRecord      : ! last event returned in event loop
dim startStopParam as long        : ! tool startup/shutdown information

! Initialize the program
userID = MMStartUp
LoadLibrary 3
UtilStartUp(userID)
StartDesk(640)
if toolError <> 0 then
  print "Unable to start up desktop."
  end
end if

call InitMenus
lastEvent.taskMask = $1FFF
ShowCursor
InitCursor

! Main event loop
done = 0
while not done
   event = TaskMaster($076E, lastEvent)
   select case event
      case wInSpecial, wInMenuBar
         call HandleMenu(lastEvent.taskData, done)
   end select
wend

EndDesk
UtilShutDown
UnloadLibrary 3
end

!
! InitMenus
!
! Initialize the menu bar.
!
sub InitMenus
  menuID% = 1     : ! menu bar resource ID
  dim height as integer           : ! height of the largest menu
  dim m as menuBarHandle          : ! our menu bar's handle

  ! create the menu bar
  m = NewMenuBar2(2, 1, NIL)
  SetSysBar(ctlRecHndl(m))
  SetMenuBar(NIL)

  ! add desk accessories
  FixAppleMenu(1)

  ! draw the completed menu bar
  height = FixMenuBar
  DrawMenuBar
end sub
```

```
!
! HandleMenu
!
! Handle a menu selection.
!
! Parameters:
! taskData - taskData field for the menu event
! done - event loop done flag
!
sub HandleMenu (taskData as long, done as integer)
  select case LoWord(taskData)
    case 256 : ! File Menu, Quit
      done = 1
  end select
  HiliteMenu(0, HiWord(taskData))
end sub
```

Listing 3-3: A Program for the Resource Description File

☒ **Tip** If you look closely, you'll see that the formatting for the switch statements has changed a lot. The format you see here works very well for the event loop and menu selection subroutine if you use a single short statement or function call to do the actual work. ☒

Problem 3-2: Type in this program, using the file name Frame.bas, or load it from the disk. Compile the Rez code from problem 3-1 with the keep name Frame.bas, then run the program.

Making Changes

When you compile a resource description file to create a resource fork, the resource compiler either erases the existing resource fork and creates a new one, or adds to the existing resource fork, depending on the flags you use. The resource compiler doesn't touch the data fork of the file at all. If there was a data fork when the resource compiler started, there will still be one when the resource compiler finishes, and its contents will not be changed.

The ORCA editor doesn't delete resources attached to the BASIC program file, so you can edit the BASIC code without having to recompile the resource description file every time. This is very convenient.

Understanding Resource Description Files

As we create the various resources in this course, we'll always do it the way we did for menus in this lesson, carefully laying out the format for the resource itself and describing what goes in each field. Someday, though, you'll need to create a resource for one of your programs that isn't covered in this course, and you'll need to have some idea of how to do it. In this section, we'll take a look at where you can get the information about these other resources.

Toolbox Resource Descriptions

In *Apple IIGS Toolbox Reference: Volume 3* you'll find an appendix that describes the various resources used by the toolbox. You'll have to do some comparing between the information in Appendix E and the tool calls that use the resources to figure out exactly what is going on, so we'll use an example that you already know something about to look at the layout of this appendix. On page E-59 you'll find a section describing `rPString`, the resource we used for the names of the menus and menu items. It looks like this:

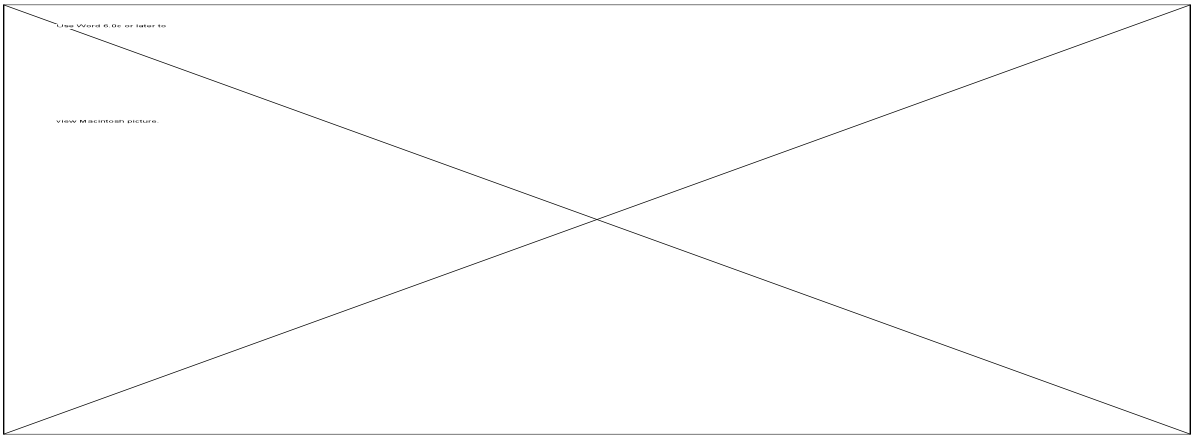

Figure 3-1: Resource Description from the Toolbox Reference Manuals

As you can see, Apple assumes you already know why you want to use the resource and basically what it is used for. You'd learn that from reading about the various tool calls or resources that make use of this resource.

The table itself gives you the internal format for the data in the resource. You can see that the data starts with a length byte and is followed by characters. That makes sense, given that this is a p-string. After all, that's the format for a p-string in the rest of the toolbox, too.

Rez Types

Appendix E often has some useful information about what sorts of things can go in the resource, but when it comes time to actually type in the resource description file, you need to load a copy of Types.Rez. Types.Rez is in the RInclude folder, which is in your libraries folder. Doing a string search in Types.Rez, you would find this entry for `rPString`:

```
#define rPString            $8006

/*---------------------- rPString ----------------------*/
type rPString {
        pstring;                    /* String */
};
```

Listing 3-6: `rPString` Resource Definition from Types.rez

Note In the C language, hexadecimal numbers start with 0x; in most other languages on the Apple IIGS, hexadecimal numbers start with $. In Listing 3-6, you can see an example of a hexadecimal constant which starts with $. The resource compiler accepts either format, so you could use 0x8006 instead.

Apple's resource description files generally use the $ character to start a hexadecimal number, so I will generally use the $ character in resource files in this course. In every case, though, you can use 0x instead.

This is the actual type declaration the resource compiler uses when it compiles your resource description file. It's sort of like a GSoft type declaration, while your resource in the resource description file is sort of like an initialized variable. This type declaration tells the resource compiler that an `rPString` resource has a resource type number of $8006, and that it contains a single piece of information, a p-string.

The only really tricky feature in a resource type is the array. Here's the type declaration for `rMenuBar`, which has an array of resource IDs for the various menus in the menu bar:

```
#define rMenuBar            $8008

/* ----------------------- rMenuBar --------------------------------*/
type rMenuBar {
    integer = 0;             /* version must be zero */
    integer = 0x8000;        /* the following refs are all menu resID's */
    array {
        longint;             /* menu template ID list */
        };
    longint = 0;
};
```

Listing 3-7: `rMenuBar` Resource Definition from Types.rez

Let's compare this to the `rMenuBar` resource from our resource description file:

```
resource rMenuBar (1) {      /* the menu bar */
   {
      appleMenu,             /* resource numbers for the menus */
      fileMenu,
      editMenu
      };
   };
```

Listing 3-8: `rMenuBar` Resource Sample

One thing that stands out is the first two integers in the `rMenuBar` type. They aren't in the resource. These are actually fixed values, set to specific values in the type itself, so we don't need to put them in the resource; you only need to code a value in a resource if it is a variable in

the type. In fact, the array of resource IDs also ends with a long integer 0, which is also coded as a constant in the resource type.

The array itself is an array of long integers. Any time you see an array in a resource type, you code as many of the values as you like in your resource, separating each value with a comma.

Finding Out More About Rez

Like I said, as we go through the course, each time we use a new resource we'll stop and lay it out carefully. If you also take the time to look up the resource in Appendix E and in Types.Rez, you'll learn quite a bit about the resource compiler. If you would like to read a very detailed technical description of the resource compiler and how it is used to create resources, you can find detailed technical information in Appendix ??.

Resource Tools

Changing Resources

The whole point of resources is that they give you a way to change the program without the source code. The Rez compiler can create a resource fork, but to change a resource in an existing program, you also need some way to find out what resources already exist. One of the many ways to do this is by decompiling the resource fork with DeRez, making some changes, and then recompiling the resource fork with Rez. In this course, we're really concerned with creating new resources, not changing the resources in programs that someone else has written, so we won't take up any time talking about the mechanics of changing a resource fork. I just thought you should know that changing the resource fork is possible, and what you need to use if you want to give it a try.

Programmer's CAD Tools

I've talked with a lot of people who want to learn to use resources in their programs because they think using resources will make it easier to create a program. Well, hopefully this lesson has shattered that completely false myth. The point of resources was never to make it easier to write the program in the first place; the point of resources is to make it possible to change a program without changing, or even having access to, the source code.

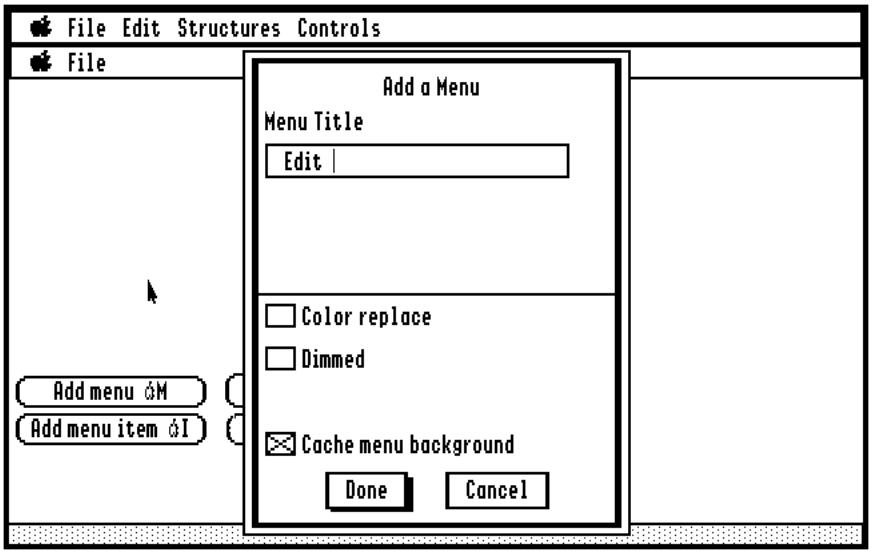

Figure 3-2: Design Master Creates a Menu Bar

One of the reasons people have this mistaken impression is that they associate programs like Design Master with resources. Design Master is one of a class of programs I like to refer to as a programmer's CAD tool. With Design Master, you don't have to type in a lot of obscure entries to create a menu bar; instead, you create a menu bar by drawing it, and then Design Master creates the code for the menu bar itself. This really isn't all that critical with menu bars, and if menu bars were the hardest thing to create for a desktop program, I don't think tools like Design Master would really be all that important, and they might never have been invented at all. When you start laying out a dozen or so buttons in a dialog, though, a tool like Design Master can save you a lot of trouble. That's when it's really nice to be able to click on a button and drag it over a few pixels, instead of changing a value in a resource description file and recompiling it.

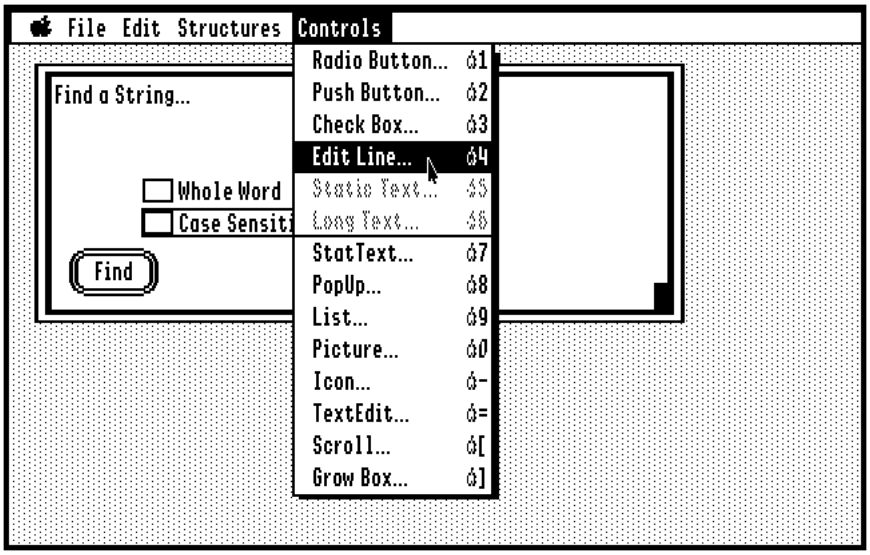

Figure 3-3: Design Master Creates a Dialog

Once you draw your menu bar, window, or whatever, Design Master essentially writes a part of a program for you, creating code that will create what you drew. In most cases you can create a resource fork directly, which is where the idea that Design Master and programs like it are resource compilers or resource editors come from; however, Design Master doesn't support generating GSoft BASIC code directly. However, Design Master is a wonderful tool. It can save you a lot of time, especially when you are creating windows and dialogs with lots of buttons and other controls. But the fact that Design Master can create resources is really incidental; it can generate code to create what you drew in a lot of formats, and resources are just one of many.

In this course, we'll always use Rez to create the resources for programs. There are two really good reasons for this:

1. A listing of a resource description file is a lot more precise on paper than telling you how to create something graphically with a tool like Design Master.
2. Rez comes with this course. You don't have to buy Design Master to create the programs in this course.

On the other hand, if you have Design Master or one of its cousins, put it to use, especially when you start creating windows and dialogs!

Summary

In this lesson we've made the switch to resources, changing one of our most complicated programs to use resources to see how it's done. We've learned what resources are, how to use Rez to create resources, and how to use scripts to automate the whole process of building the more complicated programs we're now creating.

A few other tools and concepts were mentioned briefly so you would know that they exist and basically what they are. These include Appendix E of *Apple IIGS Toolbox Reference: Volume 3*, which covers resource types used by the toolbox; Types.Rez, the main resource type file used by the resource compiler; DeRez, a utility that can decompile resources so you can changed them with Rez; and Design Master, a programmer's CAD tool that lets you draw complicated parts of a desktop program, and then create the resources for you.

Tool calls used for the first time in this lesson:

```
NewMenuBar2       SetMenuBar        SetSysBar
```

Resource types used for the first time in this lesson:

```
rMenu             rMenuBar          rMenuItem         rPString
```

Lesson 4 – Keep Alert!

Goals for This Lesson

Alerts are a simple kind of window used to display information and ask the user of the program for a simple, push-button response. This lesson covers how to create and use alerts. We'll also work on Frame.bas, the basic program we will use as the starting point for most of the programs in the rest of this course.

Alerts Present Messages

When something goes wrong in a text program, it's easy to plop in a

```
print "You goofed by doing this instead of that. "
```

in the program when the error is found. Writing a copyright message is just as easy. Letting the user pick between a couple of alternatives is a little tougher, but not much:

```
input "Do you really want to reformat your hard disk? (Y or N) ";YN$
```

Alerts are used to do this sort of thing in a desktop program. An alert is a small window, generally with no title bar (the lined thing at the top of most windows), no scroll bars, no grow box – in short, just a box on the screen where you can write some stuff. Figure 4-1 shows a typical alert. The picture at the top left of the alert is called the alert icon, several of which are predefined and used for specific purposes. The one you see here is a note alert, used in alerts that give you information, like a copyright message. This particular icon is also known as the "talking head." The message is generally text, but there's really nothing that says an alert has to be all text. The tools in the toolbox make it really easy to put text in an alert, since that is what you will need to do most of the time, but you could certainly put in some sort of a picture, or even an animation, if you prefer. Finally, most alerts have at least one button. A few, like an alert you put up to tell someone the program is busy printing a document, don't have a button; these alerts go away on their own. Most alerts have at least an OK button, though. Once you've read the text, you click on the OK button to make the alert go away so you can do something else. Alerts that offer choices will have several buttons, one for each choice.

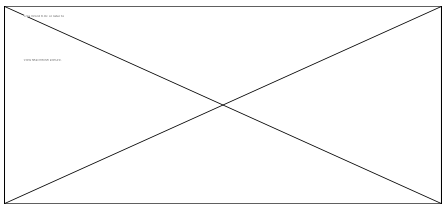

Figure 4-1: A Typical Alert

Before going too much further, I'd like to point out that there is another sort of temporary window called a dialog. This lesson does not deal with dialogs at all. Dialogs usually (but not always!) have something in them besides buttons, text messages an icons, and often support other features as well. From a visual standpoint, alerts are a specific subset of a larger class of things called dialogs, but you end up using completely different calls to create alerts and dialogs. We'll talk about dialogs later, after you know more about the various controls you can put in a dialog.

Using an Alert for an About Box

There are several ways to create an alert in your program, but one of the easiest is the `AlertWindow` call in the window manager. We'll start learning about alerts by creating an about box for our Frame program. As you know from using a lot of desktop programs, the about box is the traditional place to put the name of the program, copyright information, and the place for the programmer to take a bow. Figure 4-2 shows the about box we'll create. This about box will be drawn when we pick the About command from the apple menu, and will stay on the screen until the user clicks on the OK button.

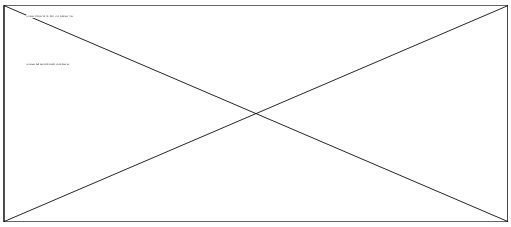

Figure 4-2: Our About Box

We'll use the Window Manager call `AlertWindow` to draw our about box.

```
sub DoAbout
  const alertID = 1
  dim b as Integer

  b = AlertWindow(awCString+awResource, NIL, alertID)
end sub
```

Listing 4-1: Subroutine to Create an Alert

This call doesn't return control to our program until the user clicks on a button, so all of the hard work is handled for us. When it does return, it returns the number of the button pushed; since our alert only has one button, we already know which one was pushed, and we throw the value away.

There are three parameters to `AlertWindow`:

`alertFlags` A flags word. As with most flags words in the toolbox, this one is a series of bit flags. Bits 1 and 2 tell what sort of parameter we are passing for `alertStrRef`:

00	`alertStrRef` is a pointer
01	`alertStrRef` is a handle
10	`alertStrRef` is a resource value

The least significant bit of `alertFlags` is a 0 if `subStrPtr` is an array of pointers to null-terminated strings (c-strings) and 1 if it is an array of pointers to strings with a length byte (p-strings).

In our case, we're passing the number 4, which indicates that the alert string is in a resource, and that substitution strings (we aren't using any) are c-strings.

See Appendix A for information about the other flag bits.

`subStrPtr` This is an array of substitution strings. Substitution strings give `AlertWindow` an awesome amount of flexibility in a very simple way. We'll look at substitution strings in detail a bit later in the lesson.

`alertStrRef` The handle, pointer, or resource ID for the alert string itself. In this course, we'll almost always pass a resource ID.

There's a lot to the alert string, and we'll get to those details a little later in this lesson. For now, let's concentrate on getting a program working with an alert string I'll provide. Since it is a resource, we'll need to add a resource description file.

```
resource rAlertString (1) {
  "43/"
  "Frame 1.0  Mike Westerfield\n"
  "Written in GSoft BASIC; includes the GSoft BASIC "
  "runtime interpreter, copyright 1998, Byte Works Inc."
  "/^#0\$00";
  };
```

Listing 4-2: Alert String for an About Box

Problem 4-1: Add the `DoAbout` function and the `rAlertString` resource to the Frame program from the last lesson. Call `DoAbout` when the About menu command is selected.

Don't peek ahead – the next section of this lesson gives the solution to this problem!

The Frame Program

I hope you worked Problem 4-1, or at least understand all of the principles behind it. If not, now is the time to take a breather and go back to review enough of what we've covered in this

course to understand exactly what is happening the the solution to Problem 4-1. The reason this problem is so important is that there is a lot of work invested in the basics of getting an event loop, menu bar, and about box all working together in a program. Virtually every desktop program you will ever use or write does these same things. It doesn't pay to reinvent the wheel for each and every program you write, so most programmers keep a basic program around that does all of this work, and simply add to it to create a new program. That's what we'll to in the rest of this course.

The two listings in this section show the GSoft BASIC source code and the resource description file for our Frame program. In almost every programming example for the rest of this course, I'll start with Frame and either add to it or make changes. In most cases, I'll just show you the new stuff, and one of the problems will be to add the new stuff to Frame to create a working program. Of course, if you get confused, the solutions to the problems are on the disk that comes with this course.

So, before you move on to the next lesson, be sure you understand this program!

```
dim done as integer               : ! tells if the program should stop
dim userID as Integer             : ! our user ID
dim event as integer                  : ! event #; returned by TaskMaster
dim lastEvent as eventRecord          : ! last event returned in event loop
dim startStopParam as long        : ! tool startup/shutdown information

! Initialize the program

userID = MMStartUp
LoadLibrary 3
UtilStartUp(userID)
StartDesk(640)
if toolError <> 0 then
  print "Unable to start up desktop."
  end
end if

call InitMenus
lastEvent.taskMask = $1FFF
ShowCursor
InitCursor

! Main event loop
done = 0
while not done
   event = TaskMaster($076E, lastEvent)
   select case event
      case wInSpecial, wInMenuBar
         call HandleMenu(lastEvent.taskData, done)
  end select
wend

EndDesk
UtilShutDown
UnloadLibrary 3
end

!
! InitMenus
!
! Initialize the menu bar.
!
sub InitMenus
  menuID% = 1     : ! menu bar resource ID
  dim height as integer           : ! height of the largest menu
  dim m as menuBarHandle          : ! our menu bar's handle

  ! create the menu bar
  m = NewMenuBar2(2, 1, NIL)
  SetSysBar(ctlRecHndl(m))
  SetMenuBar(NIL)

  ! add desk accessories
  FixAppleMenu(1)

  ! draw the completed menu bar
  height = FixMenuBar
  DrawMenuBar
```

```
end sub

!
! HandleMenu
!
! Handle a menu selection.
!
! Parameters:
! taskData - taskData field for the menu event
! done - event loop done flag
!
sub HandleMenu (taskData as long, done as integer)
  select case LoWord(taskData)
    case 256 : ! File Menu, Quit
      done = 1
    case 257:  ! Apple Menu, About
      call DoAbout
    end select
  HiliteMenu(0, HiWord(taskData))
end sub


!
! DoAbout
!
! Display the about box.
!
sub DoAbout
  const alertID = 1
  dim b as Integer

  b = AlertWindow(awCString+awResource, NIL, alertID)
end sub
```

Listing 4-3A: The Frame.bas File

```
#include "types.rez"

#define appleMenu     1
#define fileMenu      2
#define editMenu      3
#define editUndo      250
#define editCut       251
#define editCopy      252
#define editPaste     253
#define editClear     254
#define fileClose     255
#define fileQuit      256
#define appleAbout    257

resource rMenuBar (1) {                 /* the menu bar */
   {
      appleMenu,                        /* resource numbers for the menus */
      fileMenu,
      editMenu
      };
   };

resource rMenu (appleMenu) {            /* the Apple menu */
   appleMenu,                           /* menu ID */
   refIsResource*menuTitleRefShift      /* flags */
      + refIsResource*itemRefShift
      + fAllowCache,
   appleMenu,                           /* menu title resource ID */
   {appleAbout};                        /* menu item resource IDs */
   };

resource rMenu (fileMenu) {             /* the File menu */
   fileMenu,                            /* menu ID */
   refIsResource*menuTitleRefShift      /* flags */
      + refIsResource*itemRefShift
      + fAllowCache,
   fileMenu,                            /* menu title resource ID */
   {fileClose,fileQuit};                /* menu item resource IDs */
   };

resource rMenu (editMenu) {             /* the Edit menu */
   editMenu,                            /* menu ID */
   refIsResource*menuTitleRefShift      /* flags */
      + refIsResource*itemRefShift
      + fAllowCache,
   editMenu,                            /* menu title resource ID */
   {                                    /* menu item resource IDs */
      editUndo,
      editCut,
      editCopy,
      editPaste,
      editClear
      };
   };

resource rMenuItem (editUndo) {         /* Undo menu item */
   editUndo,                            /* menu item ID */
   "Z","z",                             /* key equivalents */
```

```
    0,                                  /* check character */
    refIsResource*itemTitleRefShift     /* flags */
       + fDivider,
    editUndo                            /* menu item title resource ID */
    };

 resource rMenuItem (editCut) {         /* Cut menu item */
    editCut,                            /* menu item ID */
    "X","x",                            /* key equivalents */
    0,                                  /* check character */
    refIsResource*itemTitleRefShift,    /* flags */
    editCut                             /* menu item title resource ID */
    };

 resource rMenuItem (editCopy) {        /* Copy menu item */
    editCopy,                           /* menu item ID */
    "C","c",                            /* key equivalents */
    0,                                  /* check character */
    refIsResource*itemTitleRefShift,    /* flags */
    editCopy                            /* menu item title resource ID */
    };

 resource rMenuItem (editPaste) {      /* Paste menu item */
    editPaste,                          /* menu item ID */
    "V","v",                            /* key equivalents */
    0,                                  /* check character */
    refIsResource*itemTitleRefShift,    /* flags */
    editPaste                           /* menu item title resource ID */
    };

 resource rMenuItem (editClear) {       /* Clear menu item */
    editClear,                          /* menu item ID */
    "","",                              /* key equivalents */
    0,                                  /* check character */
    refIsResource*itemTitleRefShift,    /* flags */
    editClear                           /* menu item title resource ID */
    };

 resource rMenuItem (fileClose) {       /* Close menu item */
    fileClose,                          /* menu item ID */
    "W","w",                            /* key equivalents */
    0,                                  /* check character */
    refIsResource*itemTitleRefShift     /* flags */
       + fDivider,
    fileClose                           /* menu item title resource ID */
    };

 resource rMenuItem (fileQuit) {        /* Quit menu item */
    fileQuit,                           /* menu item ID */
    "Q","q",                            /* key equivalents */
    0,                                  /* check character */
    refIsResource*itemTitleRefShift,    /* flags */
    fileQuit                            /* menu item title resource ID */
    };

 resource rMenuItem (appleAbout) {      /* About menu item */
    appleAbout,                         /* menu item ID */
    "","",                              /* key equivalents */
```

```
   0,                                   /* check character */
   refIsResource*itemTitleRefShift      /* flags */
      + fDivider,
   appleAbout                           /* menu item title resource ID */
   };

                                        /* the various strings */
resource rPString (appleMenu, noCrossBank)  {"@"};
resource rPString (fileMenu, noCrossBank)   {"  File  "};
resource rPString (editMenu, noCrossBank)   {"  Edit  "};
resource rPString (editUndo, noCrossBank)   {"Undo"};
resource rPString (editCut, noCrossBank)    {"Cut"};
resource rPString (editCopy, noCrossBank)   {"Copy"};
resource rPString (editPaste, noCrossBank)  {"Paste"};
resource rPString (editClear, noCrossBank)  {"Clear"};
resource rPString (fileClose, noCrossBank)  {"Close"};
resource rPString (fileQuit, noCrossBank)   {"Quit"};
resource rPString (appleAbout, noCrossBank) {"About Frame..."};

/*- About Box ------------------------------------------------*/

resource rAlertString (1) {
   "43/"
   "Frame 1.0 by Mike Westerfield\n"
   "\n"
   "Written in GSoft BASIC; includes the GSoft BASIC "
   "runtime interpreter, copyright 1998, Byte Works Inc."
   "/^#0\$00";
   };
```

Listing 4-3B: The Frame.Rez File

Alert Strings

Before leaving the topic of alerts, let's take a closer look at the alert string that we used in the resource fork. This alert string controls a lot of different things about the alert, from its size to the icon that is drawn. The string itself is divided into three main sections, controlling the alert's visual appearance, the string that is printed, and the buttons. Just like menu strings, the string is divided up to perform these functions based on some rigid formatting rules.

The description of the alert string is divided up into a table form to make it easier to see what field you are reading about, but the descriptions are still real text descriptions, so go ahead and read them like you would the normal text in the book.

size — This field is generally a single character controlling the size of the alert window. The character is a numeric digit, from 0 to 9. All but the first of these digits corresponds to a specific size window, but the 0 character is the first byte of a nine byte field. The other eight bytes define the size of the window by listing the edges as two-byte integers, in this format:

- v1 — Y coordinate (0 is at the top of the screen) of the top edge of the window.

h1 X coordinate of the left edge of the window.
v2 Y coordinate of the bottom edge of the window.
h2 X coordinate of the right edge of the window.

The predefined sizes from 1 to 9 vary in both width and height, giving room for more characters as the number goes up. An interesting and very useful feature of this size is that it is larger for 640 mode programs than it is for 320 mode programs, giving you about the right size box to display a given number of characters regardless of which screen you pick. That helps a lot when you are trying to define a package of alerts to move from one program to another, or when you are creating a program that can switch between 320 mode and 640 mode while it's running.

There are two ways to get a handle on what the various sizes are. One is to look at the table of sizes shown in Appendix A, and you're free to do that. A better way to get a real grasp on the issue is to try the various sizes; that's what you will be doing in Problem 4-2.

iconSpec The picture in the upper-left corner of the dialog is called an icon; it's something that you can change from alert to alert. You specify the icon in the second field of the alert string. Like the size, the icon field is generally a single character, although there is one exception for a custom icon. The alert sampler from Problem 4-2 will show you the various icons; here's a list of what they are:

Character	Use
0	No icon at all.
1	Custom icon. With a custom icon, you have to tell the window manager what icon you want, so you have to follow the single icon character with some other information. Specifically:

size	use
4 bytes	Pointer to the icon image. The image is a series of pixels.
2 bytes	Width of the image in bytes. Another way to think of this is as the length of a line of pixels.
2 bytes	Height of the image. This is the number of lines of pixels in the icon.

2	Stop icon. The stop icon is used when an error occurs. For example, if the program is trying to save a file and there isn't enough room on disk, the alert would use the stop icon.
3	Note icon. This is the icon we used for our about box; it is used in alerts that present information to the user.
4	Caution icon. This is used for alerts that are displaying some sort of warning, and they usually give you a choice of going on

or canceling the operation. An example would be a message warning the user that they are about to format a hard disk.

5 Disk icon. This icon shows a picture of a disk. It's generally used by programs that manipulate entire disks.

6 Disk swap icon. Use this icon when the program needs to read a file, but the disk containing the file is not in a drive. The message, of course, should tell the user what disk is needed.

separator The rest of the fields in the alert string are variable length strings, so you need some way of telling the Window Manager where one string ends and another begins. Since you may need to use just about any character in your strings, the Window Manager lets you tell it what character to use to separate the strings. The / character we used earlier is a traditional example.

messageText This is the string that's actually printed in the alert. You can use return characters (`'\r'`) to break a line, forcing text to a new line. You can't have more than 1000 characters in the line.

separator Put another separator character here to mark the end of the message. Be sure it is the same one you used for the first separator field!

buttonStrings You can put up to three buttons in an alert, separating each of the strings that will appear in the buttons with separator characters. The Window Manager will create buttons that are all the same size, lined up and centered at the bottom of the dialog. The total length of the text for the buttons must be 80 characters or less.

There are several special characters you can use to create the buttons. These are covered in the next table.

terminator The end of the alert string is marked with a null character, which just happens to be what GSoft BASIC puts at the end of a string anyway. In a resource description file, you can put this character in with $"00", as we did in our about box resource.

Besides the obvious points about what makes up an alert string, there are also some interesting things you can learn from this example about the Rez compiler itself. The entire `rAlertString` resource is really nothing more than a single, big string. Looking at it, though, you can see that we spread that string out over several lines. That's the first neat trick: you can create a single long string in a resource description file by writing two shorter strings. You can put spaces between the strings, put them on different lines, or even put comments in between. As long as you don't put in a comma, though, Rez combines all of the short strings to create a single, long string.

Another thing you'll see imbedded in the string is `\n`, and, at the end of the string, `\$00`. These are escape sequences; they indicate special characters ("\n" is the newline character, and \$00 is the null character indicating the end of a string).

Problem 4-2: Write a program that can display any combination of an alert size and icon. Your program should have a menu labeled Size with the nine fixed sizes, showing a check mark beside the currently selected size. Another menu, labeled Icon, should have an entry for each of the predefined icons, again using check marks to show the active icon. Add an Open command under the File menu, and use it to actually draw the dialog, displaying a dialog with the size and icon selected in the menus.

This is the only place in the course where you won't use a resource for the alert string, since it would be a little crazy to create one alert for each possible combination of size and icon. Instead, set up a string and call `AlertWindow` using this subroutine:

```
!
! SampleAlert
!
! Draw an alert window.
!
! Parameters:
!  size - size of the alert (1-9)
!  icon - alert icon number (2-6)
!
sub SampleAlert(size as Integer, icon as Integer)
  dim alertString as String
  dim sizeString as String
  dim iconString as String
  dim b as Integer

  sizeString = chr$($30+size)
  iconString = chr$($30+icon)

  alertString = sizeString + iconString
  alertString = alertString + "/Sample alert of size " + sizeString
  alertString = alertString + ", with icon " + iconString + "./^#0"

  b = AlertWindow(awCString+awPointer, NIL, @alertString)
end sub
```

Listing 4-4: Creating an `AlertWindow` from a Pointer

Substitution Strings

One of the more powerful features of the alert string is the ability to use substitution strings. Substitution strings are a special character sequence the Window Manager replaces with some other string before it actually draws the text in the alert. Using substitution strings, you can create a single alert, and then use it to display a whole wealth of information. A great example is an error handler that uses a single alert with a substitution string to show any of the errors you need to display while a program runs.

You can use a substitution string in either the text of the message or for the name of a button. They come in two flavors, and in fact, you've already used one of them without knowing it. The first kind of substitution string, and the one you've already seen, is generally used for common button names. These substitution strings consist of two characters, a # and a numeric digit.

Table 4-1 shows the string you type, along with the string the Window Manager substitutes when the alert is actually drawn.

Substitution String	String Drawn
`#0`	`OK`
`#1`	`Cancel`
`#2`	`Yes`
`#3`	`No`
`#4`	`Try Again`
`#5`	`Quit`
`#6`	`Continue`

Table 4-1: Predefined Substitution Strings

If you look back at the alert we used for the about box, you'll find `#0` used for the name of the button. From this table, you can see why the alert actually showed up with a button named OK.

There are a lot of hidden advantages to using these predefined button names, and they go way beyond saving you some typing. If your program is used in, say, France, it's pretty easy for Apple France to change the names of all of the buttons at the operating system level; they don't even need to change your resource fork to get French button names. A more subtle issue is the fact that the human interface Apple uses is an evolving concept, and it changes from time to time. It's not uncommon to find button names of Okay in older Macintosh programs. Using these predefined names has the dual affect of making your program more standard, which makes it easier to use; and making your program easier to update if Apple decides that the button really should have been called Okay after all.

The other flavor of substitution string is a little harder to use, but it's here that the real power of substitution strings comes into play. You can define up to ten substitution strings of your own. Each of them starts with an asterisk (*) and is followed by a numeric digit. One of the parameters you pass to `AlertWindow` is an array of string pointers; this array is used for the various values of the strings.

To see how this is used, let's create an error dialog. Our error dialog will be used from anywhere in our program. We'll just pass an error number, and expect the error dialog to do the rest.

```
!
! Error
!
! Parameters:
!    number - error number to show
!
sub Error(number as Integer)
  const fortuneAlert = 21
  const base = 2000
  dim substString as cStringPtr
  dim b as Integer

  substString = GetString(base + number)
  if substString <> NIL then
    b = AlertWindow(awCString+awResource, ptr(@substString), fortuneAlert)
    call FreeString(base + number)
  end if
end sub
```

Listing 4-5A: Error Subroutine

```
resource rAlertString (21) {
  "42/"
  "*0"
  "/^#0"
  $"00"
  };
```

Listing 4-5B: `AlertWindow` Resource

In this case we only need a substitution array with one element, since `*0` is the only substitution string we're using. For a single string, we can just pass the address of the address of the substitution string, which is the same thing as passing the address of an array of addresses when the array only has one element. The loaded string resource id pointed to by a variable of type cStringPtr, which represents a pointer to a C-string.

The only real concern with this mechanism is where all of the substitution strings are stored. One of the advantages to using resources for the `AlertWindow` alert string is to put the text messages in the resource fork so they can be changed by the end user. Well, we still put the alert strings in the resource fork, but we store them as `rCString` resources and load them with some simple Resource Manager calls. In good structured programming style, this mechanism is packaged in a couple of subroutines, `GetString` and `FreeString`. Here they are, along with a few sample error strings:

```
!
! GetString
!
! Get a string from the resource fork
!
! Parameters:
!    resourceID - resource ID of the rPString resource
!
! Returns: pointer to the string; NIL for an error
!
! Notes: The string is in a locked resource handle.  The caller
!    should call FreeString when the string is no longer needed.
!    Failure to do so is not catastrophic; the memory will be
!    deallocated when the program is shut down.
!
function GetString(resourceID as Integer) as cStringPtr
  dim hndl as Handle

  GetString = NIL
  hndl = LoadResource(rCString, resourceID)
  if toolerror = 0 then
    HLock(hndl)
    GetString = cStringPtr(hndl^)
  end if
end function

!
! FreeString
!
! Release a string resource.
!
! Parameters:
!    resourceID - resource ID of the rCString to free
!
sub FreeString(resourceID as Integer)
  ReleaseResource(-3, rCString, resourceID)
end sub
```

Listing 4-6A: `GetString` and `FreeString`

```
resource rCString (2001) {"Out of memory"};
resource rCString (2002) {"Could not form the full path name"};
resource rCString (2003) {"A document using this file name is already"
                          " open"};
resource rCString (2004) {"All of the objects are locked"};
resource rCString (2005) {"You must select a printer from the control panel"
                          " before setting up the page"};
```

Listing 4-6B: Sample Error String Resources

These subroutines use a couple of Resource Manager calls. `LoadResource` loads a resource into memory based on the type of the resource and the resource ID; the subroutine then locks the handle so the string won't move and returns a pointer to the string. `FreeString` just calls `ReleaseResource` to free up the memory used by the resource. There isn't much to using these

Resource Manager calls, but if you want to explore some of the options available, check out their descriptions in Appendix A.

I also snuck a new resource type into the example. Most of the toolbox is designed to work with p-strings, GSoft BASIC works best with null terminated strings (they're easily converted into GSoft's native format, which is also null-terminated, but with extra information). To make it easier on us, I've used the `rCString` resource for these strings. It looks and works just like the `rPString` resource we've been using for menus, but the `rCString` resource gives us a null terminated string instead of a string with a length byte up front.

Problem 4-3: Create a fortune cookie program. Each time the user picks Open from the File menu, randomly select a string number and display a note alert with the string. You can make up your own fortunes, or use this group of famous quotes:

Be cheerful while you are alive.
–Ptahhotpe, 24th century B.C.

I know nothing except the fact of my ignorance.
–Socrates, c.469-399 B.C.

Nothing endures but change.
–Heraclitus, c.540-480 B.C.

I have made this letter longer than usual, because I lack the time to make it short.
–Blaise Pascal, 1656 or 1657

There are three faithful friends – and old wife, and old dog, and ready money.
–Benjamin Franklin, 1738

Common sense is not so common.
–Voltaire, 1764

It is never too late to give up our prejudices.
–Henry David Thoreau, 1854

Always do right. This will gratify some people, and astonish the rest.
–Mark Twain, 1901

Everything is funny as long as it is happening to someone else.
–Will Rogers, 1924

The whole of science is nothing more than a refinement of everyday thinking.
–Albert Einstein, 1936

Summary

In this lesson we've learned to use alerts to create about boxes and display messages. In the process, we have developed a program called Frame which will be the basis for most of the programs in the rest of the course.

Tool calls used for the first time in this lesson:

```
AlertWindow       LoadResource      ReleaseResource
```

Resource types used for the first time in this lesson:

```
rAlertString      rCString
```

Lesson 5 – Why, Yes. We Do Windows!

Goals for This Lesson

In this lesson we will learn to create, draw, and manipulate windows.

Defining Our Terms

You see windows all of the time in desktop programs, so it may seem a little odd to start off by looking at what they are, but that's just what we'll do. Visually, a window is a thing on the desktop where information is presented. A standard document window, like the one in Figure 5-1, can be moved, can overlap other windows, and can even be shoved partway off of the screen. In fact, while you can't *drag* a window off of the screen (the mouse won't move that far), your program can actually *position* a window completely off the visible screen.

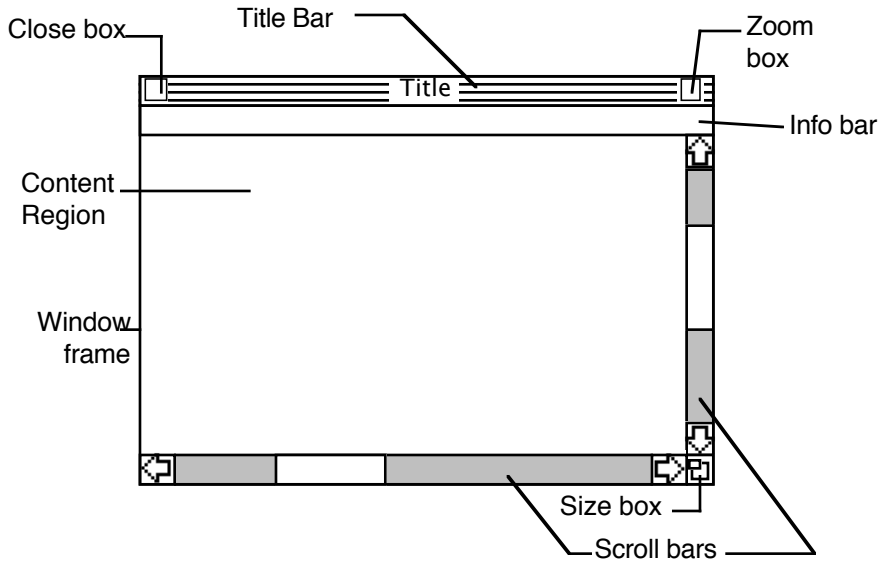

Figure 5-1: Typical Document Window

title bar The title bar is the top part of the window. It includes the name of the window and all of the box surrounding the name except for the close box and zoom box. You can move a window by dragging the title bar.

close box The close box is the small box at the left hand end of the title bar. Clicking in the close box does the same thing as selecting Close from the file menu.

zoom box The zoom box is the box at the right hand size of the title bar. Clicking on the zoom box resizes and moves the window. If the window is not in its zoomed state when you click on the zoom box, it is resized and repositioned to the zoom state.

	If the window is in the zoomed state when you click on the zoom box, the window is resized and repositioned to the last unzoomed state. In most cases, the zoomed state is when the window fills all of the available screen space.
info bar	The info bar is a part of the window right below the title bar; it is missing on most windows. The info bar is used for extra information about the window. One typical use is for rulers; you can see an info bar in the PRIZM desktop development system by showing the ruler.
content region	The content region is technically defined as all of the window that isn't something else. It's the part you draw in.
scroll bars	You can put scroll bars anywhere, but you can also have the Window Manager create the two normal scroll bars for you. These scroll bars appear to the bottom and right of the content region, and are used to scroll the content region to different parts of a document. (Technically, scroll bars are controls that are inside the content region.)
size box	The size box is in the corner formed by the two scroll bars. Dragging on the size box will change the size of the window.
window frame	The window frame is the line (or area) that surrounds the rest of the window.

Table 5-1: Parts of a Window

There are a lot of variations on this basic theme. You can leave off almost any of the parts you see in the standard window. You must have a title bar to have either a close box or a zoom box, but with that exception, you can create windows with any combination of these parts. The only things you can't leave off are the window frame and the content region – but they can be so small they don't matter, much. There are also several options available to you. Windows can have several forms of title bar and they can have a special frame called an alert frame, just to name a couple.

Opening a Window

Windows are created with the Window Manager's `NewWindow2` call. Here's the code you need to open a standard document window on the desktop:

```
  !
  ! NewDocument
  !
  ! Open a new window, returning the pointer.
  !
  ! Returns: Windows window pointer; NIL for an error.
  !
  function NewDocument as GrafPortPtr
    const wrNum = 1001              : ! Window template resource ID
    dim title as String
    dim l as Integer

    title = "MyWindow"
    title = chr$(len(title)) + title
    NewDocument = NewWindow2(pStringPtr(@title), 0, NIL, NIL, refIsResource,
wrNum, rWindParam1)
  end function
```

Listing 5-1A: GSoft BASIC Code to Open a New Window

Note The code "title = chr$(len(title) + title" is used to construct in the string variable title a p-string containing the window's title; `NewWindow2` requires a p-string for the window title.

```
  resource rWindParam1 (1001) {
    $DDA5,                        /* wFrameBits */
    nil,                          /* wTitle */
    0,                            /* wRefCon */
    {0,0,0,0},                    /* ZoomRect */
    nil,                          /* wColor ID */
    {0,0},                        /* Origin */
    {1,1},                        /* data size */
    {0,0},                        /* max height-width */
    {8,8},                        /* scroll ver hors */
    {0,0},                        /* page ver horiz */
    0,                            /* winfoRefcon */
    10,                           /* wInfoHeight */
    {30,10,183,602},              /* wposition */
    infront,                      /* wPlane */
    nil,                          /* wStorage */
    $0000                         /* wInVerb */
    };
```

Listing 5-1B: Resource for a New Window

Most of the things you need to learn about windows involve changing the resource and the parameters to the `NewWindow2` call that actually creates the window. Almost everything else about a window can be handled automatically for you by `TaskMaster`. All you really have to do is keep track of the positions of the scroll bars and draw the contents of the window, since moving windows, overlapping windows, resizing windows, and even scrolling windows is handled by `TaskMaster`.

Obviously there's a lot I haven't told you about windows yet. Your knowledge of windows will grow gradually through the lesson, and by the end, you'll understand all of the parameters to `NewWindow2` and all of the entries in the window resource. Along the way, though, I'm going to

give you a series of problems, and in most cases they build on the previous problems. You will learn about windows as you develop a program than handles them. The point is to be patient: you may not know enough about windows to understand all of the parameters, yet, but you can still type in what you see so you have a program to explore ideas with as you work through the lesson.

Problem 5-1: Add the `NewDocument` function to the Frame program, calling it whenever New or Open are selected from the File menu.

For now, just save the window pointer `NewDocument` returns in a global variable, and don't worry about the fact that you're losing track of the windows if you open more than one. As it turns out, this doesn't do any harm, since the Window Manager will dispose of all of the memory when it shuts down. We'll start handling this better later in the lesson.

You'll have to add New and Open as menu commands, of course. You File menu should look like this:

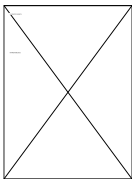

Figure 5-2: File Menu for Problem 5-1

Be sure and put in the key equivalents shown.

When you run the program, be sure you try opening more than one window. You will be able to overlap them, resize them, and drag them around the screen. The zoom box will work fine, but the close box doesn't work, yet.

Closing a Window

Closing a window is really very easy. If `wPtr` is the window pointer for the window you want to close, you close the window like this:

```
CloseWindow(wPtr)
```

The interesting part of closing a window isn't really closing it, it's finding `wPtr` for the window to close.

There are two ways to close a window, and you use a different mechanism for closing the window with each.

The first way to close a window is to select Close from the File menu. When the user picks Close from the File menu, you are supposed to close the front most window on the desktop. That, of course, means that you need a convenient way to find out what window is in front. The Window Manager's `FrontWindow` call does the trick, returning a pointer to the front window if there are any windows open, and `NULL` of there aren't any windows. Here's how you would close a window if Close is picked from the File menu:

```
wPtr = FrontWindow
if wPtr <> NIL then
   CloseWindow(wPtr)
end if
```

Checking to see if `wPtr` is `NIL` before closing the window is critical! If you don't check, and the user picks the Close command, your program could crash.

The second mechanism for closing a window is clicking on the close box. When the user clicks on the close box, `TaskMaster` returns a `wInGoAway` event.

In most programs, you can only click on the close box for the front window, so it might seem like the same code we just used would work just fine. In practice, that just won't do. Even if your program only allows the user to click on the close box for the front window, you have to keep in mind that the user might be using a desk accessory that doesn't enforce that restriction. So, instead of just closing the front window, we'll use the window pointer returned by `TaskMaster`. When `TaskMaster` returns `wInGoAway` for the event, it stuffs a pointer to the window to close in `taskData` field of the event record. We can use this window pointer to close the window, like this:

```
wPtr = GrafPortPtr(myEvent.taskData)
if wPtr <> NIL then
   CloseWindow(wPtr)
end if
```

Technically, we don't need to check to see if `wPtr` is `NIL` this time, since `TaskMaster` wouldn't return `wInGoAway` unless it also passed back a valid window pointer. In the next section, though, we're going to start doing a lot more work when we close a window, so we're going to encapsulate the code to close the window in a subroutine right away.

Problem 5-2: Add the ability to close windows to Frame. You will need to add the check for a `wInGoAway` event to the event loop, and you'll have to add code that will be called when Close is picked from the File menu. In both cases, you should call a function named `CloseDocument`, passing a pointer to the window to close. `CloseDocument` should check to be sure the window pointer is not `NIL` before closing the window.

Multiple Windows

The Frame program can already handle multiple windows, but as we start to add more things to the program, we'll need to keep track of our windows a bit better. In this section we'll look at two ways to handle more than one window effectively, and implement one of them in Frame. In the process, we'll add distinct names to our windows, so they don't all have the name "MyWindow".

Most programs need to keep track of a wide variety of information about a window. One of the best ways to organize all of this information is with a record, which we will call a document record. In fact, the reason our open and close subroutines were called `NewDocument` and `CloseDocument` was because I was going to eventually suggest calling our own records document records, and all of the information about one window a document.

For now, our document record only needs two pieces of information, a window pointer `wPtr` and a window name `wName`. As you already know, `wPtr` is a `GrafPortPtr`. The `wName` field should be a string that can contain the window's name.

How you keep track of these document records is pretty important. One very common way is to create an array of document records, and flag them as either used or unused. When you open a window, you can scan the array, looking for an entry that hasn't been used. One easy way to keep track of which entries are used and which are not is to set `wPtr` to `NIL` in an unused record. With this organization, you would have definitions something like these at the top of your program:

```
const numDocs = 4                  : ! max number of open windows

type documentStruct
  wPtr as GrafPortPtr              : ! window pointer
  wName as String                  : ! window name
end type

dim documents(numDocs) as documentStruct
```

Listing 5-2: Array Based Documents

This works, but it builds an obvious limitation into your program: there will be some maximum number of documents that the user can have open at any one time. You can pick a big number, but if it's too big, you'll waste a lot of space. (Our document records already use eight bytes of memory per document (not including the `wName` string, which is located in the GSoft BASIC string pool), and they will get bigger as we enhance our program to do more.) Also, no matter what number you pick, someone someday is going to have a very good reason for trying to open more. For that reason, I prefer a slightly more complicated but infinitely more flexible way of dealing with document records. Instead of a fixed length array, use a linked list, allocating document records as they are needed. This doesn't waste memory, and it allows the user to open as many documents as they have memory for. Using linked lists, the definitions look like this:

```
type documentStruct                    : ! information about our document
  after as pointer to documentStruct   : ! next document
  wPtr as GrafPortPtr                  : ! window pointer
  wName as String                      : ! window name
end type
type documentPtr as pointer to documentStruct  : ! document pointer

dim documents as documentPtr           : ! our documents
```

```
!
! InitGlobals
!
! Initialize the global variables
!
sub InitGlobals
  shared documents

  documents = NIL
end sub
```

Listing 5-3: List Based Documents

The only thing remotely complicated about using a linked list for our document records is adding, removing, and finding the correct document record. Here's a modified form of `NewDocument` and `CloseDocument` that handles these details, along with appropriate error checking. I've also included a subroutine called `FindDocument` which returns the document pointer for a given window pointer. These subroutines will be the basis for manipulating windows for the rest of this course.

```
!
! CloseDocument
!
! Close a document and its associated window
!
! Parameters:
! dPtr - pointer to the document to close; may be NIL
!
sub CloseDocument(dPtr as documentPtr)
  shared documents

  dim lPtr as documentPtr          : ! pointer to the previous doc.

  if dPtr <> NIL then
    CloseWindow(dPtr^.wPtr)        : ! close the window
    if documents = dPtr then
      documents = dPtr^.after      : ! ...dPtr is the first document
    else
      lPtr = documents             : ! ...dPtr isn't the first document
      while lPtr^.after <> dPtr
        lPtr = lPtr^.after
      wend
      lPtr^.after = dPtr^.after
    end if
    dispose(dPtr)                  : ! dispose of the document record
  end if
end sub

!
! FindDocument
!
! Find the document for wPtr
!
! Parameters:
! wPtr - pointer to the window for which to find a document
!
! Returns: document pointer; NIL if there isn't one
!
function FindDocument(wPtr as GrafPortPtr) as documentPtr
  shared documents

  dim done as Boolean              : ! used to test for loop termination
  dim dPtr as documentPtr          : ! used to trace the document list

  dPtr = documents
  if dPtr = NIL then
    done = true
  else
    done = false
  end if

  while not done
    if dPtr^.wPtr = wPtr then
      done = true
    else
      dPtr = dPtr^.after
      if dPtr = NIL then
        done = true
```

```
      else
        done = false
      end if
    end if
  wend

  FindDocument = dPtr
end function

!
! NewDocument
!
! Open a new window, returning the pointer
!
! Parameters:
! wName - name for the new window
!
! Returns: document pointer; NIL for an error
!
function NewDocument(wName as String) as documentPtr
  shared documents

  const wrNum = 1001              : ! window resource number
  dim dPtr as documentPtr         : ! new document pointer
  dim s as String

  allocate(dPtr)                  : ! allocate the record
  if dPtr <> NIL then
    dPtr^.onDisk = false          : ! not on disk
    dPtr^.wName = wName           : ! the name
    s = chr$(len(wName)) + wName
    dPtr^.wPtr = NewWindow2(pStringPtr(@s), 0, NIL, NIL,
                refIsResource, wrNum, rWindParam1)
    if dPtr^.wPtr = NIL then
      call FlagError(1, toolerror): ! handle a window error
      dispose(dPtr)
      dPtr = NIL
    else
      dPtr^.after = documents     : ! put the document in the list
      documents = dPtr
      call X(dPtr^.wPtr)
    end if
  else
    call FlagError(2, 0)          : ! handle an out of memory error
  end if

  NewDocument = dPtr
end function
```

Listing 5-4: Basic Document Manipulation Subroutines

This leaves two issues to deal with before we can add this new document handling mechanism to Frame. The first is how errors are flagged by `FlagError`, and the other is how window names are chosen and passed. We'll leave the topic of window names for the next section. Flagging errors is relatively simple, though: `FlagError` uses `GetString` and `FreeString` from the last lesson, appending a note about the specific tool error if a non-zero

error number is passed. The numbers refer to specific, numbered error messages. `NewDocument` used error numbers 1 and 2, which might have messages like this:

```
1  Could not open a new document
2  Out of memory
```

Installing an error handler to display these messages is part of Problem 5-3.

Window Names

There's more to picking the name for a window that you might think, at first. In most programs, each document window corresponds to a specific file. For example, in a word processor, the text from a file is displayed in a window, and the name of the window is the same as the name of a file. When a document is new, and hasn't been saved to disk, it doesn't have a file name. In that case, it's customary to name the window `Untitled x`, where `x` is a number that is incremented for each window.

The Window Manager also puts one fairly severe restriction on how we create and manipulate window names: It's up to us to allocate space for the window name, to make sure the window name stays in one place, and to make sure it doesn't get changed without making an appropriate Window Manager call to change the name, so the window gets redrawn with the new window name. All of these restrictions are handled nicely by keeping the window name in its own string buffer in our document record.

Setting the window name based on a disk file name is something we'll leave for the next lesson, when we learn how to use the Standard File Operations Tool Set to figure out what file to open or what name to use for a new file. Creating a window name for an untitled window is something we can do right away, though. The subroutine in listing 5-5 shows one way to do it; this subroutine fills in a string buffer you pass as a parameter with an appropriate untitled window name. It uses a global variable, `untitledNum`, to keep track of the number for the untitled window, resetting it to 1 if there are no untitled windows in the current list of documents.

If you look close, you'll also see that this subroutine assumes there is a new field in the document record. This field, called `onDisk`, indicates whether the file exists on disk – in other words, whether there is a valid file name for the document. We'll need this field later on, when we actually start loading and saving disk files. This field should be set to `FALSE` in `NewDocument`.

```
!
! GetUntitledName
!
! Create a name for an untitled window
!
! Returns: the new window name
!
function GetUntitledName as String
    shared documents
    shared untitledNum
    const untitled = 101
    dim wname as String                 : ! Window name
    dim sPtr as cStringPtr              : ! Pointer to resource string
    dim dPtr as documentPtr             : ! Used to trace the document list
    dim number as Integer               : ! New value for untitledNum

    dPtr = documents                    : ! If there are no untitled
    number = 1                          : ! documents, reset untitledNum

    while dPtr <> NIL
          if not dPtr^.onDisk then
                number = untitledNum
                dPtr = NIL
          else
                dPtr = dPtr^.next
          end if
    wend
    untitledNum = number
    sPtr = GetString((untitled))        : ! Get the base name
    if sPtr = NIL then
          wname = " Untitled "
    else
          wname = CToGSoftStr(sPtr)
          call FreeString((untitled))
    end if

    ! Add the untitled number

    wname = wname + str$(untitledNum)
    untitledNum = untitledNum + 1
    GetUntitledName = wname
 end function
```

Listing 5-5: `GetUntitledName` Subroutine

Problem 5-3: In this problem you will be adding support for multiple windows to Frame. This involves several steps:

1. Add `NewDocument`, `CloseDocument` and `FindDocument` from the last section. Adjust the two places in your program where a window is closed, calling `FindDocument` with the window pointer and use the result to call `CloseDocument`.
2. Add a document record, a document list variable, and the `InitGlobals` subroutine. `InitGlobals` should set the document list variable to `NULL`, indicating that there are no open documents. Be sure to include the `onDisk` field, initializing it to `FALSE` in `NewDocument`.

3. Add `GetUntitledName`, calling this subroutine before calling `NewDocument` to get a distinct window name for each new window. It's important to establish the name before calling `NewDocument`, since we will eventually call `NewDocument` to set up a new document record when we open a disk file, too.
4. Create an error handler based on `GetString` and `FreeString`. Use the description for the error handler at the end of the last lesson to write yours.

Drawing in a Window

Back in Lesson 4 you used the QuickDraw drawing commands `MoveTo` and `LineTo` to draw some lines on the screen. At that point, your program didn't have any windows, so you didn't have to worry about exactly where you were drawing; your program just wrote to the screen, figuring that it owned the whole thing. Well, at that time, it did – but no longer!

The Frame program you've created can open as many windows as you have the patience and memory to handle. Each of these windows is, among other things, a separate drawing area called a `GrafPort`. Any time you use QuickDraw to draw to "the screen" you are really drawing to a `GrafPort`, which may or may not be the whole screen. The `GrafPort` you are drawing to could also be a window, and the window could be part on the screen or part off of the screen. The window might even be covered up, part way or all the way, by another window. And, last but not least, the `GrafPort` might not even be on the screen at all. Later, you'll draw to a `GrafPort` to print, instead of to display something on the screen.

Regardless of which of these myriad of conditions exist, QuickDraw II is smart enough to keep your drawing where it belongs. You can, and generally do, just move your pen and draw wherever you want. QuickDraw II figures out if the stuff you draw is on a visible part of the screen or not, and draws the actual dots on the screen if it needs to, ignoring you if what you are drawing is not visible. This is something you'll get a chance to see for yourself in a little while.

About the only thing you really do have to do is tell QuickDraw II which `GrafPort` to use. The normal sequence of events to draw to any particular window is:

1. Get and save the "current" `GrafPort`, so you can restore it later. That way, if the subroutine that called the one you are writing assumes a specific `GrafPort` is active, it will still be active when you return. QuickDraw's `GetPort` call returns the current `GrafPort`.
2. Set the `GrafPort` to your own window using `SetPort`.
3. Unless you're sure about the current state of QuickDraw II's drawing pen, make a quick call to `PenNormal` to set things up for "normal" drawing. `PenNormal` gives you a pen that is one pixel wide and one pixel tall, sets the drawing mode to `modeCopy`, sets the pen color to black, and gives you a solid pen mask. Some of that may not make any sense; we'll get to it later. The result, though, is a setting that tells QuickDraw II to draw normal looking black lines.
4. Draw whatever you want to draw.
5. Use `SetPort` to reset the `GrafPort` to whatever it was before you started.

It's actually faster to do all of this than to describe it. Here's a subroutine that draws an X across a window. It uses `GetPortRect` to find out how big the window is. `GetPortRect`

assumes the whole window is visible, and returns its full size, even if part of the window is covered by another window or is off of the screen.

(You saw rectangles briefly in Lesson 4; a rectangle is just a structure with a top, bottom, left and right position. We'll look at them in detail in a later lesson.)

```
!
! X
!
! Parameters:
!    wPtr - GrafPort to draw in
!
sub X(wPtr as GrafPortPtr)
  dim oldPort as GrafPortPtr            : ! Previous GrafPort
  dim r as Rect                    : ! Port rectangle

  oldPort = GetPort                          : ! Get the old GrafPort
  SetPort(wPtr)                    : ! Make our port active
  PenNormal                        : ! Set default drawing pen
  GetPortRect(r)                        : ! Get the window's size
  MoveTo(r.h1, r.v1)                    : ! Draw the X
  LineTo(r.h2, r.v2)
  MoveTo(r.h1, r.v2)
  LineTo(r.h2, r.v1)
  SetPort(oldPort)                           : ! Reset the original port
end sub
```

Listing 5-6: Subroutine to Draw in a Window

Problem 5-4: Add the subroutine X to your Frame program, calling it right after the window is created.

Unlike the other changes we've been making, the function X won't become a permanent addition to Frame, so be sure you keep a separate copy of Frame that does not have this subroutine.

Try creating other windows, dragging your window off of the screen and back on, and showing and then hiding the about box with your window positioned so the about box covers part of the X, but not all of it. These experiments point out a problem that we'll deal with in the next section: your X disappears.

Updating a Window

If you tried Problem 5-4, you found out that drawing to a window isn't exactly permanent – far from it, in fact! If you drag a window partway off of the screen, then drag it back, the part of the X that was off of the screen is not redrawn. If you cover the window with another window (or dialog), the part that is covered doesn't get redrawn. In fact, about the only thing that redraws the window after covering it up is a menu that gets pulled down over part of the window.

To understand what is happening, and what we have to do to keep what we draw in the window, we need to delve into the way the toolbox handles windows. There are two ways to handle the situation when a window is covered and then uncovered, so that part of the window

needs to be redrawn. One way is for the toolbox to remember what was covered up, and redraw it when the time comes. That would work, and in fact some windowing systems do just that. The other way is to call some subroutine to redraw the contents of the window, which is what the Apple IIGS toolbox tries to do.

Whenever some part of a window becomes visible the toolbox posts an update event. Part of a window could become visible because you close a window that covers another one, move one window to uncover part of another window, or because you drag a window that was part way off of the screen back onto the visible screen. The toolbox itself also posts an update event when the window is created in the first place. In each of these situations, the toolbox adds the area of the window that needs to be drawn to something called the update region. We'll explore regions in more detail later; for now you can think of the update region as all of the parts of the window that need to be redrawn, or updated – and that's exactly what it is. An update event is then created, and your event loop will eventually find that update event and return it to you.

Once you find an update event in your event loop, you should do the following:

1. Call `BeginUpdate`, passing the window pointer for the window that needs to be updated. `BeginUpdate` does some internal magic to make sure you only redraw the part of the window that needs to be redrawn.
2. Draw whatever needs to be drawn. In most programs you would redraw everything in the window, letting QuickDraw II decide what needs to be drawn and what can be ignored. If it takes a long time to redraw the window, though, there are ways to check to see exactly what must be redrawn.
3. Call `EndUpdate`, again passing the window pointer. It is very important to make sure that you call `EndUpdate` exactly one time for every `BeginUpdate` call.

You draw in the window the same way as before. The only difference is that you don't need to save the caller's `GrafPort` or set your own; `BeginUpdate` and `EndUpdate` handle all of that for you. In fact, you *can't* call `SetPort` between calls to `BeginUpdate` and `EndUpdate` – if you do, strange and not-so-wonderful things will happen. You also can't change the origin while you're updating a window – but you don't know how to do that yet, anyway.

It isn't really quite as hard to do all of this as it sounds, but it is a bit of a pain for something that gets done all of the time, in every desktop program. With a very small change in the call to `NewWindow2`, `TaskMaster` can do all of the work for us. The only change is to pass the name of a subroutine to call when an update is needed. There are no inputs or outputs to the subroutine that `TaskMaster` calls. You will probably use the same subroutine for all of your windows, so you need some way to figure out which window you are drawing; `GetPort` does a nice job, returning the window pointer for the window to draw.

There's a problem, though. When the toolbox tries to call a function, it assumes that the function is compiled Apple IIGS code. So you can't just pass a pointer to a GSoft BASIC subroutine; it won't work. That's where the ALLOCATEPROC and DISPOSEPROC GSoft functions come into play.

ALLOCATEPROC allocates a special pointer that can be used by the toolbox to dispatch control to a GSoft BASIC function, and DISPOSEPROC deletes one allocated procedure pointer. This special pointer sets up the appropriate environment the toolbox expects and runs the GSoft BASIC subroutine or function you indicated when you first allocated the procedure pointer.

This pointer needs to be tracked on a document-by-document basis, so we add a reference to it to the documentStruct type we've already defined. Also, when a window is closed, the pointer has to be disposed of. You can see all this being done in Listing 5-7.

```
  ! Document record definition

  type documentStruct                       : ! information about our document
    after as pointer to documentStruct      : ! next document
    wPtr as GrafPortPtr                     : ! window pointer
    wName as String                         : ! window name
    onDisk as Boolean                       : ! does the file exist on disk?
    drawProc as procPtr                     : ! pointer to drawing proc
  end type
  type documentPtr as pointer to documentStruct  : ! document pointer

  !
  ! DrawContents
  !
  ! Draws the contents of the window
  !
  sub DrawContents
    dim r as Rect                           : ! Port rectangle

    PenNormal                               : ! Set default drawing pen
    GetPortRect(r)                          : ! Get the window's size
    MoveTo(r.h1, r.v1)                      : ! Draw the X
    LineTo(r.h2, r.v2)
    MoveTo(r.h1, r.v2)
    LineTo(r.h2, r.v1)
  end sub

  …
  CloseWindow(dPtr^.wPtr)                   : ! close the window
  DisposeProc(dPtr^.drawProc)               : ! dispose of the draw proc
pointer
  …

  …
  dPtr^.drawProc = AllocateProc(DrawContents)
  s = chr$(len(wName)) + wName
  dPtr^.wPtr = NewWindow2(pStringPtr(@s), 0, dPtr^.drawProc, NIL,
refIsResource, wrNum, rWindParam1)
  …
```

Listing 5-7: An Update Subroutine

Problem 5-5: Add the update subroutine from Listing 5-7 to Frame. If you are using the version of Frame from Problem 5-4, take out the function X; it isn't needed any more. Be sure and change the call to `NewWindow2` in `NewDocument` so it passes the address of `DrawContents` to the Window Manager.

With these changes in place your window behaves like you expect it to. The contents are drawn once when you create the window, so you don't even have to do it manually. If you drag the window off of the desktop, then back on, the part that was erased before is redrawn. If your window is covered up, then uncovered, the area that was covered is redrawn correctly.

The Window Port and Coordinate Systems

When you click the mouse, a variety of different things can happen, depending on where the mouse was when you pressed the mouse button. If you press on the menu bar, `TaskMaster` handles a menu event; pressing on an inactive window makes it active; and so forth. All of these different kinds of actions are separated based on where the event occurs, and `TaskMaster` returns a distinct kind of event in each case. When you click in the content region of the active window, `TaskMaster` returns `wInContent` for the event and places a pointer to the window in `wmTaskData`. As with any event, the location of the event is in the `where` field.

Programs can do any number of different things when an event occurs in the window's content area, like position a cursor, start drawing a line, manipulate a control, or even ignore the event. We're going to create a simple dot drawing program based on these events. This will give us a simple but effective way to explore multiple windows, each containing different information; to see how scrolling works; and later, to explore printing and file access. Creating the actual program will be left for Problem 5-6, but there is one key concept you need to learn about before tackling that program. You can then use the program you write to explore this concept in more detail. This important concept is the idea of using multiple coordinate systems.

Up until we started drawing the X in our window, everything we'd done used global coordinates. Global coordinates are basically screen coordinates. (Technically the screen can move, but we'll ignore that since it is rarely an issue.) Global coordinates do go past the edge of the screen, extending from -32767 to +32787 in both the horizontal and vertical direction. The visible screen appears just below and to the right of the 0,0 point, extending for 200 pixels down and either 320 pixels or 640 pixels to the right, depending on which graphics mode you are using.

Each `GrafPort` (and a window has a `GrafPort`; that's what we draw in) has its own coordinate system, called local coordinates. As with global coordinates, the local coordinates extend from -32767 to +32767 in each direction. Although you'll learn to change the origin later, for now, the 0,0 point is always at the top, left edge of the content region of the window. The content region of the window is also called the window port, which is also the port rectangle until we start changing the origin.

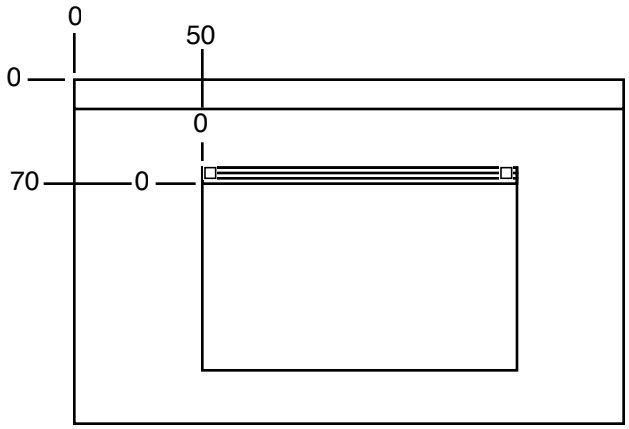

Figure 5-3: Global and Local Coordinates

All of these rather dry definitions are very important. Keep in mind that windows can move on the desktop, but you don't want the information inside the window to move relative to the window as it moves. When a window has a circle at the top left corner before you drag the window, the circle should still be at the top left corner of the window after you drag it. Local coordinates make this easy: when you use QuickDraw II drawing commands to draw in the window, it always uses the window's local coordinate system. Whether you move the window, cover it up, or drag it off of the screen, the top left corner of the window is still the 0,0 point.

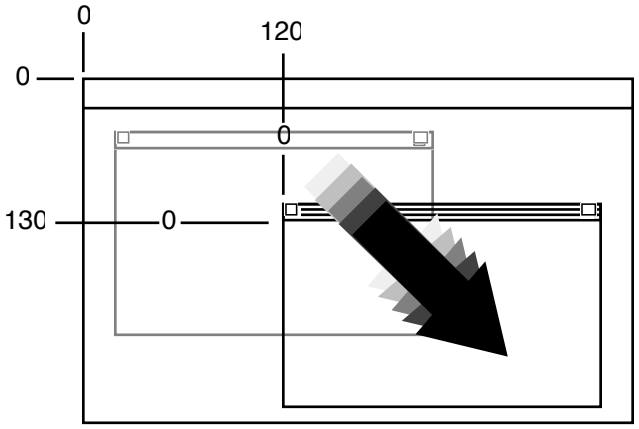

Figure 5-4: Dragging Doesn't Affect Local Coordinates

The problem is that the `where` value returned in the event record isn't connected with a particular window, so there is no practical way to decide whose local coordinates to use. So, instead of using local coordinates, the `where` field is returned in global coordinates. Dealing with this difference is not hard, but you do have to remember that there are two different coordinate systems in use, and be sure you convert between them when you need to. If you're not sure what coordinate system a tool call uses, you can always check the definition of the tool call in the *Apple IIGS Toolbox Reference* or in Appendix A, but as a general rule, tool calls that deal with a specific `GrafPort` (like the drawing commands) use local coordinates, while tool calls that do not deal with a specific port (like Event Manager calls) use global coordinates.

QuickDraw II has two easy to use commands to switch back and forth between local and global coordinates; they are `GlobalToLocal` and `LocalToGlobal`. Each takes a point as a parameter, and adjusts the point to the appropriate coordinate system. The only thing you have to be careful of is to make sure your `GrafPort` has been set before doing the conversion!

```
! convert where to local coordinates for wPtr
port = GetPort()
SetPort(wPtr)
localPoint = where
GlobalToLocal(@localPoint)
SetPort(port)
```

Problem 5-6: Starting with Frame, create a program that allows the user to draw dots in a window.

Start by adding a subroutine call in your event loop that will call a subroutine when a `wInContent` event is detected. Pass the position of the mouse in global coordinates, as well as the pointer to the window where the event occurred.

The subroutine itself should convert the position to local coordinates, then draw a point at that location. (You can draw a point with `MoveTo` and `LineTo` calls using the same coordinates.) The point should also be recorded in a new linked list of points, with the head of the list stored in the document record. (Keeping the list of points in the document record keeps the points separate for each document.)

Change the update procedure so it draws all of the points in the document's point list.

Be sure you dispose of all of the points in the linked list when you close the document!

When you test your program, be sure and try it with at least two open windows. Make sure your points stay correct for each window as the windows are dragged over one another, selected, dragged off of the screen, and resized.

Tricks With Update Events

One of the interesting things about the dot drawing program you wrote in the last section is that you had to draw the dots in two places: once in the update procedure and once in the subroutine that created the initial dots. There are some kinds of programs where handling the drawing in two different places can cause a lot of problems, both in terms of the amount of work you have to do as a programmer, and in terms of the bugs that can creep in when a single complicated task is done in two slightly different ways in two different places. (What if the drawing isn't done exactly the same way?)

There is a way to avoid drawing in two different places. Basically, you tell the Window Manager that the window needs to be updated, and let the update procedure do all of the drawing. If your update routine goes to the trouble of updating only the parts of the window that have changed, you can even tell the Window Manager to update just a portion of the window.

The Window Manager call `InvalRect` is used to force an update. You pass a rectangle as a parameter, and the Window Manager makes sure the rectangle gets redrawn by the update procedure at the next opportunity. Assuming you have already set the proper `GrafPort` with `SetPort`, this is all you need to do to force the entire window to be redrawn:

```
GetPortRect(r)
InvalRect(r)
```

The variable `r` is a `Rect`.

Problem 5-7: Start with the program you created in Problem 5-6. Instead of drawing the point when the mouse is clicked, add the point to the point list and use `InvalRect` to mark the window for update.

Scrolling

In most cases, a window only shows a small portion of the overall document. For example, in a word processor, you can only see a few lines of a document, even if you're editing an entire book. Scroll bars are used to move the window around to see different parts of the document, and if you know how to read them, they even give you clues as to the total size of the document and where you are in the document.

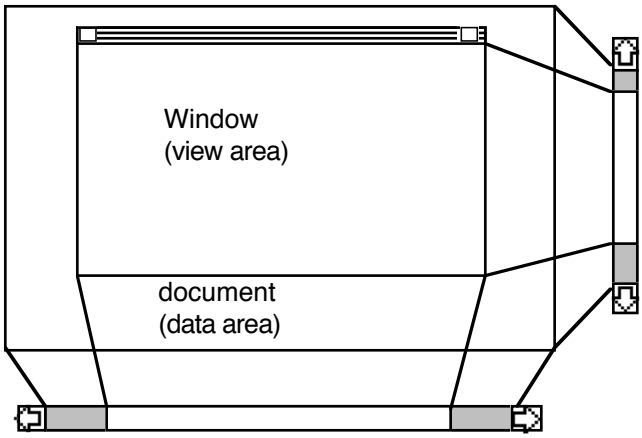

Figure 5-5: Scroll Bars and the Document

Looking at Figure 5-5, you can see that the thumb of the scroll bar (the thumb is the white part on the gray background) shows you both where you are in the document, as well as roughly how much of the document you can see. There is a minimum size for the thumb, so for a very large document, the size of the thumb doesn't mean anything.

There are five actions the user can control with the scroll bar. By clicking on one of the two arrows, the document is scrolled in the direction of the arrow by a small amount; this is generally one line in a text editor, so we call this line scrolling. By clicking in the page area (the gray area between the thumb and arrows) the user can scroll one page in either direction. A page is generally a little smaller than what you can see in the window. Finally, by dragging the thumb, the user can jump around in the document.

`TaskMaster` handles most of the details concerning the scroll bars for you, but you need to set a few parameters so `TaskMaster` knows how you want the scroll bars handled. The first of these is the total size of the document itself; `TaskMaster` uses this for three reasons. First, by comparing the total size of the document to the current port rectangle for the window,

TaskMaster can decide how big to make the scroll bar's thumb. Second, knowing the size of the document tells TaskMaster how far over the thumb should be when you have scrolled, say, two pages into the document. Finally, the size of the document tells TaskMaster whether you can scroll at all. If the entire document is visible, the thumb and paging area disappear, and if you scroll to either end of the document, TaskMaster stops you from going any farther.

The size of the data area can be set two different ways. The first is when the window is created, by setting the size of the data size field. In our programs so far, the data size field has been set to 1,1. This is a great way to set the data size for programs that have a fixed data size, like a paint program or CAD program that starts with a one-page data area. It's also a good place to set an initial default size for programs like word processors, which have a much more variable data size.

Once the window is open you can set the data size with a call to the Window Manager routine SetDataSize. You pass the width and height of the data area in pixels, along with the window pointer. This subroutine doesn't redraw the controls with the new settings, though, so you generally need to follow this up by either marking the window for update, as we did in the last section, or by drawing the controls manually with the Control Manager's DrawControls call, which redraws all of the controls in a window.

```
! Change the data size to 640 by 200
SetDataSize(640, 200, wPtr)
DrawControls(wPtr)
```

When the user scrolls the window, either using the arrows or the page area, TaskMaster needs to know how far to move. In this case, it's almost always best to set the sizes in the original resource defining the window. The line size is controlled by the field in the resource definition with the comment "scroll ver hors". (Refer back to Listing 5-1B to see these fields. We'll define these a little better later in the lesson, but for now, just change the fields based on the comments listed here.) The vertical size can be just about anything you want, but the horizontal scroll size needs to be a multiple of 8. The reason for this has to do with the way dithered colors are drawn in 640 mode, which is something we won't talk about in detail until the lesson on QuickDraw II. Basically, if a picture is shifted by something other than a multiple of 8, the colors can change in 640 mode.

The next two entries in the resource, "page ver horiz", are the number of pixels to scroll when the page area is clicked. You can set a specific value here, but then you will need to change it each time the window changes size. It's easier to do what we've done, setting these values to 0. TaskMaster will scroll by the physical pages size minus 10 pixels with this setting.

Of course, you can set these values from your program, too. The SetScroll call sets the number of pixels to move when the arrows are clicked, while SetPage controls the number of pixels to scroll when the click is in the page area. Each of these has the same parameters as SetDataSize, namely the horizontal value, the vertical value, and the window pointer.

```
! Set up for 8 pixel line scrolling
SetScroll(8, 8, wPtr)
! Set up for full page scrolling
GetPortRect(&r)
SetPage(r.h2-r.h1, r.v2-r.v1, wPtr)
```

Finally, you need to tell TaskMaster where the data area starts. In most cases, you'll start with the top left part of the data area – that would be the beginning of a text document, or the top left corner of a picture. That means you start with an origin of 0,0, which is what we've been using in our window resource. The field controlling the starting position is labeled "Origin". You can change this manually from the program using SetContentOrigin, which also takes a horizontal position, a vertical position, and a window pointer, just like the other calls we've been discussing. You might want to do this, for example, in a text editor if the user starts typing while the insertion point is not on the visible part of the screen. In that case, you could start off with a SetContentOrigin call to display the text containing the insertion point so the user can see what he's typing.

TaskMaster does almost everything for you when it comes to handling the scroll bars, but there is one thing you have to do for yourself. The LocalToGlobal and GlobalToLocal functions that convert between the global coordinate system and the local coordinate system for a GrafPort don't take the origin into account. If you scroll a window, then convert from one coordinate system to another, the result will be off by however far you scrolled. To take this into account, you need to call GetContentOrigin, which tells you what the origin is. Add the origin to a point after calling GlobalToLocal, and subtract this value from a point before calling LocalToGlobal.

The GetContentOrigin call itself is a little odd; it returns a long integer. The least significant word in this long integer is the vertical origin, while the most significant word is the horizontal origin. You've been pulling apart long integers into two words for quite a while to separate the menu ID and menu item ID from a single long integer value, and you can use the same idea to pull apart the value returned by GetContentOrigin. If you get stuck on this, you can look at the solution to Problem 5-8, which shows at least one way to handle the conversion.

Problem 5-8: Change the dot drawing program from Problem 5-7 to allow a full-screen drawing of 200 by 640 pixels. Set the line scrolling values to 8 pixels high and 16 pixels wide, and leave the page scrolling areas alone so TaskMaster will calculate an appropriate page scroll distance. With this simple change, you program can start using scroll bars, handling large documents.

Be sure to add a new point after scrolling the window. If you've taken the origin into account, everything will work correctly, but if you haven't, the points will appear in a different spot than where you see the cursor.

Customizing Your Windows

In the last section we started messing around with the values in the window resource, which we've treated as a black box up until this point. It's time to go back and shine some light in that black box.

When you create a window, you pass a lot of information to the Window Manager to describe the window you want to create. This information can be in the form of a structure using the NewWindow call, or a resource and some parameters to override the defaults in the resource, as with the NewWindow2 call. Either way, the information you supply is basically the same; only the way you pass the information changes. The rest of this section describes the various parameters

and what they are used for. Right after that, we'll look at how the values are set using two different Window Manager calls, `NewWindow2` and `NewWindow`.

One of the problems at the end of the section creates a fun, fairly useful window explorer program. It's a rather long problem, but it does help firm up the concepts in this section, and it also gives you a great visual aid to explore the various window options.

wFrameBits

The `wFrameBits` field is a word with 16 bits, each controlling some visual aspect of the window.

`fTitle` 15 If this bit is set, the window will have a title bar. If this bit is clear, the window will not have a title bar.

`fClose` 14 If this bit is set, the window will have a close box at the left side of the title bar. If this bit is clear, there will not be a close box. You might leave off a close box for the main window in a game, for example, so the main display can't be closed. You must clear this bit if the window does not have a title bar (i.e. if `fTitle` is 0).

`fAlert` 13 Set this bit to 1 for an alert-style window frame. An alert style window has a double-lined frame around the content region. Alert windows don't have much besides a window frame, so if you set this bit, you should clear `fInfo`, `fZoom`, `fFlex`, `fGrow`, `fBScroll`, `fRScroll`, `fClose` and `fTitle`.

`fRScroll` 12 If this bit is set, the Window Manager will create a scroll bar at the right side of the window to scroll up and down in the document. This scroll bar must be handled by `TaskMaster`; if you want to create your own scroll bar, leave this bit clear and use the Control Manager to create the scroll bar.

If this bit is set, `fBScroll` and `fGrow` should also be set.

`fBScroll` 11 If this bit is set, there will be a bottom scroll bar. As with `fRScroll`, you only use this bit if you want `TaskMaster` to handle scrolling for you.

If this bit is set, `fRScroll` and `fGrow` should also be set.

`fGrow` 10 If this bit is set, there will be a grow box in the corner formed by the scroll bars at the lower right corner of the window. If the bit is clear, there will not be a grow box. As with the scroll bars, this grow box must be handled by `TaskMaster`; to handle a grow box manually, you must create it yourself using the Control Manager.

If this bit is set, `fBScroll` and `fRScroll` should also be set.

`fFlex` 9 If this bit is set, the data height and width are flexible. If the bit is clear, `GrowWindow` and `ZoomWindow` (called by `TaskMaster` to resize the window) will change the origin when the window size changes.

`fZoom` 8 If this bit is set, there will be a zoom box at the right end of the title bar; if the bit is clear, there won't be a zoom box. You must set `fTitle` to get a title bar if this bit is set.

`fMove` 7 In most cases, the title bar for the window is also its drag region. In other words, you can move the window by dragging the title around. If you set this bit, things behave as you would expect; if this bit is clear, you can still have a title bar, but the window can't be moved.

`fQContent` 6 Most of the time, when you click in the content region of a window, you expect the window to become the front window, but you don't expect anything else to happen. If this bit is set, clicking in the content region of a window not only brings it to the front, but it also acts as if you actually clicked in the content region. If this bit is clear, clicking in the content region of a window that isn't the front window just brings it to the front.

`fVis` 5 If this bit is set, the window is visible; if it is clear, the window exists, but it is invisible. The window stays invisible until you use the `ShowWindow` call to make it visible. You can also hide a visible window using `HideWindow`. (We won't be using these in the course, but they are in Appendix A.)

It may seem sort of silly to create an invisible window, but there's actually a very good reason for it. In complicated windows with lots of controls (like dialogs, which are a special kind of window) it can take a lot of time to create the items that go in the window. If the window is created, and you then add the controls, the user sees all of the construction as it occurs. The same thing can happen if you are using a standard resource for a window, and making changes after the window is open.

If, on the other hand, you create an invisible window, then create the controls and make any changes, and finally use `ShowWindow` to make the window visible, the user doesn't see the construction of the window – he just sees a window, drawn all at once. Cases like this are when `fVis` should be clear.

`fInfo` 4 If this bit is set, the window will have an info bar. An info bar is an area at the top of the window, generally used for things like palettes or text editor rulers. An info bar has its own update procedure, so it acts like a little window all its own, imbedded in the main window.

		If this bit is set, the `wInfoHeight` and `wInfoDefProc` fields must have values.
`fCtlTie`	3	In most programs, when a window is not the front window, the controls look different. Scroll bars and grow boxes, for example, are hollow outlines. If this bit is set, `TaskMaster` will redraw the controls in the proper state to match the window. If the bit is clear, `TaskMaster` leaves the controls alone.
`fAllocated`	2	This flag is used internally by the Window Manager to determine if it allocated the memory for a window. It always does for your windows, ignoring whatever value you code.
`fZoomed`	1	This flag is set if the window is currently in its zoomed state, and clear if not.
`fHilited`	0	This flag is used internally by the Window Manager. You can set or clear the bit; the Window Manager ignores what you code.

wTitle

This field is a pointer to the title of a window. The title is a p-string, and must be in a fixed memory location.

The Window Manager draws any pattern in the window right up to the edge of the characters in this title, so you will normally leave at least one space on each end of the title.

If there is no title bar, this field can be set to `NIL`. It is generally set to `NIL` in resource files, and the window title is assigned when the `NewWindow2` call is made (note that in Rez, NIL is called NULL).

wRefCon

This long integer field is reserved for your use. You can set it to anything you like.

wZoom

This field is a `Rect`, with the normal complement of four integer fields (v1, h1, v2, h2) defining the edges of the rectangle. This `Rect` defines the size of the window when it is in its zoomed state. You can set the coordinates in the rectangle to 0, in which case the Window Manager picks out default values to zoom to the entire visible screen.

wColor

This is a pointer (or a resource number, in a resource description file) to a window color record. We'll look at window color records in detail in a moment.

wYOrigin, wXOrigin

These two fields define the origin for the window. As you know, scrolling a window changes its origin; setting these fields to a non-zero value lets you start off somewhere other than the top-left corner of your document.

Set these values to 0 if you are not setting the `fRScroll` and `fBScroll` bits in the `wFrameBits` parameter.

wDataH, wDataW

These values define the size of the entire document, in pixels. The values are used by `TaskMaster` to decide how far you can scroll using the scroll bars, as well as to decide how big the thumb should be in a scroll bar.

Set these values to 0 if you are not setting the `fRScroll` and `fBScroll` bits in the `wFrameBits` parameter.

wMaxH, wMaxW

These values define the maximum size for the window. If you code 0, the Window Manager will fill in values to let the window grow as big as the visible desktop.

Set these values to 0 if you are not setting the `fGrow` bit in the `wFrameBits` parameter.

wScrollVer, wScrollHor

These fields tell `TaskMaster` how many bits to scroll in each direction when the user clicks on a scroll bar arrow. `wScrollVer` is generally set to 8 or the height for a font, while `wScrollHor` is typically set to 8. Because of the way dithered colors are created in 640 mode, you really should set `wScrollHor` to some multiple of 8. The reasons are discussed in detail in a later lesson, when drawing using QuickDraw II is covered in detail.

Set these values to 0 if you are not setting the `fRScroll` and `fBScroll` bits in the `wFrameBits` parameter.

wPageVer, wPageHor

These fields tell `TaskMaster` how many bits to scroll in each direction when the user clicks in the page area of a scroll bar. In most cases, you will set these values to 0; this tells the Window Manager to pick an appropriate value. It will use a value 10 pixels smaller than the size of the window, so that paging will leave a little of the old page on the screen.

Set these values to 0 if you are not setting the `fRScroll` and `fBScroll` bits in the `wFrameBits` parameter.

wInfoRefCon

This value can be set to anything you like. It's for your use when creating info bars.

wInfoHeight

This is the height of the info bar, in pixels. The width of the info bar matches the width of the window itself, so there is no separate parameter for the width.

This value is only used if `fInfo` is set in the `wFrameBits` parameter.

wFrameDefProc

This pointer points to a subroutine that will be called when the Window Manager needs to draw the window. In all of the programs in this course, we'll set this to `NIL`, telling the Window Manager to use the standard subroutine for drawing a window frame.

While this parameter exists in any window definition, the Types.rez interface file for the Rez compiler hard-codes this value to 0, so it doesn't appear at all when you use Types.rez to format an `rWindParam1` resource.

If you are adventurous, you might want to experiment with creating your own window definition procedures. While it's pretty complicated, you can create round windows, windows in special shapes, or even hollow windows. For details, see the *Apple IIGS Toolbox Reference*.

wInfoDefProc

This pointer points to the subroutine to call when the information bar needs to be drawn. It works like `wContDefProc`, only for information bars.

Set this field to `NIL` if you did not set the `fInfo` bit in `wFrameBits`.

While this parameter exists in any window definition, the Types.rez interface file for the Rez compiler hard-codes this value to 0, so it doesn't appear at all when you use Types.rez to format an `rWindParam1` resource. That's because it has to be specified as a pointer, and the resource fork can't know the address of a subroutine in memory when the Rez source is compiled.

wContDefProc

This field points to your update procedure. We discussed the update procedure in detail earlier in the lesson.

While this parameter exists in any window definition, the Types.rez interface file for the Rez compiler hard-codes this value to 0, so it doesn't appear at all when you use Types.rez to format an `rWindParam1` resource, just like the `wInfoDefProc` value.

wPosition

This field is a `Rect`. It determines where the window is and how big it is when the window is first created. The rectangle, which you specify in global coordinates, defines the size of the content region of the window. Be sure to leave room for the window title bar, the system menu bar, and the scroll bars!

wPlane

When you create a window, you can actually define where it shows up – it doesn't have to be the frontmost window.

To make a new window the frontmost window, set this field to -1. To create a window that is behind all of the existing ones, use 0. You can also pass a specific `GrafPortPtr` in this parameter, in which case the new window will be right behind the one whose `GrafPortPtr` you pass.

wStorage

This field is used for different purposes, depending on whether you are using a resource for a `NewWindow2` call or setting up a window record for a `NewWindow` call.

For the `NewWindow` call, this parameter lets you allocate storage for a window yourself, rather than having the Window Manager set aside the memory. Setting the value to `NIL` tells the Window Manager to allocate memory for the window record itself, and that's what we'll do throughout this course.

In a window resource, this parameter is used for control lists. Later on, when we start dealing with the Control Manager up close and personal, we'll come back and use this field. For now, though, this value should be set to `NIL`.

Windows with Colors and Patterns

So far, the windows we've created have used `NIL` for the `wColor` parameter, which tells the Window Manager to use the default colors and patterns for a window. Unfortunately, the defaults are pretty gross. You get a solid black title bar, rather than the cute lined one you see in most programs. In this section, you'll finally find out how to create the coolest window frames.

The colors for the various parts of the window frame, as well as the way the title bar area is drawn, are controlled by a color table. This color table can be defined either as a resource or as a structure, but there is a restriction: like the window title, the color table must remain in a fixed area of memory. This means you have to set aside the space for the structure using a global variable or dynamically allocated memory; you cannot use a local variable, since local variables vanish when you return from a subroutine. If you are using a resource, the Window Manager makes sure the color table stays in one spot.

The color table itself is made up of five integers. These integers are divided into four-bit groups, with an occasional eight-bit group for variety. In the definitions below, the various four- and eight-bit groups are marked as letters in hexadecimal digits. These digits are then explained in the following table.

The table format you see here is very easy to refer to, but it's a lot more graphic to actually see the colors on the screen. That's one of the things you'll do with the program you write at the end of this section.

All of the colors can be any value from 0 to 15. These represent solid colors in 320 mode, and dithered colors in 640 mode. In most cases, you'll use 0xF (white) or 0x0 (black).

```
frameColor      0x00a0
titleColor      0x0bcd
tBarColor       0xeefg
growColor       0xh0ij
infoColor       0xk0m0
```

a This is the outline color for the window frame. It includes the lines around the edge of the window, the lines around the info bar, the lines outlining the close box and grow box, and the lines used to draw the small boxes inside the grow box.

b The background color of an inactive title bar. This is the color behind both the text of the title, and the color used to fill the title bar itself. In most cases, all of the windows except the front window are inactive; this is handled for you by `TaskMaster`.

c This is the foreground color of the text when the window is inactive. This is usually the same as the foreground text color for an active window (nibble d), but you could set this to a different color, say to get gray text when a window is inactive.

d This is the color of the text when the window is active.

ee These eight bits define the type of title bar. There are three types of title bars:

- 0x00 Solid title bars are the normal kind, by default a boring black. The color is actually set by digit g.
- 0x01 Dithered title bars use a checkerboard pattern, alternating between the foreground and background colors set by digits f and g.
- 0x02 Lines title bars are the sort you see most often, with lines running across the title bar. Digits f and g define the colors.

f This is one of the colors for a dithered title bar, or the color of the lines on a lined title bar.

g This is the second color for a dithered title bar, the background color for a lined title bar, or the solid color used for a solid title bar.

h This digit is only used on alert frames, where it defines the color between the outside line around the window and the heavy line just inside the main outline.

i This is the interior color for the grow box when the window is not selected.

j This is the interior color for the grow box when the window is selected.

k This digit is only used on alert frames, where it defines the color of the heavy box that runs inside of the main window outline.

m This is the interior color for an info bar.

The next section gives an example, showing you exactly how to set up a resource-based color table for a lined window.

Creating Custom Windows With Resources

In most cases, you'll want to do just what we've done so far in this lesson, using resources to create a window definition, then calling `NewWindow2` to create the window. Back at the start of this lesson, we used this resource definition to set up our standard document window:

```
resource rWindParam1 (1001) {
   $DDA5,                       /* wFrameBits */
   nil,                         /* wTitle */
   0,                           /* wRefCon */
   {0,0,0,0},                   /* ZoomRect */
   nil,                         /* wColor ID */
   {0,0},                       /* Origin */
   {1,1},                       /* data size */
   {0,0},                       /* max height-width */
   {8,8},                       /* scroll ver hors */
   {0,0},                       /* page ver horiz */
   0,                           /* winfoRefcon */
   10,                          /* wInfoHeight */
   {30,10,183,602},             /* wposition */
   infront,                     /* wPlane */
   nil,                         /* wStorage */
   $0000                        /* wInVerb */
   };
```

Listing 5-8: Resource For a Standard Document Window

Let's update that definition, adding cool, lined windows. The new resource, along with the appropriate color table, looks like this:

```
resource rWindParam1 (1001) {
   $DDA5,                       /* wFrameBits */
   nil,                         /* wTitle */
   0,                           /* wRefCon */
   {0,0,0,0},                   /* ZoomRect */
   linedColors,                 /* wColor ID */
   {0,0},                       /* Origin */
   {1,1},                       /* data size */
   {0,0},                       /* max height-width */
   {8,8},                       /* scroll ver hors */
   {0,0},                       /* page ver horiz */
   0,                           /* winfoRefcon */
   10,                          /* wInfoHeight */
   {30,10,183,602},             /* wposition */
   infront,                     /* wPlane */
   nil,                         /* wStorage */
   $0800                        /* wInVerb */
   };
```

```
resource rWindColor (linedColors) {
   0x0000,                    /* frameColor */
   0x0F00,                    /* titleColor */
   0x020F,                    /* tbarColor */
   0xF0F0,                    /* growColor */
   0x00F0,                    /* infoColor */
   };
```

Listing 5-9: Resource For a Lined Document Window

Finally, let's give this same definition again, but this time, instead of actual values, we'll use the field names that we've used during this whole, long section. That way, you can compare the description of the field you want with the place you need to put the value in the resource.

```
resource rWindParam1 (1001) {
   wFrameBits,                /* wFrameBits */
   wTitle,                    /* wTitle */
   wRefCon,                   /* wRefCon */
   wZoom,                     /* ZoomRect */
   wColor,                    /* wColor ID */
   {wXOrigin,wYOrigin},       /* Origin */
   {wDataH,wDataW},           /* data size */
   {wMaxH,wMaxW},             /* max height-width */
   {wScrollHor,wScrollVer},   /* scroll ver hors */
   {wPageHor,wPageVer},       /* page ver horiz */
   wInfoRefCon,               /* winfoRefcon */
   wInfoHeight,               /* wInfoHeight */
   wPosition,                 /* wposition */
   wPlane,                    /* wPlane */
   wStorage,                  /* wStorage */
   $0800                      /* wInVerb */
   };
```

Listing 5-10: Window Resource By Field Name

With the exception of the last one, all of these parameters have been explained in gory detail already. The big exception is the last parameter; it's the one with the comment `wInVerb`. This parameter tells the Window Manager where to look for certain things after the resource has been loaded. This field is a set of flags, used for three different purposes.

Bits 15 to 12 (the first four bits) are reserved, and should be set to 0.

Bits 7 to 0 (the last eight bits) are used when we define a control list for the window. We won't be doing that for a while, so set this part of the value to 0 for now.

The remaining four bits are divided into two two-bit fields. Bits 10 and 11 tell the Window Manager what sort of entry you're using in `wColor`, while bits 8 and 9 tell what sort of entry is in the `wTitle` field. Each of these can be 00 for a pointer, 01 for a handle, or 10 if you're using a resource ID. Back before we started using color tables, this value was set to 0x0000, which told the Window Manager to look for a pointer in both fields, and we passed `NIL` for both the color table and title. That basically told the Window Manager there was no title or color table. Now, with a color table defined via a resource, we're using a value of 0x0800, which still tells the Window Manager to use a pointer for the window title, but we're using a resource for the color table. The resource ID number for the color table goes in the `wColor` field; the resource ID is the

same number you put in parenthesis in the `rWindColor` resource. Of course, if you are using different color tables for different windows, you need to use different `rWindColor` resources and resource ID numbers, but if you'll be using cool lined color tables for all of the windows, save some space and use the same color table for all of your windows.

Some of these fields just don't work well from a resource definition, since they change for practically every window. A good example is the window title, which is generally either "Untitled x" or the name of the disk file where the information displayed when the document is actually stored. `NewWindow2` lets you override several of these parameters. Here's the `NewWindow2` call with our familiar field names instead of actual parameters:

```
NewWindow2(wTitle, wRefCon, wContDefProc, wFrameDefProc, paramTableDesc,
   paramTableRef, resourceType);
```

The last three are not normal window parameters; they are used to tell the Window Manager which resource to use. The first of these new parameters, `paramTableDesc`, tells the Window Manager what the next parameter is. We'll be using 2 all of the time, telling the Window Manager that `paramTableRef` is a resource ID number, but it is possible to tell the Window Manager to use a pointer or handle, instead. The last parameter, `resourceType`, tells the Window Manager which of two possible window resource types we're using. We'll always use `rWindParam1`.

Of course, I glossed over a lot of options by just telling you what we'll be using in the course. In a way that's unfair, since you may get the feeling I'm hiding something. The real reason for skipping the various alternatives is that we're covering the most useful ones. Adding a detailed discussion of all of the rest of them would be overkill – besides, I've optioned you to death in this section! If you really feel like you just have to know about all of those other options, check out Appendix A or *Apple IIGS Toolbox Reference*.

Using NewWindow and Window Records

There's one big disadvantage to using `NewWindow2` and a resource to define a window: It's a real pain to define windows that need something other than `wTitle`, `wRefCon`, `wContDefProc` or `wFrameDefProc` changed for each new window. (If you'll recall, these are the parameters that you can pass as parameters to the `NewWindow2` call.) The big problem at the end of this section is a window sampler, where you'll be changing most of the `wFrameBits` bit flags as well as the color table. Since those can't be passed as parameters to `NewWindow2`, we'll cover a very similar call called `NewWindow` in this section. `NewWindow` has the disadvantage of using a structure that has to be filled in field by field with assignment statements, but that's a lot easier than changing a resource on the fly.

Basically, using `NewWindow` is a lot like using `NewWindow2`; the thing that changes is how you specify all of the parameters. Instead of defining a resource, you define a window record; this window record can be a local variable, since the Window Manager takes what it needs. The color table, though, still has to be defined as a fixed variable, so we'll stuff that in our document record. Listing 5-11 shows a subroutine that will define a window using a `NewWindow`.

```
  !
  ! NewWind
  !
  ! Create a new window.
  !
  ! Parameters:
  ! dPtr - document for which to create the window
  !
  ! Returns: Pointer to me window, NIL for error.
  !
  function NewWind(dPtr as documentPtr, title as String) as GrafPortPtr
    shared frameTitle, frameClose, frameAlert, frameCtlTie
    shared frameControls, frameFlex, frameZoom, frameMove
    shared titleKind, titleSolid, titleLined, titleDithered

    dim wParms as paramList         : ! Window parameters

    dPtr^.colorTable.frameColor = $0000
    dPtr^.colorTable.titleColor = $0F00
    select case titleKind
      case titleSolid:
        dPtr^.colorTable.titleColor = $0F0F
        dPtr^.colorTable.tBarColor = $0000
      case titleLined:
        dPtr^.colorTable.tBarColor = $020F
      case titleDithered:
        dPtr^.colorTable.tBarColor = $010F
    end select
    dPtr^.colorTable.growColor = $F0F0
    dPtr^.colorTable.infoColor = $00F0

    wParms.paramLength = sizeof(paramList): ! Initialize the size
    wParms.wFrameBits = $0025

    if GetMItemMark(frameTitle) then wParms.wFrameBits =
wParms.wFrameBits+fTitle
    if GetMItemMark(frameClose) then wParms.wFrameBits =
wParms.wFrameBits+fClose
    if GetMItemMark(frameAlert) then wParms.wFrameBits =
wParms.wFrameBits+fAlert
    if GetMItemMark(frameControls) then wParms.wFrameBits =
wParms.wFrameBits+fGrow+fBScroll+fRScroll
    if GetMItemMark(frameFlex) then wParms.wFrameBits=wParms.wFrameBits+fFlex
    if GetMItemMark(frameZoom) then wParms.wFrameBits=wParms.wFrameBits+fZoom
    if GetMItemMark(frameMove) then wParms.wFrameBits=wParms.wFrameBits+fMove
    if GetMItemMark(frameCtlTie) then wParms.wFrameBits =
wParms.wFrameBits+fCtlTie

    wParms.wTitle = pStringPtr(@title)
    wParms.wRefCon = 0
    wParms.wZoom.h1 = 0 : wParms.wZoom.h2 = 0
    wParms.wZoom.v1 = 0 : wParms.wZoom.v2 = 0
    wParms.wColor = @dPtr^.colorTable
    wParms.wYOrigin = 0
    wParms.wXOrigin = 0
    wParms.wDataH = 200
    wParms.wDataW = 640
    wParms.wMaxH = 0
```

```
    wParms.wMaxW = 0
    wParms.wScrollVer = 8 : wParms.wScrollHor = 8
    wParms.wPageVer = 0 : wParms.wPageHor = 0
    wParms.wInfoRefCon = 0
    wParms.wInfoHeight = 0
    wParms.wFrameDefProc = NIL
    wParms.wInfoDefProc = NIL
    dPtr^.drawProc = AllocateProc(DrawContents)
    wParms.wContDefProc = dPtr^.drawProc
    wParms.wPosition.v1 = 30
    wParms.wPosition.h1 = 10
    wParms.wPosition.v2 = 183
    wParms.wPosition.h2 = 602
    wParms.wPlane = GrafPortPtr(topMost)
    wParms.wStorage = NIL
    NewWind = NewWindow(wParms)
  end function
```

Listing 5-11: Subroutine to Define a Window with `NewWindow`

The parameters you actually pass in this structure are basically the same old familiar ones you used with resources. Here are the differences:

`paramLength` This is the size of the window structure in bytes. Like all of the parameters that appear here but not in the `rWindParam1` resource descriptions we've used, this parameter is actually in the resource, too, but Types.rez sets it to a constant, so we normally don't worry about it in the resource description file.

`wFrameDefProc` This is the address of a subroutine that will draw the window frame. It's used for custom windows. For one odd example, by using this parameter and a few others, you could create a round window.

`wInfoDefProc` This is the address of a subroutine that draws the contents of the info bar.

`wContDefProc` This is the address of the subroutine that draws the contents of the window. Since the window is set up at run time, we can set this address here, rather than passing it as a parameter to `NewWindow`.

`wStorage` This is a pointer to the window structure itself. It's almost always set to `NIL`, forcing the Window Manager to allocate the space.

Problem 5-9: This problem makes a small change to our Frame program. This is the version we'll use in future chapters.

Add the window color table shown in Listing 5-9 to your Frame program. (Start with the version developed for Problem 5-5.) To keep the lines from butting right up against the window name, be sure and add two spaces to the left and right of the window title string.

Problem 5-10: This problem is a fairly long one, but it's well worth the effort. Even if you don't work the problem, take time to read it so you understand what it's all about, then run the

solution. The problem develops a window sampler that lets you quickly try out the various window color and `wFrameBits` options.

a. Starting with the Frame program from Problem 5-9, add a window color record to the document record. This window color record will be filled in for each window we create.

   ORCA/C's interface file for the Window Manager has a structure defined for color tables. To save you the trouble of looking it up, here's the structure you should use to define the color table:

```
type wColorTbl
   frameColor as integer
   titleColor as integer
   tBarColor as integer
   growColor as integer
   infoColor as integer
end type
type wColorPtr as pointer to wColorTbl
```

b. Add a menu called `wFrameBits`. Add these menu items to the menu:

```
fTitle   fClose   fAlert   Controls fFlex    fZoom    fMove    fCtlTie
```

   All of these but `controls` is the name of one of the bits in `wFrameBits`; `controls` is used for `fRScroll`, `fBScroll` and `fGrow`, since these have to be set as a group. Add the appropriate code to your program so the user can select or deselect these options. When the option is selected, show a check by the option in the window, erasing the check when the option is not selected.

c. Add another menu called Title. It should have three options: Solid, Lined and Dithered. Only one of these can be selected at a time.

d. Remove the code that calls `NewWindow2`, using the subroutine from Listing 5-11, instead. Form the color table and `wFrameBits` parameters based on the options that the user has selected from the menus.

Summary

This lesson has covered the basics of creating and using windows. It's been a long, long lesson, so it may seem strange to claim we only covered the basics, but there are a lot of things you can do with windows that we didn't cover – there are lots of Window Manager calls to manipulate windows that we won't use, and you can even create your own, custom windows. Still, this lesson gives you enough information about the Window Manager and windows to recreate what you'll see in the vast majority of desktop programs. If you have copies of the toolbox reference manuals, now would be a great time to browse through the chapters that cover the Window Manager.

Tool calls used for the first time in this lesson:

```
BeginUpdate       CloseWindow       DrawControls      EndUpdate
FrontWindow       GetContentOrigin  GetPort           GetPortRect
GlobalToLocal     HideWindow        InvalRect         LocalToGlobal
NewWindow         NewWindow2        SetContentOrigin  SetDataSize
SetPage           SetPort           SetScroll         ShowWindow
```

Resource types used for the first time in this lesson:

```
rWindColor        rWindParam1
```

Lesson 6 – File I/O

Goals for This Lesson

This lesson concentrates on the Standard File Operations Tool Set, which is a collection of prewritten routines that lets the user of a program specify what files to load and save. We'll follow the process through to its natural conclusion by learning to load files and save files using GS/OS, the disk operating system for the Apple IIGS. Along the way, we'll add file input and output subroutines to our Frame program.

SFO

Before we get going, let's stop and get a really good overview of what this lesson is all about. Most desktop programs need to load and save files; about the only exception would be a game of some sort. Dealing with file input and output involves two separate issues. The first is the communication that goes on between the program and the user of the program to decide which file to open or close. You've certainly gone thought the process of selecting Open from the File menu, seeing a dialog appear, and then selected a file. Saving a file is also handled by a dialog, this time a dialog that lets you pick a name for your file. If you pick a duplicate, you are warned, and so on. The process of asking for file names is so common on desktop programs, and there are so many right ways to do it, that Apple decided to create the Standard File Operations Tool Set. That's a real mouthful, so most people call it SFO. SFO displays dialogs for either opening or saving files, handles all of the interaction with the user up to the point that a file name has been picked, and then hands you the file name on a silver platter. There are some unusual cases when you might want to bypass SFO and do all of this for yourself, but they are very, very unusual. Almost all desktop programs that do file input or output use SFO to figure out what file name to use.

Once you have a file name, the second part of the problem is to actually load or save the file. You could do that with GSoft BASIC's built in file I/O commands, but there are some limitations that make GSoft BASIC's commands inadequate for a wide variety of programs. The two most severe problems are that you can't set the file type on a file you create, and GS/OS can load and save most files a lot faster than GSoft BASIC can. For both of these reasons, we'll also develop some simple, canned subroutines in this lesson that you can use to load and save files.

The Open Dialog

When you select Open from the File menu of a desktop program, you generally see something like the dialog shown in Figure 6-1.

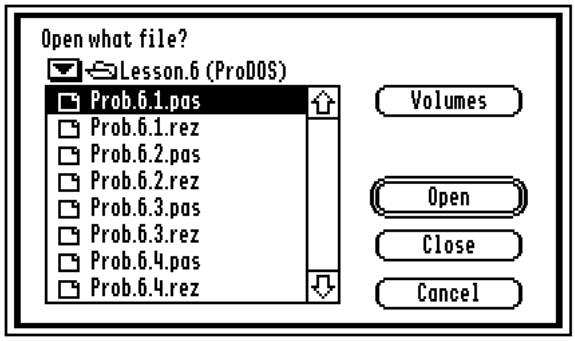

Figure 6-1: Open Dialog

I'm going to assume you know how to use this dialog to open a file. If not, you can find all of the information you need in the introductory books that came with your computer.

Creating this dialog is remarkably easy. All you have to do is call `SFGetFile2` with a few parameters. Here's a model call to `SFGetFile2` with some parameter names that we can use to talk about the various things you have to tell SFO to get a dialog like the one in Figure 6-1:

```
SFGetFile2(x, y, promptRefDesc, promptRef, filterProcPtr, typeListPtr,
   reply)
```

The first two parameters tell SFO where to put the dialog. The parameters x and y give the position of the top left corner of the dialog. Unfortunately, since the dialog can change size from one version of the system software to another, you can't really do the sensible thing and center the dialog.

There's a string, "Open what file?", at the top of Figure 6-1. You can change this string to whatever prompt you like, and that's what the next two parameters are for. `promptRefDesc` is the sort of parameter you should be getting used to by now; it tells SFO what sort of parameter you are passing in `promptRef`. If `promptRefDesc` is `refIsPointer` (a value of 0), `promptRef` is a pointer to a p-string. If it is `refIsHandle` (1), `promptRef` is the handle of a p-string. Finally, the most appropriate choice for `promptRefDesc` is `refIsResource` (2), which tells SFO that `promptRef` is the resource ID of an `rPString` resource. The reason that's the best choice is because all strings belong in the resource fork, where they can be changed by the user.

The `filterProcPtr` is a pointer to a function in your program. To understand what this subroutine is for you have to stop and remember that most programs only accept certain kinds of files. `SFGetFile2` is perfectly capable of weeding out files based on file types and auxiliary file types (you'll see how to do this in a moment), but there are occasionally strange situations where you want to be even more picky. The filter procedure gives you a way to be as picky as you want, selecting each individual file that will appear in the open dialog. We won't use filter procedures in this course, so we'll always set the `filterProcPtr` parameter to `NIL`.

The next parameter is a pointer to a list of file types you are willing to accept. In most cases, your program will only load files with a specific file type. A program that will load and display pictures, for example, would allow the picture file types, but reject GSoft BASIC program files. The process of picking out and displaying only the files that you can actually load is done automatically when you give SFO a list of file types. We'll talk about the file type list in detail in a moment.

Finally, `replyPtr` is a reply record you supply. SFO fills in the reply record with a flag telling you if the user picked a file or canceled the operation, and if a file was selected, the file type, auxiliary file type, file name, and full path name for the file. That's more than enough information to load the file.

File Type Lists

The file type list controls the files that are listed in the open dialog. You can specify files by file type, auxiliary file type, or a combination of both a file type and auxiliary file type. You can also tell SFO to list files in the dialog, but to make the files dimmed and unselectable.

A file type structure starts with an integer telling how many file type records are in the array of file types. This is followed by the correct number of file type entries, each with four parameters: a flags word, a file type, and an auxiliary file type. In GSoft BASIC, the records that are used for a file type structure list are defined by the following types:

```
type typeRec
  flags as integer
  fileType as integer
  auxType as long
end type

type typeList5_0
  numEntries as integer
  ; fileAndAuxTypes is a variable length array
  fileAndAuxTypes(9) as typeRec
end type
type typeList5_0Ptr as pointer to typeList5_0
```

If you need some different number of entries in your fileAndAuxTypes list, you should define your own type, using the desired number instead of 9 for the subscript there, and use type casting when passing references to that record to Standard File calls.

Three bits are used in the flags word. If bit 15 is set, SFO matches any file type. If bit 14 is set, SFO matches any auxiliary file type. Finally, if bit 13 is set, the files are displayed as dimmed and unselectable. The rest of the bits are reserved, and must be set to 0.

Checking the list of file types in Appendix A, you can see that a GSoft BASIC source file has a file type of 0xB0 and an auxiliary file type of 0x0104. A file type record to load BASIC source files would be set up like this:

```
types.fileAndAuxTypes (0).flags = $0000
types.fileAndAuxTypes (0).fileType = $B0
types.fileAndAuxTypes (0).auxType = $0104
```

If you wanted to load any source file, and not just GSoft source files, you would want to load any file with a file type of 0xB0, no matter what auxiliary file type the file had. In that case, you would set up the file type entry like this:

```
types.fileAndAuxTypes (0).flags = $8000
types.fileAndAuxTypes (0).fileType = $B0
```

Bits 15 and 14 can also work together, telling SFO to accept *any* file, no matter what file type or auxiliary file type the file has. Here's a complete type list record that will let you load any file at all:

```
dim types as typeList5_0

types.numEntries = 1
types.fileAndAuxTypes (0).flags = $C000
```

In this case we don't have to set the file type or auxiliary file type, since SFO is going to ignore them anyway.

The Reply Record

The reply record is where SFO tells you what file you are supposed to load. Here's the declaration for a reply record:

```
type replyRecord5_0
   good as integer
   fileType as integer
   auxFileType as long
   nameVerb as integer
   nameRef as long
   pathVerb as integer
   pathRef as long
end type
```

The first entry tells you if the user actually picked a file, or if they decided to cancel the operation. If the open has been canceled, `good` will be set to 0, and all of the other entries are invalid. You simply ignore the operation and get back to the event loop. If `good` is not zero, the rest of the fields tell you which file to load.

The `fileType` and `auxType` field are pretty obvious; they tell you the file type and auxiliary file type of the file type to load. If you only allowed one file type, or if your program doesn't depend on the actual format of the file (as in, for example, a copy program), you can ignore these fields.

SFO can return the file name and path name in a variety of ways; you control the method with `nameVerb` and `pathVerb`, which you have to set up in the reply record before calling `SFGetFile2`. The most reasonable choice (and the only one we'll talk about here) is 3, which tells SFO to allocate whatever memory is needed and return a handle in `nameRef` and `pathRef`. Handles are something we won't talk about in detail for a while yet. All you need to know for now is how to get at the actual string, and what to do with the handle when you are finished with it.

Using the `nameRef` and `pathRef` Handles

The file name and path name `SFGetFile2` returns in `nameRef` and `pathRef` are handles. We haven't talked about handles much, but you don't need to know much to use them. (Lesson 8 deals with handles in detail.) You can think of a handle as a pointer to a pointer, although there's a bit more to it than that. First off, the information a handle points to can move, so you need to use the `HLock` call to lock the handle before you try to get at the information the handle points to. Once you are finished looking at the information, you use `HUnlock` to unlock the handle again. Finally, handles are dynamically allocated memory, just like the memory you get using GSoft's `allocate` command, and you have to dispose of the memory when you are finished with it. You dispose of a handle and the memory allocated by the handle using `DisposeHandle`. All three of these calls – `HLock`, `HUnlock`, and `DisposeHandle` – take the handle that they work on as the only parameter.

The names returned by `SFGetFile2` are GS/OS output strings, also called class 1 strings. While these names may be called strings, they really aren't strings in the sense we use the term in GSoft. Instead, GS/OS is returning a structure consisting of three parts: a record length word, a name length word, and an array of characters. This array of characters can have just about anything inside, including null characters, so you have to be careful when you access the information. The biggest caution is to never use BASIC's string handling facilities on the characters in a GS/OS name, even if they have been copied into a BASIC string, since BASIC uses the null character to signal the end of a string, and null characters are legal characters in a GS/OS path name.

```
type gsosInString
   size as integer
   ; Change the array size as needed for your application
   theString(254) as char
end type
type gsosInStringPtr as pointer to gsosInString

type gsosOutString
   maxSize as integer
   theString as gsosInString
end type
type gsosOutStringPtr as pointer to gsosOutString
```

Listing 6-1: GS/OS String Definitions Used in GSoft BASIC

If you need GS/OS strings longer than 254 characters, you should define your own versions of these types.

The full path name is use when the file is opened and read, and again if the file is saved. The file name is used as the name of the window (with some spaces added on either side) and as the default name when the file is saved using the Save As... menu command. Basically, then, you need to save both the file name and the path name in the document record so they can be used later, and you also need to form a window name from the file name.

With all of this in mind, you can write a subroutine to implement the Open command, right up to the place where the file is read. There are a lot of details you have to deal with to develop a good routine, though, especially with the type casting and error handling. Listing 6-2 shows one way to handle all of these details.

```
!
! OpenDocument
!
! Handles opening a document.
!
sub OpenDocument
  const posX = 80                        : ! x position of dialog
  const posY = 50                        : ! y position of dialog
  const titleID = 102                    : ! prompt string resource ID

  dim dPtr as documentPtr                : ! pointer to new document
  dim fileTypes as typeList5_0     : ! list of valid file types
  dim gsosNameHandle as Handle     : ! handle of the file name
  dim gsosNamePtr as gsosOutStringPtr: ! pointer to the GS/OS file name
  dim i as integer                       : ! loop/index variable
  dim size as integer                    : ! GS/OS name length
  dim title as string                    : ! new document name
  dim reply as replyRecord5_0            : ! reply record

  fileTypes.numEntries = 2         : ! set up the allowed types
  fileTypes.fileAndAuxTypes(0).flags = $8000
  fileTypes.fileAndAuxTypes(0).fileType = $B0
  fileTypes.fileAndAuxTypes(1).flags = $8000
  fileTypes.fileAndAuxTypes(1).fileType = $04

  reply.nameVerb = 3                     : ! get the file to open
  reply.pathVerb = 3
  SFGetFile2(posX, posY, refIsResource, titleID, NIL, fileTypes, reply)

  if toolError <> 0 then
    FlagError(3, toolError)
  else if reply.good <> 0 then
    gsosNameHandle = Handle(reply.nameRef)
    HLock(gsosNameHandle)
    gsosNamePtr = gsosOutStringPtr(gsosNameHandle^)
    title = " "
    size = gsosNamePtr^.theString.size
    for i=0 to size-1
      title = title + chr$(gsosNamePtr^.theString.theString(i))
    next
    title = title + " "
    HUnlock(gsosNameHandle)

    dPtr = NewDocument(title)
    if dPtr = NIL then
      DisposeHandle(Handle(reply.nameRef))
      DisposeHandle(Handle(reply.pathRef))
    else
      dPtr^.fileName = Handle(reply.nameRef)
      dPtr^.pathName = Handle(reply.pathRef)

      if LoadDocument(dPtr) <> false then
        dPtr^.onDisk = true
      else
        CloseDocument(dPtr)
      end if
    end if
  end if
```

```
end sub
```

Listing 6-2: A Subroutine to Implement the Open Command

While this subroutine doesn't actually load a file from disk, it does call `LoadDocument`. That's where you would put the code to load the file itself. `LoadDocument` is a function, returning `TRUE` if the file was loaded successfully, and `FALSE` if the file couldn't be loaded for some reason. We'll assume `LoadDocument` flags its own errors and cleans up after itself, but as you see, getting rid of the document is still up to us.

This subroutine can be moved from program to program pretty easily. After all, the only thing that's likely to change is the list of file types that the program can read.

Problem 6-1: Implement the Open command, adding it to Frame. (Start with the solution to Problem 5-9.) Make sure you do all of the following:

a. Add two fields to your document record to save `nameRef` and `pathRef`. In Listing 6-2 these fields are called `fileName` and `pathName`. They should have a type of `Handle`.
b. When the New command is used to open a new document, be sure and set the file name and path name handles to `NIL`.
c. When a document is closed, be sure you check to see if the file name and path name fields have been filled in with handles. If so, dispose of the handles.
d. Add the subroutine from Listing 6-2 to your program, calling it when the user picks Open from the File menu.
e. Create a dummy `LoadDocument` subroutine. This subroutine needs to return `TRUE`, but for now, that's all it needs to do. When you're testing your program, try returning `FALSE` once to test the error handler.

The Role of Save and Save As...

There are two save commands in a standard File menu, Save and Save As... The Save command is the "quickie" save, replacing an existing copy of a file with an updated version of the same file. When the Save command is used, the program should use the full path name returned from the `SFGetFile2` call.

The Save As command is used when you want to save a document to a new file name. In that case, you use the `SFPutFile2` call, which works a lot like the `SFGetFile2` call. We'll cover the call itself in the next section. `SFPutFile2` draws the standard "save dialog," which lets you move around on a disk, or even between disks, create a new folder, and ultimately enter a file name to use when the file is saved.

This covers the case of a file loaded from disk fairly well, but there is one other possibility. When you use the Save command to save a document that was created with New, and has never been saved before, you should call the same subroutine you call when the Save As... command is used, since you need to get a file name. Way back when we first created a document record, we added a flag, `onDisk`, to tell whether a document had been created with the New or Open command, so the check to see if you should use Save or Save As... is fast and easy. Of course,

you do need to remember to set the `onDisk` flag to `TRUE` after saving a new document for the first time.

The Save As... Dialog

The save dialog itself is created with a call to `SFPutFile2`. Here's a prototype call, again with names of the parameters instead of values to make it easier to talk about the call.

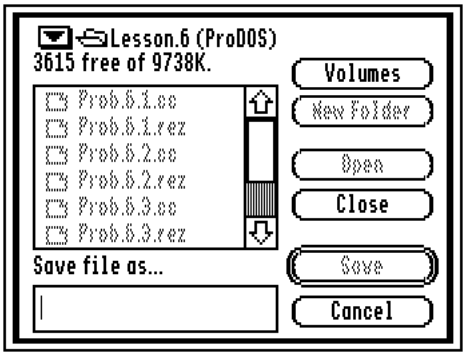

Figure 6-2: Sample Save Dialog

```
SFPutFile2(x, y, promptRefDesc, promptRef, origNameRefDesc, origNameRef,
   reply)
```

As with the `SFGetFile2` call, `x` and `y` give the position of the dialog, and `promptRefDesc` and `promptRef` give the prompt string that appears at the top of the dialog.

When the save dialog is drawn, the line edit box at the bottom where the file name is entered can have a default name. When a document exists on disk, this default name is the file name returned by the `SFGetFile2` call. If the file does not exist on disk, you can pass a string with no characters. The default name is passed in the `origNameRef` field, with `origNameRefDesc` telling `SFPutFile2` what sort of parameter is being passed. We'll use `refIsPointer` (which has a value of 0) for `origNameRefDesc`, telling `SFPutFile2` that we are passing a pointer to the default name.

Of course, that still leaves the issue of getting a pointer to a file name to pass. SFPutFile2 is expecting a GS/OS input file name. A GS/OS input file name is a length word followed by characters – sort of like an extra-long p-string. `SFGetFile2` gave us a GS/OS output file name, though, which has an extra integer at the start of the record. We can pass a pointer to this name when we call `SFPutFile2`, but we need to make sure the pointer points two bytes past the start of the value returned by `SFGetFile2`. You'll see one way to handle this in the sample code in Listing 6-3.

If the file was never saved to disk, we have a different problem. In that case there's no file name to pass, and we have to create one on the fly. You can use some sort of default name if you like, but a null string is appropriate to – that gives an immediate visual reminder that there literally is no file name for the file, yet. Creating a null file name is pretty easy, too; just pass a pointer to an integer set to 0.

The last parameter is the reply pointer. It is used exactly the same way as the reply pointer in `SFGetFile2`, but the file type and auxiliary file type fields are not filled in.

Once you get a file name, there are two other things you need to do. The first is to update the name of the window, which should change to reflect the new file name, and the second is to actually save the file.

Creating a new name for the window from the `SFPutFile2` reply record is no different than creating the window name from the `SFGetFile2` reply record. The difference is that this time the window already exists, so you need to let the Window Manager know that the name of the window has changed. You can do that with the `SetWTitle` call:

```
SetWTitle(name, wPtr)
```

The first parameter is the new name for the window; like the original name, it's a p-string. The second parameter is a pointer to the window you're changing.

Listing 6-3 shows two subroutines, `DoSave` and `DoSaveAs`, that can be used to implement all of the ideas we've covered. Both call `SaveDocument` to actually write the file to disk, and assume that `SaveDocument` will handle any errors in its own.

```
  !
  ! DoSaveAs
  !
  ! Handle the save as command.
  !
  sub DoSave
    const posX = 80                         : ! x position of dialog
    const posY = 50                         : ! y position of dialog
    const titleID = 103                     : ! prompt string resource ID

    dim dPtr as documentPtr                 : ! pointer to new document
    dim dummyName as integer                : ! used for a null file name
prompt
    dim gsosNameHandle as Handle            : ! handle of the file name
    dim gsosNamePtr as gsosOutStringPtr     : ! pointer to the GS/OS file name
    dim i as integer                        : ! loop/index variable
    dim size as integer                     : ! GS/OS name length
    dim reply as replyRecord5_0             : ! reply record
    dim s as string                         : ! used to build a p-string

    dPtr = FindDocument(FrontWindow): ! get the window
    if dPtr <> NIL then
      reply.nameVerb = 3                    : ! get the file to save
      reply.pathVerb = 3
      if dPtr^.fileName = NIL then
        SFPutFile2(posX, posY, refIsResource, titleID, refIsPointer,
clng(@dummyName), reply)
      else
        SFPutFile2(posX, posY, refIsResource, titleID, refIsPointer,
clng(dPtr^.fileName^)+2, reply)
      end if

      if toolError <> 0 then
        FlagError(3, toolError)
      else if reply.good <> 0 then
        gsosNameHandle = Handle(reply.nameRef)
        HLock(gsosNameHandle)
        gsosNamePtr = gsosOutStringPtr(gsosNameHandle^)
        dPtr^.wName = " "
        size = gsosNamePtr^.theString.size
        for i=0 to size-1
          dPtr^.wName = dPtr^.wName +
chr$(gsosNamePtr^.theString.theString(i))
        next
        dPtr^.wName = dPtr^.wName + " "
        HUnlock(gsosNameHandle)
        s = chr$(len(dPtr^.wName)) + dPtr^.wName
        SetWTitle(pStringPtr(@s), dPtr^.wPtr)

        dPtr^.fileName = Handle(reply.nameRef)
        dPtr^.pathName = Handle(reply.pathRef)
        dPtr^.onDisk = true
        call SaveDocument(dPtr)
      end if
    end if
  end sub

  !
```

```
! DoSave
!
! Save a document to the existing disk file.  If
! it's not already on disk, Save As is done
! instead.
!
sub DoSave
  dim dPtr as documentPtr                    : ! document to save

  dPtr = FindDocument(FrontWindow)
  if dPtr <> NIL then
    if dPtr^.onDisk <> false then
      SaveDocument(dPtr)
    else
      call DoSaveAs
    end if
  end if
end sub
```

Listing 6-3: Subroutines to Implement the Save and Save As Commands

A Comment About Ellipsis

An ellipsis is the series of three periods you see after some of the menu commands in desktop programs. They actually have a meaning. Any time a menu command brings up a modal dialog, forcing the user to stop and do something, the name of the menu command should be followed by three periods.

Up until now, the only menu command we've used that brought up a dialog is the about command. We've just changed the Open command so it brings up a dialog, though, and we're adding a new command (Save As) that also brings up a dialog. Be sure you change the names of your menus to take this into account.

Problem 6-2: Implement the Save As... command, adding it to Frame. Add a Save command subroutine, too, calling it when a new document is saved. Here's what your File menu should look like (and what the key equivalents should be!):

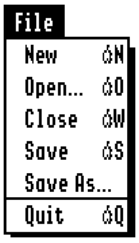

Figure 6-3: File Menu with Save and Save As

Make sure you do all of the following:

a. Add the `DoSave` and `DoSaveAs` functions from Listing 6-3, calling them when the user uses the Save or Save As... command.
b. Create a `SaveDocument` function. This doesn't need to do anything, yet, it just needs to exist since `DoSave` and `DoSaveAs` call it.

Problem 6-3: You created a program to draw dots on the screen in Problem 5-8. Combine this solution with the results from Problems 6-1 and 6-2 to create a program that can save and load lists of the dots.

You can use GSoft BASIC's built in file handling functions, `fopen`, `fclose`, `fread` and `fwrite`, to handle the file input and output. You'll want to open the output file with the command

```
open path for binary as #1
```

which opens a binary file, with a file type of 0x06 and an auxiliary file type of 0x0000. You will need those values for your `SFGetFile2` call.

Hint Just to make things easy, I saved the number of points in the file as the first integer. I saved the points by writing the point records in the point list.

Other Standard File Calls

We've covered the use of `SFGetFile2` and `SFPutFile2`, which do the job of figuring out file names for file loads and saves very well, but there are always exceptional cases that just don't quite work with the simple dialogs we've used. While we won't cover any of the other SFO calls, I'd like to mention a few of the biggies so you know they exist. If you would like to use any of the other SFO calls, you can refer to *Apple IIGS Toolbox Reference: Volume 3* for details.

It's not all that uncommon to need a few additional buttons in a dialog. One of the most common examples is a program that can save to several different file formats. In that case, you need to let the user pick the file format. A common way to do that is to add a button called Format, and open your own dialog when the button is pushed. The `SFGetFile2` and `SFPutFile2` calls don't give you any chance to add your own buttons, but two close cousins do. They are `SFPGetFile2` and `SFPPutFile2`.

In some kinds of programs, it might be nice to be able to select more than one file in an open dialog, but `SFGetFile2` only lets the user pick one file at a time. There is a way to get more than one file at a time, though, using the `SFMultiGet2` call.

All of these calls are documented in chapter 48 of the *Apple IIGS Technical Reference Manual, Volume 3*. Of course, there are a lot of other calls in SFO, too. Browsing through the chapters that cover SFO will reveal all sorts of tricks you can do with SFO.

Reading a File

Now that you know how to get a file name, it's time to actually load the file. You could load the file using GSoft BASIC's built in file handling functions, like `GET` and `PUT`, but there are some problems with these, mainly in the area of speed. We'll use GS/OS to load the file, instead.

GS/OS isn't technically a part of the Apple IIGS toolbox, although there's really no reason why it couldn't be. In fact, the Macintosh file I/O system is a tool. At the machine code level there are some very big differences in the way you call GS/OS and the way you make tool calls, but from the viewpoint of a high-level language, these differences vanish. In this course, we'll treat GS/OS as if it were a tool, and in fact, the GS/OS calls we'll use are listed in Appendix A just like the tool calls. For a complete description of GS/OS, you need to get a copy of *Apple IIGS GS/OS Reference*.

Reading a file is relatively easy with GS/OS. Here's the steps involved:

1. Call `OpenGS` to open the file.
2. Call `NewHandle` to get memory to load the file.
3. Call `ReadGS` to read the contents of the file into memory.
4. Call `CloseGS` to close the file.

Unlike the tools, every GS/OS call uses exactly one parameter, a structure that contains the various values that will be passed to the call. The GS/OS subroutines also return values in the same structure. In ORCA/C GS/OS errors are reported just like tool errors; you read the error number using the `toolerror` function, which is still 0 if there is no error to report.

Each of the GS/OS structures starts with a parameter count word that has to be filled in by you before the call is made. This parameter count word exists because you don't always have to pass the same number of parameters to a GS/OS call; you can abbreviate the parameters if you like. Here's the structure and call declarations for `OpenGS`. The structure is taken from the ORCA/C header file GSOS.h. Because of the way GS/OS calls are actually made, the function looks different in the header file; what you see here represents the way the function is used.

```
type openOSDCB
  pcount as integer
  refNum as integer
  pathName as gsosInStringPtr
  requestAccess as integer
  resourceNumber as integer
  access as integer
  fileType as integer
  auxType as long
  storageType as integer
  createDateTime as timeField
  modDateTime as timeField
  optionList as optionListPtr
  dataEOF as long
  blocksUsed as long
  resourceEOF as long
  resourceBlocks as long
end type

$20 SUB  OpenGS ((openOSDCB)
```

Listing 6-4: `OpenGS` Declaration

There are a lot of fields here, but we're only interested in four of them. The `pcount` parameter is the parameter count I mentioned; we need to set this before making a call. The `pathName` field is a pointer to a GS/OS input string; we'll pass the path name returned by the `SFGetFile2` call (after adding two to skip the initial buffer size, as before). The `dataEOF` field contains the total length of the file in bytes; we'll use this value to tell how much memory to allocate, and again when we read the file. Finally, once the file is open, GS/OS expects us to refer to the file with a number it returns; this number is called the file reference number, and is returned in the `refNum` field. We'll pass this `refNum` to both the `ReadGS` call and the `CloseGS` call.

If you look back in Appendix A at the documentation for `OpenGS`, you will find that the `pcount` field can be any value from 2 to 15. Counting the fields in the `OpenRecGS` record, `pcount` is 0, `refNum` is 1 and `pathName` is 2, so this tells us that we have to supply at least the name of the file to open, and as a minimum, GS/OS will tell us the reference number for the file so we can close it. Since you have to tell GS/OS which file to open, and GS/OS really does expect you to be polite enough to close the file when you're finished with it, it shouldn't be any surprise that you have to use at least those first two parameters.

Counting down the list, `dataEOF` is parameter number 12. We need this value, so the smallest value we can supply is 12.

There are three other input parameters in the `OpenOSDCB` record; they are the `requestAccess` parameter, the `optionList` parameter and the `resourceNumber` parameter. If we'd stuck with a `pcount` of 2, we could have ignored these parameters, but since we have used a `pcount` of 12, they must be filled in. It's a good idea to fill in the `requestAccess` parameter anyway, especially when the program may be used by people on a network. The `requestAccess` parameter is a flags word; bit 1 is set if you want to write to the file, and clear if not, while bit 0 is set if you want to read the file. The reason this pair of flags is so important is that more than one program can read a file at the same time, as long as all of them only open the file for input, but only one program can have a file open for output at any given time, and no one else can read

the file while it is open for output. For a data file on a network, opening the file for input only means that other people can load the file at the same time. That's a real possibility with a program that might be used by a class of students to open a file at the start of a class session. All of that boils down to using 1 for `requestAccess`, setting the read bit but leaving the write access bit clear.

The `optionList` parameter is something we won't cover here. Basically, it's a pointer to a buffer area GS/OS can fill in with extra information, but since we don't need any of the information, we can just set this value to `NIL` to tell GS/OS not to bother.

The `resourceNumber` parameter lets you tell GS/OS whether you want to open the data fork (use 0) or the resource fork (use 1). We'll use 0, since we're opening the data fork.

There are a wide variety of errors that can occur when you try to open a file, and it is very important to check for them with the `ToolError` function before you move on to the rest of the load process. The error checking isn't hard, you just have to remember to do it. After we finish talking about the rest of the calls, we'll collect all of the information into a single subroutine that loads a file. You can check out how the error handling is done then.

After opening the file, the next step is to reserve some memory. In this course, I'm assuming that you will load the file into memory, work on it there, and write the file once when the user uses the Save or Save As... command. There are other ways of handling files, such as leaving the file open and keeping only a small part of the file in memory. This is a good idea in some programs, like a commercial quality word processor that might need to let the user edit files larger than available memory. It's a bad idea in others, since the complexity of the program shoots up dramatically, and the program is also a lot slower.

Allocating the memory is done with the `NewHandle` call. We need to allocate `dataEOF` bytes of memory, and this memory should normally be moveable. That's a big help to the Memory Manager, as you'll discover in a couple of lessons. The call to `NewHandle` looks like this:

```
myHandle = NewHandle(ref.dataEOF, MMStartUp, $8000, NIL)
```

We'll cover `NewHandle` in a lot more detail in a later lesson; for now, use the call just like you see it.

Assuming the call is successful, `NewHandle` returns a handle to a chunk of memory. The memory will be locked at first, but after we read the file, we should unlock the handle so the Memory Manager can move the file around if it needs to. Before accessing the memory, we need to lock the file again.

Here are the declarations for the `ReadGS` call and its structure:

```
type readWriteOSDCB
  pcount as integer
  refNum as integer
  dataBuffer as ptr
  requestCount as long
  transferCount as long
  cachePriority as integer
end type

GSOS $12, $20 SUB ReadGS ((readWriteOSDCB)
```

Listing 6-5: `ReadGS` and the `readWriteOSDCB` Structure

We won't be using (or discussing) the cache priority field, so `pcount` will be set to 4. The `refNum` field needs to be filled in with the reference number returned by `OpenGS`. The `dataBuffer` field is a pointer to the place to put the bytes read; that would be `myHandle^`, which is a pointer to the first byte of memory reserved by the `NewHandle` call. The `requestCount` field tells `ReadGS` how many bytes to read; that should be filled in with `eof` from the `OpenGS` structure. The `transferCount` parameter is returned by `ReadGS`. It tells us how many bytes were actually read. That's pretty useful in situations where you are reading a file in small chunks, but we don't actually need the value, so we'll ignore it. The minimum allowed value for `pcount` is 4, though (check Appendix A for information like the minimum `pcount` value) so we have to let `ReadGS` fill in the field.

The last step in reading the file is to close the file with `CloseGS`.

```
type closeOSDCB
  pcount as integer
  refNum as integer
end type

GSOS $14, $20 SUB CloseGS ((closeOSDCB))
```

Listing 6-6: `CloseGS` and the `closeOSDCB` Structure

While the other GS/OS calls weren't all that complicated, this one is trivial. You just pass the reference number for the file to close, along with a `pcount` of 1, and GS/OS closes the file.

All of this is pulled together in Listing 6-7, which shows a subroutine that loads a file into memory. This subroutine does all of the appropriate error checking, calling our standard error handler if an error is found. (You should add message 4, "File read error", to the list of error messages in your resource file.) If the load is successful, the subroutine puts the handle to an unlocked chunk of memory containing the file in the document record; if the file can't be loaded for some reason, the subroutine sets the data handle to `NIL`.

```
!
! LoadDocument
!
! Implement this to load the document file.
!
! Parameters:
!   dPtr - pointer to the document to load
!
! Returns TRUE if successful, FALSE otherwise
!
function LoadDocument(dPtr as documentPtr) as boolean
  dim clRec as closeOSDCB               : ! CloseGS record
  dim opRec as openOSDCB                : ! OpenGS record
  dim rdRec as readWriteOSDCB           : ! ReadGS record
  dim path as gsosOutStringPtr
  dim port as GrafPortPtr               : ! caller's GrafPort
  dim r as Rect                         : ! our bounds rect

  LoadDocument = true                   : ! assume we'll succeed

  ! only open if the document pointer is good.

  if dPtr <> NIL then
    HLock(dPtr^.pathName)               : ! form the file name
    opRec.pcount = 12                   : ! open the file
    path = gsosOutStringPtr(dPtr^.pathName^)
    opRec.pathName = @path^.theString
    opRec.requestAccess = 1
    opRec.resourceNumber = 0
    opRec.optionList = NIL
    OpenGS(opRec)

    if toolerror <> 0 then
      call FlagError(4, toolerror)
      LoadDocument = false
    else
      dPtr^.pictureHandle = NewHandle(opRec.dataEOF, MMStartUp, $8000, NIL)
      if toolerror <> 0 then
        call FlagError(2, toolerror)
        LoadDocument = false
      else
        rdRec.pcount = 4          : ! read the file
        rdRec.refNum = opRec.refNum
        rdRec.dataBuffer = dPtr^.pictureHandle^
        rdRec.requestCount = opRec.dataEOF
        ReadGS(rdRec)
        if toolerror <> 0 then
          call FlagError(4, toolerror)
          LoadDocument = false
          DisposeHandle(dPtr^.pictureHandle)
          dPtr^.pictureHandle = NIL
        else
          HUnlock(dPtr^.pictureHandle)       : ! let the handle move
          port = GetPort                     : ! force an update
          SetPort(dPtr^.wPtr)
          GetPortRect(r)
          InvalRect(r)
          SetPort(port)
```

```
          end if
        end if
        clRec.pcount = 1                          : ! close the file
        clRec.refNum = opRec.refNum
        CloseGS(clRec)
      end if
      HUnlock(dPtr^.pathName)
    end if
  end function
```

Listing 6-7: Subroutine to Load a File

Writing a File

Writing a file follows pretty much the same pattern as reading one, with one exception. The `OpenGS` call assumes that a file exists; it won't create one. Instead, if the file doesn't exist, you start out by creating one. Just to keep things interesting, creating a file will fail if the file already exists, so we have to start by deleting any file that already happens to be on the disk.

Here's the procedure we will use to save a file:

1. Use `DestroyGS` to delete any file that happens to exist.
2. Use `CreateGS` to create a new file. If `DestroyGS` failed for some reason (like a locked file) this call will also fail, and we'll bail out of the save subroutine.
3. Call `OpenGS` to open the file.
4. Call `WriteGS` to write the file.
5. Call `CloseGS` to close the file.

While this is a fairly simple way to handle saving the file, there is another strategy you might want to consider. If you are saving a change to an existing file, it's possible (unlikely, but possible) to get an error after deleting the original file, but before the new information is safely on a disk. One way to avoid this problem is to save the file first, using some temporary file name, then delete the original file, and finally rename the new one. There are two disadvantages to this scheme. The minor one is that it's a little harder to implement than the way I've outlined. Since we package all of our ideas in neat subroutines that can be moved from program to program, though, this is only a minor problem. No matter how hard it is, we only have to do it once – after that, we can just copy our old subroutine. The main problem is that you can easily fill a disk while you are saving the new file. In fact, if the file you are saving is more than half the size of the capacity of the disk, you can't save a change to a file at all, since two copies will exist on the disk right before the original is deleted. That may or may not be a problem, depending on the program you are writing. In any case, we'll stick to the simple method here.

Here are the declarations for `DestroyGS`, the GS/OS call to delete a disk file, along with its structure:

```
type destroyOSDCB
  pcount as integer
  pathName as gsosInStringPtr
end type

GSOS $02, $20 SUB DestroyGS ((destroyOSDCB))
```

Listing 6-8: `DestroyGS` and the `NameRecGS` Structure

This call just needs the name of the file to delete from the disk; you pass the same file name you passed for `OpenGS` in the last section. There are three possibilities. First, the file might not exist at all. This could happen if we are saving a new file for the first time, or if we are saving a file to a new location. Either way, the `DestroyGS` call will fail, and we simply ignore that fact. If the file already exists, the most likely possibility is that the `DestroyGS` call will delete the file, making room for the new copy we are about to save. The third possibility is that the file exists, but for some reason can't be deleted. The reason a file can't be deleted is usually because it is locked, but there might be a disk error of some sort. Either way, the `DestroyGS` call will fail. We can safely ignore this possibility, since the next call to create a file will also fail. In short, we just take a stab at deleting the file – whatever happens is OK!

The next call is a bit more complicated. The `CreateGS` call, shown in Listing 6-9 with its parameter record, is used to create a new file on disk.

```
type createOSDCB
  pcount as integer
  pathName as gsosInStringPtr
  access as integer
  fileType as integer
  auxType as long
  storageType as integer
  dataEOF as long
  resourceEOF as long
end type

GSOS $01, $20 SUB CreateGS ((createOSDCB))
```

Listing 6-9: `CreateGS` and the `CreateRecGS` Structure

We need to set up the first six parameters, through `storageType`, to create the output file. The last two parameters – `eof` and `resourceEOF` – are used to set aside space for a file. In almost all cases, it's better to let the operating system figure that out for itself, so that's what we will do.

Running through the parameters that we actually need to set, `pcount` should be set to 5. The `pathName` parameter is again a pointer to the name of the file to create; since it can be a full path name, we can safely pass the same path name we use to open or delete the file.

The `access` parameter tells GS/OS who should have access to a file. This flags word should be set to $C3, which tells GS/OS to allow deleting, renaming, reading and writing, and to make the file visible. I won't go over the various bit flags here, but if you want to disable any of these options, you can check out the complete documentation for the `CreateGS` call in Appendix A.

The `fileType` and `auxType` parameters are the file type and auxiliary file type you want to use for the file. File types and auxiliary file types are assigned by Syndicomm, Inc. In a lot of

cases, there will already be a file type for the file format you want to use. For example, if you are saving a picture, there are several predefined file formats you can pick from, and you would use the file type and auxiliary file type for that format. If you need to create a completely new type of file, you can apply for a file type assignment from Syndicomm.

The last of the parameters we'll use is `storageType`, which controls the kind of file we're creating. For the normal kinds of data files we will create in this course, the value should be 1. The other common options are 13, which tells GS/OS we want to create a new folder, and 5, which creates a file that has both a data fork and a resource fork.

After creating the file, we need to open it. The only difference between opening a file for output and opening one for input, like you did in the last section, is that the `requestAccess` parameter in the `openOSDCB` structure should be set to 2 to get write access, instead of 1, which gave us read access. The `pcount` parameter can also be set to 3, since we're just using the first 3 parameters, this time.

Writing the file is also very similar to reading one. The only different is the obvious one: we tell GS/OS where the bytes to write start, and how many bytes to write, rather than telling GS/OS where to put the bytes, and how many bytes to read. In fact, as you can see in Listing 6-10, the parameter block we pass for a `WriteGS` call is even the same as the parameter for a `ReadGS` call.

```
type readWriteOSDCB
  pcount as integer
  refNum as integer
  dataBuffer as ptr
  requestCount as long
  transferCount as long
  cachePriority as integer
end type

GSOS $13, $20 SUB WriteGS ((readWriteOSDCB)
```

Listing 6-10: `WriteGS` and its Structure

There is one touchy issue with writing a file that we didn't have to worry about when we read it. When an error occurs at this point, it is usually because the disk is full. After all, if the disk is locked or completely hosed, we probably would have had trouble with the `CreateGS` call. If the disk doesn't have enough room to write the entire file, GS/OS will write as much of the file as it can, then return both the number of bytes written (in `transferCount`) and an error code. The touchy point is what your program does then. For a program that is writing a text file, it might be best to leave the file there, just in case the poor user doesn't have another disk to put the file on. At least that way he won't loose the *entire* file. For a file that has a very delicate format with a lot of interdependencies, like a linked database, it might be better to delete the entire file, and force the user to either rush out any buy a new box of disks, or cut out some information and resave the file.

The last step is to close the file. That's done exactly the same way the file was closed after reading it.

Listing 6-11 pulls all of this together into a subroutine you can use in your programs. It assumes you have added error message 5, "File write error", to the list of error messages in your resource fork.

```
!
! SaveDocument
!
! Implement this to save the document file.
!
! Parameters:
!   dPtr - pointer to the document to load
!
sub SaveDocument(dPtr as documentPtr)
  dim path as gsosOutStringPtr          : ! GS/OS output string of pathname
  dim opRec as openOSDCB                : ! OpenGS record
  dim clRec as closeOSDCB               : ! CloseGS record
  dim crRec as createOSDCB              : ! CreateGS record
  dim dsRec as destroyOSDCB             : ! DestroyGS record
  dim wrRec as readWriteOSDCB           : ! WriteGS record

  ! If the document is good (dPtr isn't NIL), save it.

  if dPtr <> NIL then
    HLock(dPtr^.pathName)               : ! lock the path name
    dsRec.pcount = 1                    : ! destroy any old file
    path = gsosOutStringPtr(dPtr^.pathName^)
    dsRec.pathName = @path^.theString
    DestroyGS(dsRec)

    crRec.pcount = 5                    : ! create the new file
    crRec.pathName = @path^.theString
    crRec.access = $C3
    crRec.fileType = $C1
    crRec.auxType = $0000
    crRec.storageType = 1
    CreateGS(crRec)
    if toolerror <> 0 then
      call FlagError(5, toolerror)
    else
      opRec.pcount = 3                  : ! open the file
      opRec.pathName = @path^.theString
      opRec.requestAccess = 2
      OpenGS(opRec)
      if toolerror <> 0 then
        call FlagError(5, toolerror)
      else
        SysBeep
        wrRec.pcount = 4                : ! write the file
        wrRec.refNum = opRec.refNum
        HLock(dPtr^.pictureHandle)
        SysBeep
        wrRec.dataBuffer = dPtr^.pictureHandle^
        wrRec.requestCount = GetHandleSize(dPtr^.pictureHandle)
        SysBeep
        WriteGS(wrRec)
        SysBeep
        if toolerror <> 0 then
          call FlagError(5, toolerror)
        end if
        HUnlock(dPtr^.pictureHandle)
        SysBeep
        clRec.pcount = 1
```

```
          clRec.refNum = opRec.refNum
          CloseGS(clRec)
          SysBeep
        end if
      end if
      HUnlock(dPtr^.pathName)
    end if
  end sub
```

Listing 6-11: Subroutine to Save a File

Screen Dumps

There are several formats for pictures, but one of the simplest is a screen dump. This file format is literally a copy of the graphics screen with a picture on the screen. In this section, we'll look at this format for a file and learn about a QuickDraw II call that can quickly draw the picture. In the problem at the end of the section you will finally get a chance to put the file input and output routines to work to create a slide show program.

A screen dump file has a file type of $C1 with an auxiliary file type of $0000. There are three parts to the file: a bitmap of the picture itself, a series of color tables that tells the computer which of the possible 4096 colors to use when displaying the picture, and a set of flag values that give some very specific hardware related information about each line that is displayed on the screen. These last two parts of the file are pretty hard to understand right now, so we'll ignore them until we get to the lesson on QuickDraw II.

The picture comes first in the file. A picture is organized as a series of 200 lines, starting from the top of the screen and moving down. Each line contains 160 bytes, organized as two pixels per byte for 640 mode pictures, and four pixels per byte for 320 mode pictures. A pixel is one colored dot on the screen. If you know enough about binary math to understand how the bits are organized in a byte, this probably makes a lot of sense. After all, in 320 mode you have 16 distinct colors available, and you need four bits to represent 16 values. That gives you two pixels to the byte, or 160 bytes for an entire 320 pixel scan line. In 640 mode, you only get four colors per pixel, which means you need two bits per pixel, and you can stuff four pixels in a byte. If you don't know much about binary math, don't get too worried about all of this just yet. You don't have to know anything about the organization of bits or pixels to draw a picture, and we'll come back to all of this in gory detail when we talk about QuickDraw II.

After you load one of these picture files into memory, the first byte of the file you loaded is also the first byte of the picture. The fastest and easiest way to draw the picture is to use QuickDraw II's `PPToPort` call. `PPToPort` stands for Paint Pixels to Port, and that's just what it does. `PPToPort` takes the entire picture and draws it in the active port. In the process, it clips the picture so it doesn't draw outside of the window, and if you ask it to, `PPToPort` can even scale the picture, changing its shape to match the space that's available. It's all a bit messy, but then you can do quite a lot with `PPToPort`.

Here's a prototype call to `PPToPort`, with names for the various parameters:

```
PPToPort(srcLocInfoPtr, srcRect, destX, destY, transferMode)
```

The first parameter is the most complex; it's a structure that tells QuickDraw II a lot about the picture it is supposed to draw. The structure is defined like this:

```
type locInfo
   portSCB as integer
   ptrToPixelImage as ptr
   width as integer
   boundsRect as rect
end type
type locInfoPtr as pointer to locInfo
```

The first field is a bit mask that tells QuickDraw II, among other things, whether we are drawing a 640 mode picture or a 320 mode picture. QuickDraw II gets pretty upset if you try to draw a 320 mode picture to a 640 mode screen, or vice versa, but it also trusts you – so we'll lie. When you are drawing to a 640 mode screen, set `portSCB` to $80; for 320 mode pictures use a value of $00. Later in the course we'll cover this byte in more detail.

The next field is a pointer to the first byte of the picture. Set `ptrToPixelImage` to point to the first byte of the picture file.

The `width` field tells `PPToPort` how long each line is, in bytes. Our lines are 160 bytes long, so we set `width` to 160.

The last field is a rectangle that gives the size for the entire pixel image. For a screen dump, this would be a rectangle with the top left point at 0,0, and the bottom right point at either 320,200 or 640,200, depending on the screen resolution.

The next parameter to `PPToPort` is `srcRect`, which is the rectangle to draw into. The obvious choice is to set this rectangle to the same size we used in the structure describing the picture, and that's what you would normally do. In fact, you can just pass `boundsRect`, from the `LocInfo` structure. If this rectangle is half the size of the original, though, the `PPToPort` call will squeeze the picture down to fit in the new rectangle. You can stretch the picture by increasing the size of the rectangle.

The next two parameters, `destX` and `destY`, tell where in the port to draw the picture. We're letting `TaskMaster` handle scrolling for us, and `TaskMaster` will move the origin of our window around to handle scrolling, so all we have to do is pass 0,0. If you wanted to draw the picture in the middle of the window you could set these parameters to some other value.

Finally, you have to tell `PPToPort` which drawing mode to use. For now we'll stick with `modeCopy`, which is what you've been using all along.

Here's a short subroutine that pulls this information together. This subroutine is set up as an update subroutine, so you can use it as a drop-in replacement for the update subroutine you've used in Frame. All you have to do is make sure the document structure defines the name of the handle for the file you load as `pictureHandle`.

```
!
! DrawContents
!
! Draws the contents of the window
!
sub DrawContents
  dim dPtr as documentPtr
  dim info as LocInfo

  dPtr = FindDocument(GetPort)          : ! get pointer to the document
  if dPtr <> NIL then
    HLock(dPtr^.pictureHandle)
    info.portSCB = $00
    info.ptrToPixelImage = dPtr^.pictureHandle^
    info.width = 160
    info.boundsRect.h1 = 0
    info.boundsRect.v1 = 0
    info.boundsRect.h2 = 320
    info.boundsRect.v2 = 200
    PPToPort(@info, info.boundsRect, 0, 0, modeCopy)
    HUnlock(dPtr^.pictureHandle)
  end if
end sub
```

Listing 6-12: Update Subroutine for Drawing a Picture

Problem 6-4: Create a slide show program, starting with the version of Frame from Problem 6-2. You program should use GS/OS to load the picture file, and it should also let the user save the file to a new name. Since you can't create new pictures, your program should not have New as an option in the File menu. Finally, the program should use 320 mode graphics.

You can test your program with any screen dump format picture. If you don't have any others handy, you can load one of the pictures from the Pictures folder on the solution disks.

Be sure to test the About box with your program. You've switched screen resolutions, and that changes the size of your alerts. (The size is adjusted automatically by `AlertWindow`, but you should check the new size to make sure it looks nice.) Of course, you'll also have to change the size of the window in your `rWindParam1` resource. If you forget, your window will be just a tad too wide for the 320 mode screen!

Summary

This lesson showed how to use the Standard File Operations Tool Set to get file names for reading and writing files. You learned two ways to load and save these files: using GSoft BASIC's standard input and output functions, or calling the GS/OS disk operating system directly. Along the way, Frame has been updated to handle SFO. While they aren't a part of the Frame program, you also have two subroutines that can be dropped into a program to read or write files using GS/OS.

The last problem in this lesson is also our first really useful program. It's a slide show program that lets you load as many screen dump format pictures as you like, displaying the pictures in multiple windows. You can even create a copy of a picture by saving it to a new name using the Save As... command.

Tool calls used for the first time in this lesson:

```
CloseGS          CreateGS         DestroyGS        DisposeHandle
HLock            HUnlock          NewHandle        OpenGS
PPToPort         ReadGS           SetWTitle        SFGetFile2
SFPutFile2       WriteGS
```