Toolbox Programming in Pascal

by Mike Westerfield

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 Pascal. 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 though 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 comes 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. Each and 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 seven kinds of controls. Are there seven? 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 disk. 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 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 Pascal language. Learning the toolbox is a big job. It's not insurmountable, but it is a big job. Learning Pascal is also a big job. Put these two big jobs together and try to learn Pascal and the toolbox at the same time, and you more than double the chance you will fail. If you don't know Pascal, stop. Put this book down. Get a good beginner book that teaches you Pascal, and learn the language 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 Pascal.

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 a Pascal compiler.

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. With system disk 6.0, 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 03 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 03, 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.

If you have the minimum amount of memory, though, you're going to have to use the text shell to compile and execute most of the programs. You will be a lot better off if you can get 1.75M of memory. With 1.75M of memory, you can use the desktop development environment to compile and test your programs, and even the text shell will work faster.

You need to have at least two disk drives, one 3.5 inch floppy drive for the compiler 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 your Pascal compiler, 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 will also need a Pascal reference manual. Your compiler came with one, and that one will do fine.

If you already know Pascal, you can use pretty much any Pascal compiler for this course. The course was written with ORCA/Pascal, and all of the sample programs and solutions have been tested with that compiler. There will be some minor differences that you will have to cope with if you use some other compiler, but you can handle those if you are patient. Of course, you won't have to deal with differences if you use ORCA/Pascal, but I wouldn't want to seem like I'm plugging my own compiler too much!

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 magazine. A hard disk only costs a few hundred dollars, and it truly changes the character of your computer.

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 52 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 three 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 three 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.

Recently, Apple released System Disk 6.0, the latest operating system for the Apple IIGS. The toolbox reference manuals only cover the tool calls up through System Disk 5.0. While Apple doesn't intend to release a new volume of the toolbox reference manual to cover the latest changes, there is a book that does. It's called *Programmer's Reference for System 6.0*. If you want to use the new features of System 6.0, you should get this book, too.

All of the books I've talked about here, plus several more, 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 staring 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 or 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 *Apple IIGS Toolbox Reference*, volumes 1 to 3, weigh 12 pounds. There are 2584 pages, covering 1072 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 and 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. ORCA/Pascal has a built-in 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.

```
StartDesk(640);
```

I can still hear my mother's cheerful, chirping voice floating out the door from my idyllic youth. "You thing 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. In ORCA/Pascal, all of the tools started with `StartDesk` can be closed by `EndDesk`:

```
EndDesk;
```

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 GS 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 you 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 of 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 would look something like this:

```
done := false;
repeat
   if KeyPress then
      HandleKeyPress
   else if MouseDown then
      HandleMouseClick
   else if DAClosed then
      HandleDAClose
   else if MenuSelected then
      HandleMenu;
until done;
```

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 record, 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:

```
StartDesk(640);
done := false;
repeat
   if GetNextEvent($xxxx, myEvent) then
      HandleEvent;
until done;
EndDesk;
```

`HandleEvent` is some hypothetical procedure that does all of the hard stuff; in a moment, we'll replace this hypothetical procedure 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 Pascal, an event record looks like this:[1]

[1]If you actually look at the definition of an event record in the interface files, 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`.

```
      eventRecord = record
          eventWhat:      integer;
          eventMessage:   longint;
          eventWhen:      longint;
          eventWhere:     point;
          eventModifiers: integer;
          end;
```

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 it 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 `eventMessage` 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 `eventMessage` 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 `eventWhere` field is a point, giving 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 makin 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.

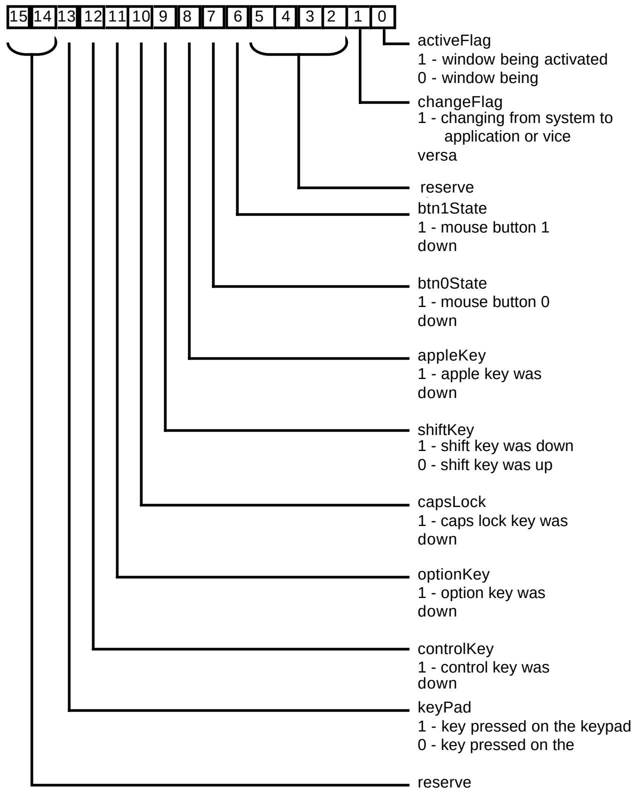


Figure 1-1: Modifier Flags

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

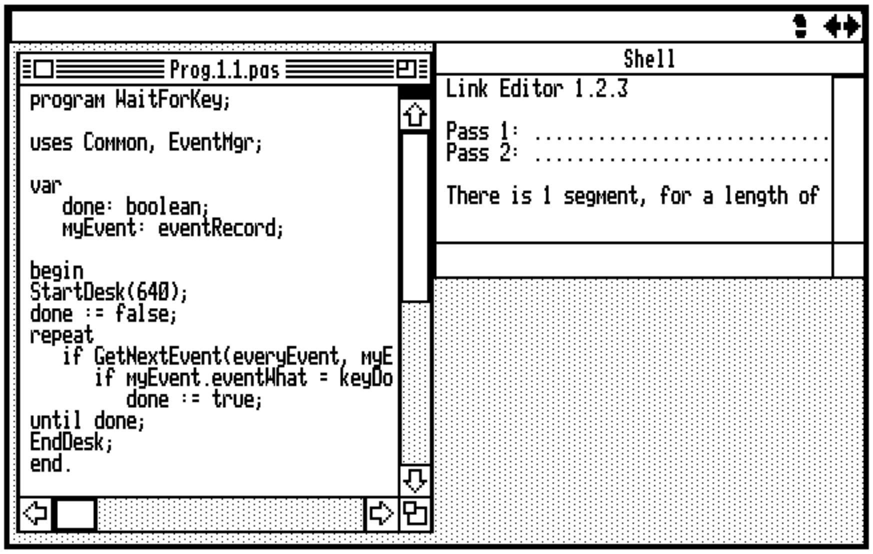
Well, we can Finally create a complete desktop program. This one doesn't to much – it just brings up the desktop and waits for us to press a key, but hey, it's a start. Even though the program doesn't do much, we've covered a lot of ground, and it's important to see some of this new knowledge put to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

```
program WaitForKey;

uses Common, EventMgr;

var
   done: boolean;                       {are we done, yet?}
   myEvent: eventRecord;                {event record}

begin
StartDesk(640);
done := false;
repeat
   if GetNextEvent(everyEvent, myEvent) then
      if myEvent.eventWhat = keyDownEvt then
         done := true;
until done;
EndDesk;
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.) Either way, take the time to compile and run the program. You can either run it form the desktop development environment or create an S16 program that you can launch from the Finder.

If you are using ORCA/Pascal's desktop development environment, you will see the menus on the menu bar disappear when you run this program, and two icons will appear on the menu bar near the right-hand edge. The blank menu bar belongs to your program. Since you haven't created a menu bar yet, there is nothing on it. The double-arrow icon is used to switch back and forth between your menu bar and the debugger's menu bar, although it doesn't work very well in the middle of an event loop, since your program and the debugger are fighting over who gets the events! The footprint lets you use the debugger's step feature without switching back to the debugger's menu bar. All of this is explained in detail in your reference manual, which gives lots of tips about debugging desktop programs. While I'm not going to repeat all of that information here, now would be a great time to browse through the section of the ORCA/Pascal reference manual that talks about debugging desktop programs.



```
program WaitForKey;

uses Common, EventMgr;

var
   done: boolean;
   myEvent: eventRecord;

begin
StartDesk(640);
done := false;
repeat
   if GetNextEvent(everyEvent, myE
      if myEvent.eventWhat = keyDo
         done := true;
until done;
EndDesk;
end.
```

```
Link Editor 1.2.3

Pass 1: ............................
Pass 2: ............................

There is 1 segment, for a length of
```

Figure 1-1: The Debug Menu Bar

Problem 1.1. Change the program so it stops when the mouse is released (not when it is pressed!).

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.pas. The source code for problem 1.2 will be in a file called Prob.1.2.pas, 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:

```
key := chr(ord(myEvent.eventMessage & $000000FF));
```

It may seem like overkill to make the effort to mask off the three undefined bytes, 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. (Ever wonder why each release of the system disk broke PaintWorks Gold, while other programs ran perfectly?)

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 in the company of the Paintworks Gold programmers, 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 shift 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. Buried in the ORCA/Pascal reference manual is the little-known but useful fact that when you start the tools with `StartDesk`, ORCA/Pascal does some technical magic behind the scenes to make the standard Pascal `write` and `writeln` procedures work with 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.

```
program KeyEvents (output);

uses Common, QuickDrawII, EventMgr;

var
   done: boolean;                        {are we done, yet?}
   myEvent: eventRecord;                 {event record}


   procedure HandleKeyPress;

   { handle a key down event                                   }

   var
      r: rect;                           {rectangle for area to erase}

   begin {HandleKeyPress}
   SetSolidPenPat(black);                {erase any old stuff}
   r.left := 0;  r.right := 100;
   r.top := 0;   r.bottom := 40;
   PaintRect(r);
   SetForeColor(white);                  {draw white text ...}
   SetBackColor(black);                  {...on a black background}
   MoveTo(10,10);                        {write the key at this location}
   writeln(chr(ord(myEvent.eventMessage & $000000FF)));
   writeln(myEvent.eventModifiers);
   end; {HandleKeyPress}
```

```
begin
StartDesk(640);
PenNormal;
done := false;
repeat
   if GetNextEvent(everyEvent, myEvent) then
      if myEvent.eventWhat = mouseUpEvt then
         done := true
      else if myEvent.eventWhat = keyDownEvt then
         HandleKeypress;
until done;
EndDesk;
end.
```

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 `writeln` 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 bits 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 the tool headers files. We'll see exactly how you can read the tool header files later, but for now, you can trust me about `black` and `white` being properly defined in the header files. 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. 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 record, 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.

```
program Sketch;

uses Common, QuickDrawII, EventMgr;

var
   done: boolean;                       {are we done, yet?}
   mouseIsDown: boolean;                {is the mouse down?}
   myEvent: eventRecord;                {event record}


   procedure BlackScreen;

   { Paint the entire screen black                              }

   var
      r: rect;                          {screen rectangle}

   begin {BlackScreen}
   r.top := 0;
   r.bottom := 200;
   r.left := 0;
   r.right := 640;
   SetSolidPenPat(black);
   PaintRect(r);
   end; {BlackScreen}


begin
StartDesk(640);                         {start the tools}
PenNormal;                              {set up normal pen parameters}
InitCursor;                             {start the arrow cursor}
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;                   {the mouse is not down}
```

```
repeat
                                        {handle the events}
   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;
                                        {draw or move the mouse}
   if mouseIsDown then
      LineTo(myEvent.eventWhere.h, myEvent.eventWhere.v)
   else
      MoveTo(myEvent.eventWhere.h, myEvent.eventWhere.v);
until done;
EndDesk;                                {shut down the tools}
end.
```

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 three 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 out to your local bookstore and pick up 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 are 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.

There is one special problem with the toolbox reference manuals, though. When Apple wrote the original toolbox reference manuals, they were also producing APW and APW C, but things had soured and they already knew there would be no APW Pascal, at least not right away. As a direct result, the toolbox reference manuals tell you exactly how to use the toolbox from the APW assembler or APW C, but they don't tell you anything at all about how to make a tool call from Pascal. As it turns out, once you know how to read the manuals, it really isn't hard to make tool calls from Pascal, but it sure helps to have some help when you first start to use them!

To see how the toolbox reference manuals work, let's look 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 from Pascal is to look at how the call is made from C; the two languages call subroutines in a very similar way. 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;
```

That's very, very close to the way the tool call is defined in Pascal. In fact, here's the definition of the `GetNextEvent` call from the ORCA/Pascal toolbox interface files:

```
function GetNextEvent (eventMask: integer; var theEvent: eventRecord):
   boolean;
```

In fact, the only major difference is the way the event record itself is passed. In C, we pass a pointer to the record, while in Pascal, we can pass the record itself.

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:

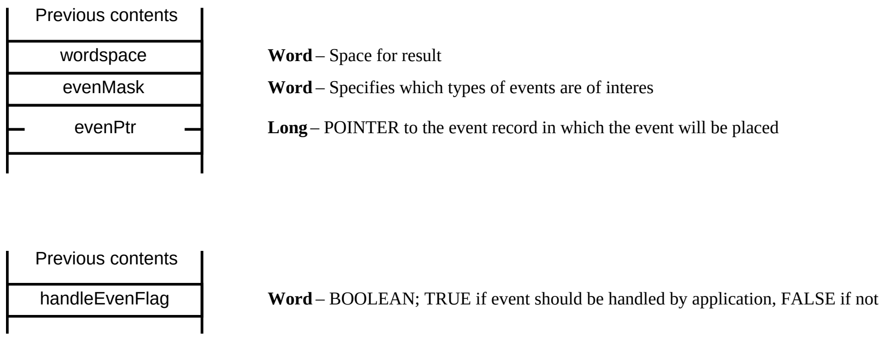

Previous contents
wordspace
evenMask
evenPtr
Word – Space for result
Word – Specifies which types of events are of interes
Long – POINTER to the event record in which the event will be placed
Previous contents
handleEvenFlag
Word – BOOLEAN; TRUE if event should be handled by application, FALSE if not

Figure 1-2: Stack Diagram for `GetNextEvent`

There's a wealth of information here, but it does take some practice and a knowledge of how Pascal passes parameters 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 is a function. The toolbox expects Pascal to put some space on the stack before making the tool call, and Pascal does just that. The wordspace area is filled in with the function return value, labeled `handleEventFlag` in the lower part of the stack diagram.

The parameters themselves are read from top to bottom. The first parameter for `GetNextEvent` is the event mask, and that's the one that appears on the stack first. The stack diagram only shows how big the parameter is, but in most cases, the type of the parameter is pretty obvious. For example, a character, integer, and boolean value all take up two bytes on the stack when Pascal passes them as a parameter (that tidbit can be found in your ORCA/Pascal reference manual), but it's pretty obvious that the event mask isn't a character or boolean.

ORCA/Pascal always passes arrays and records longer than four bytes by putting the address of the array or record on the stack. When the stack diagram shows a pointer of some sort, you almost always pass the actual record or array as a parameter in Pascal. The big exception is when the tool is returning something; you'll see that a lot with window pointers when we start opening windows on the desktop.

Besides these two methods, you can always fall back on Appendix A. For the tool calls that you use in this course, Appendix A shows the way the tool call is defined in the ORCA/Pascal header files.

Reading the Tool Interface Files

There are going to be times when the toolbox reference manuals just don't seem to tell you exactly what you want to know. For example, what if you just weren't sure if the event record was passed as a pointer or an actual record? The ultimate reference to settle questions like that once and for all is the actual interface files Pascal reads when you compile a program.

Do you remember the uses statements we've been putting at the top of our programs? Here's an example to jog your memory:

```
uses Common, QuickDrawII, EventMgr;
```

What this does is tell the compiler to go read the interfaces for three units. As it turns out, these three units happen to be the definitions for the tool calls from QuickDraw II and the Event Manager. The third unit, Common, is where all of the definitions used by more that one tool are stuffed. Things like `rect`, which is used by both QuickDraw II and the Window Manager (to name two of the many tools), are defined in Common.

The interface files that the compiler reads are in the libraries folder, and if you try to look at them you will find out that you can't. That's not because anyone is trying to keep secrets from you, though, it's simply because the compiler can read these predigested interface files faster than it can read a text file. In fact, the text versions of the interface files are also on your ORCA/Pascal disks in a folder called TOOL.INTERFACE. Whenever you aren't sure how a tool call is made, or what variable name is used for something like the kinds of events in the `eventWhat` field, you can always look at the header files themselves. In fact, I highly recommend printing the interface files and putting them in a notebook that you can use while you program.

The reason you don't need to look at these tool interface files constantly, though, is that they are created directly from the Apple IIGS toolbox reference manuals. Whenever possible, the actual names from the toolbox reference manuals are used in the Pascal interface files. When the toolbox reference manuals say that the name of the call to get an event is `GetNextEvent`, you can count on the Pascal header files spelling the name exactly that way. When the toolbox reference manual lists the event code for a null event as `nullEvt` (Table 7-6, page 7-50 of *Apple IIGS Toolbox Reference Manual: Volume 1*) you can count on a variable called `nullEvt` being defined in either the Event Manager header file or Common. (It happens to be in Common, since the same name is used for the same reason by another call in the Window Manager.)

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 ORCA/Pascal `StartDesk` and `EndDesk` procedures 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 ORCA/Pascal reference manual, and the toolbox header files from the ORCA/Pascal disks.

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.

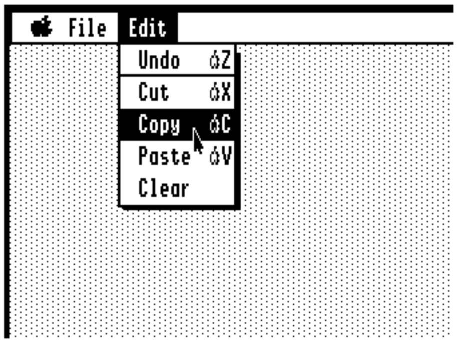


Figure 2-1: Typical Menu Bar

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 and 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 will be 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 Pascal 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.

```
new(s);                               {create the edit menu}
s^ := concat('>> Edit \N3',chr(return));
s^ := concat(s^,'--Undo\N250',chr(return));
s^ := concat(s^,'--Cut\N251',chr(return));
s^ := concat(s^,'--Copy\N252',chr(return));
s^ := concat(s^,'--Paste\N253',chr(return));
s^ := concat(s^,'--Clear\N254',chr(return));
s^ := concat(s^,'.',chr(return));
menuHand := NewMenu(s);
InsertMenu(menuHand,0);
```

This menu definition comes from the Frame.Pas program, from the ORCA/Pascal samples disk. I've modified the menu a bit, taking out some advanced features like keyboard equivalents and separators; we'll add those back 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 here with a series of string concatenation calls. The string itself is allocated dynamically using Pascal's `new` call. This is an important point about this menu string: once you call `NewMenu`, the string is used by the Menu Manager for work space and for the names of the menu and menu items. You have to make sure that each menu you create has its own, distinct string for the menu definition. In other words, you can't use the same string variable for two different menus. You also have to make sure the string doesn't move or vanish. Local variables in a Pascal procedure or function literally go away, with the memory quickly reused for other purposes, when you return from the subroutine, so you can't use local string variables for the menu strings. That leaves two practical possibilities: using globals variables, and, as you see here, using dynamically allocated memory that stays at a fixed location. You could use global string variables, but I recommend that you don't. Dynamically allocated memory doesn't take up room until the program starts to run, so you end up using less disk space, and the program that loads faster, when you use dynamically allocated memory.

The menu string itself is made up of a series of lines. In this program, I used return characters to end each line. The value for the return character is 13, and if you look at the top of the program, you will find a constant definition that declares return as 13. The Menu Manager also lets you use the character chr(0) to end a line, but you should use the same one for the entire menu. Because of the way Pascal handles strings internally, it's a little easier for us to use the return character to end a line, rather than chr(0).

Each menu has to have at least two lines, one for the menu itself, and one at the end of the list 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.

The last part of the menu name string is a set of control characters, in this case `'\N3'`. This 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 menu item 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 Pascal program.

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:

```
   menuHand := NewMenu(s);
```

While the call is fairly simple, we really should stop and look at the variables involved.

```
      menuHand: menuHandle;            {for 'handling' menus}
      s: textPtr;                      {for building menus}
```

The value returned by `NewMenu` is a menu handle, and the Menu Manager interface 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.

The other variable involved is the string itself, but it isn't just any old ordinary string. The string we pass to the Menu Manager is a pointer to a `textBlock`, a type defined in Common.Pas. The definitions for these types look like this:

```
   textBlock = packed array [1..300] of char;
   textPtr = ^textBlock;
```

These definitions bring up a number of nasty little points about strings in Pascal and the toolbox, the most important of which is that they come in two major flavors, the so-called p-strings (or Pascal strings) and c-strings. P-strings get their name from UCSD Pascal, which introduced a string type to the Pascal language that starts with a length byte and is followed by up to 255 additional characters. Despite the name, p-strings have absolutely nothing to do with standard Pascal. They are really a sort of wart on the language, introduced for no good reason by a Pascal implementation that became a model for other microcomputer Pascals. C-strings, on the other hand, are a sequence of characters ending with the null character, chr(0). That's pretty much the same thing as a standard Pascal string, with the only difference being that standard Pascal strings are fixed length. ORCA/Pascal combines standard Pascal strings and c-strings, treating chr(0) as a character that means "ignore the rest of the characters in this string."

The reason for bringing up this bit of computer language history is that most toolbox calls that use string parameters expect p-strings, but `NewMenu` is one of the exceptions. `NewMenu` expects a simple array of characters, with no leading length byte. One pretty good reason for making an exception from the general rule of using p-strings with `NewMenu` is that it isn't hard to build a menu string with more than 255 characters, which is all a p-string can handle. Standard Pascal strings, on the other hand, can have up to `maxint` characters, which is 32767 characters in ORCA/Pascal. True c-strings can be as long as available memory, but 32767 characters is more than enough for menu strings – the Apple IIGS 80 column text screen only holds 1920 characters, after all.

Taking a look at the definition of `textBlock`, you see that the string array for a menu string can have up to 300 characters. This is an artificial limit imposed by the ORCA/Pascal interface files; you can use any positive number you want. Longer strings let you define longer menus, but take up a lot more room if you don't use all of the space. Shorter strings take up less memory, but there is a bigger chance you will overflow the string. In fact, overflowing the string is a real possibility with a 300 character string. Our normal looking Edit menu uses 75 characters, and it isn't hard to imagine using slightly longer names, bumping the total up to, say, 100 characters. Our menu only has five menu items, too, and you've almost certainly seen menus with fifteen to twenty menu items. In short, very long menus can easily be too long for a `textBuffer` as defined by the ORCA/Pascal interface file. If you overflow this array, you will write on memory that does not belong to the array, causing all sorts of problems, so keep the size limit in mind when you create long menus. If your menu is too long, you have two viable options: change the ORCA/Pascal header files, or create a new type in your program and use type casting to convince the Pascal compiler you know what you are doing, and really want to pass a string with your new type to `NewMenu`.

Calling `NewMenu` creates a menu, but it doesn't do anything with the menu. 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 the handle of 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.
>32767	`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`, 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 `taskData` 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. Pulling these two numbers out of the longint `taskData` is a little tricky; here's one way to do it:

```
type
   long = record
      case boolean of
         true : (long: longint);
         false: (lsw,msw: integer);
      end;

...

menuID := long(myEvent.taskData).msw;
menuItemID := long(myEvent.taskData).lsw;
```

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, menuID);
```

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.      }
{                                                                }
{----------------------------------------------------------------}

program Quit;

uses Common, QuickDrawII, EventMgr, WindowMgr, ControlMgr, DeskMgr,
     DialogMgr, MenuMgr;

const
   return        = 13;                  {return key code}

   File_Quit     = 256;                 {Menu ID #s}

type
   long = record                        {for splitting 4 bytes to 2 bytes}
      case boolean of
         true : (long: longint);
         false: (lsw,msw: integer);
      end;
```

```
var
   done: boolean;                       {tells if the program should stop}
   myEvent: eventRecord;                {last event returned in event loop}
   menuNum,menuItemNum: integer;        {menu number & menu item number}
   where: integer;                      {where the mouse event occurred}
   wPtr: grafPortPtr;                   {window where event occurred}


   procedure InitMenus;

   { Initialize the menu bar.                                   }

   var
      height: integer;                  {height of the largest menu}
      menuHand: menuHandle;             {for 'handling' windows}
      s: textPtr;                       {for building menus}

   begin {InitMenus}
   new(s);                              {create the file menu}
   s^ := concat('>> File \N1',chr(return));
   s^ := concat(s^,'--Quit\N256',chr(return));
   s^ := concat(s^,'.',chr(return));
   menuHand := NewMenu(s);
   InsertMenu(menuHand,0);
   height := FixMenuBar;                {draw the completed menu bar}
   DrawMenuBar;
   end; {InitMenus}


   procedure HandleMenu;

   { Handle a menu selection.                                   }

   begin {HandleMenu}
   case menuItemNum of                  {go handle the menu}
      file_Quit:   done := true;
      otherwise:   ;
      end; {case}
   HiliteMenu(false, menuNum);          {unhighlight the menu}
   end; {HandleMenu}


begin {Quit}
StartDesk(640);
InitMenus;                              {set up the menu bar}
InitCursor;                             {show the cursor}
```

```
done := false;                          {main event loop}
repeat
   if GetNextEvent(everyEvent, myEvent) then
      if myEvent.eventWhat = mouseDownEvt then begin
         with myEvent.eventWhere do
            where := FindWindow(wPtr, h, v);
         if where = wInMenuBar then begin
            MenuSelect(myEvent, nil);
            menuNum := long(myEvent.taskData).msw;
            menuItemNum := long(myEvent.taskData).lsw;
            if menuItemNum <> 0 then
               HandleMenu;
            end; {if}
         end; {if}
until done;

EndDesk;
end. {Quit}
```

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.

If you decide to use PRIZM's debugger on this program be sure and 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 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.

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:

```
      s^ := concat(s^,'--Quit\N256*Qq',chr(return));
```

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:

```
      s^ := concat(s^,'--Quit\*QqN256',chr(return));
```

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 shift 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 unsifted 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/Pascal 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 they 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 (myEvent.eventModifiers & appleKey) <> 0 then
      {the command key (A.K.A. open-apple key) is down};
```

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.      }
{                                                               }
{---------------------------------------------------------------}

program Quit;

uses Common, QuickDrawII, EventMgr, WindowMgr, ControlMgr, DeskMgr,
     DialogMgr, MenuMgr;

const
   return        = 13;                  {return key code}

   File_Quit     = 256;                 {Menu ID #s}

type
   long = record                        {for splitting 4 bytes to 2 bytes}
      case boolean of
         true : (long: longint);
         false: (lsw,msw: integer);
      end;

var
   done: boolean;                       {tells if the program should stop}
   myEvent: eventRecord;                {last event returned in event loop}
   menuNum,menuItemNum: integer;        {menu number & menu item number}
   where: integer;                      {where the mouse event occurred}
   wPtr: grafPortPtr;                   {window where event occurred}


   procedure InitMenus;

   { Initialize the menu bar.                                   }

   var
      height: integer;                  {height of the largest menu}
      menuHand: menuHandle;             {for 'handling' windows}
      s: textPtr;                       {for building menus}
   begin {InitMenus}
   new(s);                             {create the file menu}
   s^ := concat('>> File \N1',chr(return));
   s^ := concat(s^,'--Quit\N256*Qq',chr(return));
   s^ := concat(s^,'.',chr(return));
   menuHand := NewMenu(s);
   InsertMenu(menuHand,0);
   height := FixMenuBar;               {draw the completed menu bar}
   DrawMenuBar;
   end; {InitMenus}


   procedure HandleMenu;

   { Handle a menu selection.                                  }

   begin {HandleMenu}
   case menuItemNum of                 {go handle the menu}
      file_Quit:   done := true;
      otherwise:   ;
      end; {case}
   HiliteMenu(false, menuNum);         {unhighlight the menu}
   end; {HandleMenu}


begin {Quit}
StartDesk(640);
InitMenus;                             {set up the menu bar}
InitCursor;                            {show the cursor}

done := false;                         {main event loop}
repeat
   if GetNextEvent(everyEvent, myEvent) then
      case myEvent.eventWhat of

         mouseDownEvt: begin
            with myEvent.eventWhere do
               where := FindWindow(wPtr, h, v);
            if where = wInMenuBar then begin
               MenuSelect(myEvent, nil);
               menuNum := long(myEvent.taskData).msw;
               menuItemNum := long(myEvent.taskData).lsw;
               if menuItemNum <> 0 then
                  HandleMenu;
               end; {if}
            end;

         keyDownEvt, autoKeyEvt: begin
            if (myEvent.eventModifiers & appleKey) <> 0 then begin
               MenuKey(myEvent, nil);
               menuNum := long(myEvent.taskData).msw;
               menuItemNum := long(myEvent.taskData).lsw;
               if menuItemNum <> 0 then
                  HandleMenu;
               end; {if}
            end;
         otherwise: {do nothing};
         end; {case}
until done;

EndDesk;
end. {Quit}
```

Listing 2-2: Quit Program with Keyboard Equivalents

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.

As it turns out, the shifted characters for 1 and 2 (! and @) don't work as key equivalents. In addition, 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.

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	Quit	Q
Edit	Undo	Z
Edit	Cut	X
Edit	Copy	C
Edit	Paste	V
Style	Plain Text	P
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. It is also very, very common to see P used as the keyboard equivalent for Print, even though this key is technically reserved for plain text.

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;                           {main event loop}
myEvent.taskMask := $001F7FFF;           {let task master do it all}
repeat
   event := TaskMaster(everyEvent, myEvent);
   case event of                         {handle the events we need to}
      wInSpecial,
      wInMenuBar: HandleMenu;
      otherwise: ;
      end; {case}
until done;
```

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.

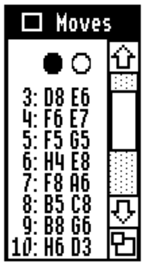
While it is called just like `GetNextEvent`, `TaskMaster` needs to get a little more information from us, and needs to be able to pass back more information to handle some complicated events we haven't run across yet. All of this extra information is passed in an extended version of the event record.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

```
eventRecord = record
    eventWhat:      integer;
    eventMessage:   longint;
    eventWhen:      longint;
    eventWhere:     point;
    eventModifiers: integer;
    {the following fields are used by TaskMaster}
    taskData:       longint;
    taskMask:       longint;
    lastClickTick:  longint;
    ClickCount:     integer;
    TaskData2:      longint;
    TaskData3:      longint;
    TaskData4:      longint;
    lastClickPt:    point;
    end;
```

In fact, if you look in the Pascal interface files, 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. For example, the Reversi game on the ORCA/Pascal samples disk has one window that lists the moves made so far in the game; you can see a copy of this window in Figure 2-2. The moves list is in two columns, with a picture of the appropriate type of piece at the top of each column. When you scroll the window, the moves move up, but the pieces at the top of the column don't. Well, as it happens, `TaskMaster` can do a fine job of scrolling a window for us, but in this case, I didn't want to scroll the entire window, only part of it – so I told `TaskMaster` to do a lot of things for me, but not to scroll the windows. That I handled for myself.



```
 ● ○
3: D8 E6
4: F6 E7
5: F5 G5
6: H4 E8
7: F8 A6
8: B5 C8
9: B8 G6
10: H6 D3
```

Figure 2-2: Reversi Moves Window

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, `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 `taskMask` 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.

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. For example, you have to set the `tmFindW` bit or `TaskMaster` will not need to even look at the value of `tmMenuSel`.

`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.

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 pallets in a later lesson, reversing the rainbow apple turns the entire apple character green. 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:

```
new(s);                              {create the apple menu}
s^ := concat('>>@\XN1',chr(return));
s^ := concat(s^,'--About...\N257',chr(return));
s^ := concat(s^,'.',chr(return));
menuHand := NewMenu(s);
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);                    {add desk accessories}
```

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 help 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 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 chr(18).
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. From Pascal, it's generally best to use N, instead.
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 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 Pascal, you must use the chr function to convert the numerical value for the character into a character the concat function can use in a string. As an example, the line

```
   s^ := concat(s^,'--Check Me!\N300C',chr(18),chr(return));
```

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.

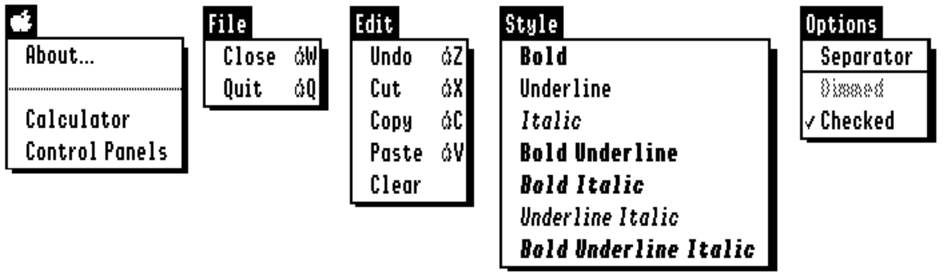


Figure 2-3: Menus Created in Problem 2.8

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);
```

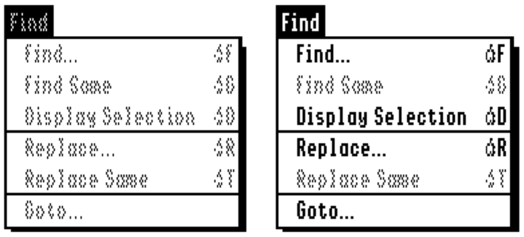


Figure 2-4: Dimmed Menu Compared to Normal Menu with Some Dimmed Items

Dimming a menu is a little more difficult. 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 read the current value for this set of flags and turn on bit 7. To enable the menu, do just the opposite, turning bit 7 off. There are two predefined masks in the interface files to make this a little easier to handle. Using these flags, here's the sequence of commands needed to enable and disable menu number 2:

```
SetMenuFlag(GetMenuFlag(2)|disableMenu, 2);     {disable menu 2}
SetMenuFlag(GetMenuFlag(2)&enableMenu, 2);      {enable menu 2}
```

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 Languages menu in the PRIZM desktop programming environment. There is a check mark beside the language for the front window. When you select a new language, the new one is checked, and the check mark is erased from the original choice.

Well, event 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 Quit menu 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.

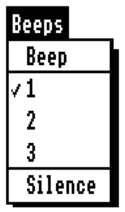


Figure 2-5: The Beeps Menu

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 Pallet" in a paint program. When the user picks this option, the paint pallet 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 Pallet" to "Show Pallet." The `SetMItem` call is used to change the name of the menu, like this:

```
SetMItem(ptr(ord4(@'--Show Pallet')+1), 263);
```

In this particular case, we're changing the text for menu item 263, which reads "Hide Pallet" 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 chr(0) or a return character, and the string has to be in a permanent, fixed location in memory. As it turns out, string constants in ORCA/Pascal happen to be in a fixed location in memory, and they also happen to be followed by a null character. The only minor hassle is that they also start with a length byte, so we need to add one to the address of the string to get a pointer to the first real character, which is the first of the dash characters. Skipping the tricks, you could do the same thing with

```
var
   ip: cStringPtr;

new(ip);
ip^ := '--Show Pallet';
SetMItem(ip, 263);
```

Well, that works. It's probably easier to see what's going on this way, but the original method, despite the tricks, is a lot easier to use in a real program once you understand what is happening.

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 Pallet" and "Show Pallet." This subroutine assumes you have defined a global variable called `paletteString`; this is the string number for the currently visible pallet string, which will be 0 for the original string, "Hide Pallet," and 1 for the alternate string, "Show Pallet." Somewhere in your program's initialization section, this variable should be set to 0. Of course, the reason the variable is a global 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 global constant called `options_Palette`, which is the menu item number for this menu item.

```
procedure ChangePalette;

{ Change the menu item string for the palette choice           }

begin {ChangePalette}
if paletteString = 0 then begin
   paletteString := 1;
   SetMItem(ptr(ord4(@'--Show Pallet')+1), options_Palette);
   end {if}
else begin
   paletteString := 0;
   SetMItem(ptr(ord4(@'--Hide Pallet')+1), options_Palette);
   end; {else}
end; {ChangePalette}
```

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.

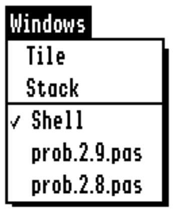


Figure 2-6: PRIZM's Windows Menu

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.

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        GetMenuFlag
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.

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 $8006 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 $8006, 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 := $8006;
   resource1.ID := 1;
   resource1.value := 'Quit';

   resource2.type := $8006;
   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:

```
   $0000: 0B000680 01000551 7569740F 00068002  '       Quit     '
   $0010: 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 $8006 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, $8006 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 Pascal. 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 Pascal compiler, 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 Pascal 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 Pascal.

Rez is a whole new language, though, so there's a lot to learn. Unfortunately, it's also patterned after the C language, so it's a little odd looking when you are used to Pascal programs.

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. As I write this, the two best places are the ORCA/M reference manual and the APW Tools and Interfaces package, but we also plan to include Rez with the next release of ORCA/Pascal.

That also brings up one other point. If you have ORCA/Pascal, but don't have any of these other packages, you need to get a copy of Rez to use with the rest of this course. There's a copy on the program disk that comes with the course. It will fit on your hard disk just fine, but if you are using floppy disk drives, you will need to use the text development system from here out – there just isn't room for PRIZM and Rez on the same program disk.

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

Looking at this source file, you can see that it doesn't look much like a Pascal program at all. You will have to learn a new syntax for writing resource description files, as Rez source files are known. It's going to look pretty odd at first, but once you know a few simple rules, writing a resource description file really isn't that hard. It's not like learning C, where you have to learn a whole new set of data statements and expression operators; all you need to learn to do is to create resource data statements.

The first line of the resource description file is

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

This line does the same thing in a resource description file that a `uses` statement does in a Pascal program. There's a file called types.rez, located at prefix 2/RInclude (13:RInclude with the 2.0 shell), that 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 interface file for Pascal defines all of the tool calls and data structures in the Menu Manager. There are some differences, of course. The biggest one is that types.rez is an editable source file, so you can look at it with the command

```
edit 2/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.

The /* and */ characters mark the start and end of a comment, just as (* and *) can be used for comments in Pascal. The rules for forming comments with the /* and */ characters are exactly the same as the rules used in Pascal to form comments with the (* and *) characters.

Each resource starts with the reserved word `resource`, sort of like each Pascal procedure starts with the reserved word `procedure`, or each Pascal record starts with the reserved word `record`. Right after this is the name of the resource we want to create; this matches one of the type declarations in types.h. 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 bar resource IDs. Arrays are variable length groups of values in a resource description file, not fixed length structures like they are in Pascal. 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. In fact, as you can start to see, the brackets in a resource description file are used for the same thing as `begin` and `end` in Pascal.

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. Unlike Pascal, you can use a character or an integer value here; a single character is equivalent in every way to a number. Also unlike Pascal, a character and a string with one character are two very different things – a string is coded with double quote marks, like the keyboard equivalents, but a character is coded with single quote marks. For example, if you want to use a '$' character to the left of the menu item instead of a check mark, you could use the line

```
      '$',                                  /* check character */
```

for the check character. For a true check mark, you would use the number 18.

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, and 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 Pascal 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.

In a resource description file, constants are defined with something called a preprocessor directive. It's actually a character substitution of one character string for another, but for our purposes, the effect is exactly like constants defined in Pascal. 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 Pascal or Shell, like you are 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.

Once the file is entered and saved to disk, it's time to compile it. If you are using PRIZM, saving the file to disk is a more important step than it is when you use Pascal. That's because the Rez compiler doesn't use the FastFile system, so the compiler doesn't see changes you've made in the editor; it only sees the new information after it has been saved to disk.

You compile the file the same way you would compile a Pascal program. When you give the file a keep name, use the same name that you will be using for the executable program. For example, if the program is called Frame.pas, and is compiled under PRIZM, the executable file name (called the keep name) is Frame. In this case, you'd want to use a file name of Frame.rez for the resource description file, so the resource fork created by Rez would also be saved in the executable file Frame.

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. Once you finish, use the CATALOG command from the text shell or from PRIZM's shell window. You should see a + beside the file type for Frame. 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 pay back 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:

```
      menuBarHand := NewMenuBar2(refIsResource, menuID, nil);
      SetSysBar(menuBarHand);
      SetMenuBar(nil);
```

The first creates the menu bar itself, using a Menu Manager call from the *Apple IIGS Toolbox Reference: Volume 3*. The first parameter, `refIsResource`, 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 procedure like `InitMenu`, 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 procedure 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.

```
{----------------------------------------------------------------}
{                                                                }
{  Frame                                                         }
{                                                                }
{  This program implements all of the support required for desk }
{  accessories.                                                  }
{                                                                }
{----------------------------------------------------------------}

program Frame;

uses Common, QuickDrawII, EventMgr, WindowMgr, ControlMgr, DeskMgr,
     DialogMgr, MenuMgr;

const
   return        = 13;                  {return key code}

   appleMenu     = 1;                   {Menu ID #s (also resource ID #s)}
   fileMenu      = 2;
   editMenu      = 3;

   editUndo      = 250;
   editCut       = 251;
   editCopy      = 252;
   editPaste     = 253;
   editClear     = 254;
   fileClose     = 255;
   fileQuit      = 256;
   appleAbout    = 257;
```

```
type
   long = record                        {for splitting 4 bytes to 2 bytes}
      case boolean of
         true : (long: longint);
         false: (lsw,msw: integer);
      end;

var
   done: boolean;                       {tells if the program should stop}
   event: integer;                      {event #; returned by GetNextEvent}
   myEvent: eventRecord;                {last event returned in event loop}


   procedure InitMenus;

   { Initialize the menu bar.                                   }

   const
      menuID = 1;                       {menu bar resource ID}

   var
      height: integer;                  {height of the largest menu}
      menuBarHand: menuBarHandle;       {for 'handling' the menu bar}

   begin {InitMenus}
                                        {create the menu bar}
   menuBarHand := NewMenuBar2(refIsResource, menuID, nil);
   SetSysBar(menuBarHand);
   SetMenuBar(nil);
   FixAppleMenu(1);                     {add desk accessories}
   height := FixMenuBar;                {draw the completed menu bar}
   DrawMenuBar;
   end; {InitMenus}


   procedure HandleMenu;

   { Handle a menu selection.                                   }

   var
      menuNum, menuItemNum: integer;    {menu number & menu item number}

   begin {HandleMenu}
                                        {separate the menu and item numbers}
   menuNum := long(myEvent.taskData).msw;
   menuItemNum := long(myEvent.taskData).lsw;
   case menuItemNum of                  {go handle the menu}
      appleAbout:  ;

      fileClose:   ;
      fileQuit:    done := true;

      editUndo:    ;
      editCut:     ;
      editCopy:    ;
      editPaste:   ;
      editClear:   ;
      otherwise:   ;
      end; {case}
   HiliteMenu(false, menuNum);          {unhighlight the menu}
   end; {HandleMenu}

begin {Frame}
StartDesk(640);
InitMenus;                              {set up the menu bar}
InitCursor;                             {show the cursor}

done := false;                          {main event loop}
myEvent.taskMask := $001F7FFF;          {let task master do it all}
repeat
   event := TaskMaster(everyEvent, myEvent);
   case event of                        {handle the events we need to}
      wInSpecial,
      wInMenuBar: HandleMenu;

      otherwise: ;
      end; {case}
until done;

EndDesk;
end. {Frame}
```

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

Problem 3-2: Type in this program, using the file name Frame.pas, or load it from the disk. Compile and link the program, saving the executable file to Frame, the same file that you save the resource fork to in problem 3-1.

This program should run correctly, now. You can even change the file type to S16 and run it from the Finder.

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 stared, there will still be one when the resource compiler finishes, and it's contents will not be changed.

When the linker creates an executable file it erases the existing data fork of the output file and creates a new data fork. The executable program is contained entirely in the data fork. If the file had a resource fork when the linker started, it still has a resource fork, with the contents unchanged, when the linker finishes.

The great thing about the way the resource compiler and linker work is that you don't have to recompile either the program source file or the resource description file each time the program is changed. If you change the resource fork, say to adjust a menu name, but don't change the program's Pascal source file, you need to recompile the resource description file with the resource compiler, but you do not have to recompile or relink the Pascal program. If the Pascal program is changed, but the resource description file stays the same, you need to recompile and relink the program, but you don't have to use the resource compiler.

Using a Script to Compile

Up until now, creating a program was pretty easy. For most programs, you just pulled down the Run menu and selected the Compile to Memory command. Now that your programs have at least two source files, things are getting a little more complicated. In larger desktop programs, you may want to use units to split the Pascal part of your program up into manageable sized chunks, making things even more complicated. Shell scripts give you a way to put all of the work back into a single, simple command.

So far, you've probably used menu based commands to create your programs. The ORCA environment gives you another way to create programs, too. Using the shell, you can compile, link, and even run a program using text commands, just like in the old days before the desktop interface became popular. While there are a lot of advantages to the desktop interface, there is one characteristic of text interfaces that has kept them popular with programmers to this day, and promises to keep the text interface alive for the foreseeable future. While typing the commands to compile a program is probably harder than using the menu commands you are used to, these commands can be collected into something called a script file. The script file is a shell program that you can create to compile your programs. Once you have created a script file, you just need to type the name of the script itself in the shell window to compile your program.

Let's look at a basic script to compile a program with two source files to see how this works. This script will compile the Pascal source file Frame.Pas and the resource description file Frame.Rez, creating an executable program called Frame. That's the same thing you just did by hand.

```
unset exit
set compile false
set rez false

newer frame.a frame.pas
if {status} != 0
   set compile true
end

newer frame frame.rez
if {Status} != 0
   set rez true
end

set exit on
if {rez} == true
   compile frame.rez keep=frame
end
if {compile} == true
   cmpl frame.pas keep=frame
end
```

Listing 3-4: A Simple Script

Let's start by stepping through this script file to see what it does. The first line, "unset exit", turns off a shell variable so the script will keep running when we use the newer command a little later on. The reason for this step is pretty technical; it's tied up in the way shell scripts handle errors and the way utilities return error codes and results. For creating your own programming scripts, just remember to use this command before any newer commands, and then use "set exit on" before you start the actual compiles, so the script will stop if it finds a problem.

Next you see two statements that are setting shell variables to false. The set command is followed by the name of a shell variable, then by the value. All shell variables are strings. The compile and rez shell variables are being used to keep track of whether we need to compile the resource description file or the Pascal source file.

The next step is to check to see which files actually need to be compiled. This script won't compile the Pascal source file unless it has changed since the last time you compiled the program; compiling the program creates the frame.a file. If the source file Frame.Pas is newer than the output from the compiler, Frame.a, the newer command sets the `{Status}` shell variable to 1; otherwise it sets `{Status}` to 0. If an error occurs – such as the script not finding Frame.a at all – newer returns some other value other than 0, usually -1. Our script checks to see if newer returned 0, and if not, sets the value of the `{Compile}` shell variable to true so we remember to compile the program later. The script then uses the same idea to check the resource description file, looking to see if it is newer than the executable program, and setting the shell variable `{Rez}` to true if we need to recompile the resource description file.

Finally, the script checks the values of the shell variables, compiling the resource description file and doing a compile and link of the Pascal source file.

This script is set up to use `newer` to check all of the dates on the files before actually compiling any of the programs. This is an important point. Once you start compiling programs, especially the Pascal part of the program, you will also start changing the modification dates used by the `newer` command, and that could cause the script to skip compiling a file that should be compiled. It's important to know what will be compiled before you start changing the time stamps!

The `newer` command used in this script is built into the ORCA shell from versions 2.0 on. If you are using an earlier version of the shell you don't have this call. The `MoreRecent` utility works just like the `newer` command, it just has a different name. If you need help installing the `MoreRecent` utility, see Appendix D. The files are in the Lesson.3 folder on the solution disks.

All of the programs from here to the end of this course are organized just like this one, with a Pascal source file and a resource description file. You could create a new script for each program by changing the word "frame" to the new program name everywhere in this script, but we can put the script language to work to help out, creating a new shell variable called `{prog}`. To create a script for a new program, just change the value of `{prog}` to the new program name. There's one other change in this script, the addition of `{parameters}` to the `cmpl` command, that we'll talk about in a moment.

```
set prog Frame
unset exit
set compile false
set rez false

newer {prog}.a {prog}.pas
if {status} != 0
   set compile true
end

newer {prog} {prog}.rez
if {Status} != 0
   set rez true
end

set exit on
if {rez} == true
   compile {prog}.rez keep={prog}
end
if {compile} == true
   cmpl {parameters} {prog}.pas keep={prog}
end
```

Listing 3-5: A Better Script

Now that we know what the script does, and how to easily change the script to work on any of the programs in this course, let's take a look at how it is used. To compile your program from PRIZM, move to the shell window and type the name of the script. This will compile and link the program. From the text shell, you don't have to move anywhere; just type the name of the script. To run the program, wait until the script finishes and type the name of the program itself.

One problem with this is that you can't use PRIZM's debugger from the shell window. You can use the "Set Execute Options" menu command to tell the debugger to kick in in step or trace mode, and you can use the Execute menu command to run the program instead of typing the program name from the shell window. These steps cause the debugger to kick in the way you are used to, but there's a catch: the compiler didn't create debug code! That's why we added the `{parameters}` shell variable where we did in the script. You can put the flags you want the compiler to use right after the name of the script in the shell window, and the flags will be used to compile the program. For example, if the script is called Frame.Make, you can type

```
frame.make +d
```

and the +d flag will be used on the `cmpl` command, causing the compiler to generate debug code. As an alternative, you could just put the +d command on the `cmpl` command permanently, removing the flag once the program is finished. If you are using the text shell, using +d will set things up so ORCA/Debugger can debug the program, after which running the `DebugBreak` utility will cause the debugger to kick in when your program starts to execute.

This section has been a pretty quick introduction to scripts, and if you've never seen them before it's understandable if you are a bit confused. Seeing this many new ideas in so short a time is enough to confuse anyone for a while! There are two important things you should take with you from this section. The first is the canned script we developed, which really will make your life easier as you write programs with resource forks. The second is the idea behind using scripts to automate complicated things you do often when you write programs. The ORCA/Pascal reference manual has detailed information on the shell, which is a whole language designed to make your life as a programmer a little easier. There's a lot to learn in the shell scripting language, but if you program a lot, or write large programs, it can be worth while to take the time to learn how to put the power of shell scripts to work for you.

Problem 3-3: Type in the script from this section and use it to compile the Frame program. Try making changes to the Pascal and resource files separately to see how the script only rebuilds what you need rebuilt.

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.

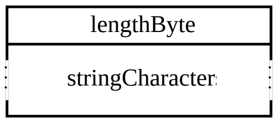
Toolbox Resource Descriptions

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:

`rPString` $8006

Figure E-27 defines the layout of resource type `rPString` ($8006). Resources of this type contain Pascal strings.

■ **Figure E-27** Pascal string, type `rPString` ($8006)



`lengthByte` Number of bytes of data stored in`stringCharacters` array.
`stringCharacters`
Array of `lengthByte` characters.

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 and in Pascal, 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

This is the actual type declaration the resource compiler uses when it compiles your resource description file. It's sort of like a record type in Pascal, 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 longint 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 the ORCA/M 2.0 reference manual or in the APW Tools and Interfaces package.

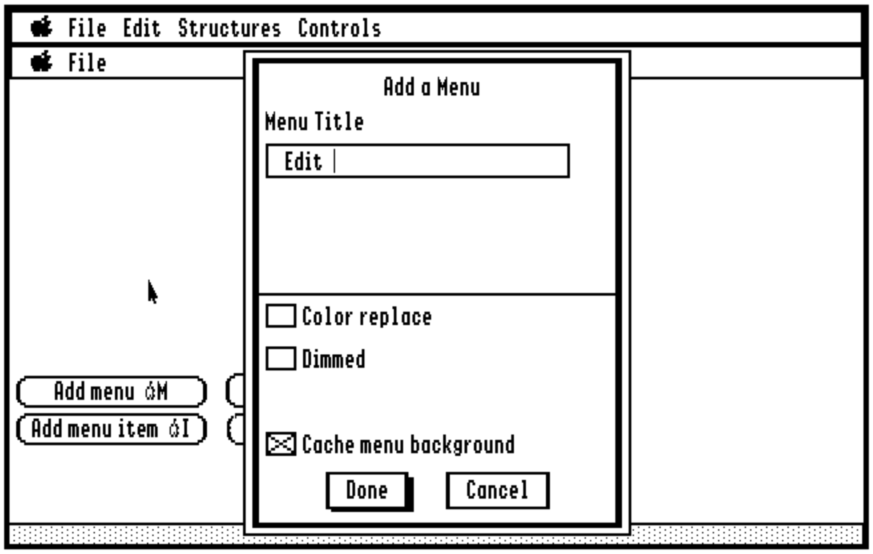
Resource Tools

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.



Menu Title

Edit

☐ Color replace

☐ Dimmed

☒ Cache menu background

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.

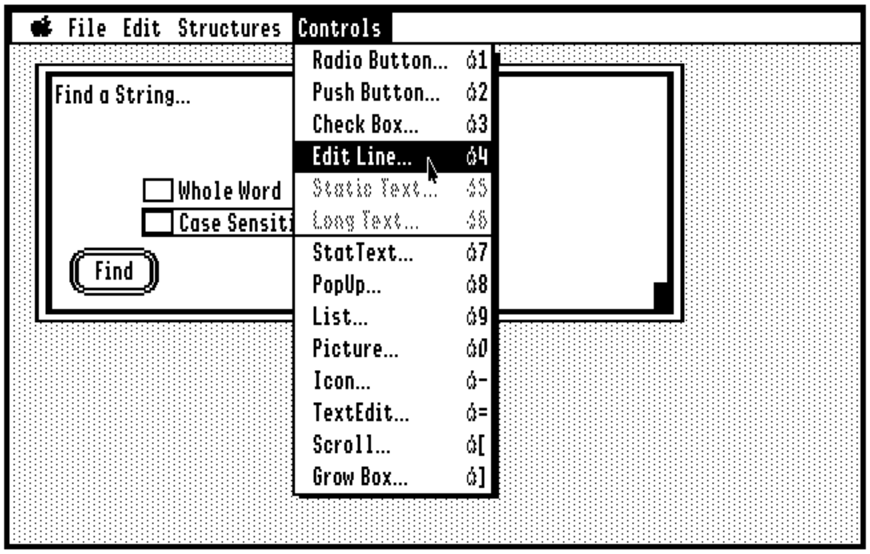


Figure 3-2: Design Master Creates a Dialog

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, but in fact, you can also create a Pascal program that doesn't use resources. In short, 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 it's 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 either resources or Pascal source code to create the same thing in your program.

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.Pas, 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

```
writeln('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:

```
write('Do you really want to reformat your hard disk? (Y or N) ');
readln(response);
```

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.

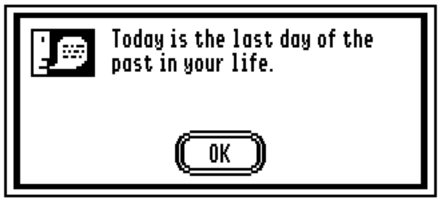
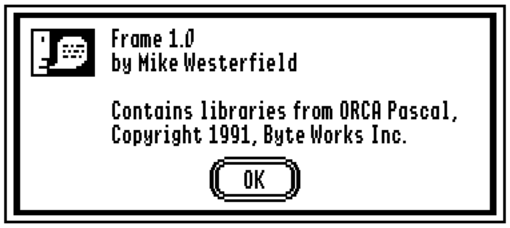

Today is the last day of the past in your life.

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 form the apple menu, and will stay on the screen until the user clicks on the OK button.


Frame 1.0
by Mike Westerfield

Contains libraries from ORCA Pascal,
Copyright 1991, Byte Works Inc.

Figure 4-2: Our About Box

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

```
procedure DoAbout;

{ Draw our about box                                        }

const
   alertID = 1;                  {alert string resource ID}

var
   button: integer;              {button pushed}

begin {DoAbout}
button := AlertWindow($0005, nil, alertID);
end; {DoAbout}
```

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, of which only two are defined. The first 13 bits are all reserved, and must be set to zero. 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).

The value of $0005 we are passing, then, tells the Window Manager that `alertStrRef` is a resource ID, and that `subStrPtr` is an array of pointers to p-strings.

`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 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 to our resource description file.

```
resource rAlertString (1) {
   "43/"
   "Frame 1.0\n"
   "by Mike Westerfield\n"
   "\n"
   "Contains libraries from ORCA Pascal,\n"
   "Copyright 1991, Byte Works Inc."
   "/^#0\$00";
   };
```

Listing 4-2: Alert String for an About Box

Problem 4.1: Add the `DoAbout` procedure 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 Pascal 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!

```
{---------------------------------------------------------------}
{                                                               }
{  Frame                                                        }
{                                                               }
{  This is a frame for other programs.  It contains a basic     }
{  event loop, a menu bar, an about box, and supports NDAs.     }
{                                                               }
{---------------------------------------------------------------}

program Frame;

uses Common, QuickDrawII, EventMgr, WindowMgr, ControlMgr, DeskMgr,
     DialogMgr, MenuMgr;

const
   return        = 13;                  {return key code}

   appleMenu     = 1;                   {Menu ID #s (also resource ID #s)}
   fileMenu      = 2;
   editMenu      = 3;

   editUndo      = 250;
   editCut       = 251;
   editCopy      = 252;
   editPaste     = 253;
   editClear     = 254;
   fileClose     = 255;
   fileQuit      = 256;
   appleAbout    = 257;

type
   long = record                        {for splitting 4 bytes to 2 bytes}
      case boolean of
         true : (long: longint);
         false: (lsw,msw: integer);
      end;

var
   done: boolean;                       {tells if the program should stop}
   event: integer;                      {event #; returned by GetNextEvent}
   myEvent: eventRecord;                {last event returned in event loop}


   procedure InitMenus;

   { Initialize the menu bar.                                   }

   const
      menuID = 1;                       {menu bar resource ID}

   var
      height: integer;                  {height of the largest menu}
      menuBarHand: menuBarHandle;       {for 'handling' the menu bar}
```

```
begin {InitMenus}
                                    {create the menu bar}
menuBarHand := NewMenuBar2(refIsResource, menuID, nil);
SetSysBar(menuBarHand);
SetMenuBar(nil);
FixAppleMenu(1);                    {add desk accessories}
height := FixMenuBar;               {draw the completed menu bar}
DrawMenuBar;
end; {InitMenus}


procedure HandleMenu;

{ Handle a menu selection.                                   }

var
   menuNum, menuItemNum: integer;   {menu number & menu item number}


   procedure DoAbout;

   { Draw our about box                                        }

   const
      alertID = 1;                  {alert string resource ID}

   var
      button: integer;              {button pushed}

   begin {DoAbout}
   button := AlertWindow($0005, nil, alertID);
   end; {DoAbout}


begin {HandleMenu}
                                    {separate the menu and item numbers}
menuNum := long(myEvent.taskData).msw;
menuItemNum := long(myEvent.taskData).lsw;
case menuItemNum of                 {go handle the menu}
   appleAbout:  DoAbout;

   fileClose:   ;
   fileQuit:    done := true;

   editUndo:    ;
   editCut:     ;
   editCopy:    ;
   editPaste:   ;
   editClear:   ;

   otherwise:   ;
   end; {case}
HiliteMenu(false, menuNum);         {unhighlight the menu}
end; {HandleMenu}
```

```
begin {Frame}
StartDesk(640);
InitMenus;                              {set up the menu bar}
InitCursor;                             {show the cursor}

done := false;                          {main event loop}
myEvent.taskMask := $001F7FFF;          {let task master do it all}
repeat
   event := TaskMaster(everyEvent, myEvent);
   case event of                        {handle the events we need to}
      wInSpecial,
      wInMenuBar: HandleMenu;

      otherwise: ;
      end; {case}
until done;

EndDesk;
end. {Frame}
```

Listing 4-3A: The Frame.Pas File

```
/*-------------------------------------------------------------*/
/*                                                             */
/*  Resources for Frame                                        */
/*                                                             */
/*-------------------------------------------------------------*/

#include "types.rez"

/*- Constants -------------------------------------------------*/

#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

/*- Menu Bar --------------------------------------------------*/

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\n"
   "by Mike Westerfield\n"
   "\n"
   "Contains libraries from ORCA Pascal,\n"
   "Copyright 1991, 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 is 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.

Character	Use
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 (chr(13)) 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 chr(0) character. In a resource description file, you can put this character in with $"00", as we did in our about box resource. If you are creating the string from a Pascal program, the Pascal language is already putting a null character at the end of and standard Pascal string or string constant. If you are using a p-string, though, be sure and put the chr(0) in explicitly.

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 called escape characters. The \n sequence gets turned into the proper character to force a new line, so we can use \n in our strings to put in a manual line feed. (`AlertWindow` is smart enough to break a long line up, but sometimes it's nice to break up lines for formatting.) Putting a hexadecimal value right after the \ character stuffs that value into the 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:

```
procedure SampleAlert (size, icon: integer);

{ Draw an alert window                                  }
{                                                       }
{ Parameters:                                           }
{    size - size of the alert (1..9)                    }
{    icon - alert icon number (2..6)                    }

var
   alertString: packed array[1..200] of char; {alert string}
   button: integer;              {button pushed}

begin {SampleAlert}
alertString := concat(chr(ord('0')+size),
   chr(ord('0')+icon),
   '/Sample alert of size ',
   chr(ord('0')+size),
   ', with icon ',
   chr(ord('0')+icon),
   './^#0'
   );
button := AlertWindow($0001, nil, @alertString);
end; {SampleAlert}
```

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 during a program.

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 p-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.

```
procedure Error (err: integer);

{ Flag an error                                                  }
{                                                                }
{ Parameters:                                                    }
{    err - error number                                          }

const
   alertID = 21;                         {alert resource ID}
   base = 2000;                          {base resource ID}

var
   substArray: pStringPtr;               {substitution "array"}
   button: integer;                      {button pushed}

begin {Error}
substArray := GetPString(base+err);
InitCursor;
button := AlertWindow($0005, @substArray, ord4(alertID));
FreePString(base+err);
end; {Error}
```

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 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 `rPString` resources and load them with some simple Resource Manager calls. In good structured programming style, this mechanism is packaged in a couple of subroutines, `GetPString` and `FreePString`. Here they are, along with a few error strings from Quick Click Draw:

```
const
   rPString = $8006;                     {resource type for p-strings}
```

```
function GetPString (resourceID: integer): pStringPtr;

{ 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 FreePString 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.         }

var
   hndl: handle;                         {resource handle}

begin {GetPString}
hndl := LoadResource(rPString, resourceID);
if ToolError <> 0 then
   GetPString := nil
else begin
   HLock(hndl);
   GetPString := pStringPtr(hndl^);
   end; {else}
end; {GetPString}


procedure FreePString (resourceID: integer);

{ Free a resource string                                        }
{                                                               }
{ Parameters:                                                   }
{    resourceID - resource ID of the rPString to free           }

begin {FreePString}
ReleaseResource(-3, rPString, resourceID);
end; {FreePString}
```

Listing 4-6A: `GetPString` and `FreePString`

```
resource rPString (2001) {"Out of memory"};
resource rPString (2002) {"Could not form the full path name"};
resource rPString (2003) {"A document using this file name is already"
                          " open"};
resource rPString (2004) {"All of the objects are locked"};
resource rPString (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. `FreePString` 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.

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
```

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 the visible screen.

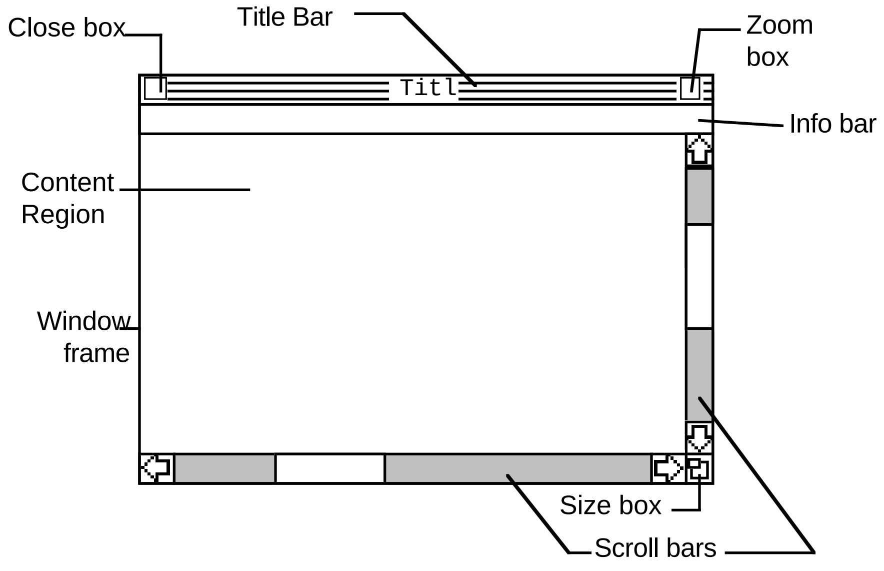


Figure 5-1: Typical Document Window

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:

```
function NewDocument: grafPortPtr;

{ Open a new window, returning the pointer                    }
{                                                             }
{ Returns: Window's window pointer; nil for an error          }

const
   rWindParam1 = $800E;                      {resource ID}
   wrNum = 1001;                             {window resource number}

begin {NewDocument}
NewDocument :=
   NewWindow2(@'MyWindow', 0, nil, nil, $02, wrNum, rWindParam1);
end; {NewDocument}
```

Listing 5-1A: Pascal Code to Open a New 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 5-1B: 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` procedure 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 loosing 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:

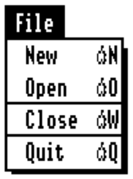


Figure 5-2: File Menu for Problem 5-1

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 nil 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);
```

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);
```

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 procedure 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 p-string long enough to hold our longest window 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 # open windows}

type
   documentRecord = record              {information about our document}
      wPtr: grafPortPtr;                {window pointer}
      wName: pString;                   {window name}
      end;

var
   documents: array[1..numDocs] of documentRecord; {our documents}
```

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 260 bytes of memory per document, and they will get bigger.) 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
   documentPtr = ^documentRecord;       {document pointer}
   documentRecord = record              {information about our document}
      next: documentPtr;                {next document}
      wPtr: grafPortPtr;                {window pointer}
      wName: pString;                   {window name}
      end;

var
   documents: documentPtr;              {our documents}
```

```
procedure InitGlobals;

{ Initialize the global variables                               }

begin {InitGlobals}
documents := nil;
end; {InitGlobals}
```

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.

```
procedure CloseDocument (dPtr: documentPtr);

{ Close a document and its associated window                    }
{                                                               }
{ Parameters:                                                   }
{    dPtr - pointer to the document to close; may be nil        }

var
   lPtr: documentPtr;                  {pointer to the previous document}

begin {CloseDocument}
if dPtr <> nil then begin
   CloseWindow(dPtr^.wPtr);            {close the window}
   if documents = dPtr then            {remove dPtr from the list when...}
      documents := dPtr^.next          {...dPtr is the first document}
   else begin                          {...dPtr is not the first document}
      lPtr := documents;
      while lPtr^.next <> dPtr do
         lPtr := lPtr^.next;
      lPtr^.next := dPtr^.next;
      end; {else}
   dispose(dPtr);                      {dispose of the document record}
   end; {if}
end; {CloseDocument}


function FindDocument (wPtr: grafPortPtr): documentPtr;

{ 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             }

var
   done: boolean;                      {used to test for loop termination}
   dPtr: documentPtr;                  {used to trace the document list}
```

```
begin {FindDocument}
dPtr := documents;
done := dPtr = nil;
while not done do
   if dPtr^.wPtr = wPtr then
      done := true
   else begin
      dPtr := dPtr^.next;
      done := dPtr = nil;
      end; {else}
FindDocument := dPtr;
end; {FindDocument}


function NewDocument (wName: pString): documentPtr;

{ Open a new document                                            }
{                                                                }
{ Parameters:                                                    }
{    wName - name for the new window                             }
{                                                                }
{ Returns: Document pointer; nil for an error                    }

const
   rWindParam1 = $800E;                 {resource ID}
   wrNum = 1001;                        {window resource number}

var
   dPtr: documentPtr;                   {new document pointer}

begin {NewDocument}
new(dPtr);                              {allocate the record}
if dPtr <> nil then begin
   dPtr^.onDisk := false;               {not on disk}
   dPtr^.wName := wName;                {set up the name}
   dPtr^.wPtr :=                        {open the window}
      NewWindow2(@dPtr^.wName, 0, nil, nil, $02, wrNum, rWindParam1);
   if dPtr^.wPtr = nil then begin
      FlagError(1, ToolError);          {handle a window error}
      dispose(dPtr);
      dPtr := nil;
      end {if}
   else begin
      dPtr^.next := documents;          {put the document in the list}
      documents := dPtr;
      end; {else}
   end {if}
else
   FlagError(2, 0);                     {handle an out of memory error}
NewDocument := dPtr;
end; {NewDocument}
```

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 `GetPString` and `FreePString` 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 it's 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`.

```
procedure GetUntitledName (var name: pString);

{ Create a name for an untitled window                                  }
{                                                                       }
{ Parameters:                                                           }
{    name - (returned) name for the window                              }

const
   untitled = 101;                      {Resource number for "Untitled "}

var
   dPtr: documentPtr;                   {used to trace the document list}
   number: integer;                     {new value for untitledNum}
   sPtr: pStringPtr;                    {pointer to the resource string}

begin {GetUntitledName}
dPtr := documents;                      {if there are no untitled         }
number := 1;                            { documents then reset untitledNum}
```

```
while dPtr <> nil do
   if not dPtr^.onDisk then begin
      number := untitledNum;
      dPtr := nil;
      end {if}
   else
      dPtr := dPtr^.next;
untitledNum := number;
sPtr := GetPString(untitled);           {set the base name}
if sPtr = nil then
   name := 'Untitled '
else begin
   name := sPtr^;
   FreePString(untitled);
   end; {else}
name := concat(name, cnvis(untitledNum)); {add the untitled number}
untitledNum := untitledNum+1;           {update untitledNum}
end; {GetUntitledName}
```

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 nil, indicating that there are no open documents. Be sure to include the `onDisk` field, initializing it to false in `NewDocument`.
3. Add `GetUntitledNum`, 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 `GetPString` and `FreePString`. 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 record with a top, bottom, left and right position. We'll look at them in detail in a later lesson.)

```
procedure X (wPtr: grafPortPtr);

{ Draw an X across the window wPtr                           }
{                                                            }
{ Parameters:                                                }
{    wPtr - pointer to the window to draw                    }

var
   port: grafPortPtr;                   {caller's grafPort}
   r: rect;                             {port rectangle}

begin {X}
port := GetPort;                        {get the caller's grafPort}
SetPort(wPtr);                          {make our port active}
PenNormal;                              {use a "normal" pen}
GetPortRect(r);                         {get the size of our window}
MoveTo(r.h1,r.v1);                      {draw the X}
LineTo(r.h2, r.v2);
MoveTo(r.h1, r.v2);
LineTo(r.h2, r.v1);
SetPort(port);                          {reset the caller's port}
end; {X}
```

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 procedure X won't become a permanent addition to Frame, so be sure you keep a separate copy of Frame that does not have this procedure.

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 to 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 is one special requirement for this subroutine. Any time a subroutine is called from the toolbox, as this one is, the subroutine can have trouble reading global variables. The reason this is true has to do with the memory models used by compilers on the Apple IIGS. In ORCA/Pascal, you need to tell the compiler that the subroutine will be called by the toolbox so the compiler will add some special code to the procedure to fix the data bank register. You do that with the `DataBank` directive, as you can see in Listing 5-7.

```
{$databank+}

procedure DrawContents;

{ Draw the contents of the active port                          }

var
   port: grafPortPtr;                     {caller's grafPort}
   r: rect;                               {port rectangle}

begin {DrawContents}
port := GetPort;                          {get our port}
PenNormal;                                {use a "normal" pen}
GetPortRect(r);                           {get the size of our window}
MoveTo(r.h1,r.v1);                        {draw the X}
LineTo(r.h2, r.v2);
MoveTo(r.h1, r.v2);
LineTo(r.h2, r.v1);
end; {DrawContents}

{$databank+}

   .
   .
   .
dPtr^.wPtr := NewWindow2(@dPtr^.wName, 0, @DrawContents, nil, $02, wrNum,
   rWindParam1);
```

Listing 5-7: An Update Procedure

Problem 5-5: Add the update procedure from Listing 5-7 to Frame. If you are using the version of Frame from Problem 5-4, take out the procedure 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 `taskData`. As with any event, the location of the event is in the `eventWhere` 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 it's 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.

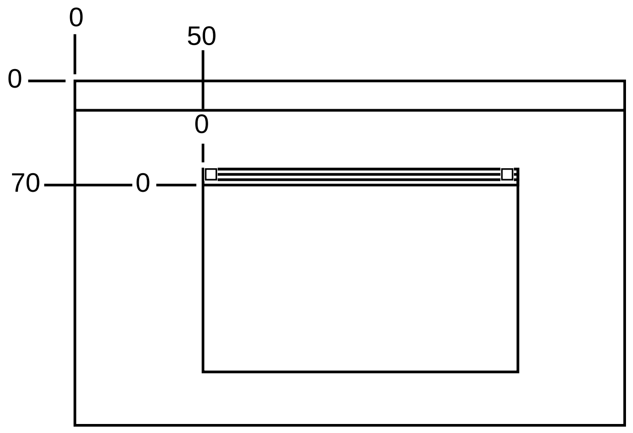


Figure 5-3: Global and Local Coordinates

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.

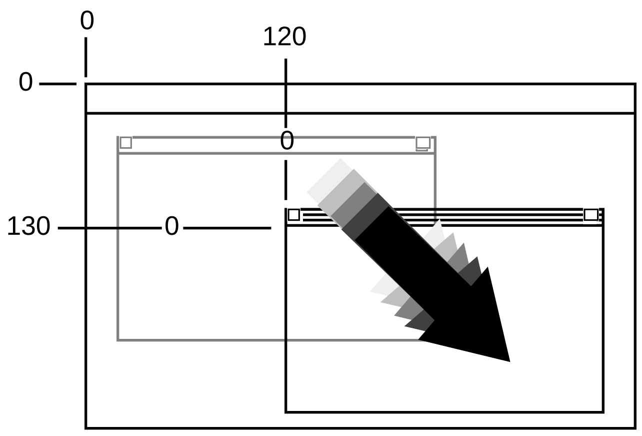


Figure 5-4: Dragging Doesn't Affect Local Coordinates

The problem is that the `eventWhere` 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 `eventWhere` 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 eventWhere to local coordinates for wPtr}
port := GetPort;
SetPort(wPtr);
localPoint := myEvent.eventWhere;
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 procedure 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.

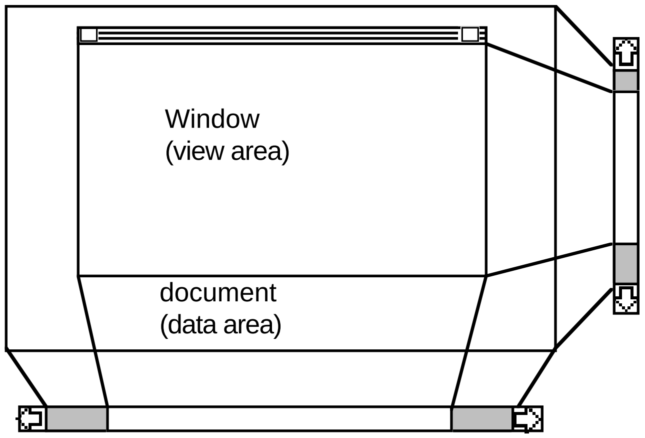


Figure 5-5: Scroll Bars and the Document

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 area 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` procedures 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 longint value, and you can use the same idea to pull apart the longint 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 record 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`.

The problem 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 it's own update procedure, so it acts like a little window all it's 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 it's 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.

wRefCon

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

wZoom

This field is a rect record, with the normal complement of four integer fields (v1, h1, v2, h2) defining the edges of a 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 it's 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 procedure 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.

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.

wPosition

This is a rect record. 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 front most window.

To make a new window the front most 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 window's 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 record, 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 record 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 $F (white) or $0 (black).

```
frameColor     $00a0
titleColor     $0bcd
tBarColor      $eefg
growColor      $h0ij
infoColor      $k0m0
```

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:

$00 Solid title bars are the normal kind, by default a boring black. The color is actually set by digit g.

$01 Dithered title bars use a checkerboard pattern, alternating between the foreground and background colors set by digits f and g.

$02 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 */
   1001,                       /* 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 (1001) {
   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 $0000, 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 $0800, 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 $800E, for `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 record that has to be filled in field by field with Pascal 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`.

```
function NewWind: grafPortPtr;

{ Create a new window                                         }
{                                                             }
{ Returns: Pointer to the window; nil for error               }

var
   wParms: paramList;                  {parameters for NewWindow}

begin {NewWind}
with wParms do begin
   paramLength := sizeof(wParms);
   wFrameBits := $DDA5;
   wTitle := @'  Untitled  ';
   wRefCon := 0;
   wZoom.h1 := 0; wZoom.h2 := 0;
```

```
    wZoom.v1 := 0; wZoom.v2 := 0;
    wColor := nil;
    wYOrigin := 0; wXOrigin := 0;
    wDataH := 200;
    wDataW := 640;
    wMaxH := 0;
    wMaxW := 0;
    wScrollVer := 8; wScrollHor := 8;
    wPageVer := 0; wPageHor := 0;
    wInfoRefCon := 0; wInfoHeight := 0;
    wFrameDefProc := nil;
    wInfoDefProc := nil;
    wContDefProc := @DrawContents;
    wPosition.v1 := 30;
    wPosition.h1 := 10;
    wPosition.v2 := 183;
    wPosition.h2 := 602;
    wPlane := pointer(topMost);
    wStorage := nil;
    end; {with}
 NewWind := NewWindow(wParms);     {open the window}
 end; {NewWind}
```

Listing 5-11: Subroutine to Define a Window with `NewWindow`

The parameters you actually pass in this record are basically the same old familiar ones you used with resources. Here are the differences:

`paramLength`	This is the size of the window record 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 record 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/Pascal's interface file for the Window Manager has a record defined for color tables. To save you the trouble of looking it up, here's the record you should use to define the color table:

```
wColorTbl = record
    frameColor: integer;
    titleColor: integer;
    tBarColor:  integer;
    growColor:  integer;
    infoColor:  integer;
    end;
```

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 Pascal's built in file I/O mechanism, but there are some limitations that make Pascal's file I/O procedures 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 Pascal. 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.

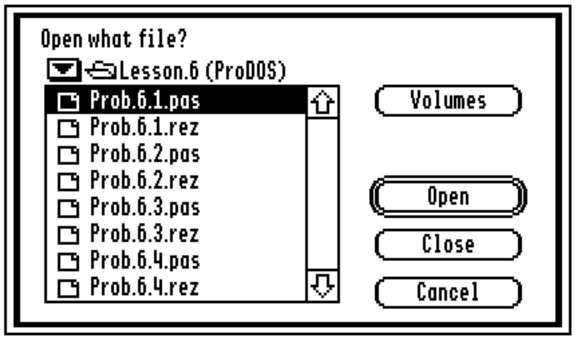


Figure 6-1: Open Dialog

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,
    replyPtr);
```

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 parameter 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 0, `promptRef` is a pointer to a p-string. If it is 1, `promptRef` is the handle of a p-string. Finally, the most appropriate choice for `promptRefDesc` is 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 procedure in your program. To understand what this procedure 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 Pascal source code 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 record 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 ORCA/Pascal, the definitions that are used for a file type record list are:

```
typeRec = record
    flags:     integer;
    fileType:  integer;
    auxType:   longint;
    end;

typeList5_0 = record
    numEntries:       integer;
    fileAndAuxTypes:  array [1..10] of typeRec;
    end;
```

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 Pascal source file has a file type of $B0 and an auxiliary file type of $0005. A file type record to load Pascal source files would be set up like this:

```
with types.fileAndAuxTypes[1] do begin
   flags := $0000;
   fileType := $B0;
   auxType := $0005;
   end;
```

If you wanted to load any source file, and not just Pascal source files, you would want to load any file with a file type of $B0, no matter what auxiliary file type the file had. In that case, you would set up the file type entry like this:

```
with types.fileAndAuxTypes[1] do begin
   flags := $8000;
   fileType := $B0;
   end;
```

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:

```
var
   types: typeList5_0;

...
types.numEntries := 1;
types.fileAndAuxTypes[1].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.

There's one thing you need to keep in mind when you are creating a file type list. The list is supposed to be a variable length array, but there isn't really any such thing. Some sort of fixed array size had to be picked. In the ORCA/Pascal interface file for SFO, the size of the array of file type entries is 10, so you can't create a file list with more than 10 entries. If you need more than that, you'll have to create your own private array or override the size of the array by changing the ORCA/Pascal interfaces. I sort of prefer creating my own array. One easy way to do that is with a variant record that overlays the ORCA/Pascal definition with your own, like this:

```
type
   bigTypeList5_0 = record
      numEntries:       integer;
      fileAndAuxTypes:  array [1..50] of typeRec;
      end;

   myTypeList = record
      case boolean of
         true:  (types1: typeList5_0);
         false: (types2: bigTypeList5_0);
      end;
```

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:

```
   replyRecord5_0 = record
       good:         integer;
       fileType:     integer;
       auxFileType:  longint;
       nameVerb:     integer;
       nameRef:      longint;
       pathVerb:     integer;
       pathRef:      longint;
       end;
```

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 `auxFileType` 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 Pascal's `new` procedure, 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 Pascal. Instead, GS/OS is returning a record 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 Pascal's string handling facilities on the characters in a GS/OS name, even if they have been copied into a Pascal string, since Pascal uses the null character to signal the end of a string, and null characters are legal characters in a GS/OS path name.

```
gsosInString = record
    size:  integer;
    theString:  packed array [1..254] of char;
    end;
gsosInStringPtr = ^gsosInString;

gsosOutString = record
    maxSize:    integer;
    theString:  gsosInString;
    end;
gsosOutStringPtr = ^gsosOutString;
```

Listing 6-1: GS/OS String Definitions Used in ORCA/Pascal

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.

There are a few dicey type issues you have to handle as you wind your way from a reply record to a handle to a name for a window, too. `SFGetFile2` can return the file name and path name a lot of ways, but they all use four bytes. The reply record sets aside those four bytes as a longint. In your document record, though, you'll be dealing with these values using Memory Manager commands that expect a handle, so your document record variables should define the space for these values as type `handle`. Finally, when you lock the handle and start plucking characters from it, you'll need to use a more specific type. The handle is a pointer to a pointer to a `gsosOutString`, so one way to handle this is to declare a work pointer of type `gsosOutStringPtr` and set it up like this:

```
gsosNamePtr := pointer(dPtr^.fileName^);
```

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.

```
procedure DoOpen;

{ Open a file                                                   }

const
   posX = 80;                          {X position of the dialog}
   posY = 50;                          {Y position of the dialog}
   titleID = 102;                      {prompt string resource ID}

var
   dPtr: documentPtr;                  {pointer to the new document}
   fileTypes: typeList5_0;             {list of valid file types}
   gsosNameHandle: handle;             {handle of the file name}
   gsosNamePtr: gsosOutStringPtr;      {pointer to the GS/OS file name}
   i: integer;                         {loop/index variable}
   name: pString;                      {new document name}
   reply: replyRecord5_0;              {reply record}

begin {DoOpen}
with fileTypes do begin                {set up the allowed file types}
   numEntries := 2;
   with fileAndAuxTypes[1] do begin
      flags := $8000;
      fileType := $B0;
      end; {with}
   with fileAndAuxTypes[2] do begin
      flags := $8000;
      fileType := $04;
      end; {with}
   end; {with}
reply.nameVerb := 3;                   {get the file to open}
reply.pathVerb := 3;
SFGetFile2(posX, posY, 2, titleID, nil, fileTypes, reply);
if ToolError <> 0 then
   FlagError(3, ToolError)             {handle an error}
else if reply.good <> 0 then begin
                                       {form the file name}
   gsosNameHandle := pointer(reply.nameRef);
   HLock(gsosNameHandle);
   gsosNamePtr := pointer(gsosNameHandle^);
   name := '  ';
   for i := 1 to gsosNamePtr^.theString.size do
      name := concat(name, gsosNamePtr^.theString.theString[i]);
   name := concat(name, '  ');
   HUnlock(gsosNameHandle);
   dPtr := NewDocument(name);          {get a document record}
   if dPtr = nil then begin        {in case of error, dispose of the names}
      DisposeHandle(handle(reply.nameRef));
      DisposeHandle(handle(reply.pathRef));
      end {if}
   else begin                          {otherwise save the names}
      dPtr^.fileName := handle(reply.nameRef);
      dPtr^.pathName := handle(reply.pathRef);

      if LoadDocument(dPtr) then       {read the file}
         dPtr^.onDisk := true          {file is on disk}
      else                             {handle a read error}
         CloseDocument(dPtr);
      end; {else}
   end; {else if}
end; {DoOpen}
```

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 it's 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.

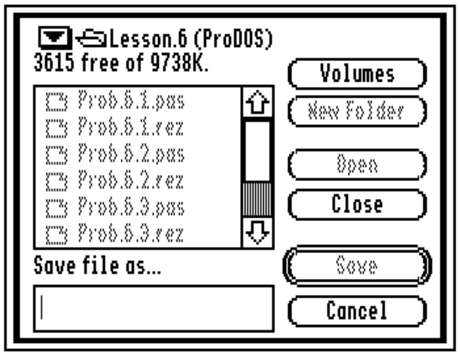


Figure 6-2: Sample Save Dialog

```
SFPutFile2(x, y, promptRefDesc, promptRef, origNameRefDesc, origNameRef,
   replyPtr);
```

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 a value of 0 for `origNameRefDesc`, telling `SFPutFile2` that we are passing a pointer for 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 it's own.

```
procedure DoSaveAs;

{ Save a document to a new name                                }

const
   posX = 80;                          {X position of the dialog}
   posY = 50;                          {Y position of the dialog}
   titleID = 103;                      {prompt string resource ID}

var
   dPtr: documentPtr;                  {document to save}
   dummyName: integer;                 {used for a null file name prompt}
   gsosNameHandle: handle;             {handle of the file name}
   gsosNamePtr: gsosOutStringPtr;      {pointer to the GS/OS file name}
   i: integer;                         {loop/index variable}
   reply: replyRecord5_0;              {reply record}

begin {DoSaveAs}
dPtr := FindDocument(FrontWindow);
if dPtr <> nil then begin
   reply.nameVerb := 3;                {get the new file name}
   reply.pathVerb := 3;
   if dPtr^.fileName = nil then begin
      dummyName := 0;
      SFPutFile2(posX, posY, 2, titleID, 0, @dummyName, reply);
      end {if}
   else
      SFPutFile2(posX, posY, 2, titleID, 0,
         pointer(ord4(dPtr^.fileName^)+2), reply);
   if ToolError <> 0 then
      FlagError(3, ToolError)          {handle an error}
   else if reply.good <> 0 then begin
                                       {form the new window name}
      gsosNameHandle := pointer(reply.nameRef);
      HLock(gsosNameHandle);
      gsosNamePtr := pointer(gsosNameHandle^);
      dPtr^.wName := '  ';
      for i := 1 to gsosNamePtr^.theString.size do
         dPtr^.wName :=
            concat(dPtr^.wName, gsosNamePtr^.theString.theString[i]);
      dPtr^.wName := concat(dPtr^.wName, '  ');
      HUnlock(gsosNameHandle);
      SetWTitle(dPtr^.wName, dPtr^.wPtr);
```

```
                                    {save the names}
      dPtr^.fileName := handle(reply.nameRef);
      dPtr^.pathName := handle(reply.pathRef);
      dPtr^.onDisk := true;         {file is on disk}
      SaveDocument(dPtr);           {save the file}
      end; {else if}
   end; {if}
end; {DoSaveAs}


procedure DoSave;

{ Save a document to the existing disk file                     }

var
   dPtr: documentPtr;               {document to save}

begin {DoSave}
dPtr := FindDocument(FrontWindow);
if dPtr <> nil then
   if dPtr^.onDisk then
      SaveDocument(dPtr)
   else
      DoSaveAs;
end; {DoSave}
```

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!):

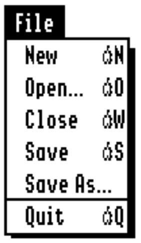


Figure 6-3: File Menu with Save and Save As

Make sure you do all of the following:

a. Add the `DoSave` and `DoSaveAs` procedures from Listing 6-3, calling them when the user uses the Save or Save As... command.
b. Create a `SaveDocument` procedure. 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 Pascal's built in file handling procedures, `read`, `write`, `reset` and `rewrite`, to handle the file input and output. Assuming you create a file of points, ORCA/Pascal will create a binary file, which has a file type of $06 and an auxiliary file type of $0000. You will need those values for your `SFGetFile2` call.

Hint: Just to make things easy, I made the file a `file of integer`, and saved the number of points in the file as the first integer. I saved the points as a pair of integers.

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, `SFPGetFile2` and `SFPPutFile2` do.

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 Pascal's built in file handling procedures, like `Read` and `Readln`, 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 record that contains the various values that will be passed to the call. The GS/OS subroutines also return values in the same record. In ORCA/Pascal 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 records 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 record and call declarations for `OpenGS`, taken from the ORCA/Pascal header file for GS/OS:

```
     openOSDCB = record
         pcount:          integer;
         refNum:          integer;
         pathName:        gsosInStringPtr;
         requestAccess:   integer;
         resourceNumber:  integer;
         access:          integer;
         fileType:        integer;
         auxType:         longint;
         storageType:     integer;
         createDateTime:  timeField;
         modDateTime:     timeField;
         optionList:      optionListPtr;
         dataEOF:         longint;
         blocksUsed:      longint;
         resourceEOF:     longint;
         resourceBlocks:  longint;
         end;

  procedure OpenGS (var parms: 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 `openOSDCB` 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 that is built into ORCA/Pascal 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(dataEOF, userID, $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's the declarations for the `ReadGS` call and it's record:

```
   readWriteOSDCB = record
      pcount:         integer;
      refNum:         integer;
      dataBuffer:     ptr;
      requestCount:   longint;
      transferCount:  longint;
      cachePriority:  integer;
      end;

procedure ReadGS (var parms: readWriteOSDCB);
```

Listing 6-5: `ReadGS` and the `ReadOSDCB` Record

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 dataEOF from the OpenGS record. 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.

```
   closeOSDCB = record
       pcount: integer;
       refNum: integer;
       end;

procedure CloseGS (var parms: closeOSDCB);
```

Listing 6-6: CloseGS and the CloseOSDCB Record

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.

```
function LoadDocument (dPtr: documentPtr): boolean;

{ Load a document file from disk                               }
{                                                              }
{ Parameters:                                                  }
{    dPtr - pointer to the document to save                    }
{                                                              }
{ Returns: true if successful, else false                      }

var
   clRec: closeOSDCB;                  {CloseGS record}
   opRec: openOSDCB;                   {OpenGS record}
   port: grafPortPtr;                  {caller's grafPort}
   r: rect;                            {our port rect}
   rdRec: readWriteOSDCB;              {ReadGS record}

begin {LoadDocument}
LoadDocument := true;                  {assume we will succeed}
```

```
opRec.pcount := 12;                    {open the file}
HLock(dPtr^.pathName);
opRec.pathName := pointer(ord4(dPtr^.pathName^)+2);
opRec.requestAccess := 1;
opRec.resourceNumber := 0;
opRec.optionList := nil;
OpenGS(opRec);
if ToolError <> 0 then begin
   FlagError(4, ToolError);
   LoadDocument := false;
   end {if}
else begin
   dPtr^.dataHandle :=                 {allocate memory for the file}
      NewHandle(opRec.dataEOF, userID, $8000, nil);
   if ToolError <> 0 then begin
      FlagError(2, ToolError);
      LoadDocument := false;
      end {if}
   else begin
      rdRec.pcount := 4;               {read the file}
      rdRec.refnum := opRec.refnum;
      rdRec.dataBuffer := dPtr^.dataHandle^;
      rdRec.requestCount := opRec.dataEOF;
      ReadGS(rdRec);
      if ToolError <> 0 then begin
         FlagError(4, ToolError);
         LoadDocument := false;
         DisposeHandle(dPtr^.dataHandle);
         dPtr^.dataHandle := nil;
         end {if}
      else begin
         HUnlock(dPtr^.dataHandle);     {let the data move in memory}
         port := GetPort;               {force an update}
         SetPort(dPtr^.wPtr);
         GetPortRect(r);
         InvalRect(r);
         SetPort(port);
         end; {else}
      end; {else}
   clRec.pcount := 1;                  {close the file}
   clRec.refnum := opRec.refnum;
   CloseGS(clRec);
   end; {else}
HUnlock(dPtr^.pathName);               {unlock the name handle}
end; {LoadDocument}
```

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's the declarations for `DestroyGS`, the GS/OS call to delete a disk file, along with its record:

```
     destroyOSDCB = record
         pcount:   integer;
         pathName: gsosInStringPtr;
         end;

  procedure DestroyGS (var parms: destroyOSDCB);
```

Listing 6-8: `DestroyGS` and the `DestroyOSDCB` Record

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 lest 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.

```
     createOSDCB = record
         pcount:        integer;
         pathName:      gsosInStringPtr;
         access:        integer;
         fileType:      integer;
         auxType:       longint;
         storageType:   integer;
         dataEOF:       longint;
         resourceEOF:   longint;
         end;
```

```
  procedure CreateGS (var parms: createOSDCB);
```

Listing 6-9: `CreateGS` and the `CreateOSDCB` Record

We need to set up the first six parameters, through `storageType`, to create the output file. The last two parameters – `dataEOF` 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 Apple Computer. 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 Apple Computer.

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` record 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 record for a `ReadGS` call.

```
     readWriteOSDCB = record
         pcount:         integer;
         refNum:         integer;
         dataBuffer:     ptr;
         requestCount:   longint;
         transferCount:  longint;
         cachePriority:  integer;
         end;

  procedure WriteGS (var parms: readWriteOSDCB); prodos ($2013);
```

Listing 6-10: `WriteGS` and its Record

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.

```
procedure SaveDocument (dPtr: documentPtr);

{ Save a document file to disk                                  }
{                                                               }
{ Parameters:                                                   }
{    dPtr - pointer to the document to save                     }

var
   clRec: closeOSDCB;                   {CloseGS record}
   crRec: createOSDCB;                  {CreateGS record}
   dsRec: destroyOSDCB;                 {DestroyGS record}
   opRec: openOSDCB;                    {OpenGS record}
   wrRec: readWriteOSDCB;               {WriteGS record}

begin {SaveDocument}
HLock(dPtr^.pathName);                  {lock the path name}
dsRec.pcount := 1;                      {destroy any old file}
dsRec.pathName := pointer(ord4(dPtr^.pathName^)+2);
DestroyGS(dsRec);
crRec.pcount := 5;                      {create a new file}
crRec.pathName := pointer(ord4(dPtr^.pathName^)+2);
crRec.access := $C3;
crRec.fileType := $C1;
crRec.auxType := 0;
crRec.storageType := 1;
CreateGS(crRec);
if ToolError <> 0 then
   FlagError(5, ToolError)
else begin
   opRec.pcount := 3;                   {open the file}
   opRec.pathName := pointer(ord4(dPtr^.pathName^)+2);
   opRec.requestAccess := 2;
   OpenGS(opRec);
   if ToolError <> 0 then
      FlagError(5, ToolError)
   else begin
      wrRec.pcount := 4;                {write the file}
      wrRec.refnum := opRec.refnum;
      HLock(dPtr^.dataHandle^);
      wrRec.dataBuffer := dPtr^.dataHandle^;
      wrRec.requestCount := $8000;
      WriteGS(wrRec);
      if ToolError <> 0 then
         FlagError(5, ToolError);
      HUnlock(dPtr^.dataHandle^);
      clRec.pcount := 1;                {close the file}
      clRec.refnum := opRec.refnum;
      CloseGS(clRec);
      end; {else}
   end; {else}
HUnlock(dPtr^.pathName);                {unlock the name handle}
end; {SaveDocument}
```

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 0. There are three parts to the file: a bit map of the picture itself, s 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 to 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 it's 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(srcLocInfo, srcRect, destX, destY, transferMode);
```

The first parameter is the most complex; it's a record that tell's QuickDraw II a lot about the picture it is supposed to draw. The record is defined like this:

```
locInfo = record
   portSCB:          integer;
   ptrToPixelImage:  ptr;
   width:            integer;
   boundsRect:       rect;
   end;
```

The first parameter 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 parameter is a pointer to the first byte of the picture. Set `ptrToPixelImage` to the first byte of the picture file.

The `width` parameter tells `PPToPort` how long each line is, in bytes. Our lines are 160 bytes long, so we set `width` to 160.

The last parameter 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 record describing the picture, and that's what you would normally do. In fact, you can just pass `boundsRect`, from the `locInfo` record. 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 procedure, so you can use it as a drop-in replacement for the update procedure you've used in Frame. All you have to do is make sure the document record defines the name of the handle for the file you load as `pictureHandle`.

```
{$databank+}

procedure DrawContents;

{ Draw the contents of the active port                          }

var
   dPtr: documentPtr;                    {document to draw}
   info: locInfo;                        {record for PPToPort}

begin {DrawContents}
dPtr := FindDocument(GetPort);
if dPtr <> nil then begin
   HLock(dPtr^.pictureHandle);
   with info do begin
      portSCB := $00;
      ptrToPixelImage := dPtr^.pictureHandle^;
      width := 160;
      with boundsRect do begin
         h1 := 0; h2 := 320;
         v1 := 0; v2 := 200;
         end; {with}
      end; {with}
   PPToPort(info, info.boundsRect, 0, 0, modeCopy);
   HUnlock(dPtr^.pictureHandle);
   end; {if}
end; {DrawContents}

{$databank+}
```

Listing 6-12: Update Procedure 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 and 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 Pascal'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
```

Lesson 7 – Move Over Guttenberg

Goals for This Lesson

This lesson shows how to use the Print Manager to print anything you can draw in a window. We also add printing to Frame, our framework program that will be the basis for most of the programs in the book.

How the Print Manager Works

Printing could have been really hard on the Apple IIGS. After all, virtually every desktop program has to print graphic images, not the simple text you would expect from a program on an IBM PC or a text-based Apple II. Printing text is hard enough, but with graphics you have to know how to address the individual bits on each and every printer someone might try to hook up to their Apple IIGS. Apple's Print Manager takes over the job of figuring out what sort of printer is hooked up to the computer – whether it can print color or just black and white, how many individually addressable dots there are in each direction, and how big the paper is. All you have to do is bring up a few dialogs to let the user pick options, and when the time comes, draw to the printer just like you would draw to a window.

There's a lot of information that the user can pick to control the printing process – things like whether they want a full color image, whether the image should be printed normally or in landscape mode, and a host of smaller details. Of course, no one wants to pick all of those details each time something is printed. Well, maybe not all of them. Things like how many copies are printed should at least be pointed out each time a document is printed. You wouldn't want to get forty copies of something by mistake because forty copies were printed the last time!

To handle this varying level of detail, the Print Manager actually uses three distinct levels of dialogs. The first level is something you don't even worry about. It's possible to have more than one printer hooked up to an Apple IIGS – mine is hooked up to two LaserWriters over a network, and generally to an ImageWriter on my desk. When I pick out a printer, it's something that I generally only do once, and then I use the same printer all of the time. The same thing would be true if you have just one printer hooked up. You want to tell your computer what sort of printer it is, and what printer port it is plugged into, just once. After that, every single application should be able to find out about the printer you've picked. That first level of information is selected from the Control Panel. You and your program don't do anything at all.

The next level of information is the sort of stuff you want to do one time for each document, and then leave it pretty much the same after that. For example, if you're using a paint program to create a picture, you're may want to print the picture in portrait mode or landscape mode, but you're probably not going to switch between the two for the same picture. This second level of information is done with the page setup dialog. The Print Manager will handle all of the details for you, even using a different page setup dialog for each kind of printer. You do have to get involved, though, even if it's just a little bit. You have to make a Print Manager call to draw the dialog.

The last level of information is the sort of information that could change each time a document is printed, like the number of copies that will print, and whether the document should be printed quickly with low quality output; or whether the document is finished, so it's worth the time to let the printer do the best job it can. This is in the print job dialog. Like the page setup dialog, it's something you have to ask the Print Manager to do, but the Print Manager will draw the dialog and figure out what happens for you.

That's a lot of information to remember about how to print something, and some of that information should be remembered for the lifetime of the document. The Print Manager uses a chunk of memory called a print record to remember all of this information. The print record is something you create when the document is first created with the New command. Normally, you would save the print record right along with the document, loading it and asking the Print Manager to check it for accuracy each time the document itself is loaded. That way, if the user picks landscape mode printing today, your program will be smart enough to print the document the same way tomorrow.

The rest of this lesson covers the mechanics of making all of this happen, as well as outlining the print loop which actually prints the information. You'll get to add printing to three programs: Frame, so you can put it in all of your future programs with a minimum of hassle; the point drawing program, so you can play with printing in a simple setting; and finally in the slide show program you wrote as part of a problem in the last section, so it becomes not just a slide show program, but a picture printing program. Incidentally, this finishes Frame. After this lesson, all of the things you learn will be a lot more specific to the kind of program you are writing.

Starting the Tools (Again)

Well, after all that, the first call we'll talk about isn't even in the Print Manager. You see, there's a minor problem with the Print Manager. It isn't started by `StartDesk`, the quick and dirty tool start up subroutine built into ORCA/Pascal, so we need to start the Print Manager for ourselves. We'll face the same problem later in the course with tools like TextEdit, so it's a topic I though we should cover fairly carefully.

Back before System 5.0, starting tools was a piecemeal process. Each tool had to be started individually, and many of them had specific requirements. A very common one was the need for a chunk of direct page memory. Getting over this initial hurdle was pretty hard for a beginning toolbox programer, and that's why I put `StartDesk` into ORCA/Pascal. Well, Apple must have agreed with me, because there is a nifty new tool call in the toolbox starting with System Disk 5.0 that starts the tools almost as smoothly as `StartDesk`. It had a major problem, though: for programs that used resources, Apple's `StartUpTools` call didn't work correctly from a programming environment like ORCA. This problem has been fixed starting with System Disk 6.0, so there's no reason at all not to take advantage of all of the help `StartUpTools` gives us.

`StartUpTools` needs two things from us: a user ID, so it knows who owns the memory it allocates for the tools that need it; and a record containing a list of the tools you want started. The record also has a few other details, like whether you want to use 320 mode or 640 mode. It passes back a pointer to another record which you'll pass to `ShutDownTools` when your program quits. Here's the code you need:

```
var
   startStopParm: longint;

begin
startStopParm := StartUpTools(UserID, 2, 1001);
if ToolError <> 0 then
   {handle the error here}

...

ShutDownTools(1, startStopParm);
```

Listing 7-1A: Using `StartUpTools`

```
type rToolStartup (1) {
   mode640;                  /* mode to start QuickDraw */
      {
      1,$0300,               /* Tool number, version */
      };
   };
```

Listing 7-1B: The `rToolStartup` Resource

As you can see, one way to give `StartUpTools` a list of tools is with a resource, and that's the method we'll use. That second parameter of a 2 to `StartUpTools` tells `StartUpTools` that the last parameter is a resource ID for an `rToolStartup` resource. If you would rather use a record that's built inside of your program, you can check out the complete description of the tool call in the *Apple IIGS Toolbox Reference: Volume 3*.

`StartUpTools` returns a handle to a record that you need to pass along to `ShutDownTools` when your program quits. That parameter has to be stored in a safe, global variable so you can get to it at the proper time. The first parameter to `ShutDownTools` tells it that the second parameter is a handle; the other possibility is a pointer, which you would only use if you are setting up the record manually.

Of course, it is possible for `StartUpTools` to fail. One of the tools you ask for might not be on the boot disk, or there might not be enough direct page memory left, or there might be a disk error, or... well, a lot could go wrong. No matter what does happen, though, your program is left in the lurch. You can't go on, since you don't have all of the tools available, and you can't even tell the user what happened, since the tools you use to put up an alert may not be going. Handling this error is a tough one, and the only reliable way to do it is with a Miscellaneous Tool Set call called `SysFailMgr`. This tool call prints a message and a number, then displays that familiar sliding apple, completely shutting down the machine until the computer is rebooted. That's a horribly drastic action for your program to take, but the only other possibility is to just exit, without even telling the user why you quit, and that's even worse.

Here's a call you could use for this rather severe error:

```
SysFailMgr(ToolError, @'Could not start tools: ');
```

Just for fun, when you're working the first problem, be sure and fake a tool start up error. It's a good idea to test your program in sensitive places like this anyway, so you can make sure errors are handled properly, and it also lets you see exactly what you're doing to the user.

The only other topic we need to cover before we get back to the Print Manager is the tool numbers and versions for the various tools used in this course. Table 7-1 shows the tools, their tool numbers, and the version numbers as of System Disk 6.0. Some tools depend on other tools, too. For example, the Print Manager uses the Dialog Manager to create the various printer dialogs. The table shows the tools you need to start to use a particular tool in the last column.

If there is a more recent system disk, or if your program needs to work with older system disks, you will have to check the release notes for the system disk or use a program like NiftyList to find the version numbers.

tool number	tool version	tool name	depends on…
1	3.1	Tool Locator	
2	3.2	Memory Manager	1
3	3.2	Miscellaneous Tool Set	1, 2
4	3.7	QuickDraw II	1, 2, 3
5	3.4	Desk Manager	1, 2, 3, 4, 6, 14, 15, 16, 20, 21, 22
6	3.1	Event Manager	1, 2, 3
8	3.3	Sound Tool Set	1, 2, 3
11	3.0	Integer Math Tool Set	1
14	3.3	Window Manager	1, 2, 3, 4, 6, 15, 16, 20, 27, 30
15	3.3	Menu Manager	1, 2, 3, 4, 6, 14, 16, 30
16	3.3	Control Manager	1, 2, 3, 4, 6, 14, 15, 30
18	3.4	QuickDraw II Auxiliary	1, 2, 3, 4, 27
19	3.1	Print Manager	1, 2, 3, 4, 6, 14, 15, 16, 18, 20, 21, 27, 28
20	3.3	LineEdit Tool Set	1, 2, 3, 4, 6, 18, 22, 27
21	3.4	Dialog Manager	1, 2, 3, 4, 6, 14, 15, 16, 18, 20, 27
22	3.1	Scrap Manager	1, 2
23	3.3	Standard File Operations	1, 2, 3, 4, 6, 14, 15, 16, 20, 21
25	1.4	Note Synthesizer	1, 2, 8
27	3.3	Font Manager	1, 2, 3, 4, 11, 14, 15, 16, 20, 21, 28
28	3.3	List Manager	1, 2, 3, 4, 6, 14, 15, 16
30	1.2	Resource Manager	1
34	1.3	TextEdit Tool Set	1, 3, 4, 6, 14, 15, 16, 18, 22, 27, 30

Table 7-1: Tools `StartUpTools` Can Start

The Tool Locator, Memory Manager and Resource Manager are started automatically, and don't need to be listed. As a general rule, you should start all of the other tools except the Sound Tool Set, Print Manager, Note Synthesizer and TextEdit Tool Set for every program.

Problem 7-1: Change Frame so it uses `StartUpTools` to start all of the tools checked in Table 7-1. Use `SysFailMgr` to handle any errors reported by `StartUpTools`.

Getting a Print Record

Getting back to the Print Manager, the first thing we have to do is allocate a print record. The print record needs to be a handle, and it needs to be 140 bytes long. Since you need a print record in every document, it makes sense to allocate one when you create the document record and open the window.

```
dPtr^.prHandle := NewHandle(140, userID, 0, nil);
```

What you do with this record depends on whether or not you are creating the record for a new document or opening an existing one that will get a print record from the document's file. If you are creating a new print record, you need to fill in the print record with default values for all of the fields. `PrDefault` does that for you; you should probably call it in the procedure you have hooked up to handle the menu command New.

```
PrDefault(dPtr^.prHandle);
```

If you are opening a file that has a print record, the natural thing to do is to fill in the print record from the file. That's fine as far as it goes, but the print record might have been created for a different printer, or even for a different version of the Print Manager that had slightly different requirements. Either way, you need to give the Print Manager a chance to check all of the information in the print record; you do that like this:

```
changed := PrValidate(dPtr^.prHandle);
```

`PrValidate` returns a boolean result that tells you if the print record was changed. If so, you might want to set a "changed" flag in your document record to make sure the user gets a chance to save the document with the new print record, even if there are no other changes. Then again, you might feel like I do. I figure the same call will be made the next time the document is opened, and the user isn't likely to know or care about a change that happens automatically, behind their back. I just ignore the result.

It's possible to get errors back from any of these calls, so you need to handle those appropriately. I think one good way to handle errors here is to flag the error using our normal error handler, then dispose of the print record (if it was even allocated) and set it to nil. Later, you can check to see if there is a valid print record before trying to print. That lets the user of the program keep going, even if there is some problem with printing.

The Page Setup Dialog

There are two menu commands that every program using the Print Manager should include in the File menu. The first is "Page Setup...", and the second is "Print..." The Page Setup command calls `PrStlDialog` to let the user pick out that second level information we talked about earlier; things like whether the page is in portrait or landscape mode. The call is pretty simple:

```
dPtr^.changed := dPtr^.changed or PrStlDialog(dPtr^.prHandle);
```

The call returns a flag, just like `PrValidate`, that tells whether or not changes were actually made. For `PrValidate`, I mentioned that I normally throw the flag away, since the user doesn't really know we are calling `PrValidate`, anyway. This time, though, the user picked a specific menu command to make specific changes to a document, so I think it's appropriate to make sure the changes get saved. If you aren't saving the print record with the document, though, you should still throw the value away, and not set the document's changed flag.

You might notice that there's no figure here telling you what the dialog looks like. There are two very good reasons for that. The first is that the dialog is different for different printers and different versions of the Print Manager, so you'd have to check the program to see what was happening, anyway. The most important reason, though, points out one of the neat things about the way the toolbox works: *we don't care what the dialog looks like*! Oh, sure, the user cares, but as a programmer, it just doesn't matter to us. The Print Manager worries about all of the details, and fills in the print record for us. All we really care about are a few fields in the print record itself.

The Print Dialog

The last of the dialogs is the one that lets the user pick things that will change from one time the document is printed to the next, like the number of copies that will be printed. This is the first step you take when the user picks the Print command from the File menu. The second step, of course, is to actually print the document.

Like the other calls that set up information in the print record, `PrJobDialog` takes the print record as a parameter, and returns a boolean result. If the result is true, the user picked out some options and asked you to print the document; if the result is false, you don't do anything.

```
   if PrJobDialog(dPtr^.prHandle) then
      {print the document};
```

Dimming the Menus

One of the nice things a program can do is to let the user know when a command is available, and when it can't be used. Normally you will want to dim any menu command that isn't available. That's not something I've made a big deal about so far, since most of our programs have been frameworks for future programs, but now it's time to start thinking about when commands are available and when they are not.

The idea is to dim a menu command when it can't be used, and undim it as soon as it can be used. In the case of the print commands, that means the commands should be dim if there isn't an active document window, and they should not be dimmed if a window is open. The big issue is actually when you dim or undim the windows.

There are a lot of ways to handle this, but let's talk about two: the obvious solution, and the one I hope you will use. The obvious solution is to go into the program and insert a check where documents are created and closed, dimming or undimming the print commands then. That works, but it makes the program very complicated. It means that you have to think about each and every menu command, and pepper code throughout the program to turn menu commands on and off. Let me make a very safe prediction: you'll mess it up. This method is a sure-fire way to create bugs, and they'll be very hard to track down.

Let's think about this another way. Instead of peppering the code to handle dimming throughout the program, collect it all in one spot. For each menu command that may need to be disabled, check the command in this one place for all of the possible conditions and either dim or undim the menu. This way, if there is a bug, you know exactly where the problem is.

Of course, you still need to call this subroutine. In most cases, you can just put the subroutine in the event loop, right before the call to `TaskMaster`. That way, if the event you just handled changed the status of the menus, the menu bar is updated before `TaskMaster` gets a shot at an event that might end up pulling down a menu.

The obvious problem with this method is that it takes time, even if the menus don't need to be changed. Well, sometimes that's important, and sometimes it isn't. After all, most of the time spent in the event loop is just time spent waiting on the organic computer at the keyboard to do something, so checking the menu status is sort of the computer equivalent of twiddling your thumbs. Time consuming work is done outside of the event loop, when the subroutines are called. Still, maybe the time will be a factor. To see, try commenting out the call to the subroutine during testing. After all, you know when the commands are available, right? If the program runs noticeably faster, go back and make the subroutine itself more efficient, or perhaps set it up so it only gets called when there's a null event.

I doubt if you'll notice a speed difference, though. At least, not in this program. Later in the course, we'll write a program that is a little sensitive to this subroutine, and we'll make a few minor changes – but only for that program.

Problem 7-2: Update Frame to handle the standard print dialogs. You will need to add the menu commands "Page Setup..." and "Print..." to your File menu, calling subroutines to make the appropriate Print Manager calls in each case. You should also set up Frame to create a print record and call `PrDefault`. For now, assume that your program does not save the print record with a document, and set things up appropriately.

Be sure the print commands are dimmed if no document is open.

The Print Loop

Once the print record has been filled in by calls to the various Print Manager subroutines it's time to actually print the document. The print loop is a little complicated, so we'll take it in stages.

Let's start by summarizing the entire print loop. After that, we'll look at the various steps in detail, learning about several fields in the print record along the way. Finally, we'll look at an actual print loop.

The print loop is broken up into two parts. In the first part, you make the calls that actually draw to the printer port. With some printers, it's possible to spool the printing. Spooling means that the printing isn't actually done when you issue the various drawing commands; instead, all of the information is saved to a disk file, then the printing is done after you've finished. That's why there is a second part of the printing process. After you've finished with the drawing part of the loop, you need to tell the Print Manager it's time to open the spool file and actually start printing. Of course, if spooled printing wasn't used, the Print Manager knows that and ignores the call.

Here's an outline of the print loop:

1. Call `PrOpenDoc` to tell the Print Manager to get ready to print.
2. For each page in your document, do the following:
   a. Call `PrOpenPage` to get the Print Manager ready for a new page.
   b. Draw the page, just as you would draw the information in a window.
   c. Call `PrClosePage` to tell the Print Manager you're through with this page.
3. Call `PrCloseDoc` to tell the Print Manager you're finished printing.
4. Call `PrError` to see if there were any errors while printing. If not, call `PrPicFile`, which will print the document if it was spooled.

Opening the Printer's GrafPort

The first step in actually printing a document is to open a `grafPort`. Opening a `grafPort` for the printer is a lot like opening a window in the sense that both of these actions give you someplace to draw. The only difference from your perspective is how you open the `grafPort` and how you go about finding out things like how big the drawing area is. In both cases you use exactly the same drawing commands. In fact, in a lot of programs, you might even call the same update routine you use to update the document window to draw to the printer's `grafPort`.

You open a printer `grafPort` with `PrOpenDoc`, like this:

```
prPort := PrOpenDoc(dPtr^.prHandle, nil);
```

The value returned is a `grafPortPtr`, just like a window, so you declare `prPort` as a `grafPortPtr`. Naturally, one of the things you pass is the print record that has been set up by all of the Print Manager calls we've talked about up to this point. The last parameter is the `grafPortPtr` you want to use; in almost all cases, you just pass nil, telling `PrOpenDoc` to create a new grafPort just for printing the document.

Page Sizes and Multiple Pages

Up until now, we've treated the print record as a black box; something the Print Manager sets up and uses, and that we don't need to know anything about. There is one thing we have to delve into the print record to find out, though, and that's the size of the page. The reason, of course, is that we need to know whether the document we're printing will fit on one page, or whether we need to break it up into multiple pages.

The information we need is actually imbedded in a subrecord inside the main print record. The subrecord, called `prInfo` in the print record, is of type `prInfoRec`; the type is defined like this:

```
   prInfoRec = record
       iDev:  integer;
       iVRes: integer;
       iHRes: integer;
       rPage: rect;
       end;
```

The first field doesn't really interest us; it tells what kind of printer is being used. The two most common values are 1, for an ImageWriter; and 3, for a LaserWriter. The next two fields give the vertical and horizontal resolution of the printer in pixels per inch. There are a lot of ways this information can be used, depending on what is being printed. One common example would be a CAD program, which needs to draw a 1 inch square when you tell it to, no matter what printer is used. The CAD program could use these fields to find out how many pixels to move in a particular direction to draw a 1 inch line.

The last field, `rPage`, is the one we can't do without. This rectangle defines the actual printable area on a page. If our document is wider than `rPage.h2-rPage.h1` we have to print two pages wide, and if the document is taller than `rPage.v2-rPage.v1`, we have to print more than one page high. In some cases, you might even want to use these values to figure out how much information you will let the user put in the document. A common example is a word processor, which would make sure that the page wasn't any wider than `rPage.h2-rPage.h1`.

Figuring out just how to map a document onto a printed page depends a lot on the kind of program you are writing, so I'd strongly recommend packaging the process into a subroutine. That makes Frame a lot easier to change, since all of the information you need to change to print a document is in one neat little subroutine. The only other change you might have to make is for programs that can save the print record with a document, in which case you have to add the code to save, load, and validate the print record.

Here's a simple version of the subroutine that assumes the document record has a field called `docRect` which defines the total size of the document, and also assumes that you want to print the document with one screen pixel mapping to one printer pixel.

```
procedure GetPageCount(dPtr: documentPtr; var h, v: integer);

{ Figure out how many printer pages are needed for the document }
{                                                               }
{ Parameters:                                                   }
{    dPtr - document to print                                   }
{    h - (returned) number of horizontal pages in the document  }
{    v - (returned) number of vertical pages in the document    }

var
   infoRec: prInfoRec;                  {printer info record}

begin {GetPageCount}
HLock(dPtr^.prHndl);                    {get a copy of the info record}
infoRec := dPtr^.prHndl^^.prInfo;
HUnlock(dPtr^.prHndl);
pwidth := infoRec.h2 - infoRec.h1;      {get the page size}
pheight := infoRect.v2 - infoRec.v1;
with dPtr^.docRect do begin             {get the document size}
   dwidth := h2 - h1;
   dheight := v2 - v1;
   end; {with}
h := dwidth div pwidth;                 {get the page counts}
if h*pwidth < dwidth then
   h := h+1;
```

```
v := dheight div dheight;
if v*pheight < dheight then
   v := v+1;
end; {GetPageCount}
```

Listing 7-2: Sample `GetPageCount` Subroutine

In Frame, we'll just set h and v to 1, and remember to replace these values with more appropriate calculations when Frame is adapted for a specific program.

The Main Print Loop

You make a pass through the print loop once for each page that needs to be printed. In the print loop itself, you do three things: call `PrOpenPage` to start a fresh page, draw all of the information for that page, and call `PrClosePage` to tell the Print Manager you are finished with the page.

The first step is to call `PrOpenPage`. This is a pretty simple call:

```
PrOpenPage(prPort, nil);
```

The `prPort` parameter, of course, is the printer `grafPort` returned by the call to `PrOpenDoc`. The second parameter can point to a rectangle, which the Print Manager will use to automatically scale your drawing. If you aren't using the automatic scaling feature (and we won't in this course) the parameter is set to nil.

Before you start printing, be sure and set up your drawing tools. Most of the time, that means calling `PenNormal` right after the call to `PrOpenPage`.

The next step is to draw the information that should appear on this page of the document. By "this page," I'm assuming that you're taking into account which page is actually being printed. Naturally, you should loop over each of the horizontal and vertical pages the call to `GetPageCount` told you to take into account.

There's one other touchy issue about printing multiple pages, and that's that the Print Manager doesn't shift your image over for you. For example, let's assume you're printing a 3 page by 3 page document, like the one shown in Figure 7-1. If you're printing the middle page, it's up to you to subtract the height and width of one sheet of paper from the horizontal and vertical coordinates used for each drawing command.

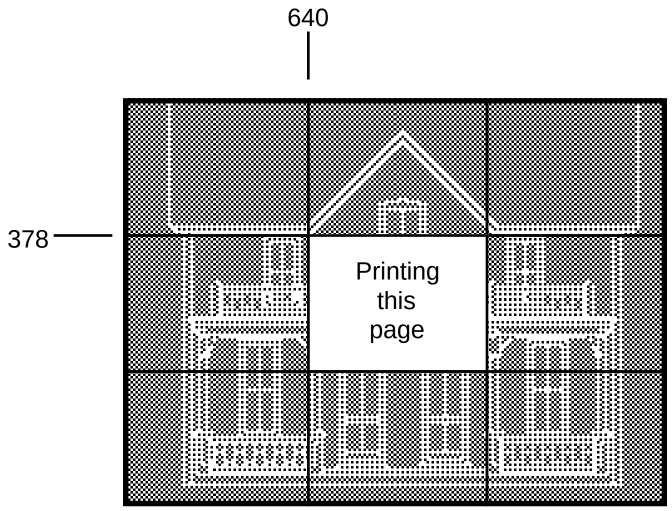


Figure 7-1: Shifting for Multiple Page Documents

Let's assume that the Print Manager tells us that a page is 378 pixels high, and 640 pixels wide. If the document is a drawing a picture that is 1134 pixels high and 1920 pixels wide, we might end up with something a lot like the document we just looked at in Figure 7-1. In that case, when we printed the top left page, everything would work just fine. The entire picture won't be on the page, but the Print Manager figures that out and only prints the part of the picture that should be printed. When we print the next page to the right, though, it's very important to subtract 640 from each horizontal coordinate. For the page being printed in Figure 7-1, we would subtract 640 from each horizontal coordinate, and 378 from each vertical coordinate.

The simplest thing to do when you draw a page is to just draw the entire document and let the Print Manager figure out what should be on the page and what should be ignored. That works fine for a lot of programs, but if it takes a long time to do all of the drawing commands, you might want to to some preprocessing yourself. For example, if you are writing a word processor that will print a 100 page book, you don't want to force QuickDraw II to reformat all 100 pages of text each time a single page is printed. In a case like that, you should go to the trouble of skipping over the lines that have already been printed, then stopping when you get to the first line that will be on the next page.

Once a page is printed, call `PrClosePage` to tell the Print Manager you're through with the page:

```
PrClosePage(prPort);
```

Finishing the Print Process

Once all of the pages have been printed in the main print loop, close the document.

```
PrCloseDoc(prPort);
```

If the user didn't ask for spooled printing, or if the Print Manager ignored the user because the printer being used doesn't support spooled printing, this is actually the end of the print loop. Just in case the document has been spooled, though, you should keep going. The next step is to check to see if there were any errors flagged during the print loop with a call to `PrError`. If not, you call `PrPicFile` to print the document.

```
var
   status: PrStatusRec;

...

if PrError = 0 then
   PrPicFile(dPtr^.prHandle, nil, @status);
```

The first parameter is the print record, which is natural enough. The next parameter is the `grafPort` you want the Print Manager to use while printing the spooled file. This is *not* the same `grafPort` as the one used when the drawing commands were used in the main print loop; that one was closed by `PrCloseDoc`, and doesn't exist anymore. By passing nil , we're telling the Print Manager to create a working `grafPort` for itself. The last parameter points to a status record. This record is filled in constantly while the Print Manager prints the spooled document, and you can set up a heartbeat interrupt task that will watch this record and keep the user informed about what is happening. That's a lot of trouble for a very meager return on the amount of effort you have to expend, so we won't get into that here. All you have to do is make sure you pass a pointer to a valid status record.

Problem 7-3: Add the print loop code to Frame. Use a default `GetPageCount` subroutine that always says the document is one page high and one page wide.

While you aren't printing anything yet, the printer should still cycle, printing a blank page each time you ask the program to print a document. If you like, add code to print an X.

Problem 7-4: Add printing to the slide show program you developed as a solution to Problem 6-4. Keep things simple by assuming that the entire picture will fit on a single page. With most printers, it will.

A Few Tips and Tricks

There are several odd little things you should know about the Print Manager that I won't go into in detail in this course. I'll cover a few of them in this section so you know the features exist, but for details you'll have to refer to *Apple IIGS Toolbox Reference: Volume 1*.

Printing a Subrange

It's possible for the user to pick a subrange of pages to print. For example, if someone is using a word processor to write a book or long paper, they might want to print just a few pages from a larger document. The Print Manager lets the user pick a subrange of pages, and it will automatically skip printing the pages that the user didn't ask for. It's possible for you to poll some fields in the print record to find out if the user asked for a subrange of pages, and with some appropriate tricks, not print the pages that need to be skipped at all. In some kinds of programs a trick like that could save a lot of time, while in other kinds it just isn't worth the effort, since drawing the document doesn't take much time anyway.

Aborting with Open-Apple Period

While the Print Manager is printing a document, it checks for an open-apple period abort key. If the user holds down the open-apple key and presses the period, the Print Manager will eventually figure that out and stop printing, flagging an error. If you like, you can draw an alert window with an appropriate message (like "Press open-apple period to cancel") before you drop into the print loop, then close the window after the print loop finishes.

The 10K Check

`PrPicFile` needs one fairly good size chunk of memory, namely one 10K chunk. There are some situations where `PrPicFile` will fail because it can't get this block of memory, but it could have succeeded if you had known in advance that there wasn't enough memory. This generally happens when your program uses chunks of memory itself for temporary results. For example, a paint program often keeps a large buffer or two open for something called an off-screen grafPort. This memory could be freed up if your program knew it was needed.

If your program falls into this category, you should try to allocate a 10K block of memory before calling `PrPicFile`. If you can't get the memory, stop and dispose of some of your program's temporary buffers, and try again. Of course, if your program doesn't have any temporary buffers it can dispose of, you can just call `PrPicFile` and let it flag an error and return.

Page Gaps and Ribbon Printing

One of the options the user can pick with the ImageWriter printer is "No Gaps Between Pages," which does pretty much what it says. When the user picks this option, the Print Manager gives you a vertical resolution for the page that goes literally from one fold in the paper to the other. The neat thing about this option is that it gives you a way to print continuous ribbons from a large document. For example, you can print a sign sideways, printing a continuous picture that runs right across page gaps. You can also print a large house plan in a CAD program so the user only has to paste the strips together, and didn't have to tape the drawing together in individual pages.

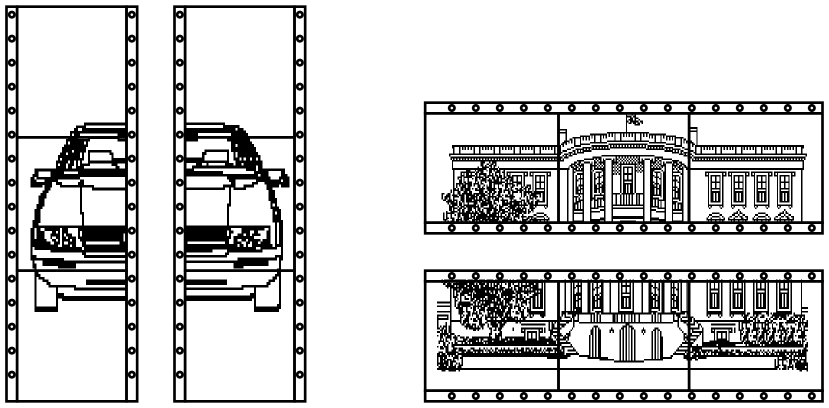


Figure 7-2: Ribbon Printing in Portrait and Landscape Modes

To make this work, of course, you have to print the pages in a particular order, and the order changes depending on whether the document is being printed in portrait or landscape mode. In portrait mode, you should print the document in columns, printing an entire column from top to bottom and then moving on to the next column. In landscape mode, the document it printed sideways, so you have to switch the printing order, too, printing a row from left to right, then moving to the next row.

Summary

This lesson covered the basics of printing from desktop programs using the Print Manager. Along the way we finished the Frame program, which gives you a great start on almost any desktop program, and learned a new way to start the tools.

Tool calls used for the first time in this lesson:

```
PrDefault          PrJobDialog        PrCloseDoc         PrClosePage
PrError            PrOpenDoc          PrOpenPage         PrPicFile
PrStlDialog        PrValidate         ShutDownTools      StartUpTools
SysFailMgr
```

Resource types used for the first time in this lesson:

```
rToolStartup
```

Lesson 8 – Thanks for the Memory

Goals for This Lesson

In past lessons we've made use of Memory Manager calls like `NewHandle` and `HLock`. In this lesson, we'll learn about these calls in more detail, exploring the organization of memory on the Apple IIGS and how to effectively use the Memory Manager to get the most out of that memory. We'll also find out what a handle really is, and look into the advantages and disadvantages of the Memory Manager compared to the standard memory management functions of Pascal, new and dispose.

Memory on the Apple IIGS

I'd like to start off with a quick run down on how memory is organized. I'm doing this mostly to define a few terms, but also to remind you of some computer related facts you don't have to use often in Pascal. If you've never heard of a byte, bit, or hexadecimal number, this is probably going to be a little to quick. In that case, I'd recommend looking for a good run down on bits, bytes, binary math, hexadecimal math, and memory organization in a book that goes into a little more detail. One good source for information like that is an introductory book on assembly language. Bits and bytes are the bread and butter of assembly language programming, so the beginner books dive into it right away, and generally have a pretty complete description.

Like most modern computers, the memory of the Apple IIGS is organized into addressable units called bytes. Each byte of memory contains 8 bits, and each bit is an on-off switch. With a little math, it's easy to see that a byte can hold up to 256 distinct values.

Each byte of memory has a specific address. The very first byte of memory has an address of 0, the next byte has an address of 1, and so on. We generally use hexadecimal math when we talk about memory addresses, mostly because it's easy to see the individual bytes of the address, which is important for reasons we'll talk about in a moment. The Apple IIGS uses the 65816 CPU. The 65816 has a three-byte address, and there are two hexadecimal digits in each byte, so we'd write the addresses as $000000, $000001, and so forth. In practice, we actually use a four byte chunk of memory to store an address, but the fourth byte is not used by the 65816, and rarely written when we're talking about addresses. We just assume it's zero.

Because the addresses for bytes of memory are stored in bytes, the computer has some natural chunk sizes for memory. A page of memory is all of the memory with the same two values in the first two bytes. Another way of looking at a page of memory is that it's 256 consecutive bytes, numbered $xxxx00 to $xxxxFF, where xxxx has to be the same value. There are some advantages to keeping some kinds of memory aligned to a page boundary. The most common is to align something called the direct page register to a page boundary, which makes some kinds of machine language programs run just a bit faster. Some instructions in the 65816 instruction set also run a tad slower when the memory you are using crosses a page boundary. Finally, many of the tools ask specifically for memory in page-size chunks, and require that the pages start at an even page boundary. The Memory Manager has to be able to handle all of this, so it has a flag that lets you tell it to give you a chunk of memory that starts on a page boundary.

The next natural chunk of memory is called a bank of memory, and it's all of the memory with the same first byte. For example, $xx0000 to $xxFFFF would be one bank of memory. A bank of memory is 256 by 256, or 65536 bytes long; we call this 64K, where one K (or Kilobyte) is 1024 bytes. Banks are important for a lot of reasons. First, the registers on the 65816 hold two bytes, which means that the computer is a lot more efficient when it deals with information that is in the same bank of memory. Second, a program can't lie across a bank boundary. Please notice that I did not say a program can't be in several banks, or that a program can't be bigger than a bank of memory – it can. The restriction is that a program can't be put in memory so any one piece of it crosses a bank boundary, so all of the pieces must be smaller than 64K, and the all have to be in a specific bank. The Memory Manager can give you memory that is all in one bank.

Memory is certainly not created equal. The 65816 has some special instructions which only work in something called direct page, and direct page memory has to come from bank zero. (Bank zero is the bank with addresses of $000000 to $00FFFF. Bank $E1 would be the bank of memory from $E10000 to $E1FFFF, and so on.) The hardware stack on the 65816 is also restricted to bank zero. Finally, the Apple IIGS uses a lot of memory in banks $00, $01, $E0 and $E1 for special purposes. The only memory that's completely free for use by a program is the memory that starts at bank $02, and continues up from there.

Using the terms from the toolbox reference manuals, the memory that can be used for anything is called "allocatable memory."

Some memory has a special use, but can be used for something else if you don't need it for the special use. A good example is the chunk of memory from $E12000 to $E19FFF, which is where the pixels on the screen are stored; that's the only place the screen can go, but if you aren't using the super high resolution graphics screen, you can use that chunk of memory for something else – like your program. All of this memory is called "special memory," and of course you want to avoid it if you can, just in case it's needed later for its special purpose. The Memory Manager will let you avoid allocating special memory.

Memory which can't be used for anything but a very specific purpose, like the language card area or memory mapped I/O areas, is called "unmanaged memory." The Memory Manager simply won't give you that memory, and you should never try to make use of it unless you know exactly how the Apple IIGS uses that memory, and follow all of the rules imposed by the operating system.

Handles

In Pascal, C, and most other languages, when you ask for a chunk of memory, you get back a pointer to the first byte of the chunk you get to use. The pointer, of course, is an address of a byte of memory. The Apple IIGS Memory Manager doesn't give you a pointer, though. Instead, you get back something called a handle. Back when we first started making Memory Manager calls, I told you it was safe to think of a handle as a pointer to a pointer, which is true up to a point. (Pun intended.) Actually, a handle is not just a pointer to a pointer, it's a pointer to a record, and the first thing in the record is a pointer. This is an important distinction, as you'll see in a moment.

Why Handles?

Before we go any further, though, let's stop and see why the Apple IIGS uses handles instead of pointers. After all, with all of the confusion and extra code needed to deal with handles, there must be a darn good reason for using them instead of a pointer.

Let's play a simple mind game, allocating and deallocating memory. We'll start off and allocate 10 chunks of memory, each 5 bytes long. Once we're finished, the memory is divided up like you see in Figure 8-1.

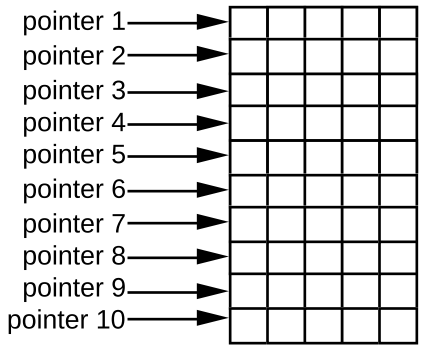


Figure 8-1: Memory After Initial Allocation

This is a pretty typical state of affairs in a program that uses dynamic memory, but as you know, it's also fairly common to dispose of memory, and the memory isn't always disposed of in a neat order. For example, if the program ends up disposing of every other chunk of memory, we end up with memory looking like Figure 8-2.

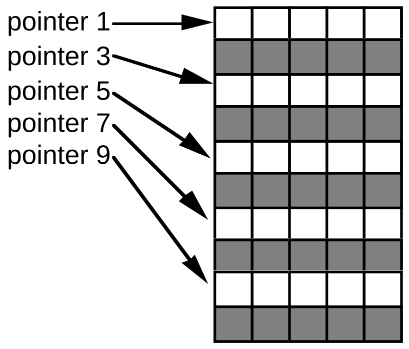


Figure 8-2: Memory After Deallocation

At this point, half of the memory is actually free, for a total of 25 free bytes of memory, but the larges piece of memory available is only 5 bytes long. If a program asks for a 6 byte piece of memory, it can't get it – and the program might easily fail with an out of memory error, even though there's more than four times as much memory available as it asked for.

The problem, of course, is that the memory that is available isn't in a single place. The solution would be to move the allocated pieces of memory together so the 25 free bytes are all in one place, but if we do that, all of the pointers in the program are invalid – they all point to the wrong place.

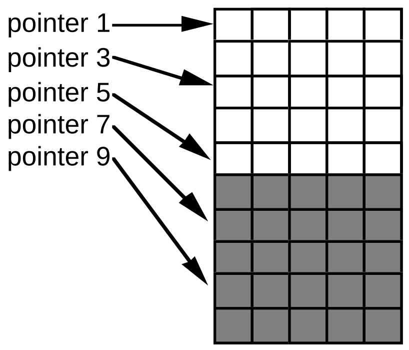


Figure 8-3: Memory After Compaction

Handles solve this problem very neatly. Instead of keeping a pointer to the chunk of memory, the program has a pointer to a chunk of memory that, in turn, points to the memory. That way, when the Memory Manager moves the memory around to combine the smaller pieces, the program doesn't get messed up. After all, the program doesn't keep a pointer to the memory at all, it keeps a pointer to a pointer, and the Memory Manager is free to update the actual pointers as the memory is moved around.

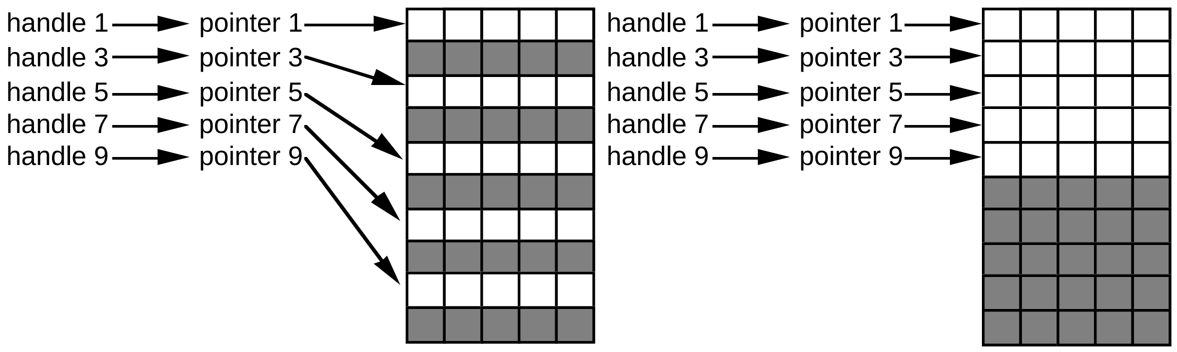


Figure 8-4: Compaction with Handles

Locking Memory

Of course, there are some problems with this scheme. The obvious problem is that there are a lot of cases when you don't want something to move. If you know much about machine language programs, you can imagine the havoc it would play with a program to move the program itself, or even the program's data, while the program was trying to run! Even for your own dynamic memory, there may be times when you need to keep pointers to the memory, and it would blow your program out of the water if something moved. Finally, even if it's OK for a chunk of memory to me moveable most of the time, it needs to hold still while you are trying to read or write to the area – and on a machine like the Apple IIGS, which has a lot of interrupt handlers running in the background, it's quite possible for something to make a Memory Manager call that will shift memory around even if you aren't making any tool calls in your subroutine. In short, the ability to move chunks of memory around to combine small free pieces into larger ones is pretty neat, but there are a lot or problems with this scheme.

The Memory Manager handles all of this fairly smoothly with the concept of locked memory. A locked chunk of memory can't be moved. If you'll think back, we've been locking and unlocking memory all along, and this is why we've been doing it. When you are about to read or write to a chunk of memory through a handle, the first step is to lock the memory. If you like, you can then get the actual pointer; that's called dereferencing the handle, and in some cases it will make your program run a lot faster. Finally, when you're through with a piece of memory, you unlock it so the Memory Manager can move it again if it needs to.

So far things look pretty good, but what if you accidentally unlock a chunk of memory that should *never* be moved, like a chunk of memory that holds some executable code? To prevent that, the Memory Manager lets you ask for memory that is fixed, which basically means that it is always locked. You can still use `HLock` and `HUnlock`, and you won't get an error, but the Memory Manager will never move the memory, even if it is unlocked.

Purgeable Memory

Another really useful feature of the Memory Manager is it's ability to mark memory as purgeable. When memory is purgeable, it's still there, but the Memory Manager is allowed to get rid of it if someone asks for memory, and there isn't enough around to satisfy the request.

It may seem strange to have such a thing as purgeable memory, and in a lot of programs you'll never use this idea, but you're still getting the advantage of purgeable memory all of the time. One way you can take advantage of purgeable memory is to write programs that don't depend on the initial values of global variables. Basically, that means never assuming that a global variable really starts off with a value of 0. (Just for the record, the Pascal language doesn't guarantee the initial value of variables anyway, and technically it's an error to read the value of any variable in Pascal until you've actually assigned some value to the variable. ORCA/Pascal just doesn't enforce that restriction.) When you do this, your program can be restarted. What that means is that when you leave the program, it stays in memory – and if the user restarts your program before anything else uses the memory, it starts up from memory, rather than taking the time to reload the program from disk.

Another place this idea is used is in the ORCA development environment. When you load a file from disk to edit it or compile it, then close the file, the ORCA shell doesn't actually get rid of the file. Instead, the file is written to disk, but it stays in memory, too. Then, when you edit the file again, the copy that's already in memory can be used, saving a lot of time.

These are the sorts of things you might use purgeable memory for in your own programs, too. Basically, any time you have temporary information – something you could get from disk or recalculate if you need it again – you should store the information in purgeable memory. That way, if someone else needs the memory, your purgeable chunk can be used. In all likelihood, whoever used the memory will free up the memory again before you need it. All you have to do is check the handle to make sure the memory hasn't been purged before you lock it. And, incidentally, locking the memory also prevents it from being purged, so the information is safe until you unlock it again.

We'll look at the calls to mark memory as purgeable, as well as looking at some details about purge levels and the like, a little later in this lesson.

How Memory is Allocated

All of this is starting to get pretty involved. Memory can be fixed or moveable, purgeable or not purgeable, locked or unlocked – with all of these options, it might seem like it would be pretty easy to end up with memory that was almost as fragmented as with pointers. Well, it would be – except for a convention the Memory Manager uses when it allocates memory, and that you have to follow to make things work smoothly.

If your program deals exclusively or mainly with fixed chunks of memory, and does a lot of allocating and deallocating in no particular order, it won't be long before memory gets pretty fragmented. Also, the current Memory Manager won't change the order of chunks of memory; it just pushes moveable chunks as far towards the end of memory as it can when it needs to make room. Because of this, if moveable and fixed memory is more or less randomly scattered through memory, you'll still fragment the memory pretty quickly, since the moveable chunks won't be moved around the fixed chunks.

Obviously, things will work better if the fixed memory is kept near the bottom of memory, and doesn't have many gaps; and if moveable memory is kept near the top of memory, and doesn't have any fixed memory stuffed in the middle. That's just what the Memory Manager tries to do for you. If you ask for fixed memory, the Memory Manager gives you the first chunk of memory it can find that meets all of your requirements, starting its search from the lowest memory address and working up. If you ask for moveable memory, the Memory Manager scans from the end of memory, and works towards the beginning.

You have to help, though. If you are asking for a fixed piece of memory, you should make sure it's a chunk that will stay around for a while. If you won't need the memory long, it might be best to ask for a moveable piece of memory and just keep it locked. That way, the fixed memory area won't get fragmented. And, if you ask for moveable memory, you should do everything you can to keep it moveable as much as possible.

Writing Efficient Programs

One of the problems with the Memory Manager is that it doesn't deal with huge numbers of handles very well. As your program, the tools, and the various desk accessories and Inits start to allocate memory, some operations will slow down noticeably compared to a configuration that uses fewer handles. To keep this problem at bay, I'd recommend allocating a few large chunks of memory instead of a lot of small ones. That's what ORCA/Pascal does when you use the `new` and `dispose` procedures, so if you need a lot of small pieces of fixed memory, just use `new` and `dispose`. You get better type checking, anyway.

So What's a Handle?

Even up until this point, everything we've said about a handle is that it's a pointer to a pointer. For the most part, that's exactly how you should think of it. If you stop and think about it, though, you can see that the Memory Manager needs to know a lot more about a chunk of memory than where it is at. The Memory Manager has to keep track of whether the memory is fixed or moveable, whether it's locked at the moment, and whether the memory is purgeable. It also needs to know how big the chunk of memory is, whether it has to remain in a fixed bank, whether it is page aligned or bank aligned, and even who owns the memory. All of that information, and more, is in the record the Memory Manager uses for each and every handle. The first field in that record is certainly a pointer to the memory, but that isn't all that's in the handle. So what is actually in the record? Well, to put it bluntly, it's none of our business. Except for the first field, the format for a handle is private, and only the Memory Manager knows or cares just what the record looks like. That's good in most respects, because it means Apple can change and improve the Memory Manager without worrying (much) about breaking other programs.

The point, though, is that a handle is really not just a pointer to a pointer. If some tool call asks for a handle, you darn well better give it one. Don't assume that it's OK to just pass the address of a pointer, or even a pointer to a pointer. Even if you try it and it works, the toolbox may change in some future version, and break your program. If the description of a tool call says you are supposed to use a handle, accept no substitutes: pass a handle that was actually created by the Memory Manager.

Allocating Memory

With all of the newfound knowledge, let's go back and take a closer look at the `NewHandle` call, and especially at the flags word. The `NewHandle` call looks like this:

```
hndl := NewHandle(size, userID, attributes, location);
```

The size is just the number of bytes you want, and of course the result is a handle. (If there is an error, nil is returned.) It's the last three parameters that can be more useful than you would expect from what was mentioned in earlier lessons.

The `userID` is a unique number assigned to your program when it was started. You can actually ask for your own user ID, but it's rarely a good idea to do that. The reason is simple: the program launcher that starts your program knows its user ID, and when your program finishes, the program launcher can make sure all of its memory is deallocated, even if your program messes up. If the program allocates another user ID of its own, the program launcher can't double-check to make sure all of the memory gets deallocated.

On the other hand, it's nice to be able to set aside some user ID's for yourself, mostly because there are a few Memory Manager calls that can work on a whole group of handles, doing something to all of the handles with a particular user ID. A common example is `DisposeAll`, which disposes of all of the handles that use a particular user ID at the same time. You can get four distinct sub-user ID's to help make efficient use of the calls that work on more than one handle without resorting to allocating an entirely new user ID. All you have to do is or the main user ID with $0100, $0200, or $0300. In fact, ORCA/Pascal does exactly that – when you use ORCA/Pascal's `UserID` function, it returns the program's main user ID ored with $0100.

The attributes flag is the parameter that gives you the most control over the way memory is actually allocated. This parameter is a series of bit flags.

15	`attrLocked`	This bit is set when a handle is locked, and clear when it is not locked. It's pretty handy when you're allocating memory, since you can allocate the memory in an initial locked state, saving yourself the trouble of an explicit call to `HLock` to lock the new handle.
14	`attrFixed`	This bit is set for fixed memory, and clear for moveable memory. Both bits 14 and 15 must be clear before the Memory Manager will actually move the memory.
13-10		Reserved; set to 0.
9-8	`attrPurge`	These two bits define the initial purge level, which can be 0 (00), 1 (01), 2 (10) or 3 (11). Purge levels are described later in this lesson.
7-5		Reserved; set to 0.
4	`attrNoCross`	If this bit is set, the Memory Manager will make sure that all of the memory is in a single bank of memory. This can be pretty important for some applications. For example, you should always set this bit if you are bypassing Pascal's `new` and `dispose` calls, but will be using Pascal to access arrays or pointers in the memory. A common mistake is to ask for more than 64K of memory but set this flag. Since a bank of memory can only hold 64K, there is no way to allocate more than 64K without crossing a bank boundary, and the Memory Manager returns an out of memory error.
3	`attrNoSpec`	If this bit is set, the Memory Manager won't give you any special memory.
2	`attrPage`	If this bit is set, the Memory Manager will make sure the chunk of memory it returns starts on a page boundary. Another way of looking at it is that the last two digits of the address will be 00.

		The most common use for this flag is when you are starting tools without the assistance of `StartDesk` or `StartUpTools`. Many of the tools ask for page aligned chunks of memory from bank zero, and you need to set this flag to satisfy the "page aligned" part of the requirement.
1	`attrAddr`	The toolbox documentation is a little misleading when it describes this flag; it says the flag forces the memory to *remain* at a fixed address. Actually, this flag forces the memory to be *allocated* at a particular location; specifically, wherever the last parameter, `location`, points. The `attrFixed` flag is the one that causes a chunk of memory to stay at a fixed location once it is allocated. You should almost never use this flag. The only excuse is when you need a specific piece of memory that is set aside for a particular purpose. For example, QuickDraw II would use this flag to allocate the screen buffer for the super high resolution graphics screen, which has to be at $E12000. Some animation programs might also use this flag to allocate the shadow buffer.
0	`attrBank`	If this flag is set, the memory will come from a specific bank, but it can come from anywhere in the bank. The most common use for this flag is also for allocating direct page space when initializing tools. When this flag is set, the last parameter, `location`, should point to the proper bank. It doesn't matter where in the bank `location` points; the last two bytes of this address are ignored.

The last parameter, `location`, is only used if `attrAddr` or `attrBank` is set.

Most of the time you'll ask for memory with a flags word of $8000 or $8010, giving you a moveable chunk of memory that starts locked. The later case of $8010 also makes sure the memory doesn't cross a bank boundary. The next most common values would be $C000 or $C010, which gives you a chunk of memory that will never be moved. In rare cases, you might ask for memory with a flags word of $C015, which says that the memory is fixed, may not cross a bank boundary, is page aligned, and comes from a particular bank. That's the correct setting for allocating memory from bank zero when you're initializing a tool. For example, the Print Manager needs one page of memory, so you would get it like this:

```
   printHandle := NewHandle(256, userID, $C015, pointer(0));
```

Even better is to let `StartUpTools` start the Print Manager so you don't have to worry about this sort of detail.

Purgeable Memory

At the start of the lesson we talked about a lot of the different things the Memory Manager can do with memory, and one of these was to mark memory as purgeable. The Memory Manager is allowed to dispose of purgeable memory if it needs to to try and satisfy a `NewHandle` call.

Of course, some things are more important than others, so there are purge levels. Memory that can't be purged has a purge level of 0. Purge levels of 1, 2 and 3 are successively more likely to be purged; the Memory Manager will purge all of the memory with a purge level of 3 before it gets desperate and starts in on the memory with a purge level of 2. Purge level 3 is actually reserved for the System Loader, which uses it for restartable programs.

You can set the purge level when the memory is originally allocated, or you can set the purge level later with `SetPurge`:

```
SetPurge(level, myHandle);
```

Either way, the memory won't be purged if it is locked.

So what happens if the memory is purged? Well, the handle is still there, but the pointer is nil (which has a value of 0). So, if you are using purgeable memory, always start by locking the memory, then check to see if the pointer is nil. If it is, you need to reallocate the memory and recalculate or reload whatever should be in the chunk of memory.

Finding Free Memory

There are three Memory Manager calls that can give you some idea of how much memory is available, although none tell you enough to tell if a particular request is going to succeed. The first is `FreeMem`, which doesn't use any parameters, and returns the total number of free bytes. This does not include memory that can be made free by purging; it only counts the bytes that are actually unused when `FreeMem` is called.

If you want to know how much memory is available, use `RealFreeMem`. Like `FreeMem`, this call doesn't need any parameters, and returns a longint value that is the number of free bytes. This time, though, the call also counts any bytes that are in an unlocked, purgeable chunk of memory.

`FreeMem` and `RealFreeMem` tell you the total amount of free memory, but they don't give you a clue about how that memory is organized. The most useful single number about how memory is organized is the largest single piece of free memory, which is returned by `MaxBlock`. Like the other calls, it doesn't need any parameters. `MaxBlock` returns the size of the largest piece of free memory. Like `FreeMem`, it doesn't take into account purgeable memory, so you might be able to allocate a bigger block of memory than `MaxBlock` says is available.

In the end, these numbers don't mean much to a running program. The only way to tell if you can get a particular chunk of memory is to try. These calls can be useful, though, to tell the user how much memory is lying around.

Problem 8-1: Change Frame so the about box shows the total free memory (as returned by `RealFreeMem`) and the largest block of free memory (as returned by `MaxBlock`). Hint: Use substitution strings.

Summary

This lesson has covered the concepts behind the Memory Manager. You learned what a handle really is, how memory is managed by the Memory Manager, how to use purgeable memory, when to use the Memory Manager and when to use Pascal's `new` and `dispose` procedures, and how to find out how much free memory is available.

It's worth repeating something that I pointed out at the start of the course. This is an introductory toolbox course, and there is no attempt at covering all of the tool calls available. There are a lot of other Memory Manager tool calls, and it would be a great idea to browse through *Apple IIGS Toolbox Reference*, volumes 1 and 3, to see what else is available.

Tool calls used for the first time in this lesson:

```
FreeMem            MaxBlock            RealFreeMem        SetPurge
```

Lesson 9 – Drawing on the Front Side of the Screen

Goals for This Lesson

This lesson concentrates on the graphics capabilities of the Apple IIGS. Of course, we've been using a GUI (Graphical User Interface) all along, but up until now we've concentrated on using existing tools to draw menus, windows, and so forth. This lesson looks at the graphics package for the Apple IIGS, QuickDraw II. It also looks at the hardware capabilities of the Apple IIGS that allow it to display colors, and how you can select colors from the palette of 4096 that are available. Finally, we'll look at a few topics, like regions, that add a lot of power to the basic drawing commands of the Apple IIGS.

Super High Resolution Graphics

The Super High Resolution Graphics Screen

The Apple IIGS toolbox always draws using super high-resolution (SHR) graphics. This graphics mode uses a single 32K screen buffer, located at $E12000 in the Apple IIGS's memory. There are actually three distinct parts of the screen buffer, all of which are read by the display hardware of the Apple IIGS to draw the colored pixels you see on the screen.

The first chunk of memory is a 32000 byte buffer that contains the actual pixels that are displayed on the monitor. The buffer is arranged as 200 scan lines, one for each of the lines of pixels the Apple IIGS can display. The lines are arranged top to bottom, so that the top scan line is at $E12000, the next scan line at $E120A0, and so forth.

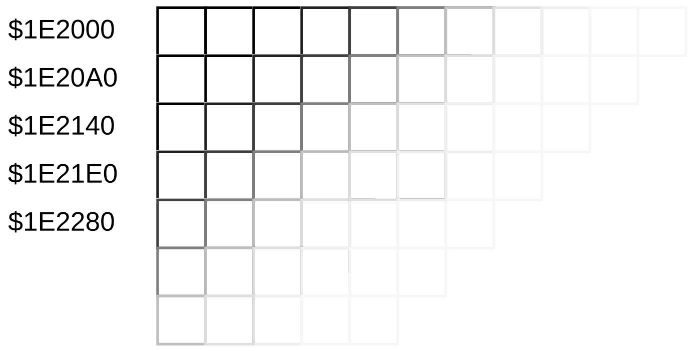


Figure 9-1: Mapping of Scan Lines

In the 640 graphics mode we've used for most of our programs, each of the 160 byte scan lines contains 640 pixels. The pixels are arranged in a simple, linear map, with the leftmost pixel in the most significant two bits of the first byte of the scan line, and so forth across the screen. This gives four pixels in each byte, with two bits per pixel. In 320 mode, each pixel uses four bits, so only two pixels fit in each byte of the scan line.

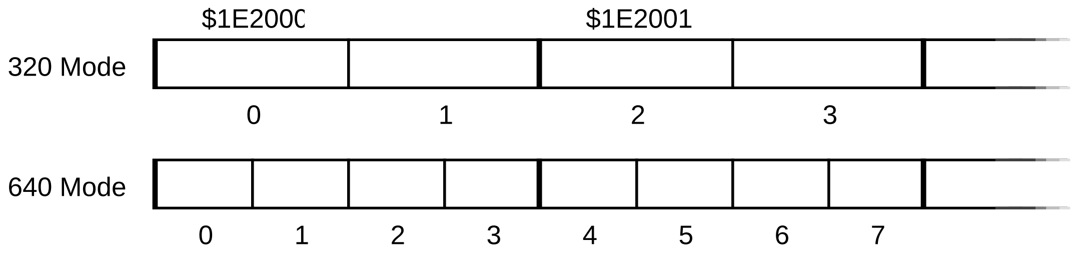


Figure 9-2: Mapping of Pixels in a Scan Line

Right after the 32000 bytes that map to pixels are 200 additional bytes, one for each scan line. Once again, a simple linear order is used: the first byte corresponds to the top line of pixels on the screen, the next byte to the second line, and so on. These bytes are the scan line control bytes, and each one has control of a number of different characteristics for the scan line. We'll talk about the scan line control bytes in detail later, after we've looked at colors.

The third area of the screen buffer is a series of 16 color tables, each needing 32 bytes. These are pretty involved, so we'll break the discussion of colors out into a separate section.

If you've been counting, we're still 56 bytes short of 32K. The extra 56 bytes are at the end of the 200 scan line control bytes, right before the color tables. The extra 56 bytes aren't used for anything, although a few QuickDraw commands will set these bytes in the process of changing the rest of the scan line control bytes.

Colors in 320 Mode

Each pixel on the 320 mode graphics screen is set using four bits of memory, so it's easy enough to see that you can set any one of 16 values. You might expect that each of these values would correspond to a particular color, and you'd be right, up to a point. Each value does correspond to a particular color, but that color is not fixed. Instead, the Apple IIGS uses a color table, with a two-byte entry for each of the 16 possible pixel values.

Each of the two byte entries in a color table is broken up into four four-bit fields, like this:

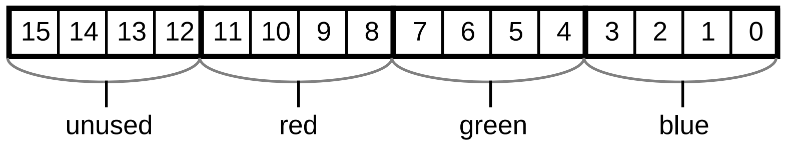


Figure 9-3: Master Color Value

Each of these color words is called a master color value. The first four bits aren't used, but the next three groups of bits control the red, green, and blue intensity of the dot, with 0 turning the color off, and 15 giving the brightest value possible for the color. If you're at all familiar with the physics of color, you'll recognize these three colors as the primary colors; you can get (almost) any color by mixing these three colors together. If you're not familiar with the physics of color, and you'd like an introduction that is fascinating, easy to read, and very informative, I'd suggest Chapter 35 of *The Feynman Lectures on Physics*. The title is probably pretty scary, but this particular chapter is the only short article I've ever read which is easy to understand and still manages to explain things like where brown comes from. (Think about it: you can mix any color from a primary color, right? Then where is brown in the rainbow?) Anyway, the book is famous, and should be available in any library.

A color table has 16 of these master color values, one for each of the 16 possible values of a pixel. You can set each master color value to anything you'd like – for a fun treat, try creating an

Init that periodically sets all of the colors to $0000 (black) or inverts all of the color values (making the screen look like a color negative).

To change the colors used to display the screen, you use two QuickDraw II calls, `GetColorTable` and `SetColorTable`. Each takes two parameters, the color table number and a variable with a type of `colorTable`. The `colorTable` type is defined in Common.pas:

```
     colorTable = array [0..15] of integer;
```

As a general rule of thumb, you should always set color 0 (the first element of the color table) to black ($0000) and color 15 to white ($0FFF). If you don't the screen will look, well, strange. These are the colors used by the toolbox to draw window frames, erase the window to white, and so on.

Here's the color table that you get if you don't make any changes:

Number	Color	Value	Number	Color	Value
0	black	$0000	8	beige	$0FA9
1	dark gray	$0777	9	yellow	$0FF0
2	brown	$0841	10	green	$00E0
3	purple	$072C	11	light blue	$04DF
4	blue	$000F	12	lilac	$0DAF
5	dark green	$0080	13	periwinkle blue	$078F
6	orange	$0F70	14	light gray	$0CCC
7	red	$0D00	15	white	$0FFF

Figure 9-4: Standard 320 Mode Colors

The Scan Line Control Byte

I mentioned that there are 16 color tables, which seems pretty silly at first. After all, there are only 16 possible values for a pixel, and each color table has 16 color entries, so why do we need 16 color tables? The reason is that you can actually pick a different color table for each scan line. In fact, you can pick all sorts of things for each scan line.

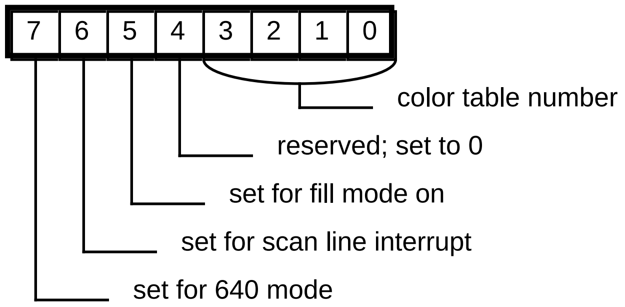


Figure 9-5: The Scan Line Control Byte

Bits 6 and 5, used for scan line interrupts and fill mode, are generally used by assembly language programmers doing advanced animation programs; in any case, we won't get into them in this course. The two values you should know about are the `scbColorMode` bit, which tells you if the scan line is in 320 or 640 mode, and the color table number. There are a couple of fairly useful tricks you can do by manipulating these SCB values directly:

1. Since the SCB for each scan line is recorded in a dump of a picture file, you can check the most significant bit of the SCB to figure out if a picture should be displayed in 320 mode or 640 mode.

2. You can use the QuickDraw II call `SetAllSCBs` to change all of the scan line control bytes at once. One useful feature is to quickly change the color table that you are using, something you could do to switch between the colors for two different pictures. Here's a quick example that changes a 320 mode screen so it uses color table 1, instead of the default color table of 0:

```
SetAllSCBs($01);
```

3. The Apple IIGS can only display 16 colors, right? Well, yes and no. You can only display 16 colors on a single scan line, but since you can use a different color table on each scan line, you can actually use up to 256 colors at once. This isn't very useful in most desktop programs, but in a game it might be fun to use one color table for a control panel at the bottom of the screen, and another color table for a picture at the top of the screen. You could even use 640 mode in one area for crisp text, and 320 mode in another area for great color pictures!

   Here's a quick sample that shows how to use color table 0 for the top half of the screen, and color table 1 for the bottom half:

```
for i := 0 to 99 do begin
   SetSCB(i, $00);
   SetSCB(i+100, $01);
   end;
```

There are all sorts of creative ways you can use these multiple color tables and the ability to use different resolutions on different lines. In fact, some clever assembly language programmers have even written programs that switch the color tables *while the Apple IIGS is drawing the screen!* That's a bizarre thing to do to a defenseless piece of silicon, but the result is a staggering ability to display pictures with virtually any of the 4096 colors the Apple IIGS can display, all at the same time. Unfortunately, while it's pretty, this isn't very useful – all of the computer's time is used up switching color tables, rather than running a program to use all of those colors.

Problem 9-1: Starting with Frame, create a color explorer. Your program should open a single window with no close box, no scroll bars or grow box, and no zoom box. You should also remove all of the commands except Close and Quit from the file menu. Close is there to support desk accessories; your program should ignore this command. In short, the user only gets one window.

Switch the program to use 320 mode graphics.

Next, add three new menus, labeled Red, Green and Blue. Each should have 16 entries, the numbers 0 to 15. These are the levels of each color, and the current level should be checked in each menu.

Inside the window, draw four rectangles. Along the left third of the screen, draw three rectangles , one on top of the other. Paint these rectangles with colors 1, 2 and 3. Set each of these colors to a pure color from the color menu. For example, if the user picked a value of 3 for red, set color 1 to $0300.

The right two-thirds of the screen should be painted with color 4, which, of course, is set to the combination of all three colors.

This is a pretty simple program, but it's great fun. Using this program, you can try all sorts of color combinations quickly.

Colors in 640 Mode

Compared to 320 mode graphics, 640 mode graphics swaps increased resolution for a decrease in the number of colors that can be displayed. Instead of using four bits per pixel, with 16 distinct colors, 640 mode uses two bits per pixel, giving 4 distinct colors.

If that was all there was to the story, you would be limited to four colors per scan line in 640 mode, but as it turns out, there are some tricks you can use to get more than just four colors at a time. The trick is called dithering, and it relies on the fact that alternating colored dots in 640 mode get mixed to produce a third color. For example, if you draw alternating black and white pixels across the 640 mode screen, you'll see a gray screen, not a striped one. Considering how crisp black and white text looks in 640 mode, that may be a hard one to swallow. Frankly, I didn't believe it myself. Just for fun, give it a try with this program:

```
program Dither (input);

uses Common, QuickDrawII;

var
   color: integer;                         {pen color}
   i: integer;                             {loop variable}

begin
StartGraph(640);
color := 0;
for i := 0 to 639 do begin
   SetSolidPenPat(color);
   MoveTo(i, 0);
   LineTo(i, 199);
   color := color ! 3;
   end; {for}
readln;
EndGraph;
end.
```

Listing 9-1: A Simple Dithering Demonstration

Without dithering, the result would be a black and white striped screen, but as you can see, you get a gray screen.

At first blush, it looks like the color table used for 320 mode graphics is a bit large for 640 mode graphics. After all, 640 mode only needs the first 4 entries, not all 16 like 320 mode graphics. Apple's engineers went for another trick, though, that extends the dithered colors quite a bit. Instead of using the first four entries in the color table, each of the pixels in a byte uses a different set of four colors. You can certainly set each of the minipalettes up to have the same four colors, and in fact black and white are normally in all four of the minipalettes, but you can also mix the colors. Since the screen is dithered, that gives you a great mix of colors. By default, you get black and white for colors 0 and 3 in all four minipalettes. Pixels in odd columns are colored blue for color 1, and yellow for color 2; while pixels in even columns are colored red and green for colors 1 and 2. Painting a large area with color 1 mixes the blue and red to give an annoying purple, while color 2 mixes the green and yellow to give a sickly green. Naturally, you can change the color tables to get other dithered colors.

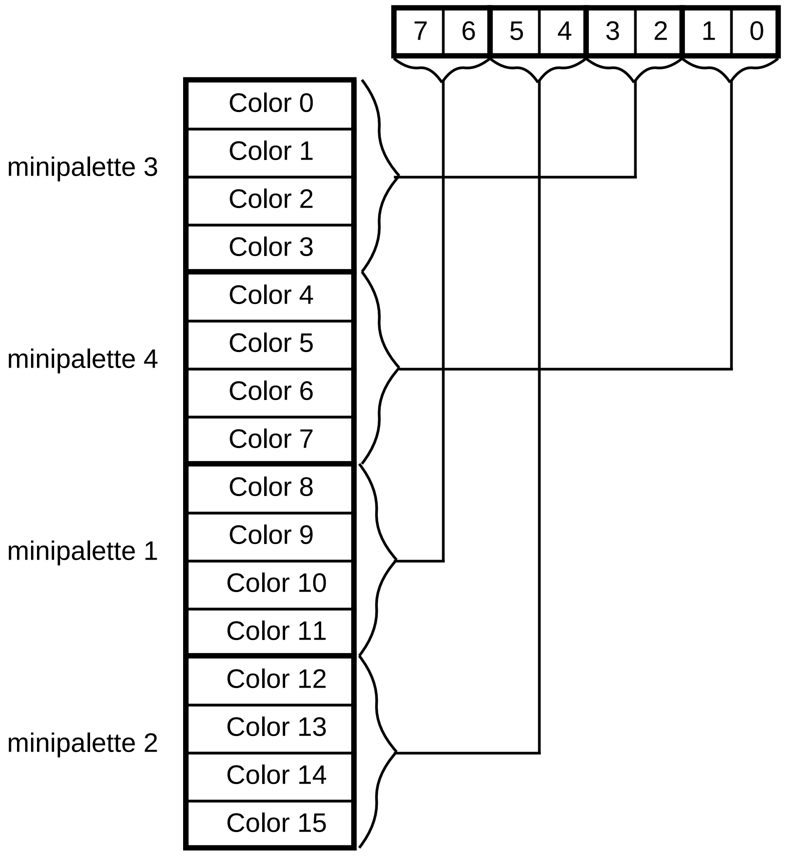


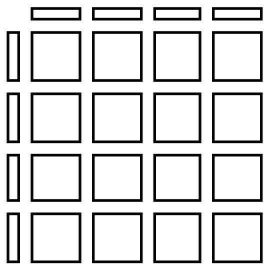
Figure 9-6: Minipalettes in 640 Mode Graphics

Dithered colors are pretty neat; they give you a way to get 640 pixels on a scan line for precision, and still get 16 solid colors for things like graphs or colored icons. There are some disadvantages to dithering, though. The big one is that you can't draw colored objects with very fine detail, since the colors tend to mix. Another problem is that an object will shift colors a bit if you move it over one pixel, since the color table alternates by pixel position. You can avoid that problem by setting all of the minipalettes to the same colors, or by making sure that you always move pictures right and left by even multiples of four pixels. That's one of two reasons why I said to always be sure you tell `TaskMaster` to scroll your window left and right in multiples of eight pixels. The other reason has to do with pen patterns, which we'll talk about in a moment.

The last problem with dithered colors is that it's really tough to plan a color palette. With 320 mode graphics, you could pick any 16 colors you want; but with 640 mode, each color in a palette affects several dithered colors. Frankly, I have such a tough time with planning 640 mode color palettes that I almost always use the default color table. If you want to create a new 640 mode color table, I'd suggest trial and error as a good way to go about it. While it won't be assigned as a problem, a good way to do a lot of trials and see those errors quickly would be to write a 640 mode color explorer, sort of like the 320 mode color explorer from Problem 9-1.

Problem 9-2: Write a short program to display all 16 possible dithered colors. This isn't a complicated desktop program; it's just a quickie program to see the actual dithered colors for yourself. You can base it on the Dither program from this section.

The screen will be divided into boxes like the ones you see below.



Color the small rectangles using colors 0 to 3 from left to right along the top row, then using 0 to 3 again from top to bottom along the left column. The larger boxes in the middle should be drawn using a combination of the colors from the pure color to the top and left of the box. You should alternate colors every other vertical line, just like the Dither program from this section alternated black and white to get gray.

Screen Dump Files

One of the graphics file formats for the Apple IIGS is just a snapshot of the screen memory. It's a pretty good format for simple paint programs or slide shows, but of course it limits a picture to the size of the screen, and there are a lot of cases where that isn't big enough.

Screen dump files have a file type of $C1 and an auxiliary file type of $0000. If that looks familiar, it's because the pictures you've loaded in a few of the problems have used these screen dump files.

Problem 9-3: Now that you know the format for a screen dump file, go back and add two features to finish the slide show program you wrote in Problem 7-4. First, add checks to make sure the picture can be displayed by your program. Specifically, you should make sure only one color table is used, and that all of the SCBs are for 320 mode. If not, warn the user that the picture may not look quite right. Next, read the color table used by the program and set the current color tables to use the correct colors for the picture.

Be sure you can handle pictures with different color tables! An easy way to do this is more or less the way you handle menu dimming, by simply setting the correct color table each time through the event loop. In a time-critical program you would have to get a bit more sophisticated, but this method will do here.

You can test your program on the pictures in the Pictures folder. They use several different color palettes.

Setting the Pen Size

Most of the QuickDraw drawing commands use the idea of a pen. You've used this pen already: positioning it with `MoveTo`, setting the color of the pen with `SetSolidPenPat`, and so forth. As it turns out, there's a lot more to the drawing pen than you might think. In addition to setting the size and color of the pen, you can also set the pen pattern so it can draw something besides just a solid color. You can even change the way the pen works, telling it to reverse the color in bits instead of drawing normally.

Up until now, we've only used the pen to draw lines, or perhaps set the pen color for some of the commands like `PaintRect` that fill a large area with the pen color. In all cases, we've used the default pen size of 1 pixel by 1 pixel that is set by `PenNormal`. You can actually set the size of the pen to a lot of other values, though. In fact, the pen can be as large as eight pixels by eight pixels. One of the most common reasons to change the size of the pen is that horizontal lines are thicker in 640 mode than vertical lines, since each pixel in 640 mode is a little more than twice as high as it is wide. When I'm doing line drawings in 640 mode, I almost always set the pen to be one pixel high and two pixels wide. `SetPenSize` does the trick:

```
 SetPenSize(2, 1);                                {use a pen that is two pixels wide}
```

Pen Patterns

We've used `SetSolidPenPat` to change the color of the lines and objects we draw with the pen, but you can also change the pen pattern. With a pen pattern, you set up an eight by eight array of colored dots, using literally any color you like.

The pen pattern itself is actually a tiny picture, eight pixels wide and eight pixels high. Since pixels are not the same size in 640 mode and 320 mode, this means that the pen pattern is also not the same size. In fact, a pen pattern in 320 mode is 32 bytes long, while a pen pattern in 640 mode is only 16 bytes long.

You can set up the bytes for the pen pattern a lot of different ways. In ORCA/Pascal, a pen pattern is defined as an array of bytes, like this:

```
      pattern = array [0..31] of byte;
```

(The definition was pulled from the interface file Common.pas.) What `SetSolidPenPat` actually does is to set all of the pixels in a pen pattern to the same color.

Here's one way to set up a pen pattern. In 320 mode, this pen pattern will give you alternating lines of red (color 7) and white (color 15, or $F), with the lines running vertically.

```
   for i := 0 to 31 do
      myPattern[i] := $F7;
   SetPenPat(myPattern);
```

When most people see pen patterns for the first time, it seems like they would be a great way to draw small pictures: just set up a pattern, then do a `MoveTo`, `LineTo` at a single point. That sort of works, but the problem is that patterns are fixed to a very specific grid system. To visualize how they work, imagine painting over patterned wallpaper, then erasing part of the paint. You see the original pattern showing through, but nothing short of ripping off the paper will shift the pattern over an eighth of an inch. Patterns work the same way: they are fixed to the physical coordinate system of the graphics screen itself. That's why you want to make sure you always scroll left and right by multiples of eight pixels, and why `TaskMaster` makes sure your window moves by multiples of eight pixels. If you moved or scrolled the window by two pixels, then drew part of a pattern again, the old and new areas would clash where they joined together.

Pen Modes

The normal thing to expect when you draw a black line across a white screen is for a black line to appear. That's what QuickDraw does most of the time, but there are a lot of cases when it is very useful to have QuickDraw do something else entirely. The most useful is to have it reverse the bits, something called exclusive oring the bits. You can do that, and a lot more, by setting the pen mode with `SetPenMode`.

Actually, this isn't the first time you've seen the `SetPenMode` call. Way back in Lesson 1 you learned to put a `SetPenMode(modeCopy)` call into your program to make sure QuickDraw did what you normally want it to do: just copy the colors you tell it to onto the screen.

`modeCopy` is the first of four drawing modes. The others are `modeOR`, which tells QuickDraw to overlay the bits that are being drawn with the ones already on the screen; `modeXOR`, which tells QuickDraw to perform an exclusive or on the bits being drawn and the ones already on the screen; and `modeBIC`, which flips all of the bits in the pen, then ands them with the image. If you're familiar with truth tables and boolean logic, you'll recognize most of these operations. Figure 9-7 shows how they work when you use each of the pen modes to draw a white X across a white box.

The color white has all of the pen bits set, so white would be the equivalent of true in boolean logic, while black would be false. Of course, some judicious changes in the color table could change those meanings!

Each of the pen modes also has a "not" version, where all of the bits in the pen are reversed before drawing. Figure 9-7 also shows these "not" modes.

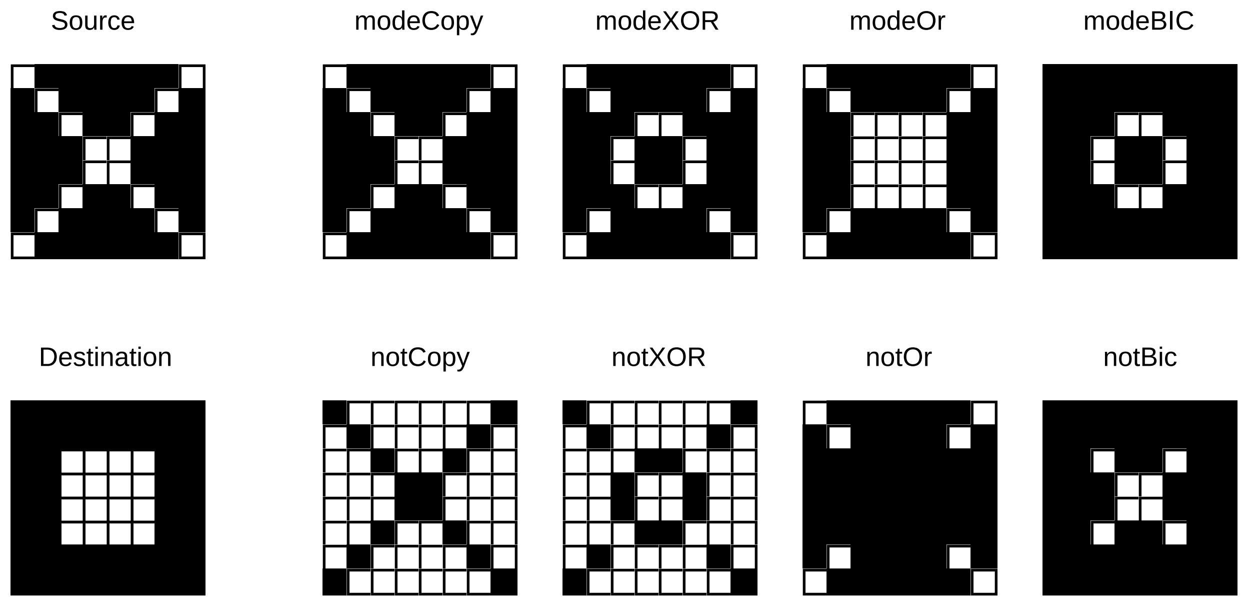


Figure 9-7: Pen Modes

It's very important to keep in mind that these operations are bitwise operations. The distinction doesn't matter when we're looking at black and white pictures, where drawing a white pixel on a white background in `modeXOR` will make the pixel black. It matters a lot when we use colors, though. For example, if you use `modeXOR` to draw a white pixel in 320 mode (value $F, or all four bits set) on a red background (value $7, or 0111 in binary), the result is a beige pixel (value $8, or 1000 in binary).

Rubber-Banding

`modeXOR` has a very useful feature. If you draw the exact same image twice in `modeXOR`, you end up drawing the image and then erasing it. It doesn't matter how complicated the picture is that you draw, or how complicated the background is, or even what colors are involved: draw the image once and it appears, draw it again and it vanishes. We can put this technique to work for something called rubber-banding.

When you use a drawing program to draw a line, you start by pressing the mouse at one end of the line, then you drag until you get to the other end. While you're doing this, you see the line that you would get if you let up on the mouse button – move the mouse a little, and the line gets redrawn in the new spot. That's what rubber-banding is. You get to "stretch" a line or object from one point to another. The way it's done is with `modeXOR`, drawing the image once so you see it, and again to erase it. You'll put this to use a bit later in a program.

It messes up the event loop to try and handle rubber-banding in the main even loop, so it's best to create a second event loop that handles the rubber-banding, and only exits once the user is finished dragging the shape around. For such a simple task as following the mouse, though, it doesn't make sense to set up a full-fledged event loop, where you would have to allow for all of the various possible kinds of events. Instead, you can use two calls, `StillDown` and `GetMouse`, to track the mouse. `StillDown` checks to see if the mouse button is still down, returning true if it is and false if it isn't. You stay in your rubber-banding loop as long as `StillDown` returns true. `GetMouse` reads the position of the mouse (in local coordinates!). Putting these two commands to work, here's a simple loop that will handle rubber-banding for a line. This subroutine is a little more complicated than it has to be, mostly because it bothers to check to see if the mouse has really moved before redrawing the line. This avoids a nasty problem in simpler rubber-banding subroutines, where the line can flicker while the mouse is held still. Also, this subroutine just handles the rubber-banding – it's up to the caller to draw the finished line after this subroutine returns. The caller is also responsible for setting up the pen mode it wants to use; this subroutine leaves the pen mode in `modeXOR`.

```
procedure RubberBand (start: point; var finish: point);

{ Entered to draw a line in rubber-band mode.                    }
{                                                                }
{ Parameters:                                                    }
{    start - initial point on the line                           }
{    finish - (output) end point for the line                    }

var
   newPoint: point;                      {mouse position}

begin {RubberBand}
{get ready to rubber-band}
finish := start;
PenNormal;
SetSolidPenPat(15);
SetPenMode(modeXOR);
{draw the initial line}
MoveTo(start.h, start.v);
LineTo(finish.h, finish.v);
{rubber-band loop}
while StillDown(0) do begin
   GetMouse(newPoint);
   if (newPoint.h <> finish.h) or (newPoint.v <> finish.v) then begin
      MoveTo(start.h, start.v);
      LineTo(finish.h, finish.v);
      finish := newPoint;
      MoveTo(start.h, start.v);
      LineTo(finish.h, finish.v);
      end; {if}
   end; {while}
{erase the line}
MoveTo(start.h, start.v);
LineTo(finish.h, finish.v);
end. {RubberBand}
```

Listing 9-2: Rubber-Banding

Pen Masks

The pen mask is another eight pixel by eight pixel pattern, just like the pen itself. Unlike the pen, though, the pen mask is a simple bit map, one bit per pixel in the pen. If a bit in the pen mask is set, the corresponding bit is drawn on the screen, but if the pen mask bit is clear, that pixel isn't drawn on the screen.

Probably the most common use for pen masks is to draw something in an unhighlighted or unselected state. A good example of this is a dimmed menu item. By setting the pen mask to a checkerboard pattern, and then drawing something, only every other pixel is actually drawn. That's exactly what you normally see for dimmed menus.

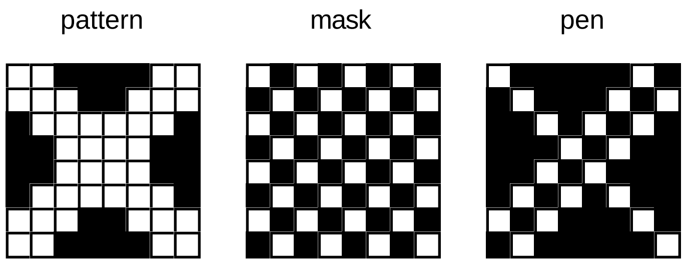


Figure 9-8: Pen Mask

Here's a sample section of code that could be used to set up a dimmed pen mask. To set the pen back to normal, you could either use `SetPenMask` a second time with a mask set to all $FFs, or you could use `PenNormal`.

```
procedure DimmedMask;

{ Set the pen mask for drawing dimmed items                     }

var
   i: integer;                          {loop variable}
   myMask: mask;                        {pen mask}

begin {DimmedMask}
for i := 0 to 3 do begin
   myMask[i*2] := $55;
   myMask[i*2+1] := $AA;
   end; {for}
SetPenMask(myMask);
end; {DimmedMask}
```

Listing 9-3: Set the Pen Mask for Dimmed Drawing

Drawing Shapes

We've used `PaintRect` to fill in large, rectangular areas of the screen, and occasionally to clear a window or even draw a rectangle. It's almost that easy to draw a lot of other shapes on the screen, too. In addition, you can do a lot more than just fill them in with the current pen pattern, like `PaintRect` does. Each shape can also be filled; filling works a lot like painting, but you give QuickDraw a pattern along with the shape, so you don't have to set the pen pattern before drawing the object. You can also erase a shape, which fills in the shape with the background color for the window (generally white). Shapes can be inverted, which flips all of the bits in the shape. Inverting a shape has the same effect as setting the pen color to white, setting the pen mode to `modeXOR`, and then drawing the shape – but you don't have to set the pen mode or color first, and you don't have to change them back when you've finished drawing. Finally, you can frame a shape, which outlines the shape using the current pen color and size. Framing a shape is just like using `MoveTo` and `LineTo` to outline the shape.

In the sections below we'll talk about each of these shapes in detail. Here's a quick map before we start, in the form of a typical picture of each of the basic shapes.

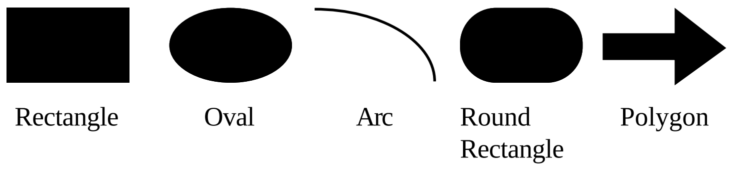


Figure 9-9: The Basic Shapes in QuickDraw II

Rectangle

```
procedure EraseRect (r: rect);
procedure FillRect  (r: rect; p: pattern);
procedure FrameRect (r: rect);
procedure InvertRect(r: rect);
procedure PaintRect (r: rect);
```

You've already seen and used rectangles. The `rect` record gets filled in with the top, bottom, left and right dimension of the rectangle, then you make the appropriate call.

There is one subtle point about how rectangles are drawn that is real easy to miss, and very unexpected (at least to me). When you paint this rectangle:

```
   r.h1 := 1;  r.h2 := 10; r.v2 := 1; r.v2 := 10;
```

you might expect to see a picture that looks like the one you would get from this loop:

```
   for h := 1 to 10 do begin
      MoveTo(h, 1);
      LineTo(h, 10);
      end; {for}
```

Well, it's not. Actually, `PaintRect`, `EraseRect` and `FillRect` all fill the *interior* of the rectangle outlined by the points in the `rect` record. That means the right and bottom edge are filled up to, but not including the coordinates you give. In short, the rectangle really looks like the one drawn by this loop:

```
   for h := 1 to 9 do begin
      MoveTo(h, 1);
      LineTo(h, 9);
      end; {for}
```

Framing a rectangle works the same way, so framing our example rectangle would produce a drawing that would look like this one:

```
   SetPenSize(1, 1);
   MoveTo(1, 1);
   LineTo(9, 1);
   LineTo(9, 9);
   LineTo(1, 9);
   LineTo(1, 1);
```

There's one other subtlety, though. The idea is that `FrameRect` should only affect pixels that are entirely inside the boundary set by the rect record. What happens when you set the pen width to something bigger, say 2 by 2? The answer is that QuickDraw moves in as many pixels as it needs to. With a pen size of 2 by 2, you would have to pass 8 to `MoveTo` and `LineTo` to mimic `FrameRect`.

While I'm not going to assign a problem about drawing rectangles, if you are not quite sure what I mean about rectangles being drawn inside the border you set up in the rect record, I'd suggest writing a few short programs to try out the loops from this section, along with a few more that use `FrameRect` with pen widths other than one by one. You can use a short, graphics only program like the Dither program from earlier in this lesson, so it isn't that hard.

Oval

```
procedure EraseOval (r: rect);
procedure FillOval  (r: rect; p: pattern);
procedure FrameOval (r: rect);
procedure InvertOval(r: rect);
procedure PaintOval (r: rect);
```

Ovals are basically squashed circles. Of course, if you draw a "square" oval, with the height and width the same size, you get a perfect, unsquashed circle.

The only thing you have to tell QuickDraw to draw an oval is how big it is. QuickDraw figures out all of the messy math and draws an oval. It even lives up to it's name, drawing the oval pretty quickly.

As with rectangles, the oval is entirely *inside* the border you set up in the rect record. For example, this code looks like it would draw a box around an oval:

```
   r.h1 := 1;
   r.h2 := 10;
   r.v1 := 1;
   r.v2 := 5;
   SetSolidPenPat(5);
   PaintOval(r);
   SetSolidPenPat(1);
   MoveTo(1, 1);
   LineTo(10, 1);
   LineTo(10, 5);
   LineTo(1, 5);
   LineTo(1, 1);
```

If you think it will, you should try it in a short program. What actually happens is the top and left lines hit the topmost and leftmost pixels, but the bottom and right lines are one pixel below and to the right of the oval.

Arc

```
procedure EraseArc (r: rect; startAngle, arcAngle: integer);
procedure FillArc  (r: rect; startAngle, arcAngle: integer; p: pattern);
procedure FrameArc (r: rect; startAngle, arcAngle: integer);
procedure InvertArc(r: rect; startAngle, arcAngle: integer);
procedure PaintArc (r: rect; startAngle, arcAngle: integer);
```

An arc is a chunk cut from an oval, more or less the way you would cut a slice of pie. You give a start angle and an arc angle, which tell QuickDraw how big of a pie slice you want, and QuickDraw draws just the part of the oval from the start angle to the start angle plus arc angle.

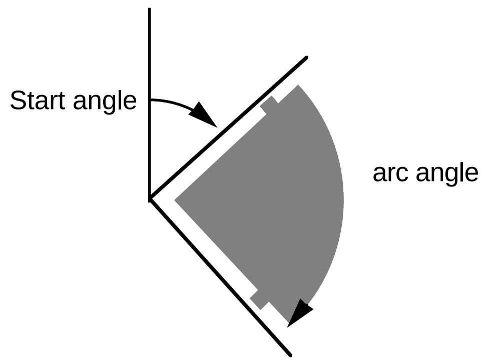


Figure 9-10: Angles in an Arc

If you've ever done much with angles on computers, you are used to dealing with radians instead of degrees. Well, surprise! You can never get to complacent, because waiting around the corner is some piece of software that blows away all of your conventions. The arc commands use angles specified in degrees. They also use compass style headings, with 0 degrees at the top of the circle, 90 degrees to the right, and so on.

Round Rectangle

```
procedure EraseRRect (r: rect; ovalWidth, ovalHeight: integer);
procedure FillRRect  (r: rect; ovalWidth, ovalHeight: integer; p: pattern);
procedure FrameRRect (r: rect; ovalWidth, ovalHeight: integer);
procedure InvertRRect(r: rect; ovalWidth, ovalHeight: integer);
procedure PaintRRect (r: rect; ovalWidth, ovalHeight: integer);
```

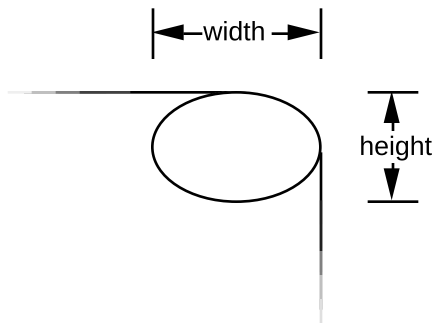


Figure 9-11: Oval Size in a Round Rectangle

A round rectangle is a rectangle with little quarter-ovals at the corners instead of the normal square corners. They're pretty cool for button outlines, and you see them sometimes for windows with a slightly rounded corner.

Of course, you have to tell QuickDraw just how round those corners should be. You do that by giving the height and width of an oval to take the chunk from. Remember, you're telling QuickDraw the total size of the oval, not the size of the rounded part of the corner!

Polygon

```
procedure ErasePoly (p: polyHandle);
procedure FillPoly  (p: polyHandle; p: pattern);
procedure FramePoly (p: polyHandle);
procedure InvertPoly(p: polyHandle);
procedure PaintPoly (p: polyHandle);
```

All of the shapes we've talked about so far are useful, but fairly simple. They are used for the vast majority of cases where you want to draw regular objects, like the outline of a window or button. Polygons are harder to use, but they fill the gap between these simple shapes and the industrial strength shapes you occasionally need for more complicated objects, like arrows in a scroll bar.

A polygon is a closed object, so it does eventually get back to it's starting point. A polygon is made up of lines, so you could always duplicate `FramePoly` with a series of `MoveTo`, `LineTo` calls. (And, unlike the other shapes we've been studying, this time you get exactly the same thing from `MoveTo` and `LineTo` as you would from `FramePoly` – none of the edges get shifted over.) The lines themselves can be as short as you like, so you could even create an oval with a polygon.

Of course, since you can put as many line segments as you want in a polygon, you can't use a simple data structure like the `rect` record to create a polygon, and that's the main drawback. You have to create a polygon by giving QuickDraw all of the points that make up the edge of the polygon. You start with a call to `OpenPoly`, which creates a new polygon record and gives you back a handle for the record. After that, you draw the object once with an initial `MoveTo` call to get to the first point, followed by a series of `LineTo` calls to outline the polygon. Once the polygon is outlined, you call `ClosePoly` to finish off the polygon definition. Creating an arrow like the ones on a scroll bar (but a little bigger!) would look like this:

```
   poly := OpenPoly;
   if ToolError <> 0 then
      poly := nil
   else begin
      MoveTo(50, 50);
      LineTo(100, 50);
      LineTo(100, 25);
      LineTo(140, 75);
      LineTo(100, 125);
      LineTo(100, 100);
      LineTo(50, 100);
      LineTo(50, 50);
      ClosePoly;
      if ToolError <> 0 then begin
         KillPoly(poly);
         poly := nil;
         end; {if}
      end; {else}
```

It's worth looking at this code fragment carefully. Polygons are created with dynamic memory records, so there is a very real chance that an error will pop up while you're creating one, especially if it's a very complex polygon and your program is running low on memory. In short, the error checking is very important, and you also need to make sure `poly` isn't nil before you try to use one of the polygon drawing commands.

Setting up the polygon is the hard part, though. Once the polygon exists, you can do all sorts of wonderful things with it very quickly. Filling, painting, erasing, inverting and framing all work fine, and quickly, too. Since drawing a polygon is done entirely inside QuickDraw, with no overhead for individual tool calls, even framing a polygon is faster than drawing the same thing with `LineTo` calls. In short, polygons are a great choice anytime a complex shape needs to be drawn more than once, or anytime you have to fill, paint, erase or invert a complex shape.

Once you are through with a polygon, you use `KillPoly` to get rid of the polygon QuickDraw created. You can see an example of `KillPoly` in the sample code fragment that we used to show how to create a polygon.

There is one other useful call that you should know about when you start using polygons, and that's the `OffsetPoly` call. Without `OffsetPoly`, you would have to create a separate polygon for each and every object you wanted to draw, in each and every position it might appear in. Obviously, that's pretty wasteful for something like a scroll bar arrow, which will be drawn in a lot of different places, but doesn't change shape. `OffsetPoly` lets you move a polygon to a new location without changing it's shape. You give QuickDraw the polygon and a horizontal and vertical offset, and QuickDraw moves the polygon.

```
{move a polygon to the right 5 pixels and down 10 pixels}
OffsetPoly(poly, 5, 10);
```

Problem 9-4: Write a short 320 mode "graphics only" program, like Dither, to explore polygons.

Start by filling the entire screen with gray (14), setting the pen mode to `modeCopy`, setting the pen color to blue (4), and creating the polygon you saw in the text – the one that draws a large arrow.

Now draw the polygon in different positions on the screen using the five polygon drawing commands. Use `OffsetPoly` to draw the five polygons without recreating the polygon each time. For `FillPoly`, set up a pattern with red and white vertical stripes.

Problem 9-5: For this program you will create a simple draw program that will let you draw lines, rectangles and ovals in any one of eight pen colors. You'll use rubber-banding to let the user see the shapes as they are drawn.

a. Switch Frame to 320 mode.

b. Add menus and menu items to let the user pick the pen color and select between lines, rectangles, and ovals. These three shapes are the three basic shapes the program will let the user draw. You should support the following colors:

0	Black	2	Brown	7	Red	6	Orange
9	Yellow	10	Green	4	Blue	3	Purple

c. Create a list in the document record that is sort of like the one you used for the point plotting program, but this time, the list needs to be a list of variant records, since each element can be a line, rectangle or oval, and you need to be able to tell the difference between them.

d. When the user presses the mouse button, look to see which shape is selected. Enter a drawing routine for the appropriate shape. In the drawing routine, use rubber-banding and framing to let the user create the shape. On exit, draw the new object and add it to the list for the document.

   I would suggest implementing and testing each of the three shapes individually. Do lines first, rectangles next, and ovals last.

e. Create an update routine that will draw all of the objects in the object list. Be sure and test overlapping objects, making sure that the object that was created first is drawn first. Otherwise the picture will change when the window is updated!

f. Add support for printing.

g. Add support for saving and loading files.

h. This is actually the first time in this course we've created a program that can load, change, and save a file, so be sure you handle this! If the user makes a change to a document, then closes it, double-check to see if the user really wants to close the document. You can do that with a three-button alert window with a caution icon. The question should be "Save changes to *0 before closing?" with buttons of Yes, No and Cancel. When the user quits, do the same sort of check on all of the documents, letting the user save the document, close the document, or cancel the quit operation.

Regions

If you look through the chapter on QuickDraw II in *Apple IIGS Toolbox Reference: Volume 2*, you will find one kind of object that I didn't cover in the last section: regions. I didn't include regions with the "basic" drawing objects for two reasons: regions are generally used for drastically different reasons than the other objects, and there are some restrictions on when you can do with regions.

Let's talk about that restriction, first. With some kinds of printers, notably the LaserWriter, you can't print a region. Any of the drawing commands we talked about in the last section will work just fine, drawing just the way you would expect, but regions don't work. Because of that restriction, it's a good idea to make sure that all of the parts of your document are drawn without using any regions.

That's a pretty severe restriction, and a pretty severe recommendation against using regions, too. If regions can't be used well, they must have a darn good reason for existing, and they do. To understand this reason, we'll have to explore what regions are.

Like polygons, regions are built up by drawing other objects. Unlike polygons, regions are a collection of solid areas, and they don't have to be connected. You can also have a hole in the middle of a region, something that doesn't work very well with polygons. (You can sort of fake a hole with a polygon by drawing a line from the outer edge to the inner hole, then retracing your line on the way out, but the connecting line shows up when you call `FramePoly`.)

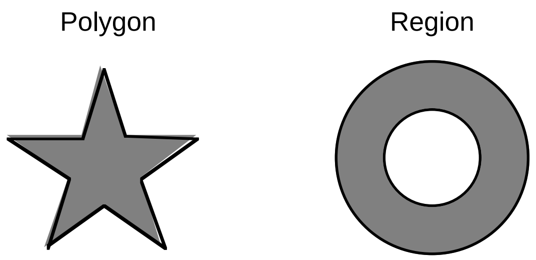


Figure 9-12: Comparing Polygons and Regions

By far the most common use for a region is to form a complex drawing mask, and that's the only use we'll look at in this course. That's not the only way to use regions effectively – they're also very handy for things like flood fills in a paint program – but I'll leave you to explore the other uses for regions on your own.

There are three very important regions associated with any window. Two are maintained by the Window Manager, and the other is one you can use for your own purposes.

The regions that are maintained by the Window Manager are called the visible region and the update region. The visible region is the part of the window's content region that you can actually see. Since widows can overlap other windows, and can hang partway off of the screen, the visible region is not always a simple rectangular area. The Window Manager keeps track of the visible region as you shuffle windows around or open and close windows.

Using the Update Region

The update region is the part of a window that needs to be redrawn. When a program is just sitting there, doing nothing, the update region is empty, but if you drag one window so it exposes part of a window that used to be covered, the newly exposed area has to be redrawn. The Window Manager automatically figures out what needs to be drawn, creating an update region that includes all of the pixels that need to be drawn. The Window Manager also tells the Event Manager that the window needs to be updated, and the Event Manager reports the update event to your program as soon as all of the higher priority events have been handled.

Just before `TaskMaster` calls your update subroutine, it calls `BeginUpdate`. `BeginUpdate` is a Window Manager call that temporarily replaces the visible region with the pixels that are in both the visible region and the update region. QuickDraw will never draw a pixel unless it is in a grafPort's visible region, so any drawing command outside of the area that needs to be redrawn is ignored. Once your update routine finishes updating the window and returns to `TaskMaster`, `TaskMaster` calls `EndUpdate`, which restores the original visible region for the window. All of this happens inside `TaskMaster`, so you normally don't have to worry about the process unless you are writing your own event loop without the aid of `TaskMaster`, but understanding how it all works will help you see how you can take advantage of this process to make your program simpler and faster.

The first way to take advantage of this process is to create programs that only draw in the update procedure. For example, in a simple version of the drawing program from Problem 9-5, all of the objects could be drawn in two places: once when the user draws the object for the first time, and once in the update routine. Another way to handle this is to not draw the object when the user creates it at all. Instead, you use the Window Manager's `InvalRect` call to tell it to mark the area where the object should be as invalid. The Window Manager will take it from there, making sure that the window gets updated at the next convenient moment. The actual drawing takes place in the update routine, so your program is simpler, and you have less chance of making a mistake or creating an incompatibility in the way the object is drawn in the two different places.

```
{accumulate the area covered by the rectangle r in the update region}
InvalRect(r);
```

Incidentally, you've seen `InvalRect` before. There was a brief discussion about it back in Lesson 5, but I though it would be a good idea to go over the ideas again, now that you know a little more about how the Window Manager and QuickDraw handle updates.

The second way to take advantage of the update region helps you draw the window faster. In a lot of cases – like after `TaskMaster` scrolls a window, for example – most of the visible region is perfectly OK, and doesn't need to be redrawn. Also, in a lot of programs, more of the document is actually outside of the window than inside. For example, you can only see a few lines from a book on the screen at any one time. Drawing the entire document, as we've done so far, wastes a lot of time. If you check first, and only draw the parts of the document that are in the update region, you can save a lot of time. In some kinds of programs, the time you save can be pretty substantial.

They key to using the update region is twofold: first, you have to find out what it is, and second, you have to check to see if the object you are about to draw is in the update region.

Getting the region that needs to be updated is pretty easy, but if you look through the toolbox reference manual, it can also be pretty confusing. The problem is that you don't really want the update region, and when you stumble across the Window Manager call to return the update region, it's real tempting to try to use that call. The reason you don't want the update region is because `BeginUpdate`, which is called by `TaskMaster` before your update routine is called, has already set the visible region to the area that actually needs to be updated, and then cleared the update region. In short, the update region is already empty when your update routine is called. What you want is the visible region; you get it with a call to `GetVisHandle`:

```
   visRegion := GetVisHandle;
```

With the region in hand (or at least in a handle), the next step is to test to see if the thing you need to draw is in the update region. There are several tool calls that can help, but I'm just going to cover the easiest one to use here. That's `RectInRgn`, which checks to see if there are any points in common in a rectangle and a region. If you pass the `visRegion` and a rectangle that surrounds the object you are about to draw, and `RectInRgn` returns false, you can skip drawing the object.

```
   if RectInRgn(objectRect, visRegion) then
      DrawMyObject;
```

It's important to use some common sense here. Testing to see if an object is in a region takes time. If you do this blindly, you could end up with an update region that takes *longer* than a simpler one that just draws everything. That's because you're playing a game of averages. To come out ahead, two things must be true: First, it has to take less time to check an object to see if it should be drawn than it does to draw the object. That's not always true. You may need to do some fancy footwork to get faster tests, doing things like sorting graphics objects or using information about the document itself, like the fact that lines in a text document come one after the other, so you can even skip the tests on a lot of objects. Second, the total amount of time you save on objects that are not drawn has to exceed the amount of time it takes to test the objects that end up getting drawn.

In a lot of cases, these conditions will hold some of the time, but not other times. For example, if you close a window, exposing a second window that was completely covered, the whole second window will have to be redrawn. Chances are, it will take longer with the "smart" update routine than with a simple dumb one that redraws the entire document. On the other hand, of you scroll the document down one line, the smart update routine could win big, since only a small part of the screen really needs to be redrawn. Since it's probably more important to update the screen quickly while scrolling than it is to redraw the screen quickly after the entire window has been uncovered, the program will seem faster. You've won the game of averages, even though, in a few specific cases, your program might be slower.

All of this really points out the need to use the old neural network stuffed between your ears for more than just remembering rules. You have to exercise some judgement, too. In most cases, you end up trying it both ways to see which method works best for a particular program. Another tried and true method is to try it the simple way first: if it works, and the program seems fast enough, there's no reason to bother with the more complicated update routine. If the program seems too slow, you can always go back and try the smart update method later.

Using the Clip Region

There's another region that QuickDraw maintains for every `grafPort` that can be very useful; it's called the clip region. You already know from experience that QuickDraw won't draw where it isn't supposed to. For example, if you try to draw outside of the visible part of the window, QuickDraw figures that out and doesn't put any pixels there. That's due to the fact that QuickDraw never draws outside of the visible region. As it turns out, QuickDraw is even more selective: not only does it not draw outside of the visible region, but it also doesn't draw outside of the clip region.

The clip region is for you to use. One of the most frequent uses is to mask off part of the screen so you don't accidentally draw on part of your window you want to leave alone. That can save you a lot of time and effort that you would otherwise spend checking to see if what you are about to draw will overwrite something that should be protected.

One simple way to set the clip region is with ClipRect, which sets the clip region to a particular rectangle. Everything outside of the rectangle is masked off, and QuickDraw won't draw to it. Of course, you need to make sure you fix the clip region when you are through; otherwise, scroll bars, palettes, and anything else in the window will never get updated! Here's a fairly typical way to use the clip region, assuming drawRect is a rectangle that includes the part of the window you want to draw to, and r is a work rectangle. The second call to ClipRect restores the clip region to the entire window.

```
ClipRect(drawRect);
{draw the contents here}
GetPortRect(r);
ClipRect(r);
```

There are several other commands that give you more detailed control over the clip region. While they can be very useful in a lot of specific cases, I'm not going to cover them here. After all, this is an introductory course! We don't need to get so bogged down in details we miss the overall picture.

Problem 9-6: Modify your drawing program from Problem 9-5 so it only draws an object if the object is in the update region.

GrafPorts

Throughout this course, we've talked about windows and grafPorts, and by now you're probably getting pretty used to the terms, not to mention how to make use of them. I'd like to finish off this lesson with a more detailed definition of these terms.

A grafPort is something QuickDraw II can draw to. In a simple graphics program, like the Dither sample from the beginning of this lesson, the grafPort is the entire screen. The clip region and visible region we've talked about are attached to a grafPort, and each grafPort has it's own clip region and visible region. There are a lot of other properties tied to a grafPort, with each one keeping separate copies, too. Pen characteristics are a good example. Changing your pen to modeXOR affects the current grafPort, but the pen mode for all of the other grafPorts doesn't get changed.

In theory, every grafPort covers the entire drawing plane, extending from -32767 to 32767 in each direction. In practice, the real drawing area is always smaller. It's contained in a locInfo record that gives, among other things, the actual size of the drawing rectangle. You saw locInfo records back when you drew a picture into a window with a single command, PPToPort.

Every window has a grafPort, and in fact, the record that defines a grafPort is the first thing you find in the record that defines a window. That's why we can play so fast and loose with grafPorts and windows: when you pass a window record to a QuickDraw command that expects a grafPort, it doesn't miss a beat, since the window record starts with a grafPort. In fact, in the header files for Pascal, the Window Manager calls are actually defined to accept grafPorts, not window records. Of course, window records have more information after the grafPort, but that information is private to the Window Manager. We can read, and when necessary set, all of that information with various Window Manager calls.

While every window has a grafPort, every grafPort is definitely not a window. The drawing screen in a non-toolbox graphics program like Dither is also a grafPort, and you've also used a grafPort when you used the Print Manager. It's also possible to create a grafPort that will never be visible. These grafPorts, generally called offscreen grafPorts, are very useful when you want to take some time to create a complicates picture, then draw it all at once on the screen. Graphics programs use offscreen grafPorts a lot, as do programs doing animation from the toolbox. (You have to resort to assembly language for really fast, crisp animation on the Apple IIGS, but a clever programmer can do a lot with toolbox calls from a high-level language.)

There's a lot to QuickDraw that we haven't covered in this lesson; it's just to big, and many of the features (like offscreen `grafPorts`) are only used in very specific circumstances. In the next lesson, we'll explore another aspect of QuickDraw II and learn about the Font Manager. It would be a great idea, though, to spend some time now browsing through the chapters in the toolbox reference manual that cover QuickDraw. Some of the things we've talked about are in the Window Manager, so try and spend some time there, too. I'd suggest reading the introductory sections at the front of both of those tools. After that, spend some time flipping through the rest of the chapter, browsing through the descriptions of any calls that catch your eye. There's a lot to discover!

Summary

This lesson has concentrated on the graphics capabilities of the Apple IIGS. We've spent most of our time learning to use QuickDraw, but several important ideas that involve the Window Manager have also been explored. We've learned how to use colors in both 320 mode and 640 mode, including how to change the color palette to get something other than the default colors. We've seen how dithering can be used to get more than four colors in 640 mode.

QuickDraw II uses the idea of a drawing pen, and we've expanded on our ability to use this pen. We've learned to use pen modes for things like rubber-banding, and seen how to use pen patterns and pen masks.

QuickDraw II has a wide variety of commands used to draw shapes, including rectangles, ovals, arcs, round rectangles, and polygons. You can also do a variety of things with each shape, including painting the shape, filling it, erasing it, inverting it, or even framing a shape to draw it's outline. We've seen how these ideas are implemented as a set of well-ordered calls.

We've also touched on regions, although we just covered the bare essentials. We learned about visible regions, update regions, and clip regions, and saw how to use these effectively to simplify and speed up our programs.

The lesson ended with a short discussion that defined `grafPorts` and windows. We've used these all along, but with some knowledge of QuickDraw and some experience to guide us, we were able to get a little more formal about the definitions.

Tool calls used for the first time in this lesson:

```
ClipRect          ClosePoly         EraseArc          EraseOval
ErasePoly         EraseRect         EraseRRect        FillArc
FillOval          FillPoly          FillRect          FillRRect
FrameArc          FrameOval         FramePoly         FrameRRect
FrameRect         GetColorTable     GetMouse          GetVisHandle
InvertArc         InvertOval        InvertPoly        InvertRect
InvertRRect       KillPoly          OffsetPoly        OpenPoly
PaintArc          PaintOval         PaintPoly         PaintRRect
RectInRgn         SetAllSCBs        SetColorTable     SetPenMask
SetPenPat         SetPenSize        SetSCB            StillDown
```

Lesson 10 – Fonts

Goals for This Lesson

This lesson covers the use of fonts. You will learn how the Apple IIGS uses font families, sizes and styles to organize a vast array of character shapes you can choose from. You will also learn how to write text to a `grafPort`, including colored text. Finally, you will learn how to create programs that let the user pick their own fonts.

Defining Some Terms

By now, you're probably getting pretty used to the names of the various tools, so when we start to talk about fonts, you mind probably turns to the Font Manager. I know mine did the first time I needed to use text in a desktop program. As it turns out, the Font Manager isn't the only tool you need to worry about, and in many ways it's the least important. QuickDraw II has a lot to do with fonts, too, and QuickDraw II Auxiliary even gets involved a bit. There is a pretty clear division of labor, though. QuickDraw is used to draw text in a `grafPort`, and the Font Manager is used to select and manipulate the font QuickDraw II will use to draw.

In this lesson, we'll start with the drawing commands, so we'll be starting in QuickDraw II. Rather than getting bogged down in large desktop programs, we'll use short graphics-only programs that just write some text on the screen. They will look a lot like the Dither program we used in the last lesson to explore dithered colors. Later in the lesson we'll start to learn how to change to different fonts, and at that point we'll go back to full desktop programs.

Drawing text really isn't much different from drawing anything else in a `grafPort`. In fact, drawing text is actually quite a bit easier than some of the basic shapes, like polygons. You just position the pen and make a call to a QuickDraw routine to print the text. There are a lot of ways to tell QuickDraw what text to draw – you might use a p-string, a c-string, or just a plain text buffer – so there are a lot of different calls to actually draw the text. They all work pretty much the same way, though. The hard part, surprisingly enough, is figuring out where to position the pen to get the text to show up at a particular spot. To understand how text is positioned on the screen, we'll have to define some terms.

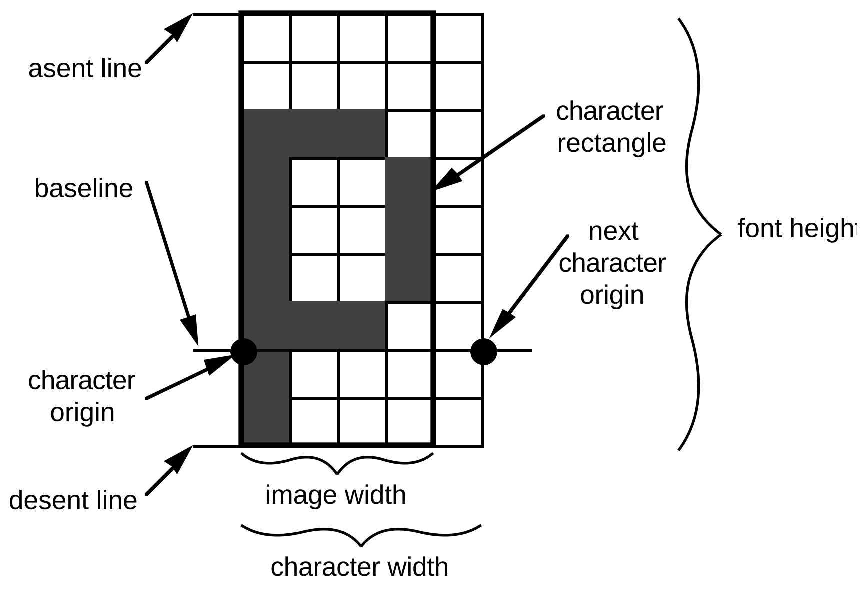


Figure 10-1: Terms Used to Describe Characters

Base Line	The base line is a horizontal line drawn through the character origin. In terms of handwritten characters, the base line does exactly the same thing as a line on a piece of paper.
Character Origin	This is the spot the pen is at when QuickDraw starts to draw a character. When I was learning about fonts, I expected this point to be at the top left corner or bottom left corner of the character, but it isn't. In terms of writing, it's where the line would be on a piece of stationary; in the terms used by QuickDraw, the origin is on the base line. The part of the character above the character origin is "above the line;" characters like "W" and "a" appear entirely above this point, resting on the base line. Characters like "y", which extend below the base line when you write them by hand on lined paper, also extend below the base line in QuickDraw fonts. When QuickDraw II draws a character, it starts at the current pen location, which is set with `MoveTo`, just like you were getting ready to draw a line. After a character is drawn, the pen moves to the right a few pixels to the next character origin.
Ascent Line	The ascent line is a line over the top of the character. In most fonts capitol letters and tall lowercase letters (like "k") extend from the ascent line to the base line.
Descent Line	The descent line is a line below the bottom of the character. In most fonts, characters like "y" extend below the base line, touching the descent line.
Kern Max	Kerning is a pretty wild concept for those of us who had no background in printing or typesetting before we started writing desktop programs. It turns out that typesetters sometimes place letters so close together that part of one letter can extend further to the right than the leftmost part of the next character. For example, if the characters "Ta" are typed, the "a" can be scooted over to the left to appear right next to the "T", or maybe even under the overhang of the top bar

in the "T". The process of pushing characters together like this is called kerning. In bit mapped fonts like those used on the Apple IIGS, kerning is done by extending part of a character to the left of the character origin.

Font Rectangle The font rectangle is the rectangle that completely encloses the character.

Some of this information, like the ascent and descent distances, is the same for all characters. QuickDraw's `GetFontInfo` call returns a record with all of the global information. A typical call would look like this:

```
var
   fontInfo: fontInfoRecord;

begin
GetFontInfo(fontInfo);
...
```

`GetFontInfo` tells you about the current font. In other words, it returns information about the font QuickDraw II will use to draw text if you draw the text before picking another font.

The font record itself looks like this:

```
fontInfoRecord = record
    ascent:  integer;
    descent: integer;
    widMax:  integer;
    leading: integer;
    end;
```

The `ascent` and `descent` fields are pretty obvious; they're the number of pixels between the base line and the ascent and descent lines.

The `widMax` field tells you the width of the widest character in the font. There are other calls to find out the width of a particular character; we'll look at those in a moment.

`Leading` is the recommended number of pixels you should leave between lines. Once you think about it, it's pretty obvious: you should leave more space between lines of 1 inch characters than you would leave between lines of 1/10 inch characters. Incidentally, this word is pronounced like the metal lead, not like lead a horse. The term comes from the not so distant past, when thin strips of lead were stuck in between lines of hand set type to increase the space between lines.

Drawing Text

It might seem pretty frightening to spend that much time learning how to figure out the horizontal and vertical values you need to pass to `MoveTo` to put text where you want it, but take heart. We're at the easy part, now. To actually write text to the screen, all you do is pass a string to QuickDraw. Here's a short program that writes "Hello, world." on the graphics screen.

```
program Hello (input);

using Common, QuickDrawII;
```

```
begin
StartGraph(320);
PenNormal;
MoveTo(50, 50);
DrawString(@"Hello, world.");
readln;
EndGraph;
end.
```

Listing 10-1: Writing Text to the Graphics Screen

There are a lot of different ways to represent strings in memory, and QuickDraw supports several of them with different text drawing commands. Listing 10-1 uses `DrawString`, which takes a pointer to a p-string. P-strings have a leading length byte, followed by up to 255 characters. In ORCA/Pascal, any string defined using `string[x]`, where `x` is less than 255 characters, is a p-string. You can also get a p-string by defining a variable as `packed array[0..x] of char`. Finally, if you take the address of a string constant with the @ operator, like the sample program in Listing 10-1 does, you get the address of a p-string.

Ironically, Pascal strings have nothing to do with Standard Pascal. When Nicholas Wirth designed Pascal, he defined strings in a very specific way, as a packed array of characters. There is no length byte, and in fact, no way to set the length of a string in Standard Pascal. That's why the folks who designed UCSD added the string type with a leading length byte – they wanted to have a string that could have a variable length. Unfortunately, they weren't firing on all neurons that day, and we're stuck with the name Pascal strings applied to this abominable data type to this day.

Well, enough editorializing. ORCA/Pascal also supports true Standard Pascal strings, which can have a length up to 32767 characters – a rather distinct advantage over the UCSD Pascal string. ORCA/Pascal defines the character `chr(0)` with a rather peculiar meaning; it means to ignore all of the characters from there to the end of the character array. With that one convention, ORCA/Pascal accomplishes a lot. First, Standard Pascal strings have, in effect, a variable length. Second, as long as the string is one character smaller than the character array holding the string, the string ends with a null character (`chr(0)` is called the null character). That means that ORCA/Pascal strings have the same format as C strings.

You can get a Standard Pascal string using a type of `string[x]`, where `x` is 256 or more. You can also get a Standard Pascal string using a type like `packed array[1..x] of char`. The difference between a Standard Pascal string and a UCSD Pascal string, then, is that the first array element is 1 for Standard Pascal, and 0 for UCSD Pascal. The 0th array element holds the length byte in UCSD Pascal. As long as the character array is one element longer than the current string length, the string will also be a C string with a terminating null character.

Finally, to get a C string from a string constant, add one to the address returned by the @ operator. In ORCA/Pascal, all string constants start with a leading length byte and are followed by a terminating null, so you can use either format you like.

Well, finally I get to the point: to print a C string, pass the address of the string to `DrawCString`. It works just like `DrawString`, but expects a string with a terminating null character.

There are some times when you just need to get some characters out to the screen, either individually, or plucking characters from a large chunk of text. To print an individual character, pass the character to `DrawChar`.

```
DrawChar('a');
```

To print a block of text, call `DrawText`. This time you need to pass more than the address of the first character in the text block; you also need to pass the number of characters.

```
   DrawText(@gsosString.theString, gsosString.size);
```

Problem 10-1: Create a program that shows how text is placed on the screen. Start with a white screen in 320 mode graphics, then draw a light gray (color 14) line at the baseline, and light blue lines (11) at the ascent and descent lines. Next, draw a vertical red line (7) through the character origin. Finally, write the string "The quick gray Apple IIGS jumped over the lazy black IBM PC" to the screen.

We'll get to what these calls mean in a moment, but for now, just add them right after you set up QuickDraw:

```
   SetForeColor(0);
   SetBackColor(15);
   SetTextMode(modeForeCopy);
```

Actually, the entire line won't fit in 320 mode, but you get the idea. We'll fix the line width problem in Problem 10-3.

Colorful Text

The text we've drawn so far is black text on a white background. That's fine, as far as it goes. Black text on a white background is the way we usually see it on paper and in windows on the desktop. Of course, it would be a lot more fun to see colored text every once in a while.

`SetForeColor` tells QuickDraw to write using colored text. By foreground, we mean all of the pixels that make up the character itself; the pixels that are in the character's rectangle but that are not a part of the character are called the background. The foreground color for text can be any of the solid colors, but you can't use dithered colors in 640 mode. `SetForeColor` takes a single parameter, the number of the color you want QuickDraw II to use.

You can also set the background color of the text using `SetBackColor`. This causes QuickDraw II to fill the rectangle surrounding each character with the background color, usually white. Like `SetForeColor`, you can use the number of any solid color as a parameter for `SetBackColor`.

The pen modes you used in the last lesson to get all of the great effects, like rubber-banding, don't affect the way text is drawn at all. QuickDraw uses a separate text mode to draw text. The text modes work just like pen modes, they're just separate from pen modes to make it easier to draw text in one mode and graphics in another. In fact, all of the pen modes can also be used as text modes, and they work exactly the same way. You can also use eight other modes. These work just like the original eight pen modes, but when you use them, QuickDraw II only draws the text – the background is not affected at all.

The call to set the text mode is `SetTextMode`. It takes a single integer parameter as the text mode.

Here's a complete list of the pen modes, along with a short description of each one. For more details, check out the toolbox reference manual. Be careful, though: The toolbox reference manual has some mistakes in the description of `SetTextMode`. These mistakes are outlined in Chapter 43 of Volume 3 of the toolbox reference manual.

`modeCopy`	Copy the foreground and background to the `grafPort`.
`notCopy`	Reverse the bits in the character, then copy the foreground and background to the `grafPort`.
`modeOR`	Or the bits in the foreground and background with the pixels in the `grafPort`.
`notOR`	Reverse the bits in the character, then or the bits in the foreground and background with the pixels in the `grafPort`.
`modeXOR`	Exclusive or the bits in the foreground and background with the pixels in the `grafPort`.

`notXOR`	Reverse the bits in the character, then exclusive or the bits in the foreground and background with the pixels in the `grafPort`.
`modeBIC`	Reverse the pixels in the foreground and background of the text, then and the pixels in the text with the pixels in the `grafPort`.
`notBIC`	And the pixels in the foreground and background of the text with the pixels in the `grafPort`.
`modeForeCopy`	Like `modeCopy`, but only the foreground of the text is used.
`notForeCopy`	Like `notCopy`, but only the foreground of the text is used.
`modeForeOR`	Like `modeOR`, but only the foreground of the text is used.
`notForeOR`	Like `notOR`, but only the foreground of the text is used.
`modeForeXOR`	Like `modeXOR`, but only the foreground of the text is used.
`notForeXOR`	Like `notXOR`, but only the foreground of the text is used.
`modeForeBIC`	Like `modeBIC`, but only the foreground of the text is used.
`notForeBIC`	Like `notBIC`, but only the foreground of the text is used.

Problem 10-2: Write a graphics program in 320 mode that shows all of the text modes. Start by drawing a red (color 7) and yellow (color 9) checkerboard pattern across the entire screen. Set the text foreground color to blue (color 4), and the background color to white (color 15). (These colors are carefully chosen for maximum eyestrain.) Finally, draw the name of the various text modes in the appropriate mode.

Formatting Text

QuickDraw figures its job is to take a character number, look up the character in the font, and draw whatever it finds in the current `grafPort`. As far as QuickDraw is concerned, issues like tabs, return characters, and formatting numbers for output are your problem. It only wants text. Of course, that means you might have to get a little creative to write certain things on the screen.

Using Write and Writeln

As it turns out, ORCA/Pascal can do a lot of formatting for you. When you start the tools with `StartDesk` or `StartGraph`, ORCA/Pascal automatically hooks up the standard Pascal text output routines, `write` and `writeln`, so they write the text to the graphics screen instead of writing the text to the text screen. ORCA/Pascal uses `DrawChar` and `DrawString` to write the text, so all of the color and text mode options work just fine. ORCA/Pascal will also handle return characters, moving down one text line and moving back to the horizontal pen location that was set when you started writing text.

There are certainly a lot of times when you need better control over text than you get from `write` and `writeln`, but when they work, they're just about the easiest way to write formatted text.

Using String Conversion Subroutines

The biggest problem with writing text is numbers. Converting a number, especially a floating-point number, to a string by hand is a harrowing experience. ORCA/Pascal has several built-in functions that can do the conversions for you, though. You can then combine the various strings representing the numbers with any other text, and use the standard QuickDraw text drawing commands to draw the strings.

The ORCA/Pascal functions that convert numbers to strings are `Cnvds`, `Cnvis` and `Cnvrs`.

Using the Size of Text

Of course, you may just be trying to dump a block of text to a window. In that case, the major problem you will have is figuring out when to split a line and start a new one. There are two sets of calls that let you do this. One set returns the width of a character or string, while the other returns a rectangle surrounding the character or string. There's one of each of these calls for each of the string formats, as well as one for characters. The width calls are the simplest, and since you already know the height of the text from the `GetFontInfo` call, they are the ones you will use most often, too. Each of them takes the same sort of parameter as the corresponding text drawing command, and each returns an integer that is the width of the string or character in pixels.

```
width := CharWidth('A');
width := StringWidth(@'Hello, world.');
width := CStringWidth(pointer(ord4(@'Hello, world.')+1));
width := TextWidth(@gsosString.theString, gsosString.size);
```

The calls that return a rectangle surrounding the text all look like the text drawing commands with an extra parameter at the end of the parameter list. The extra parameter is the rectangle you want the call to fill in.

```
CharBounds('A', r);
StringBounds(@'Hello, world.', r);
CStringBounds(pointer(ord4(@'Hello, world.')+1), r);
TextBounds(@gsosString.theString, gsosString.size, r);
```

Problem 10-3: The text from Problem 10-1 didn't all fit on a single line. Find where the text should be split, and draw it on two lines. Be sure to split the text on a space, not in the middle of a word. Use leading to figure out how much space to leave between the lines.

The Standard Character Set

When you hear someone talk about a character font, it's natural to assume that all of the printing characters you're used to are in the font. The alphabet, the numeric digits, and the other characters you see on a standard keyboard certainly ought to be there. On the other hand, folks in France have a different idea about what a standard keyboard looks like than the people in the United States do, and the Germans and British have their own ideas, too.

One of the nice things about a graphics based font is that all of these differing viewpoints can be accommodated. Figure 10-2 shows how Apple has defined the font space available with eight bit characters on the Apple IIGS and Macintosh. It's worth pointing out that very few fonts actually have all of these characters, but if a font uses these characters, they are almost always in the right spot. It's also worth pointing out that these are conventions, not rules. Greeks and Japanese have a different enough view of a standard keyboard that this character set is scrapped and replaced, and there are some special purpose fonts that use symbols instead of letters. The most common is Cairo, but I've also seen an Egyptian hieroglyphics font.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0			space	0	@	P	`	p	Ä	ê	†	∞	¿	–		
1			!	1	A	Q	a	q	Å	ë	°	±	¡	—		
2			"	2	B	R	b	r	Ç	í	¢	≤	¬	"		
3			#	3	C	S	c	s	É	ì	£	≥	√	"		
4			$	4	D	T	d	t	Ñ	î	§	¥	ƒ	'		
5			%	5	E	U	e	u	Ö	ï	•	µ	≈	'		
6			&	6	F	V	f	v	Ü	ñ	¶	∂	∆	÷		
7			'	7	G	W	g	w	á	ó	ß	∑	«	◊		
8			(	8	H	X	h	x	à	ò	®	∏	»	ÿ		
9			)	9	I	Y	i	y	â	ô	©	π	…			
A			*	:	J	Z	j	z	ä	ö	™	∫	space			
B			+	;	K	[	k	{	ã	õ	´	ª	À			
C			,	<	L	\	l	|	å	ú	¨	º	Ã			
D			-	=	M	]	m	}	ç	ù	≠	Ω	Õ			
E			.	>	N	^	n	~	é	û	Æ	æ	Œ			
F			/	?	O	_	o		è	ü	Ø	ø	œ			

- The characters from the space ($20) to the tilde ($7E) are all standard printing ASCII characters.
- While they have standard definitions, the characters $11..$14, $AD, $B0..$B3, $B5..$BA, $BD, $C2..$C6 and $D6 tend to be rare in most fonts.
- Character CA is the non-breaking space.

Figure 10-2: The Standard Graphics Characters

Font Families

So far we've been using the default font, also known as the system font. Of course, there are a lot of other fonts out there. To organize all of these fonts, Apple uses font families, font sizes, and font styles.

Font families are the different kinds of fonts. Each font family has a name and a number. Sometimes it's easiest to use the name and sometimes it's easiest to use the number, but you can always count on a particular font number always matching a font name.

You can get a font family number directly from a name, but in most cases you won't have to. If you decide to get a font family number from a name, you will need to know the exact font name beforehand, and in most cases, if you know the font name, you know it's number, too. As a general rule, the Font Manager calls that let the user pick fonts deal with names and give you back a number, so for this lesson, we'll assume you already have a font number.

Of course, you don't, yet, so here's a few to get you started. I've also shown the font name. If you try to use any of the Font Manager calls that need a name, be sure and pass the font name just as it is shown, character for character. You can't even change an uppercase character to a lowercase character. If you do, the Font Manager won't recognize the font.

```
2     newYork
3     geneva
4     monaco
20    times
21    helvetica
22    courier
-2    shaston
```

Figure 10-3: Some Font Family Names and Numbers

The Font Manager interface file has constants defined for all of these fonts (plus a few more), so you can use the font name when you are writing programs.

The system font doesn't have a name, but it does have a number. It's font number 0. The system font is the default font, and it's also the one you see on menus, in dialogs, and so forth.

Font Sizes

Each font family comes in a variety of sizes. Font sizes are specified in points, a unit of measurement used by printers. In rough terms, a 72 point font is one inch high – at least, it's one inch high on the printer. There are some funky scaling issues that take place to display fonts on the Apple IIGS screen, especially in 320 mode. Also, while a 12" monitor is pretty common on the Apple IIGS, it's not required, so trying to measure a font on the screen isn't going to work. Finally, some fonts are a little bigger than others, even when you use the same point size. For example, in this book I've used 12 point Times for text, and 10 point Courier for listings. They look about the same size to me, though, and 12 point Courier definitely looks a lot bigger than 12 point times.

You can ask for a font size from 1 to 255 points. A 1 point font will look pretty weird; about all it's good for is to frustrate people. Lawyers love it. For the most part, you should start with about 12 or 14 point for the screen or a dot matrix printer, and 10 or 12 point for a laser printer. You can always adjust the size later.

It would take a lot of disk space to save every font in every possible font size, so of course, each font is only stored in a few sizes. If you ask for a font that isn't available, the Font manager will politely make one for you. That may not be what you want, since a font created on the fly by the Font Manager doesn't look as good as one created pixel by pixel by an artist, but at least it works. With some kinds of printers, the output will even be pretty good. Some laser writers may even have the font built in in the correct size, even if the Font Manager can't find it to display on the screen.

There are ways to find out what font sizes exist already, and even ways to tell the Font Manager not to scale the fonts, but just to pick the closest fit. If you're writing a program that does a lot with fonts, like a word processor or page layout program, it would be worth your while to spend some time pouring through the chapter on the Font Manager in the toolbox reference manual to find out about all of these things, but for most programs it isn't an issue, so I'm not going to get bogged down in those details here.

Font Styles

In addition to sizes, fonts can have styles. Unlike sizes, styles are generally created on the fly, although you can have them created in a bit mapped form.

Plain Text	Plain text is the unadorned font, with no extra styling applied.
Outline	Outlined text is drawn as an outline around the bits that would normally be displayed.

Shadow	Shadowed text is a lot like outlined text, but the border is thicker on the bottom and right, giving a shadowed effect.
Underline	Underlined text has a line under it.
Italic	Italicized text is slanted. The pixels on each line or two are moved over one column to get this effect.
Bold	Bold text is darker than the normal font. On the screen, this is done by printing the character twice, with the second character one bit to the right of the first. Of course, the font is actually preformed, but that's the effect.

To be really annoying, you can use these styles in combination. With a little effort, you can take a perfectly readable sentence and apply enough styles so it would take an expert to figure out what the letters really are.

For the most part, you only need to have QuickDraw II and the Font Manager started to use the capabilities we're talking about in this lesson. (A quick check of Appendix A or the toolbox reference manual shows that the Font Manager also requires the Memory Manager and Tool Locator, but those are running in any Pascal program.) If you are using the styles shadow, outline or italic, though, you also have to load QuickDraw II Auxiliary. Both `StartDesk` and `StartGraph` load QuickDraw II Auxiliary, but if you're using `StartUpTools` you have to remember to include QuickDraw II Auxiliary in your list of tools.

Installing a Font

All of the information you need to specify a font – the family number, size, and style – can be packed into a single record called a font ID record.

```
fontID = record
    famNum: integer;
    fontStyle, fontSize: byte;
    end;
```

The family number and size are pretty obvious. The family number is a number like one of the ones in Figure 10-3, and the size is the size of the font in points. The font style is a bit map, with one bit for each style element:

bit	style
0	Bold
1	Italic
2	Underline
3	Outline
4	Shadow

In each case, a style bit of 1 says the style is in use, while a style bit of 0 says the style is not in use. Plain text, then, would have a style byte of 0, while underlined text would have a style byte of 4.

For the system font, all three fields are zero.

When you want to use a new font, you first create the font ID, then call `InstallFont`. `InstallFont` does all of the work, loading the font if it needs to be loaded. It also makes the font the current font, so the QuickDraw text drawing commands will use the font until the next time you change it. In the couple of sections, we'll cover neat ways to let the user pick the font, but even now you can load a font by hard-coding the font ID, like this:

```
{load 10 point plain courier}
id.famNum := courier;
id.fontStyle := 0;
id.fontSize := 10;
InstallFont(id, 0);
```

The first parameter to `InstallFont` is pretty obvious; it's the font ID for the font you want to install. The second parameter is called the scale word. If it's zero, you're giving the Font Manager permission to scale the font. If you pass 1, the Font Manager uses the closest font it can find, but doesn't scale the font.

There are a lot of cases where you want to use the current font, but change the font size or style. One way to do that is to read the current font ID, change it, and call `InstallFont` with the new font ID. `FMGetCurFID` reads the current font, but there is a catch: Pascal functions can't return a record, so the value is returned as a long integer. All you have to do is remember to put in a type cast. If you forget, the compiler will give you a gentle reminder.

```
{switch to bold}
intID := FMGetCurFID;
id := fontID(intID);
id.fontStyle := id.fontStyle | 1;
InstallFont(id);
```

Problem 10-4: Write "Hello, world." to the 320 mode screen in 36 point bold times. Use red letters.

Try the same program with 640 mode graphics. As you see, the font is not the same.

You can use `StartDesk` for this program, then paint the screen white, just like you have done in the short programs that use `StartGraph`. You're loading a lot of tools you don't need, but the point of this problem is to let you see a couple of font calls in action, not spent the afternoon starting tools in a custom way!

Choosing Fonts with ChooseFont

There are two common ways to let the user pick a font. One involves menus with explicit menu items for each font family, style and size, and the other one just uses one menu item to call up a standard Font Manager alert. In my own opinion, programs that don't do much with text, but do let the user pick fonts, should use the dialog. That saves space on the menu bar for other important jobs. Programs like text editors and page layout programs ought to use the menus, since you can pick the options quicker with menus.

`ChooseFont` is the call that brings up the Font Manager's font chooser dialog. It's really pretty simple to use. You need to install a menu item named "Choose Font" somewhere in one of your menus. When the user picks this item, you call `ChooseFont`, passing the current font ID and a word of family specification bits. `ChooseFont` returns the font ID. Like `FMGetCurFID`, `ChooseFont` returns the font ID as a long integer.

```
id := ChooseFont(currentID, 0);
```

The family specification bits word only has one bit that's currently used. It's bit 5, which is set if you only want to allow base families and clear otherwise. All of the other bits should be clear Base fonts are fonts that actually exist in the font folder, as opposed to those that the Font Manager builds for you by changing the base fonts.

Problem 10-5: Create a font sampler program. Your program should call `ChooseFont` to let the user pick a font. If any windows are open, it should apply the font ID to the topmost window; if not, the value should be saved and used when the next window is opened.

The window update routine should print the font ID at the top of the window, using the system font. It should then print column and row numbers, similar to Figure 10-2, again using the system font. The table should be filled in with the printable characters from the font selected by the user.

Be sure to take the size of both the font being sampled and the system font into account when the table is created!

Printed fonts don't always look quite like you would expect, so be sure you use a version of the Frame program that supports printing as a basis for your program. There's no point is supporting disk I/O, though.

Once again, the point is to get some practice working with the Font Manager, not to develop the world's best font sampler. Keep it simple. Allow a document size of 640 by 400 – that will print fine on most printers – but don't worry about it when a font is too big to display or print. Just show what you can.

Choosing Fonts with Full Menus

If you want to use full menus to select the fonts, you need to provide three things: a way to select the font family, a way to select the font size, and a way to select the font style.

The font style is the easiest. There are five flags you can set, so you can just put five menu items, one for each style, in a menu. A simple check works best for showing which styles are selected. The Menu Manager has some flags that can be used to apply font styles to the menu itself, so, for example, you can use outlined text to write "Outline" in the menu. That's a classy touch I highly recommend. (But remember: underline won't work, since the system font doesn't support it.) It's also traditional to include an option for plain text; it just turns off any style bits that have been selected all in one whack.

The size isn't that much harder to handle. Most programs give a few common sizes as menu items that can be selected, checking them when a particular size is the current size. The common font sizes to include are 9, 10, 12, 14, 18 and 24. Of course, that's just a small selection from the 255 possible font sizes, so most programs also give you the option of using `ChooseFont`, and a few even create a custom dialog that lets the user type in a font size in a editline item. That's something you don't know how to do yet, but it's a pretty simple dialog to create once you've learned a few things about controls.

There is one thing you should do in a font size menu that's a little tough. When the Font Manager builds a font, the result isn't generally as good as a font that is created by an artist. The user may want to pick fonts that will look good, so you need some way to tell them which font sizes actually exist. The traditional way to show which font sizes exist as bit mapped fonts is to show those sizes using outlined or bold text. That means you need some way to outline a menu item name, and you also need some way to tell if a font exists on disk.

`CountFonts` is one way to see if the font exists on disk. There are a lot of ways to use `CountFonts`, but we'll only discuss this one.

You pass two parameters to `CountFonts`, a font ID with the proper font family number and size, and a flags word, which for our purposes should be set to $0A. The font ID should also have the style word set to $FF. `CountFonts` returns the number of fonts that fit the bill. If it returns 0, the size should be printed in normal text in the font size menu. If something other than 0 is returned, the size should be printed with outlined text.

You can change the style for the text in a menu item using the Menu Manager's `SetMItemStyle` call. You pass the menu item number and a style word. The style word is basically a copy of the font style byte expanded to a word length, so for outlined text you would pass $0008, and for normal text you would pass $0000.

Putting all of this together, here's the sort of code you would need to set the style on a font size menu item:

```
intID := FMGetCurFID;
id := fontID(intID);
id.fontStyle := $FF;
id.fontSize := 10;
if CountFonts(id, $0A) = 0 then
   SetMItemStyle(0, size_10)
else
   SetMItemStyle(8, size_10);
```

The last of the three things you need to put in your menus is a list of the font families, listed by name. The Font Manager has a call that will do this for you. `FixFontMenu` builds a list of font family names, starting with a menu ID for the menu to put the font names into, a starting ID, which will be the first menu ID assigned to one of the font family names; and a family specification bits word, like the one used for `ChooseFont`. You have to tell the Menu Manager to rebuild the menu size after adding these new items, too. A typical call looks like this:

```
FixFontMenu(Font_Menu_ID, First_Font_ID, 0);
height := FixMenuBar;
```

The menu item IDs are assigned sequentially, starting with `First_Font_ID`. Your `HandleMenu` subroutine should be updated to check for these menu numbers; to make the check easy, you might use a value like 1000 for `First_Font_ID`, then make sure all other menu IDs are much smaller.

Once your menu handler detects a font family choice, pass it to `ItemID2FamNum`. This Font Manager call takes the item ID as input and returns the correct family number. By combining this family number with the size and style information from the other two menus, you can create a complete font ID.

```
id.famNum := ItemID2FamNum(menuItemNum);
```

There's also an inverse to this call. `FamNum2ItemID` returns the item ID for a particular family number, something you need to handled checking and unchecking of the family names as your application switches windows.

```
menuItemNum := FamNum2ItemID(id.famNum);
```

These three different aspects of the font – the family, size and style – don't generally get their own private menus. I've seen it done that way, but in a lot of programs, menu bar space is precious enough that you just can't use three separate menus. If you combine any of these menus, though, be sure to put the font families at the bottom, so the other font information doesn't get crowded off of the menu bar if the user has too many fonts installed!

Problem 10-6: Rework the font sampler from Problem 10-5 so it uses three font menus, labeled "Font", "Style", and "Size". Be sure to support checking of the current font family, style and size, and be sure to outline the sizes that exist as bit mapped fonts on disk. Also, add "Plain Text" as one of the font styles, and handle it appropriately.

The first entry in your Font menu should be the call to Choose Font.

Be sure you keep the proper items checked and outlined! As the user picks different windows, the checks and outlines should be updated to reflect the front document window. If something other than a document window is front (like a desk accessory) or if all of the document windows are closed, the checks and outlines should reflect the font that would be used for a new window.

Summary

This lesson has covered fonts – how to use them and how to select them. You have learned to use QuickDraw to draw text in a variety of colors and pen modes. You've also learned how to measure text so you know when to wrap lines. Finally, you learned to use the Font Manager to select fonts.

Tool calls used for the first time in this lesson:

```
CharBounds        CharWidth         ChooseFont        CountFonts
CStringBounds     CStringWidth      DrawChar          DrawCString
DrawString        DrawText          FamNum2ItemID     FixFontMenu
FMGetCurFID       GetFontInfo       ItemID2FamNum     SetMItemStyle
SetTextMode       StringBounds      StringWidth       TextBounds
TextWidth
```

Lesson 11 – TextEdit

Goals for This Lesson

This lesson introduces one of the most specialized tools in the toolbox, TextEdit. TextEdit is a fairly sophisticated tool for displaying and editing text. In fact, you can create a reasonably good text editor using little more than TextEdit – and that's what we'll do in this lesson. This text editor will be used later in the course to help explore other parts of the toolbox in a realistic program.

An Overview

You may not have thought about it quite this way, but you already know enough to write most of a text editor. About the only things you would need that we haven't covered are some dialogs. After all, you know how to choose and use fonts. You know how to figure out how big a string is in a particular font, how to scroll windows, how to draw and erase a line in `modeXOR` to create a flashing cursor, how to scroll the screen, how to load and save files, and how to print. In short, you know all of the things you need to know to actually write an effective text editor up to the point where you would add dialogs.

Of course, knowing how something is done doesn't mean it is easy, or that you would ever want to do it. I know how to walk from Albuquerque to Chicago, but I don't ever plan to put that knowledge to use. It would be a big job, and if I ever really wanted to get to Chicago, there are a dozen better ways.

The TextEdit tool is a better way to handle something you already know how to do for yourself. The TextEdit tool can display text in one or several fonts, scroll the text, update the window, and even let the user enter new text. It gives you relatively easy to use mechanisms to print the text or examine and change it from your program. In short, it's all you need to put together a very effective little editor. About the only major things it can't do are wrap text around a picture and scroll horizontally.

There are two good reasons for learning about the TextEdit tool in an introductory course. The first is that most programs need to display text, and many need to edit text. TextEdit gives you a way to get the program finished enormously faster than if you had to do all of that work for yourself. The second reason we'll use TextEdit is to actually get that word processor. It's not that I expect you to really need a word processor; you probably have two or three that do more than the one we'll write. The real reason for wanting the editor is to give us a simple but realistic program we can use later.

Using TextEdit

TextEdit has to be started, and it's not one of the tools that we've been starting so far. There's not much to adding another tool to the list of tools to start, but you should be sure you add TextEdit to your `StartUpTools` tool list. The tool number for TextEdit is 34 ($22); the version number, as of System Disk 6.0, is 1.3 ($0103 in the resource description file).

The easiest way to use TextEdit is to create a TextEdit control. That's something we haven't done before, but creating a control is a lot easier than you might think. For the most part, it's all done with resources.

Back in lesson 5, when you learned how to set up a window with an `rWindParam1` resource, there were two fields at the end that I basically told you to not worry too much about. They were right at the end of the 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 */
   };
```

Listing 11-1: Resource For a Standard Document Window

The field labeled `wStorage` is actually a reference to a list of controls. The last field, `wInVerb`, is named a little more appropriately; it is a flags word that gives some detailed information about the control reference, color table reference, and window title reference. You used the color table bits, bits 10 and 11, back in Lesson 5 to create a window with a custom color table. In this lesson, we'll use bits 7-0, which describe the control list.

(Incidentally, the comments for the fields come from the Types.rez header file that comes with the Rez compiler. I left the names alone so you would feel comfortable reading that file later. `wStorage` is a horrible comment for what the field is used for. It's a hold over from the pre-resource days; this position in the window record tells where the window is stored.)

To create a TextEdit control, the first step is to modify the window resource so it looks for a list of controls. The modified resource looks like this:

```
rWindParam1 (1001) {
   $C1A5,                      /* wFrameBits */
   nil,                        /* wTitle */
   0,                          /* wRefCon */
   {0,0,0,0},                  /* ZoomRect */
   linedColors,                /* wColor ID */
   {0,0},                      /* Origin */
   {0,0},                      /* data size */
   {0,0},                      /* max height-width */
   {0,0},                      /* scroll ver hors */
   {0,0},                      /* page ver horiz */
   0,                          /* winfoRefcon */
   0,                          /* wInfoHeight */
   {30,10,195,630},            /* wposition */
   infront,                    /* wPlane */
   1001,                       /* wStorage */
   $0802                       /* wInVerb */
   };
```

Listing 11-2: Resource For a Window with Controls

The 1001 for `wStorage` is a resource number for an `rControlTemplate` resource. A value of 2 in bits 7-0 of the `wInVerb` parameter tells the Control Manager that the `wStorage` parameter is a resource ID, and that it points to a single control, not a list of controls. (We'll use lists of controls in Lesson 13.)

The next step is to create the TextEdit control itself. Here's the `rControlTemplate` resource we'll use in the problems to create a TextEdit control. We'll talk about the individual fields in a moment.

```
resource rControlTemplate (1001) {
   1001,                         /* Application defined ID */
   {0,0,165,620},                /* controls bounding rectangle */
   editTextControl {{
      $0000,                     /* flags */
      $7C00,                     /* more flags */
      0,                         /* refcon */
      $42280000,                 /* text flags */
      /*-------------------------------------------------*/
      {-1,-1,-1,-1},             /* indent rect */
      /*-------------------------------------------------*/
      $FFFFFFFF,                 /* vert bar */
      0,                         /* vert Amount */
      nil,                       /* hor bar */
      0,                         /* hor amount */
      /*-------------------------------------------------*/
      nil,                       /* style ref */
      /*-------------------------------------------------*/
      0,                         /* text descriptor */
      nil,                       /* text ref */
      0,                         /* text length */
      /*-------------------------------------------------*/
      nil,                       /* max chars */
      nil,                       /* max lines */
      nil,                       /* Max chars per line */
      nil,                       /* max height */
      /*-------------------------------------------------*/
      nil,                       /* color ref */
      4,                         /* drawing mode */
      /*-------------------------------------------------*/
      nil,                       /* Filter Proc Ptr */
      }}
   };
```

Listing 11-3: `rControlTemplate` for a TextEdit Control

As you can see, there's a lot to this record. While there's a lot of work setting up this resource, one of the really neat things about the resource is that it's all you have to do. When you add this resource to your window, TextEdit, the Control Manager, and the Window Manager all take over and work a small miracle. Suddenly, when your program runs, a cursor appears in the window. When you close the window, the control and all of its associated data structures are cleaned up. The user can type text, erase and delete text, cut copy and paste, and all sorts of wonderful things. We'll get into a lot more detail about that later. With a few rather simple additions to your program, the user can even change fonts, format the text, load and save files, and print the text. You'll learn to do all of that shortly.

Here's the details about the parameters in the `rControlTemplate` resource.

```
1001,                                    /* Application defined ID */
```

This is the control ID. You need this number for some Control Manager and TextEdit calls. It can be anything except 0 or 1, but it causes a little less confusion if you use the same number for the resource ID and the control ID, as we've done here.

(Actually, 1 would work as a control ID, but it's not a good choice, for reasons we'll explore in a later lesson.)

```
{0,0,165,620},                     /* controls bounding rectangle */
```

This rectangle is the size of the TextEdit control. For our text editor, the size is the same as the window size, and later on we'll set up some flags to tell TextEdit to change the size of the control to match the size of the window. You can create a TextEdit control that doesn't take up the whole window, though. That way you can have text in the same window with other controls or pictures. You can even put more than one TextEdit control in the same window. You'll learn how to handle more than one control in the same window in Lesson 13.

```
$0000,                              /* flags */
```

All of the bits but one are unused in this flags word – at least, they are unused for a TextEdit control. This same resource type is used for a lot of different kinds of controls, and for some of the other controls, more of the bits are used.

Bit 7 should be clear, as it is in our resource, if you want the control to be visible.

```
$7C00,                              /* more flags */
```

All of the bits in this flags word are used for TextEdit controls.

bit 15	If this bit is set, the control is the current target. That means that if the user presses a key, this control is the one that gets the character. TextEdit sets the bit for itself, though, and we're supposed to leave in clear when the control is defined.
bits 14-12	These bits have to be set to 1. TextEdit uses them for internal flags, and will turn them off if it needs to.
bit 11	Setting this bit tells TextEdit to create it's own size box, and to hook that size box to the window. It replaces the size box we've used in the past, so we'll need to get rid of the size box and scroll bars we generally ask `TaskMaster` to handle for us. If you are creating a TextEdit control that will be a small part of a larger window, you would clear this bit.
bit 10	This bit must be set to 1. It's another one of those internal bits that TextEdit insists must be set when you create the control.
bits 9-4	These bits must be set to 0.
bits 3-2	You can create a separate color table for the TextEdit control, and there is a field later on in the resource that refers to the color table. These bits tell what kind of a color table reference we're using. In this case, we've set the bits to 00 for a pointer reference, and then

set the color table pointer to nil, telling TextEdit to use the default color table.

bits 1-0	These bits define the reference type for a list of styles. We will cover styles a little later in this lesson, but we won't use styles from a resource at all, so once again the reference is 00, and the pointer that appears later in the resource is nil.

```
0,                                         /* refcon */
```

The Control Manager sets aside this long word field for our own use. We're not using it, so it's set to 0.

```
$42280000,                                 /* text flags */
```

The text flags long word controls all sorts of things. This parameter finishes the job started by the flag words, letting you tell TextEdit just how you want the text to look, and how you want it to work.

bit 31	This bit is set if the TextEdit control is supposed to actually be a control. You can create a TextEdit control and hide it from the Control Manager by clearing this bit, but then you have to do all of the nasty work for yourself.
bit 30	This bit must be set to 1, telling TextEdit that there will only be one format used for all of the text. It's there so Apple can expand TextEdit later.
bit 29	If this bit is clear, like it is in our resource, TextEdit will allow any number of different styles of text. If it's set, only one font, size, and so forth can be used.
bit 28	For something like a program listing, you might want to let a line disappear off of the right edge of a page if it is too long. In that case, you would set this bit. Text editors normally break a line when they get to the end of a page, starting the next word on a new screen line. If you clear this bit, as we've done, that's what TextEdit will do with the lines. Another way to think of this is to label a physical line of text as anything ending with a carriage return character. If this bit is set, TextEdit only uses one screen line for each physical line, no matter how long the line is. If this bit is clear, TextEdit treats a physical line as a paragraph, splitting it up so you can see all of the words in the available space.
bit 27	Setting this bit disables scrolling.
bit 26	TextEdit can be used to display text, as well as edit it. If you set this bit, the user can read the text, but can't change it.
bit 25	TextEdit can handle cutting and pasting text, but there are some problems with simple methods of cutting and pasting. The problems are related to how you handle cutting and pasting words of

text. If you set this bit, it turns on smart cuts and pastes, which cause TextEdit to apply a few extra rules when it cuts and pastes text. Specifically, blanks are removed from the start of text when you cut it. If there are no blanks at the start of the text, blanks at the end are removed. When text is pasted, blanks are inserted if the pasted text would butt up against characters already in the window.

What this does is apply some reasonable rules to help you cut and paste words. When you select a word and cut it, TextEdit tries to remove extra spaces from the text you cut. Then, when you paste the text, TextEdit tries to keep spaces between the words. The method isn't perfect, but it works more often than it fails.

bit 24 If you are using more than one TextEdit control in a window, or if you're mixing TextEdit controls with other controls that can use keys, the Tab key is normally used to move from one control to another. If you're not using more than one control that uses keys, though, you should let the user type tabs. Setting this bit tells TextEdit to move to the next control when you type Tab; clearing it, as we've done, tells TextEdit to put tabs in the text.

bit 23 If you set this bit, TextEdit draws a box around the control. That's probably what you would want if the control is a small control in a larger window, but in our case the control fills the entire window, so we left this bit off.

bit 22 This bit must be set to 0.

bit 21 TextEdit uses a ruler to decide how wide the text is. If this bit is set, resizing the window automatically resizes the ruler. To create a program that works a little more like a real text editor, the ruler size should depend on the width of the printed page, not on the width of the window. In that case, the bit would be clear, and the program would have some other way of setting the ruler width.

bit 20 If this bit is clear, TextEdit lets the user select text. If it is set, the user can't select text.

In most cases, if you're going to let the user edit the text, you should let them select it, too.

bit 19 Imagine two open windows on the desktop, both with TextEdit boxes, and both with selected text. When you select one of them, the controls are drawn differently. The controls in the window that isn't active are just outlines, while the controls in the active window have arrows, thumbs, and so forth. If you set this bit, the selections will change, too. In the active window, selected text will be inverted, like you're used to seeing it. In the window that isn't active, TextEdit draws an outline around the text. I sort of like that. If you don't, clear this bit in your resource.

bits 18-0 These bits are not used, and should be set to 0.

```
{-1,-1,-1,-1},                          /* indent rect */
```

You don't want text smashed right up against the edge of the window. This rectangle tells how big of a margin to leave on each side. The values of -1 we've used tell TextEdit to use some default values.

```
$FFFFFFFF,                              /* vert bar */
```

This is the handle for the vertical scroll bar. If you don't want a vertical scroll bar, use nil. If you want TextEdit to create one for you, use $FFFFFFFF.

```
0,                                      /* vert Amount */
```

This tells TextEdit how far to scroll when the scroll bar arrow is used. A value of 0 tells TextEdit to use it's internal default.

```
nil,                                    /* hor bar */
```

This field is for the horizontal scroll bar. Unfortunately, TextEdit doesn't use it. Maybe someday it will.

```
0,                                      /* hor amount */
```

If horizontal scroll bars are ever added to TextEdit, this field will be the number of pixels to scroll.

```
nil,                                    /* style ref */
```

This is a reference to the styles to use in the TextEdit control. We won't be using predefined styles, so this field is set to nil in our resource.

```
0,                                      /* text descriptor */
```

This field tells what sort of value is in the next one. We've set it to 0, for a pointer.

```
nil,                                    /* text ref */
```

This field is a reference to the initial text for the TextEdit record. For a help screen, you might want to point this at another resource. For our text editor, we'll load the text manually, so this field is set to nil.

```
0,                                      /* text length */
```

This is the length of the text in the previous field.

```
nil,                                    /* max chars */
```

You can limit the number of characters in the TextEdit control, or tell TextEdit it can allow as many characters as it wants. That's what we've done by setting the field to nil.

```
nil,                                    /* max lines */
```

In later versions of TextEdit, you may be able to limit the number of lines in a TextEdit control. So far, this field isn't actually used, and must be set to nil.

```
nil,                                    /* Max chars per line */
```

In later versions of TextEdit, you may be able to limit the number of characters on a line. So far, this field isn't actually used, and must be set to nil.

```
nil,                                    /* max height */
```

I'm really not sure what this field might be used for some day, but it isn't used yet. It must be set to nil.

```
nil,                                    /* color ref */
```

This field is used for color tables. We aren't using one, so the field is set to nil.

```
4,                                      /* drawing mode */
```

You get some control over the drawing mode TextEdit uses. We've picked a drawing mode of 4, which is the numeric value for `modeForeCopy`. It's not very fast, but it uses the correct colors. If you're willing to play with the text colors a bit, you can speed up the text drawing by using one of the other drawing modes.

```
nil,                                    /* Filter Proc Ptr */
```

This parameter is for a filter procedure, which gives you control over some of the internal functions of TextEdit. We won't be using filter procedures in this course.

TextEdit does almost everything for you, but there is one chore you do have to take care of. Your window update procedure has to make one call to `DrawControls` to make sure the control actually gets drawn.

```
DrawControls(GetPort);
```

Problem 11-1: Start with the solution to Problem 10-6, which was the font sampler that used full menus. We'll use that program as the basis for our text editor, which will be used in several problems throughout the rest of the course. Remove the font sampler code, so all you have left is Frame with the font selection menus. Now add the TextEdit resource to the document window.

When you worked problem 10-6, you took out disk I/O. We'll put that back in later. You should leave the printing hooks in your program, but for now, don't try to implement printing. There are some details about TextEdit we need to talk about before you try to print a document. Finally, while we're starting with the font sampler for the very good reason that we'll implement font selection, we haven't covered how to do it yet. This problem is just about getting the basic window up. You should be able to type text, use the mouse, and use cut, copy, paste and clear. Leave everything else for later.

Be sure and change the frame bits word in the window resource to get rid of the `TaskMaster` scroll bars and grow box. TextEdit uses it's own, and you don't want two sets of controls in the same window. Also, remember that TextEdit is a new tool – you need to update your tool startup list appropriately.

Editing with TextEdit

TextEdit supports quite a few editing operations, doing all of the work for you. We'll explore them in this section, using the program you wrote as a solution to Problem 11-1.

Naturally, you can type characters from the keyboard. The return key starts a new line, and unlike most other tools, TextEdit supports the tab key. You can even change where the tab stops are located, although we won't deal with that in this lesson.

TextEdit uses the standard editing keys, and supports the standard mouse editing features.

Key	**Description**
left arrow	Backs up one character. Hold down the command key to back up to the start of the last word, or the option key to move to the start of a line of text. You can hold down the shift key while you move the cursor to select text.
right arrow	Moves forward one character. Hold down the command key to move to the next word, or the option key to move to the end of the line. You can hold down the shift key while you move the cursor to select text.
up arrow	Moves up one line. Hold down the command key to move to the top of the page, or the option key to move to the start of the document. You can hold down the shift key while you move the cursor to select text.
down arrow	Moves down one line. Hold down the command key to move to the bottom of the page, or the option key to move to the end of the document. You can hold down the shift key while you move the cursor to select text.
delete	If any text is selected, the delete key deletes the text. If nothing is selected, the character to the left of the insertion point is deleted.
clear	This key works like the delete key if anything is selected, but is ignored if nothing is selected.
control-F	This is a forward delete; it deletes the character to the right of the selection point. If any text is selected, it works like delete and clear, selecting the selected text.
control-Y	Deletes all of the characters from the insertion point to the end of the line.
control-X	Copies the selected text to the clipboard, then deletes it.
control-C	Copies the selected text to the clipboard.
control-V	Pastes the text in the clipboard into the document. If any text is selected, the selected text is deleted first.

Table 11-1: TextEdit Editing Functions

You can also use the mouse, of course. You can click to move the selection point, or click and drag to select characters. You can also double-click, which selects words, or even triple-click to select lines.

Finally, TextEdit knows about your Edit menu. After all, the Cut, Copy, Paste and Clear commands all have standard menu ID numbers. Well, TextEdit recognizes them, taking appropriate action when the commands are used.

Changing the Ruler

TextEdit can do some limited formatting of the text it displays – things like left or right justification, or picking out where the tab stops should go. The values that control these formatting options are collected in a ruler record that looks like this:

```
teRuler = record
    leftMargin:     integer;
    leftIndent:     integer;
    rightMargin:    integer;
    just:           integer;
    extraLS:        integer;
    flags:          integer;
    userData:       longint;
    tabType:        integer;
(* Change size of array for application. *)
    tabs:           array [1..1] of teTabItem;
    tabTerminator:  integer;
    end;
```

Listing 11-4: The Ruler Record

Setting things like tab stops and margins works best with dialogs or interactive rulers. We haven't talked about dialogs, and we won't talk about interactive rulers, so I'm going to ignore a lot of the fields in this record for now. If you want to read about all of the fields, flip back to Appendix A. We'll also talk about a few more of these fields later, when we create some sample dialogs. For now, I'd like to concentrate on the `just` field, which controls text justification.

Text justification is used to line up lines of text. Unjustified text, called left justified by TextEdit, is the sort you would normally get with a typewriter. The left edge of all of the lines match, but the right edges don't – they end after the last character of the last word. Right justification does just the opposite, lining up the right edges of each line, leaving a ragged left edge. Fill justification lines up both the left and right edges by inserting small amounts of space between each word (or character, in some editors). Fill justification is what you normally see in books, like this one. The last type of justification is centering, which puts the same amount of space on the left and right edge of the text.

The `just` field records the kind of justification TextEdit uses. You can use any of these values:

0	left justification
-1	right justification
1	center justification
2	fill justification

Since there are other fields in the record besides the `just` field, the best way to change the way text is justified is to read the current ruler record, change it, and then send the changed record back to TextEdit. `TEGetRuler` is used to read the record.

```
procedure TEGetRuler (rulerDescriptor: integer; rulerRef: univ longint;
   teH: teHandle);
```

`rulerDescriptor` This field tells whether `rulerRef` is a pointer, an existing handle, a resource ID, or a pointer to a location to put a handle TextEdit allocates

	for us. Since the ruler record doesn't have a fixed length, it's best to let TextEdit figure out how much room it needs and allocate a handle to hold it. For that, `rulerDescriptor` is set to 3.
`rulerRef`	This is the address where we want TextEdit to place the handle. If the variable where the handle will be stored is called `rulerHandle`, you would use a parameter of `@rulerHandle`.
`teH`	If there are several text edit controls in a single window, you can pass the handle for a specific control for this last parameter. We just use nil, which tells TextEdit to look at the active control. Since there's only one of them in our window, we know it will be the right one. Just be sure you use `SetPort` so your window is the active window.

After you change the ruler, call `TESetRuler` to force TextEdit to use the new values. The parameters are almost the same as for the `TEGetRuler` call. The only difference is that you pass 1 for `rulerDescriptor` instead of 3, and you pass the actual handle instead of a pointer to the handle.

Of course, you have to check for errors after the `TEGetRuler` call, since you wouldn't want to modify something you thought was a handle when `TEGetRuler` had run out of memory or something. Be sure and lock the handle before you use it, too, and dispose of the handle after the `TESetRuler` call.

Rulers do have one unfortunate limitation. Each ruler applies to the entire TextEdit document.

Problem 11-2: Add justification to the editor you wrote in Problem 11-1. Start by adding four new menu items to the Style menu that you already have for the font style. These new menu items should be at the bottom, and there should be a separator under the last font style. The new items are "Left Justify", Right Justify", "Fill Justify", and "Center". Use a check mark to show which one is currently in use, reading the setting from TextEdit to set the check mark.

When the user picks one of these menu items, set the appropriate value for `just` in the ruler record. Dim the menu items if there are no open windows. Check the appropriate style menu in your `CheckMenu` subroutine (the one you call from the event loop to highlight and unhighlight menus), reading the ruler to find out what style to check. If that seems unnecessary, try setting one style in one window, and another style in a second window, and switch back and forth – I would expect the style that is checked to switch, too.

Note: When I added these final checks to my `CheckMenu` subroutine, I finally noticed an impact on the program. The repeated calls to get the ruler were interfering with the flashing of the cursor. To fix the problem, I just moved the `CheckMenu` call to a spot right after the case statement that handles events, then did a simple check, calling it only if the event that was just handled wasn't a `nullEvt`. That way, the subroutine was only called if something happened. So, why not make this check earlier? Because the call wasn't a problem earlier, and there is no point in making a program more complicated until you have a reason!

Changing Fonts

TextEdit uses a style record to hold information about the current font. The style record looks like this:

```
teStyle = record
    teFont:     fontID;
    foreColor:  integer;
    backColor:  integer;
    userData:   longint;
    end;
```

Listing 11-5: `teStyle` Record

Knowing what you do about fonts, the fields in this record should make a lot of sense. The first one is the font ID; it's exactly the same one you used in the last lesson. The `foreColor` and `backColor` fields are the foreground and background colors. The last field is for your own use.

Setting the style is pretty easy. `TEStyleChange` changes the style for all of the text in the current selection. If there is no selection, `TEStyleChange` changes something called the null style record. That's just a 50¢ name for the default style.

```
procedure TEStyleChange (flags: integer; var stylePtr: teStyle;
   teH: teHandle);
```

`flags` You don't always want to change everything about a particular style. For example, if the user selects some text and picks a font size of 10, you wouldn't want to set the font style, too. After all, some of the text might be bold, and some normal text. You can set and clear bits in this flag to tell `TEStyleChange` just what you want to change, and what you want left alone.

bits 15-7	Reserved; set to 0.
bit 6	If this bit is set, the font family is changed.
bit 5	If this bit is set, the font size is changed.
bit 4	If this bit is set, the foreground color is changed.
bit 3	If this bit is set, the background color is changed.
bit 2	If this bit is set, the user data is changed.
bit 1	If this bit is set, the style attributes are changed.
bit 0	If this bit is set, the style attributes are reversed. This only happens if all of the text that is selected already has a style that matches the attributes in the style record. If the selected text doesn't match the font style in the style record, setting this bit would have the same effect as setting bit 1. (More on this in a moment.)

`stylePtr` This is the style record itself. In the ORCA/Pascal interfaces, this parameter is defined as a var parameter, so you just pass a variable of type `teStyle`.

`teH` This is the handle of the TextEdit record to change. If you pass nil, the active TextEdit record is changed.

Most of this is pretty obvious, but bits 0 and 1 of the `flags` parameter may seem a bit odd. The reason for the difference is that a lot of editors will undo a style if you apply it twice. For example, if you select a word and pick out Bold from the style menu, the word changes to bold text. If you pick bold again, the text changes back to the way it was. Setting bit 0 when you change the font style will cause TextEdit to work that way. Setting bit 1, instead, tells TextEdit to just do it. You have to pick one or the other, though. If you set both bits, `TEStyleChange` will flag an error.

Towards the end of a programming project, you'll sometimes hear programmers muttering about the 80-20 rule. Normally that rule says that a program spends 80 percent of its time in 20 percent of the code. (I've also heard it as the 90-10 rule. I guess it depends on the kind of programs you write.) Towards the end of a programming project, though, the 80-20 rule takes on a new meaning: a programmer spends 80 percent of his programming time writing 20 percent of the program. Or, saying the same thing a different way, it's the details that sneak up and cause you grief. And that's what happens with changing fonts, too.

Everything we've talked about works just fine. There's just one tiny problem. When you pull down the font menu, you sort of expect to see a check mark beside the name of the font that you're using. Multiple fonts create a minor headache, though, since moving the cursor can change the font. Selecting text presents another problem, since the selected text might include more than one font. A good way to handle this is not to check a font name unless all of the selected text uses the same font, but that assumes you know what font is being used. Well, you can find out.

`TEGetSelectionStyle` is a little more complicated than the call to set a selection. It does have to return more than one style, after all.

```
function TEGetSelectionStyle (var commonStyle: teStyle;
   styleHandle: TEStyleGroupHndl; teH: teHandle): integer;
```

`commonStyle` This is the field we actually need. `TEGetSelectionStyle` looks for all of the style elements that are the same in all of the selected text. Those values are placed in this style record. Of course, some style elements might be the same, while others are different – the selection might use the same font family, for example, but part of the selection could be bold while part is not. `TEGetSelectionStyle` also returns an integer flag word with bits telling which fields are valid.

bit 5	This bit is set if the font family is the same throughout the selection.
bit 4	This bit is set if the font size is the same throughout the selection.
bit 3	This bit is set if the foreground color is the same throughout the selection.
bit 2	This bit is set if the background color is the same throughout the selection.
bit 1	This bit is set if the user data field is the same throughout the selection.
bit 0	This bit is set if the font style attributes are the same throughout the selection.

`styleHandle` You have to pass a handle for this value, and it should be one that can be moved. `TEGetSelectionStyle` will resize it and put all of the style records that are used throughout the selection into the buffer, one after another. The very first word of the buffer is the number of style records.

`teH` As with the other calls, this is the handle for the text edit control you want to check. If you pass nil, TextEdit uses the active text edit control.

Problem 11-3: In this problem, you add font selection to the editor that you created in Problem 11-2.

Start by adding a call to `TEGetSelectionStyle` in your `CheckMenu` subroutine. Check and uncheck the various font menus, leaving all of the items unchecked if a characteristic is mixed. For example, if the font family varies, none of the font names should be checked.

Now call `TEStyleChange` to change the style when the user picks a new font family, font style, or font size.

Get rid of the call to `ChooseFont`, and get rid of the Choose Font command, to. There's nothing really wrong with this command, but it makes handling the various style selections a lot more complicated than it needs to be for our purposes, which is to learn about the toolbox.

You can also get rid of the font variable in the document record, since TextEdit is handling all of that for you, now.

Loading and Saving

The next step in writing the editor is to load and save text files. TextEdit has two calls that help. The first is `TEGetText`, which returns the text and the style information for the text in a TextEdit control. The text and style information come back in two separate buffers.

`TESetText` puts text into a TextEdit control. It's inputs are the same buffers returned by `TEGetText`.

```
function TEGetText (bufferDescriptor: integer; bufferRef: univ longint;
   bufferLength: longint; styleDescriptor: integer;
   styleRef: univ longint; teH: teHandle): longint;
```

`bufferDescriptor`	This flags word tells `TEGetText` what we are passing in `bufferRef`. It also tells `TEGetText` what format to use for the text. We'll use $0019, which tells `TEGetText` that `bufferRef` is the address of a variable, and that it should put a handle there. `TEGetText` will figure out how big the handle needs to be and allocate the memory. Of course, we need to remember to dispose of the handle when we're finished saving the text. This flags word also tells `TEGetText` to use a null terminated string for the format of the text.
`bufferRef`	We'll pass the address of our pointer variable in this spot. You can look up the call in the toolbox reference manual if you would like to find out what the other possible parameters are.
`bufferLength`	This field isn't used the way we're calling `TEGetText`. Pass a 0.
`styleDescriptor`	Pass a 3, which tells `TEGetText` that the `styleRef` field is a pointer to a variable where it should store a handle. `TEGetText` will allocate the memory and return the handle.
`styleRef`	This is where you pass the address of the variable where the style handle will be stored.
`teH`	This is the handle of the TextEdit control we want to know about. Pass nil, as usual, to tell `TEGetText` to use the active TextEdit record.
`TEGetText`	`TEGetText` is a function; it returns the total length of the text.

Both the text and style records need to be saved. `TEGetText` returns the length of the text buffer, so we can use that as part of the output file. You can write the length as a four byte value at the start of the file, so that you can easily read the length when you load the file. Right after that, you would write the actual text. Next would come the length of the style information, followed by the data returned in the style buffer. The only trick here is to figure out how long the style record is. You can do that by looking in the style record itself, which has this format:

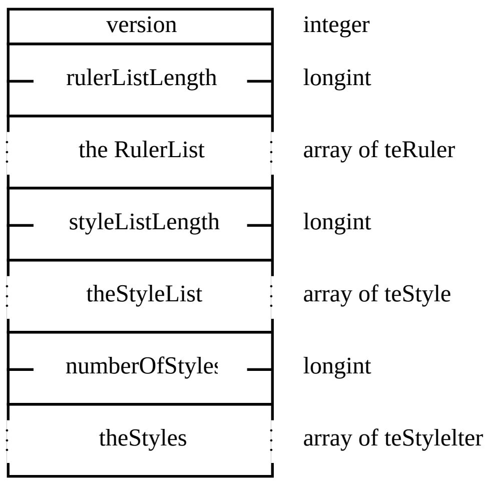


Figure 11-1: The `TEFormat` Record Returned by `TEGetText`

It will take a little pointer math, but you can scan through the record, adding up the lengths of the three long integers. Of course, you need to add an extra 14 to the length to account for the lengths themselves and the version number at the start of the record.

The part of the program that reads a file from disk can start by loading the entire file and figuring out where the text and style information start. With these values in hand, call `TESetText`.

```
procedure TESetText (textDescriptor: integer; textRef: teTextRef;
   textLength: longint; styleDescriptor: integer;
   styleRef: teStyleRef; teH: teHandle);
```

`textDescriptor`	Pass 5, telling `TESetText` that `textRef` is a pointer to a text buffer, and that `textLength` contains the length of the text.
`textRef`	Pass a pointer to the first byte of the text. (The `teTextRef` type is actually a longint, so pass the ord4 of the pointer.)
`textLength`	Pass the length of the text here.
`styleDescriptor`	Pass 0, telling `TESetText` that `styleRef` is a pointer. (The `teStyleRef` type is actually a longint, so pass the ord4 of the pointer.)
`teH`	Pass nil, telling `TESetText` to write to the active TextEdit control.

I've only told you enough to use these calls one way. There are lots of other formats you can use. As with any new call, it would be a good idea to look these up in the toolbox reference manual or Appendix A and read the complete description.

Problem 11-4: Add file input and output to the editor from Problem 11-3. Use a file type of $06 and auxiliary file type of $0001, which is a general binary file.

Hint: The subroutine we developed in Lesson 6 for writing a file using GS/OS calls was designed to write the file in one block. You could do it that way, but since TextEdit gives you the file in two pieces, and you have to add some length words, it will be tough. A better way to handle writing the file would be to do several writes. That's something we talked about in Lesson 6, but didn't actually do.

Printing

TextEdit has a fairly simple call that lets you print the document. It's called `TEPaintText`.

```
function TEPaintText (thePort: grafPortPtr; startingLine: longint;
   var destRect: rect; flags: integer; teH: teHandle): longint;
```

`thePort`	This is the `grafPort` where the text will be drawn. `TEPaintText` can actually draw to any `grafPort`. When you are printing, this parameter should be the `grafPort` the Print Manager sets up.
`startingLine`	This is the line number for the first line to print. It should be zero for the first call. `TEPaintText` is a function; it returns the line number for the next line to print, so you need to keep track of that value and pass it on the second and all future calls. `TEPaintText` will return -1 when there are no more lines to print.
`destRect`	This is the size of one page. I actually covered the information you need to find the page size way back in Lesson 7, but that was a long time ago. To review, you need to look in the `prInfo` record, which is imbedded in the print record. Inside that record is a rect data structure called `rPage`; `rPage` is a rectangle that's the same size as the page. You might want to look at the data structures listed in Appendix A or flip back to Listing 7-2 for a detailed look at the `prInfo` record.
`flags`	You can ask `TEPaintText` to just calculate the lines that would be printed, without actually printing them. The big use for this feature is to skip pages. To calculate the proper number of lines without printing, set bit 14. For normal printing, pass a 0. All bits except bit 14 must be 0.
`teH`	This is the handle for the TextEdit control to print. This should be the actual handle; we'll look at how to find it in the next section.

We'll add printing to the text editor in Problem 11-5.

Did the Document Change?

The last step to finish up the basic editor is to see if the document has changed. The reason we need to know is so we can warn the user if they attempt to close a document that has been changed since the last time it was saved. You've done this before, back in Lesson 9. In fact, if you've been snatching code from that lesson, you may have already added most of what you need.

If the file has changed, you need to bring up an alert that warns the user and asks if he wants to save the file. The alert should have three buttons: Yes (the default), No, and Cancel (which stops the close operation). You can create the alert itself with `AlertWindow`, which you've been using for a long time.

The problem is when to display the alert. After all, TextEdit is doing so much of the work for you that it's hard to tell if the user even typed anything! To find out, you need to delve into the TextEdit record itself to grab a flag called `fRecordDirty`. This flag is bit 6 of the `ctrlFlag` byte, which is located 16 bytes into the text edit record. The text edit record is the thing we've been ignoring up until now, passing nil whenever we were supposed to pass a record handle. All you need to get the handle is the control ID number and the window pointer; after that, you call `GetCtlHandleFromID`, passing the window pointer and the control ID. `GetCtlHandleFromID` returns the control handle.

Putting all of that together, here's a subroutine that checks to see if a text edit record has changed.

```
function TextEditChange (id: integer; wPtr: grafPortPtr): boolean;

{ Check to see if the text or style information has changed     }
{                                                               }
{ Parameters:                                                   }
{    id - control ID for the record to check                    }
{    wPtr - window containing the record                        }
{                                                               }
{ Returns: true if the record changed, false if not or error    }

var
   ctlHandle: ctlRecHndl;               {control handle}
   ptr: ^byte;                          {work pointer}

begin {TextEditChange}
ctlHandle := GetCtlHandleFromID(wPtr, id);
TextEditChange := false;
if ToolError = 0 then begin
   HLock(ctlHandle);
   ptr := pointer(ord4(ctlHandle^)+16);
   TextEditChange := (ptr^ & $40) <> 0;
   HUnlock(ctlHandle);
   end; {if}
end; {TextEditChange}
```

Listing 11-6: Subroutine to See if Text Has Changed

Problem 11-5: Add printing and checks for changes to your program. Before quitting, make sure all windows are closed. Before closing a document, check to see if the text has changed. If so, bring up a dialog like the one discussed in this section.

If the user picks Cancel, your program should not close the window. If you are in the process of quitting, you should stop the process.

If the user picks No, go ahead and close the window.

If the user picks Yes, save the document and then close the window.

One last detail: your save subroutine needs to clear the `fRecordDirty` bit, just in case the user saves the document, but leaves it open. TextEdit has no idea when (or if) you save the document, so it depends on you to clear the `fRecordDirty` bit after you've done a save. You can do that with a subroutine based on the one in Listing 11-6.

Summary

This lesson introduced TextEdit, a very handy tool for displaying and editing text. The problems in the lesson ended with a simple text editor.

Tool calls used for the first time in this lesson:

```
GetCtlHandleFromID      TEGetRuler              TEGetSelectionStyle
TEGetText               TEPaintText             TESetRuler
TESetText               TEStyleChange
```

Resource types used for the first time in this lesson:

```
rControlTemplate
```

Lesson 12 – Scraps

Goals for This Lesson

This lesson introduces the Scrap Manager. The Scrap Manager is the tool that is used for cutting, copying and pasting. When you use the Scrap Manager for these operations, you can create programs that can cut, copy and paste information to and from other applications.

The Scrap Manager

The obvious difference between programs written for the Apple IIGS or Macintosh and programs written for MS DOS computers is the desktop interface. Another difference that, in many ways, is just as important but a lot less obvious is that programs on the Apple IIGS and Macintosh have a remarkable ability to share data. You can bring up a spread sheet, do some pretty heavy duty calculations, then move some of the cells to a word processor to create a table for a report. Pictures move just as easily as text. The Scrap Manager makes all of this possible with amazingly little effort.

There are two problems that the Scrap Manager has to handle to make cutting and pasting between applications work smoothly. The first is where the information is stored. After all, on the Apple IIGS at least, you have to shut down one application and start another to move information from a spread sheet to a text editor. The second problem the Scrap Manager has to deal with is the format for the information. The information in a spread sheet is obviously very different from the information in a word processor, and the Scrap Manager has to handle this some way.

Scraps

Scraps are the chunks of information you cut and paste. The Scrap Manager keeps track of scraps for you, saving the information your program passes when it cuts or copies, and passing the information back when it pastes. When the program cuts or copies information, it calls `PutScrap` to tell the Scrap Manager to save the information.

```
PutScrap(numBytes, scrapType, srcPtr);
```

The first parameter is the number of bytes you want the Scrap Manager to remember. This parameter is a long integer, so you don't have to worry about 64K limits. The next parameter is the scrap type, which tells the Scrap Manager what kind of information it is remembering. We'll talk a little more about the scrap type in a moment. The last parameter is a pointer to the information the Scrap Manager should remember.

The Scrap Manager makes a copy of the information, so you can safely change the original information right after the call. For example, if you are cutting a word out of a text buffer, you can pass a pointer to the start of the word in `srcPtr`, then remove the word right away. Since `PutScrap` made it's own copy, you don't have to preserve the buffer.

When your program needs to get the information that's in the scrap to paste it into the document, you start with a call to `GetScrapSize` to find out how big the scrap is. Of course, if the size of the scrap is zero, there's nothing to paste.

```
size := GetScrapSize(scrapType);
```

Once you know how big the scrap is, you can ask for the handle.

```
scrapHandle := GetScrapHandle(scrapType);
```

You can lock the handle and copy the information out, just like you would with any other handle. Be sure to unlock the handle when you are through. Don't delete the handle when you're though, though: this is the Scrap Manager's copy, so all you should do is copy the information into your document.

Scrap Types

One of the problems I said the Scrap Manager had to solve was the problem of what format to use for all of the information. In a sense, the Scrap Manager solves this problem by ignoring it. Instead of worrying about the format any particular program wants to use, the Scrap Manager lets you specify any one of 65536 different scrap types! Well, that's obviously enough to handle more than just a couple of formats, but it presents a problem for your program. You simply can't handle that many kinds of input data.

The whole issue is resolved by having two different kinds of scrap, private scraps and public scraps. A private scrap is yours and yours alone. You pick out a number for the scrap type, and you decide what format to use for the information. If you are writing a dungeon style game, your data format might include pictures for animating the figure, hit points, and so forth. For a spread sheet, you might include the formula used to compute the value in a cell.

The other kind of scrap is the public scrap. There are two kinds of public scrap. Text scraps have a scrap type of 0. Text scraps are just ASCII character sequences. To find the length, you need to look in the length of the scrap. Scrap type 1 is set aside for picture scraps. Pictures are a kind of recorded drawing that you can create or draw using QuickDraw II.

Let's imagine a sophisticated word processor to see how these scrap types work together in a real program. Let's say you are using the text editor, and select a chapter from a book like this one, then copy it. The text editor will actually create two different scraps, one text scrap and one more complicated scrap that includes font information, the position and contents of pictures, tab settings, and so forth.

Now let's imagine pasting this scrap into two different programs, the original word processor and a paint program. When you paste the scrap into the word processor, it looks for its private scrap type – and finds it. The text editor uses the private scrap type, using all of the font, tab, and picture information it saved in the private scrap.

Now imagine pasting the same scrap in a paint program. The paint program doesn't have any idea what to do with the text editor's private scrap. In fact, it doesn't even look for it. Instead, the paint program probably looks for it's own private scrap type, which might contain things like the color palette to use with a picture. It doesn't find it, so its next step is to look at one of the public scraps. This is the key point that makes the Scrap Manager work: the paint program, like every other program, should be able to read and use either one of the public scrap types. Naturally, it will look for the picture scrap first. Since the text editor didn't create a picture scrap, it will then grab the text scrap and draw the text in the picture window.

Now, it's pretty obvious that the text won't look quite the way it did in the text editor. Any pictures are lost. Tabs, fonts, and other formatting information is also lost. But the text itself does get passed. A spread sheet would also create a text scrap, and the cell contents could be pasted into the word processor or onto a graph in the paint program. In short, because every program must create at least one of the public scrap types, and every other program is responsible for being able to use either one of them, you can always move information from one program to another.

In practice, there are problems. Almost any program can read and use text, but not all programs can handle a picture. Our small text editor can't, for example. In that case, the program treats the scrap as if it were empty. It's not perfect, but it works in the vast majority of cases.

Most people wouldn't expect to be able to paste a picture into a text only editor, anyway, so they won't even be surprised when it doesn't work.

Multiple Scraps

Creating more than one scrap type is really pretty easy. Just call `PutScrap` one time for each scrap type, passing the proper scrap type as a parameter each time. The Scrap Manager handles all of the details.

Using multiple scraps is almost as easy. When you call `GetScrapSize`, check first for your private scrap type. If the Scrap Manager returns an error or a scrap length of zero, check for the public scrap type that is easiest to handle. A text editor would check for a text scrap, for example. If you still come up blank, check the other public scrap type.

It's actually possible to accumulate information in a scrap. For example, if you call `PutScrap` ten times in a row with the same information, then paste the result, you'll get ten copies of the information. That can be useful in situations where you have to preprocess the information, and may not have enough memory to create the entire scrap in one chunk. Most of the time, though, it's something you want to avoid, and you avoid it by calling `ZeroScrap` just before you start copying information into the scrap with the `PutScrap` calls. `ZeroScrap` doesn't have any parameters, and doesn't return anything. It just deletes the current scrap, so the Scrap Manager creates a new scrap with the next series of `PutScrap` calls.

The Scrap File

The second problem the Scrap Manager has to solve is where to put the scrap. For the short term, the answer is pretty obvious: the scrap is kept in memory. This doesn't work well when you switch between programs, though, since the Scrap Manager, like all of the tools, is shut down in the process of leaving your program. To handle this situation, the Scrap Manager lets you save the scrap to disk by calling `UnloadScrap`. `UnloadScrap` doesn't have any parameters. It just tells the Scrap Manager to save the scrap.

You can actually tell the Scrap Manager where to save the scrap, but that's rarely a good idea. After all, the idea is for another program to be able to find your scrap. Unless you're trying to hide the scrap, you want the Scrap Manager to be able to find it in its normal place. That's in the system folder; you need to know that so you are aware that your program may ask for the system disk. Beyond that, the location, type, and all other information about the scrap file is really not your concern. You can look it up if you want, but I wouldn't bother. (And didn't.)

When your program starts, you need to make a call to `LoadScrap`. That tells the Scrap Manager to look for the scrap file on disk, loading it into memory where it can be used by `GetScrapSize` and `GetScrapHandle`. Like `UnloadScrap`, `LoadScrap` doesn't have any parameters.

Problem 12-1: Add scrap file handling to the text editor you created in Problem 11-5. All you really have to add are the calls to `LoadScrap` and `UnloadScrap`. `LoadScrap` should be called right as your program starts, and `UnloadScrap` just before you shut down the tools.

A Scrapbook Program

We'll put a lot of this information to use by creating a scrapbook program. There are two tool calls I'd like to describe which are needed for this program but don't really fit in with the lesson as a whole, and some program design issues I'd like to talk about, too. So, rather than jump right into the problem, I'd like to discuss those other issues first.

A scrapbook is a program that let's you collect a lot of scraps in a single, convenient location. Our program, like most scrapbooks, will be pretty simple. It will handle multiple scraps, displaying the first one in a window when the program starts. It will call `LoadScrap` when it starts, and when you select Paste, it will create a new scrap entry and put the pasted scrap in the new spot. Cutting will remove a scrap from the scrapbook, placing it in the scrap. Copying will also place a scrap in the clipboard, but in this case the scrap won't be removed from the scrapbook. Finally, Clear will remove a scrap from the scrapbook without calling the Scrap Manager to put it in the clipboard.

Scrapbooks generally only display one scrap at a time, and ours is no exception. For now, we'll use a menu called Move with two items, Next Scrap and Last Scrap, to move through the scraps in the scrapbook.

The scrap itself will be displayed in the scrap window, displaying it as a picture if possible, and as a text scrap if there is no picture. If a scrap is loaded which doesn't have a text or picture scrap type, the scrapbook should post an appropriate message.

Figure 12-1 summarizes the program, showing the menus and the scrap window from the solution to Problem 12-2.

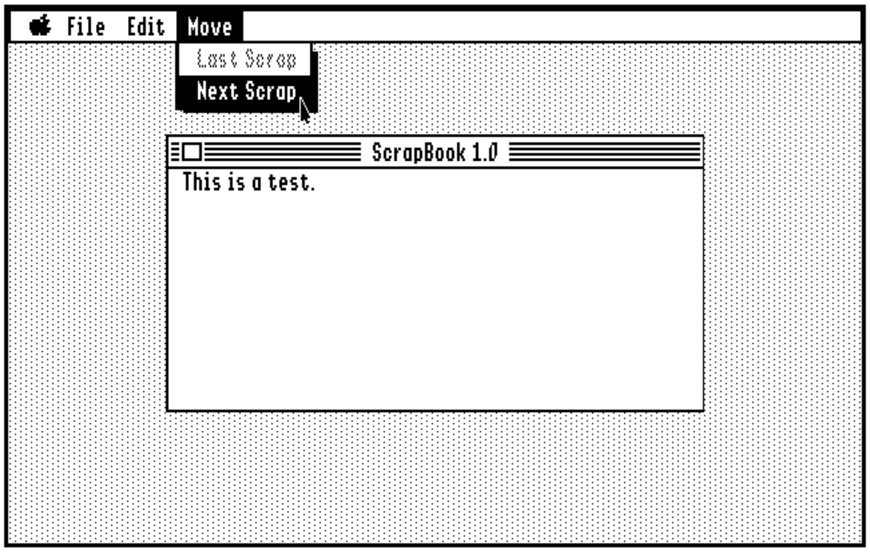


Figure 12-1: The Scrapbook Program

There are three internal design issues you will have to deal with before you can write this program. The first is the matter of how to store the scraps in the scrap file. The solution disk has a scrap file for you to use, so you should use the same file format the solution uses. The file consists of a series of scraps. Each scrap starts with a count word; this simply tells you how many scrap types exist for a particular scrap. If there are no more scraps, the count word is zero. The count word is followed by the scrap records, one for each scrap type. Each scrap record consists of three fields. The first is a two-byte scrap type. The next field is a four-byte scrap length. The last field is variable length; it's the scrap contents. It's worth pointing out that we really don't care what the scrap contents are; all we need to know is the length. To skip to the next scrap, then, you would add the scrap length plus six to the location of the first scrap. The type for the file is $06, a general binary file, so you can use standard Pascal file I/O.

The next issue that you need to deal with is how to figure out what the scrap types are, so you can paste all of them in the scrapbook. After all, it would be a shame to miss the preferred scrap type for a program, but it just wouldn't make much sense to find the scrap length for each and every possible scrap type. The `GetIndScrap` call is used by scrapbooks to find and read all of the scrap types for a scrap. It looks like this:

```
scrapBuffer = record
   scrapType: integer;
   scrapSize: longint;
   scrapHandle: handle;
   end;

GetIndScrap(index: integer; var buffer: scrapBuffer);
```

To get all of the scraps, start calling `GetIndScrap` with an index of 1. It will fill in the scrap buffer you pass with the scrap type, scrap size, and scrap handle for the first scrap. Repeat this with an index of 2, and so forth, until `GetIndScrap` returns an error. At that point, you've read all of the scrap types.

The last design issue is how to display the scrap. You already know how to draw text strings, so displaying a text scrap is no problem. What about picture scraps, though? The answer is, fortunately, very simple. QuickDraw II Auxiliary has a call called `DrawPicture`. To draw a picture scrap, you just pass the handle to `DrawPicture`, along with the destination rectangle.

```
procedure DrawPicture (picHandle: handle; var destRect: rect);
```

There is still one overriding problem with this program, and that is that using it is pretty awkward. To put a scrap in the scrapbook, you have to cut or copy it from a program, quit, run the scrapbook, and paste it into the scrapbook. To paste a scrap from the scrapbook into a document, you have to quit the original program, run the scrapbook, copy the scrap, rerun the program, and paste the scrap. To be technical, ick.

The solution is to turn the scrapbook into a desk accessory, so it can be used by almost any desktop program without leaving the program. While there won't actually be a problem assigned to convert this program into an NDA, you'll eventually learn all of the things you need to know to make the conversion.

Problem 12-2: Write the scrapbook program. To be useful, this program would eventually be turned into a desk accessory, so it can't depend on any custom menu items, and must have a single window. That makes the detailed requirements look a little strange for an application!

Your program should have a single, fixed size window that stays open all of the time. The window should be moveable, though. The program should support cut, copy, paste, clear and close, all of which are standard menus. (Close should do what Quit normally does; for now, you can hook the two menu items together, so both quit.) The program should look for a scrap file when it starts, loading it if there is one, and automatically write the scrap file before quitting. Use the name "ScrapBook" for the file; you'll find one in the Lesson 12 folder on your disks. The single new menu, Move, should have two items, Next Scrap and Last Scrap.

Internally, I'd suggest setting up the scraps as a doubly linked list of records, one record for each scrap. That makes it easier to insert and delete scraps. Inside each record, you'll need another linked list; this one lists the various scrap styles for a particular scrap. The scraps themselves should be in moveable handles.

I'd suggest starting with the solution to Problem 9-1, which also uses a single window.

Test your program by moving a scrap from your text editor to the scrapbook, then by moving a different scrap back to a text editor document. You might try some of your other desktop applications, too. Some of them will support the Scrap Manager, while some will not.

Summary

This lesson covered the Scrap Manager, and how it is used to create programs that can cut, copy and paste information. It showed how programs share information, too.

Tool calls used for the first time in this lesson:

```
DrawPicture        GetIndScrap        GetScrapHandle     GetScrapSize
LoadScrap          PutScrap           UnloadScrap        ZeroScrap
```

Lesson 13 – Controls, Part 1

Goals for This Lesson

In this lesson, you will learn to create windows with your own controls. We'll cover a few basic controls, concentrating on how the program interacts with the Control Manager and Window Manager to create and use controls. More advanced controls will be covered in Lesson 14.

A Quick History Lesson

Before getting started, I'd like to caution you a little bit about the Control Manager and how it has grown over the years. I hope you are reading *Apple IIGS Toolbox Reference* as you go through the course. If so, it would be pretty natural to flip open volume 1 and read the introductory section of the chapter on the Control Manager. If you do that, you're going to find something that doesn't look much like what this lesson covers.

The Control Manager has changed a lot over the years. One of the major changes is the use of resources, with a clean, unified mechanism for defining controls, coupled with a mechanism for attaching them to a particular window from inside the resource fork. None of this was possible when the original two volumes of the toolbox reference manual were released. Back then, all controls were created by setting up a record and making a Control Manager call. In the next two lessons, we'll use the newer, resource based method exclusively.

Control Records and Control Lists

Actually, you've already created a control, and it's the most complicated control of all. The text editor you wrote in Lesson 11 used a text edit control. The resource you filled in to create the text edit record is actually a special case of a much more complicated resource. The complete `rControlTemplate` resource, which includes the text edit control as a subset, is shown in Listing 13-1. This is the raw definition from the Types.rez header file used by the Resource Manager. Seen all in one chunk, it can be very intimidating, but we'll break it down in byte size pieces over the next two lessons. I'm showing you the resource here so you have something to refer back to.

```
type rControlTemplate {
        integer = 3+$$optionalcount (Fields);/* pCount must be at least 6 */
        _mybase_ longint;                    /* Application defined ID */
        rect;                               /* controls bounding rectangle */
        switch {

        case SimpleButtonControl:
            key longint = 0x80000000;        /* procRef */
            optional Fields {
                _mybase_ integer;            /* flags */
                _mybase_ integer;            /* more flags */
                _mybase_ longint;            /* refcon */
                _mybase_ longint;            /* Title Ref */
                _mybase_ longint;            /* color table ref */
                KeyEquiv;
            };
        case CheckControl:
            key longint = 0x82000000;       /* procRef */
            optional Fields {
                _mybase_ integer;           /* flags */
                _mybase_ integer;           /* more flags */
                _mybase_ longint;           /* refcon */
                _mybase_ longint;           /* Title Ref */
                integer;                    /* initial value */
                _mybase_ longint;           /* color table ref */
                KeyEquiv;
            };

        case RadioControl:
                key longint = 0x84000000;   /* procRef */
            optional Fields {
                _mybase_ integer;           /* flags */
                _mybase_ integer;           /* more flags */
                _mybase_ longint;           /* refcon */
                _mybase_ longint;           /* Title Ref */
                integer;                    /* initial value */
                _mybase_ longint;           /* color table ref */
                KeyEquiv;
            };
        case scrollControl:
            key longint = 0x86000000;       /* procRef */
            optional Fields {
                _mybase_ integer;           /* flags */
                _mybase_ integer;           /* more flags */
                _mybase_ longint;           /* refcon */
                integer;                    /* Max Size */
                integer;                    /* viewSize */
                integer;                    /* initial value */
                _mybase_ longint;           /* color table ref */
            };

        case statTextControl:
            key longint = 0x81000000;       /* procRef */
            optional Fields {
                _mybase_ integer;           /* flags */
                _mybase_ integer;           /* more flags */
                _mybase_ longint;           /* refcon */
                _mybase_ longint;           /* Text Ref */
                integer;                    /* text size */
                integer leftJust,
                        centerJust,
                        fullJust,
                        rightJust = -1;     /* text justification */
            };

        case editLineControl:
            key longint = 0x83000000;       /* procRef */
            optional Fields {
                _mybase_ integer;           /* flags */
                _mybase_ integer;           /* more flags */
                _mybase_ longint;           /* refcon */
                integer;                    /* Max Size */
                _mybase_ longint;           /* resource ID of the text */
                integer;                    /* password character - 6.0 */
            };

        case PopUpControl:
            key longint = 0x87000000;       /* procRef */
            optional Fields {
                _mybase_ integer;           /* flags */
                _mybase_ integer;           /* more flags */
                _mybase_ longint;           /* refcon */
                integer;                    /* Title Width */
                _mybase_ longint;           /* menu Ref */
                integer;                    /* Initial Value */
                _mybase_ longint;           /* Color table ref */
            };

        case ListControl:
            key longint = 0x89000000;       /* procRef */
            optional Fields {
                _mybase_ integer;           /* flags */
                _mybase_ integer;           /* more flags */
                _mybase_ longint;           /* refcon */
                integer;                    /* list size */
                integer;                    /* List View */
                _mybase_ integer;           /* List Type */
                integer;                    /* List Start */
                longint =0 ;                /* member drawing routine */
                integer;                    /* ListMemHeight */
                integer;                    /* List Mem Size */
                _mybase_ longint;           /* List Ref */
                _mybase_ longint;           /* Color Ref */
            };

        case growControl:
            key longint = 0x88000000;       /* procRef */
            optional Fields {
                _mybase_ integer;           /* flags */
                _mybase_ integer;           /* more flags */
                _mybase_ longint;           /* refcon */
                _mybase_ longint;           /* color table ref */
            };

        case PictureControl:
            key longint = 0x8D000000;       /* procRef */
            optional Fields {
                _mybase_ integer;           /* flags */
                _mybase_ integer;           /* more flags */
                _mybase_ longint;           /* refcon */
                _mybase_ longint;           /* picture ref */
            };

        case editTextControl:
            key longint = 0x85000000;       /* procRef */
            optional Fields {
                _mybase_ integer;           /* flags */
                _mybase_ integer;           /* more flags */
                _mybase_ longint;           /* refcon */
                _mybase_ longint;           /* text flags */
                rect;                       /* indent rect */
```

```
                _mybase_ longint;           /* vert bar */
                integer;                    /* vert Amount */
                _mybase_ longint;           /* hor bar */
                integer;                    /* hor amount */
                _mybase_ longint;           /* style ref */
                _mybase_ integer;           /* text descriptor */
                _mybase_ longint;           /* text ref */
                longint;                    /* text length */
                longint;                    /* max chars */
                longint;                    /* max lines */
                integer;                    /* Max chars per line */
                integer;                    /* max height */
                _mybase_ longint;           /* color ref */
                _mybase_ integer;           /* drawing mode */
                _mybase_ LongInt;           /* Filter Proc Ptr */
            };

        case IconButtonControl:
            key longint = 0x07FF0001;       /* procRef */
            optional Fields {
                _mybase_ integer;           /* flags */
                _mybase_ integer;           /* more flags */
                _mybase_ longint;           /* refcon */
                _mybase_ longint;           /* Icon Ref */
                _mybase_ longint;           /* Title Ref */
                _mybase_ longint;           /* color table ref */
                _mybase_ integer;           /* Display mode */
                KeyEquiv;
            };

        case rectangleControl:
            key longint = 0x87FF0003;       /* procRef */
            optional Fields {
                _mybase_ integer;           /* flags */
                _mybase_ integer;           /* more flags */
                _mybase_ longint;           /* refcon */
                integer;                    /* pen height */
                integer;                    /* pen width */
                hex string[8];              /* penmask */
                hex string[32];             /* penpattern */
            };

        case thermometerControl:
            key longint = 0x87FF0002;       /* procRef */
            optional Fields {
                _mybase_ integer;           /* flags */
                _mybase_ integer;           /* more flags */
                _mybase_ longint;           /* refcon */
                integer;                    /* value for pos of mercury */
                integer;                    /* scale for mercury */
                _mybase_ longint;           /* color table reference */
            };

        };
};
```

Listing 13-1: The Complete `rControlTemplate` Resource

When you created the text edit control for the text editor, you told the Window Manager to look for a single `rControlTemplate` resource. It's pretty rare to have a window with just one control, so in most cases, we want to tell the Window Manager to look for a list of controls. We do that by coding 9 as the last byte of the last parameter in the `rWindParam1` resource; that's the field with the comment of `/* wInVerb */`. In the text editor, this field is $0802; now it will be $0809. This tells the Window Manager that the resource ID in the previous field (labeled `wStorage`) is the resource ID for an `rControlList` resource, rather than an `rControlTemplate` resource. (The $x8xx part is used for the window color table.) The `rControlList` resource is a list of resource IDs for `rControlTemplate` resources. Listing 13-2 shows a typical `rControlList` resource. This one lists three controls, with resource IDs of 1001, 1002, and 1003. If you need more controls, just tack a few more resource IDs on the end of the list.

```
resource rControlList (1001) {
   {
      1001,
      1002,
      1003
      }
   };
```

Listing 13-2: A Typical `rControlList` Resource

Simple Buttons

The first control we'll look at is the simple button. Simple button controls are the (usually) oval shaped buttons with text in the middle, like the ubiquitous OK button shown in Figure 13-1.

Figure 13-1: A Typical Simple Button Control

You can click on a simple button, but that's about it. When you click on the simple button, the program does something right away. The OK button, for example, is usually the button you press to accept some choices you've made in a complicated dialog, or to acknowledge that you have read the message in an alert.

Defining the Control

Listing 13-3 shows the `rControlTemplate` resource for an OK button.

```
resource rControlTemplate (1001) {
   1,                                /* control ID */
   {100,10,0,0},                     /* control rect */
   SimpleButtonControl {{
      $0001,                         /* flags */
      $3002,                         /* more flags */
      0,                             /* refcon */
      1001,                          /* Title Ref */
      0,                             /* color table ref */
      {"\$0D","\$0D",0,0}            /* key equivalents */
      }};
   };
```

```
 resource rPString (1001) {"OK"};
```

Listing 13-3: Resources for an OK Button

The first two lines are common to all controls. The first of these is the control ID, which works sort of like a menu ID. Each control in the window has a unique ID number, and it's that ID number that we usually use to identify a control. In some cases, the control handle is used, instead.

A control ID of 1 has a special meaning in dialogs. The control with an ID of 1 is the default control. The default control is frequently a simple button, and when it is a simple button, it's the one with a double outline. The default control should be hooked to the return key, so that the window or dialog acts like the default control was picked if the return key is pressed. If there is no default control, there shouldn't be a control with a control ID of 1, either.

The default control should represent the "normal" action for the control, which should also be the safe action. For example, if you are about to reformat a hard disk, and you're asking the user if you should go ahead, the default action should be to not format the disk. Make the user pick the dangerous action by hand! The default button isn't always the OK button.

The next line is a rectangle that tells the Control Manager where to put the control. The first two coordinates are the top and left coordinates for the control, respectively, while the last two are the bottom and right edges for the control. Of course, there is the minor issue that you may not know how big the button should be, especially if the user picks out an alternate system font or changes the button message to another language. For most controls, the Control Manager can figure out the correct bottom and right coordinates for you. To ask the Control Manager to pick out the size, set the last two coordinates for the rectangle to 0, as we did in the example.

The next line is the one that finally determines just what sort of control we are creating, in this case a `SimpleButtonControl`.

The next four entries can change from control to control, but they don't change very often. All of the controls defined as of System 6.0 have two integer flag words and a long integer field you can use for your own purposes, and most controls also have the fourth parameter, a title reference. There's a lot of variation in how the two flag words are used, though, so we'll give a little table with each control that lists the meaning of each of the flag bits.

`flag` bits	use
15-8	Reserved; set to 0.
7	0 for a visible control, 1 for an invisible control.
6-2	Reserved; set to 0.
1-0	Determines the style for the button. There are four button styles: 00 An oval button with a single line outlining the button. 01 An oval button with a double outline. The double outline should only be used for the default button. 10 A square button with a single line outlining the button. 11 A square button with a drop shadow. This is the default button form for the square button.

`moreFlags` bits	use
15-14	Reserved; set to 0.
13	Set this bit to 1 if the button has a keystroke equivalent.
12	This bit tells the Control Manager that you are using a standard control. It must be set to 1.
11-4	Reserved; set to 0.
3-2	Defines the type of reference in the color table field. 00 Color table reference is a pointer. 01 Color table reference is a handle. 10 Color table reference is a resource ID.

1-0	Defines the type of reference in the title field. 00 Title reference is a pointer. 01 Title reference is a handle. 10 Title reference is a resource ID.

The next field is the `refcon` field. It's a long integer value you can use for your own purposes.

The title reference field is the resource ID for an `rPString` resource. This is the string that will appear in the button.

The next field is the color table reference. It can be pretty distracting to have chartreuse buttons on a mustard background, but with a little effort, you can create truly annoying colored buttons. We'll cover that topic in detail later, but for now, use a pointer of 0 – which tells the Control Manager to pick a boring black on white for the button color.

The last field is used to define keyboard equivalents for the button. It this example, we've set up a key equivalent of `"\$0D"`, coding the numeric equivalent of the return key (13, or $0D) as an escape sequence. The first two entries in the key equivalent record are the uppercase and lowercase version of the key. Since the return key works the same way, whether or not the shift key is pressed, we've set both values to 13. You should use characters if they are printable. For example,

```
{"C","c",0,0}
```

would work just fine.

The last two items are the key modifiers and a flags word called the key care bits. These two words let you do fancy tricks, like only allowing the return key, and not allowing the enter key. I'm not going to go over them in detail here. If you want to treat the keys some special way, check out the detailed description in the toolbox reference manual.

Using the Control

One of the things `TaskMaster` checks for as it preprocesses an event is whether or not a keypress or mouse down event involves a control. If it does, `TaskMaster` does all of the work to track the mouse and highlight the control, then returns `wInControl`. Some fields in the event record tell us what actually happened:

`taskData2`	The handle for the control will be in this field. The control handle is used by several Control Manager calls, some of which we'll look at later.
`taskData3`	The part code for the control is in `taskData3`. The part code tells us what part of the control was hit by a mouse down event. For most controls, including simple buttons, there is only one part, so this value isn't very useful. We'll use the value later, though, for scroll bars.
`taskData4`	The control ID is placed here. This is the value we really need; it tells us which control was hit.

Problem 13-1: Create a program that opens a single alert style window. Use 320 mode graphics, since we'll add some colored objects to this program later in the lesson, and we want to be able to use lots of colors. Get rid of everything in the File menu except Quit and Close. (Close is required to support desk accessories, but your program should ignore it.)

Create two simple buttons in the window. The first should be called "Beep Once." It is the default button, and should have a double outline and use the return key as a key equivalent. The other button should be called "Beep Twice." It should have a normal outline, and a key equivalent of 2.

Your program should detect hits on the buttons, calling `SysBeep` once when the first button is pressed, and twice when the second button is pressed.

We're going to use this same program as a test bed over the next two lessons, creating a control sampler. Because of that, you need lots of room, so make the window as big as you can. The controls should be layed out more or less like you see in Figure 13-2.

Hint: Don't forget to call `DrawControls` in your window update procedure, just like you did for the text editor. The controls aren't drawn unless you do.

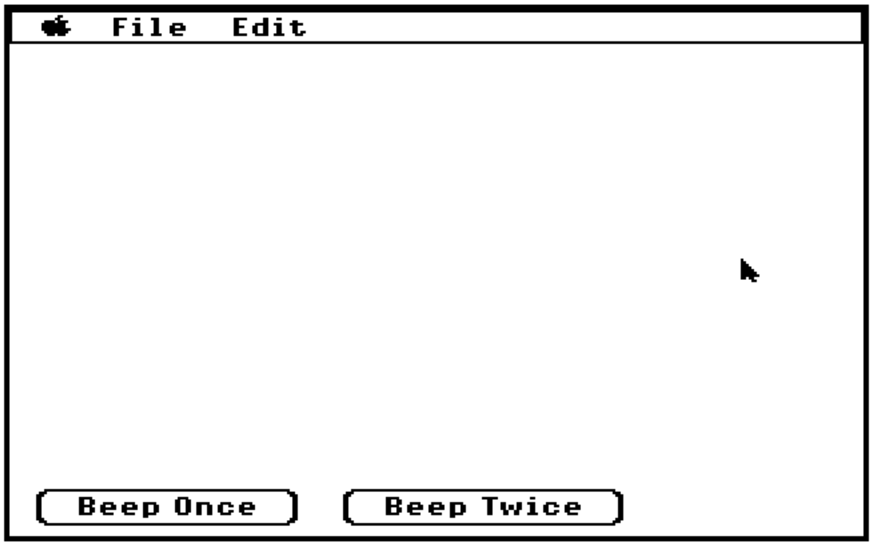


Figure 13-2: Control Sampler Window

Colored Controls

It's possible to create a control that isn't made up of black text and lines on a white background by adding a color resource. Listing 13-4 shows the OK button with a typical color resource that draws all of the parts that are normally black in blue – at least, it's blue on a 320 mode screen with the standard color palette.

```
resource rControlTemplate (1001) {
   1,                                 /* control ID */
   {100,10,0,0},                      /* control rect */
   SimpleButtonControl {{
      $0000,                          /* flags */
      $300A,                          /* more flags */
      0,                              /* refcon */
      1001,                           /* Title Ref */
      1001,                           /* color table ref */
      {13,13,0,0}                     /* key equivalents */
      }};
   };

resource rPString (1001) {"OK"};

data rCtlColorTbl (1001) {
   $"3000 F000 3000 F300 3F00"
   }
```

Listing 13-4: A Blue OK Button

There are two steps needed to add the color resource to the `rControlTemplate`. The first step is to set bits 3-2 for the `moreFlags` parameter to 10, telling the Control Manager that the color table reference field is a resource ID for a color table. The second step, of course, is to put the resource number for the color table in the color table reference field. Following our long standing convention, we've used the same resource number of 1001 for the color table reference that we've used for all of the other resources associated with this control.

The resource for the color table itself looks a little odd. That's because there isn't a resource defined in Apple's header files for a color table, so we have to put in the fields as raw data. Unfortunately, that wasn't just an oversight on their part. There's a very good reason for not putting in a color table resource type: the color table format varies from control to control.

Other than the bizarre format, though, there's nothing special about using raw data instead of a resource defined via a type. The resource as you see it defines five integers, each entered as a hexadecimal value. The bytes appear in the string in the same order they appear in memory, so the least significant byte comes first. In normal hexadecimal format, the five numbers you see are $0030, $00F0, $0030, $00F3 and $003F.

All five integers are bit maps, divided up into various fields to control the color of the button. It's possible, for example, to create a simple button with a blue outline and red letters. Here's a detailed breakdown of the color table for a simple button, in the same order you see them in Listing 13-4.

Button Outline

The first word contains the button outline color, which controls the color of the oval or box around the outside of the button. This includes the double outline of a default button and the drop shadow that appears under a square button. The color itself is in bits 7-4. They are generally set to $0, which gives a black outline. All other bits are unused, and should be set to 0.

Unhighlighted Button Interior

This word controls the color of the interior of the button when it's just sitting there on the screen. The button looks different when you are pressing it with the mouse.

Bits 7-4 control the background color. They are generally set to $F, which gives a white background. All of the other bits are unused, and should be set to 0.

Highlighted Button Interior

This word controls the color of the interior of the button when you are pressing on the button with the mouse. The toolbox reference manuals call this the highlighted state or the selected state. Once again, only bits 7-4 are used. These bits are normally set to $0 for black; in the example, you see them set to $3, for blue.

Unhighlighted Button Text

The fourth word controls the color of the text when the button is not selected. The most significant 8 bits are unused, and should be set to 0. The next four bits, bits 7-4, control the background color for the text. These bits should almost always match the background color for the button itself, so they are usually set to $F. Bits 3-0 control the foreground color for the text. For the standard black text, the value would be $0. For our blue button, these bits are $3.

Highlighted Button Text

The last word controls the highlighted color for the text. The bits are used the same way as for unhighlighted text, but since the button normally inverts when you highlight it, the "black" part becomes "white", and the "white" part becomes "black."

I've included this section mostly for your information. If I were ever inclined towards bad puns, I might mention that it's here for local color. This section tells you enough about control colors and how they are used for you to be able to find and read the color table definitions for the various controls in the toolbox reference manual, but I won't spend a lot of time going over all of the color tables for each and every control.

There are three reasons for skipping over the color table information. The first I've mentioned: once you know basically how to use them, it's pretty simple to get the information you need to color a particular control from the toolbox reference manuals. The second is that I have a fair amount of respect for the number of trees the paper would consume, not to mention the boredom factor for making you skip over all of those detailed bit maps of color tables you may never use. The most important, though, is that color tables are rarely used in most applications. Black buttons on a white background are the default for a reason: they are easy to see on any screen, color or black and white. They can be seen by people who are color blind. Black and white are rarely changed in the color palette, so they are rarely messed up by changing the color palette. Even on a color screen with a normally sighted person, getting carried away with colors can lead to a very distracting program. In fact, about the only group of programmers who regularly use colored buttons are game programmers, and they have such particular requirements for things like shadowed toggle switches or 3-D buttons that they often create custom controls instead of using simple color tables, anyway.

That's not to say colors are not useful. The odd program here or there that uses blue or green buttons can be enough of a subtle change to make the program interesting without being garish. You might want to use colored buttons in a simple children's game, too. Colors are like the C programming language: they aren't bad in and of themselves, but they are sometimes used too often or for the wrong reason.

Static Text

The next control we'll look at is the static text control. Static text controls just write some text on the screen. It's possible to have the program do something when the user clicks on a static text control, but most of the time, static text controls are just used for labels in a complicated window or dialog. A typical resource definition for a static text control is shown in Listing 13-5.

```
resource rControlTemplate (1001) {
   10,                            /* control ID */
   {10,10,21,200},                /* control rect */
   statTextControl {{
      $0000,                      /* flags */
      $1002,                      /* more flags */
      0,                          /* refcon */
      1001                        /* Title Ref */
      }};
   };

resource rTextForLETextBox2 (1001) {"Control Test Platform"};
```

Listing 13-5: A Typical Static Text Control

Here's how the bits are used in the two flag words:

`flag` bits	use
15-8	Reserved; set to 0.
7	0 for a visible control, 1 for an invisible control.
6-2	Reserved; set to 0.
1	Set this bit for text substitution, or clear it if text substitution is not used.
0	Set this bit if the text is a Pascal string, and clear it for a C string. This bit isn't used if the text is in a resource.

`moreFlags` bits	use
15-13	Reserved; set to 0.
12	This bit tells the Control Manager that you are using a standard control. It must be set to 1.
11-2	Reserved; set to 0.
1-0	Defines the type of reference in the title field. 00 Title reference is a pointer. 01 Title reference is a handle. 10 Title reference is a resource ID for an `rTextForLETextBox2` resource.

In a static text control, the title field is actually the text. The text is in a new type of resource, `rTextForLETextBox2` . This resource works just like an `rPString` resource, but it creates a simple block of text, which can be longer than 255 characters. The Control Manager figures out how many characters to draw by looking at the size of the resource.

Setting bit 1 of the flag word tells the Control Manager to perform text substitution. This lets you substitute text in a static text string, more or less like you did in an alert window. It's a useful capability, but I'm not going to go over it in detail here. I want you to be aware that text substitution is possible, but I'll leave it up to you to dig the details out of the toolbox reference manual if you need to.

The only really tricky part of defining a static text control is picking the size of the control rectangle. Unfortunately, you can't let the Control Manager do it for you, like we did with simple buttons. The only foolproof way to do it is to set up the text in a sample program and make the proper QuickDraw II calls to see how big the text actually is. Even then, you have to make some assumptions. The basic assumption is that the system font is being used. In practice, you can allow 11 pixels for each line, then experiment with the width of the control rectangle to make sure it is wide enough that none of the text is chopped off. For a start, try 10 pixels per character for the width. This is also a great time to put a program like Design Master to work, since you can just draw and let Design Master tell you how big the rectangle should be.

There are also two optional fields for the control record. The first is the length of the text; it's used when you use a pointer to the text. Since we're using resources, it isn't an issue. The other is a text justification flag. Using it, you can have the Control Manager left justify (0, `leftJust`), fill justify (2, `fullJust`), right justify (-1, `rightJust`) or center the text (1, `centerJust`) in the control rectangle. Here's what the static text resource would look like for centered text:

```
resource rControlTemplate (1001) {
   10,                              /* control ID */
   {10,10,21,200},                  /* control rect */
   statTextControl {{
      $0000,                        /* flags */
      $1002,                        /* more flags */
      0,                            /* refcon */
      1001,                         /* Title Ref */
      0,                            /* text length; not used */
      centerJust                    /* center the text */
      }};
   };
```

Problem 13-2: Start with the solution to Problem 13-1, and add a title to the window using a static text control. The title should say "Control Test Platform," just like the sample resource definition.

Make sure you can detect hits on the control by calling `SysBeep` 3 times when the control is selected, but take the calls to `SysBeep` out afterwards. It's generally not a good idea to have the program do something when the user clicks on static text.

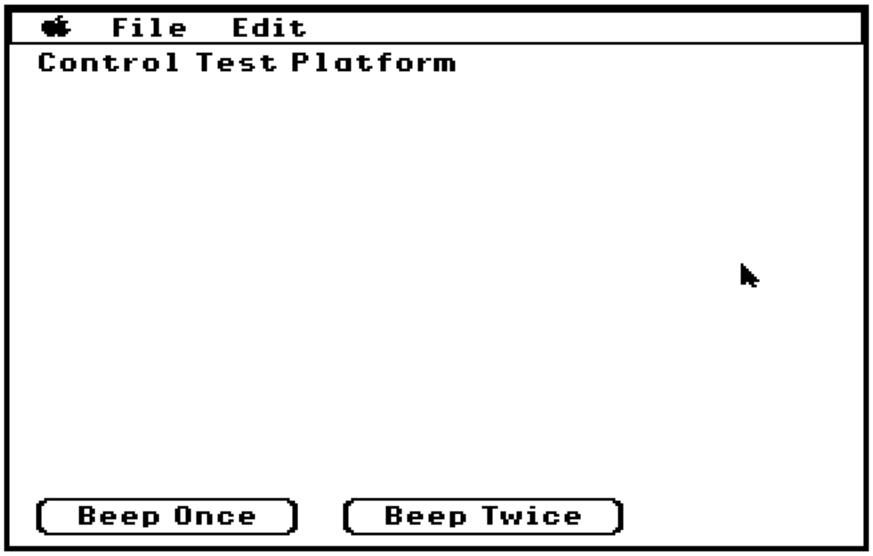
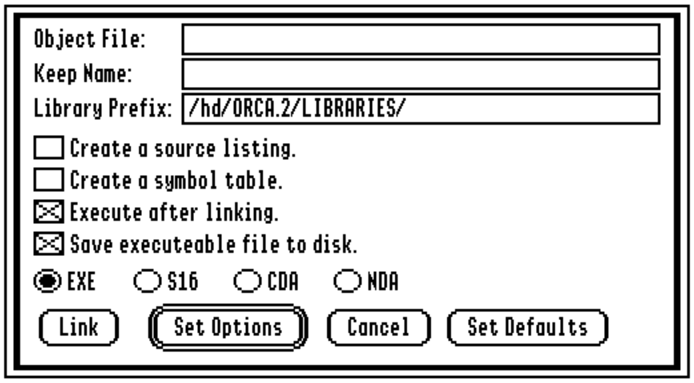


Figure 13-3: Control Sampler with Static Text Title

Radio Buttons

Radio buttons are used when you want to present a choice, and the user is supposed to pick exactly one of the alternatives. For example, PRIZM's link dialog let's you pick one of four executable file types: EXE, S16, NDA or CDA. Figure 13-4 shows the link dialog that contains the radio buttons that let you pick the file format.



Object File:
Keep Name:
Library Prefix: /hd/ORCA.2/LIBRARIES/

☐ Create a source listing.
☐ Create a symbol table.
☒ Execute after linking.
☒ Save executeable file to disk.

● EXE ○ S16 ○ CDA ○ NDA

Figure 13-4: PRIZM's Link Dialog Shows Some Typical Radio Buttons

Radio buttons are grouped in families. Only one of the radio buttons in any particular family can be selected at any one time. You don't really have to do any work to make this happed; the Control Manager automatically deselects one radio button when the user clicks on one of the other radio buttons in the same family, and `TaskMaster` selects a radio button when the user clicks on it. The active radio button is the one with the dot in the middle of the circle. The others have a circle with nothing inside.

```
resource rControlTemplate (1001) {
   10,                                /* control ID */
   {10,20,0,0},                       /* control rect */
   RadioControl {{
      $0001,                          /* flags */
      $3002,                          /* more flags */
      0,                              /* refcon */
      1001,                           /* Title Ref */
      0,                              /* initial value */
      0,                              /* color table */
      {"G","g",0,0}                   /* key equivalents */
      }};
   };

resource rPString (1001) {"Green"};
```

Listing 13-6: A Typical Radio Button Control

Like simple buttons, the Control Manager will be happy to figure out the size for the control, so you can (and usually should) set the last two entries in the control rectangle to 0.

Also like the simple button, you can use key equivalents for radio button controls. The last two fields are optional, though. If you aren't using color tables or key equivalents, you can safely leave the entries off entirely. If you are using key equivalents, though, you do have to put in something for the color table, even if you aren't using one.

The initial value field should be 0 if the control is not supposed to be the one that is selected with the window is created, and 1 if it should be selected. Exactly one of the controls in a particular family should be selected.

The flags for the radio button control are:

`flag` bits	use
15-8	Reserved; set to 0.
7	0 for a visible control, 1 for an invisible control.
6-0	This is the family number for the radio button.

`moreFlags` bits	use
15-14	Reserved; set to 0.
13	Set this bit to 1 if the button has a keystroke equivalent.
12	This bit tells the Control Manager that you are using a standard control. It must be set to 1.
11-4	Reserved; set to 0.
3-2	Defines the type of reference in the color table field. 00 Color table reference is a pointer. 01 Color table reference is a handle. 10 Color table reference is a resource ID.
1-0	Defines the type of reference in the title field. 00 Title reference is a pointer. 01 Title reference is a handle. 10 Title reference is a resource ID.

Problem 13-3: Start with the program from Problem 13-2. Add a family of three radio buttons, labeled Red, Green and Blue. Group these controls to the right of a rectangle that is outlined in black and filled with the color whose radio button is active.

Use key equivalents for the various colors, assigning the first letter of the color as the key equivalent.

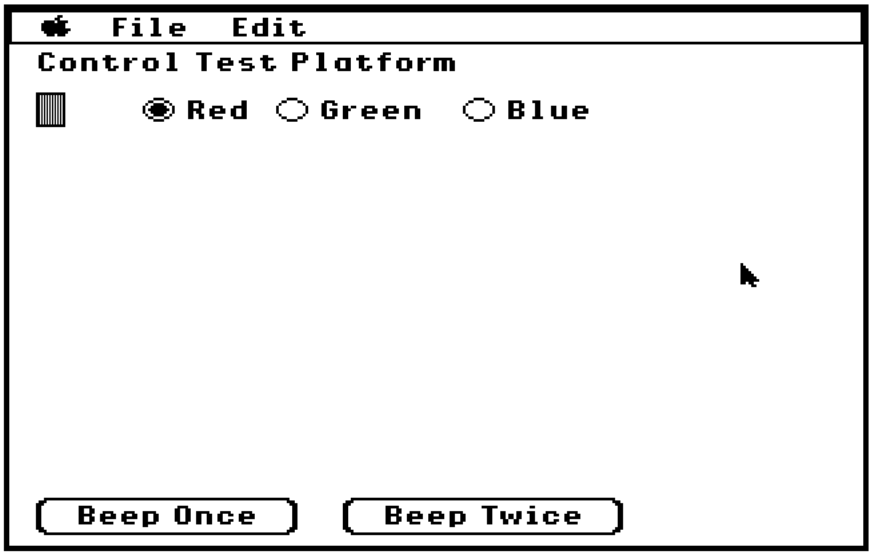


Figure 13-5: Control Sampler with Radio Buttons

Check Boxes

Check boxes are used for yes/no choices. It's pretty common to have a series of related yes/no choices grouped together, but unlike radio buttons, check boxes don't have families. Any grouping you see is there for organization, not because the Control Manager attaches any special significance to more than one check box.

Figure 13-6 shows a typical use of check boxes. It's the Find dialog from PRIZM, which uses check boxes to let you select some options for the string search. If the option is on (selected) the box will have an X. If the option is off (not selected) the box is empty.

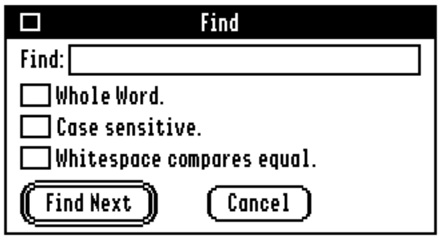


Figure 13-6: A Typical Use of Check Box Controls

Listing 13-7 shows a typical resources for a check box control.

```
resource rControlTemplate (1001) {
   10,                                 /* control ID */
   {10,20,0,0},                        /* control rect */
   CheckControl {{
      $0000,                           /* flags */
      $3002,                          /* more flags */
      0,                              /* refcon */
      1001,                           /* Title Ref */
      0,                              /* initial value */
      0,                              /* color table */
      {'3','#',0,0}                   /* key equivalents */
      }};
   };

resource rPString (1001) {"Check Here"};
```

Listing 13-7: A Typical Check Box Control

This looks a lot like the the resource for a radio button, and it is. The only difference, other than the fact that you specify `CheckControl` instead of `RadioControl`, is that there is no family number. Here's the official list of flags, just for completeness, though:

`flag` bits	use
15-8	Reserved; set to 0.
7	0 for a visible control, 1 for an invisible control.
6-0	Reserved; set to 0.

`moreFlags` bits	use
15-14	Reserved; set to 0.
13	Set this bit to 1 if the button has a keystroke equivalent.
12	This bit tells the Control Manager that you are using a standard control. It must be set to 1.
11-4	Reserved; set to 0.
3-2	Defines the type of reference in the color table field. 00 Color table reference is a pointer. 01 Color table reference is a handle. 10 Color table reference is a resource ID.
1-0	Defines the type of reference in the title field. 00 Title reference is a pointer. 01 Title reference is a handle. 10 Title reference is a resource ID.

Problem 13-4: Start with the solution to Problem 13-3. Add a check box labeled "Sound On." When the program starts, the control should be selected (i.e. there should be an X in the box). When this control is not selected, the simple buttons that call `SysBeep` shouldn't make any sound.

This control should not have a key equivalent, so the last two parameters (the color table and key equivalents parameters) should be left off. Be sure you check the flag words, too – bit 13 of `moreFlags` should be clear!

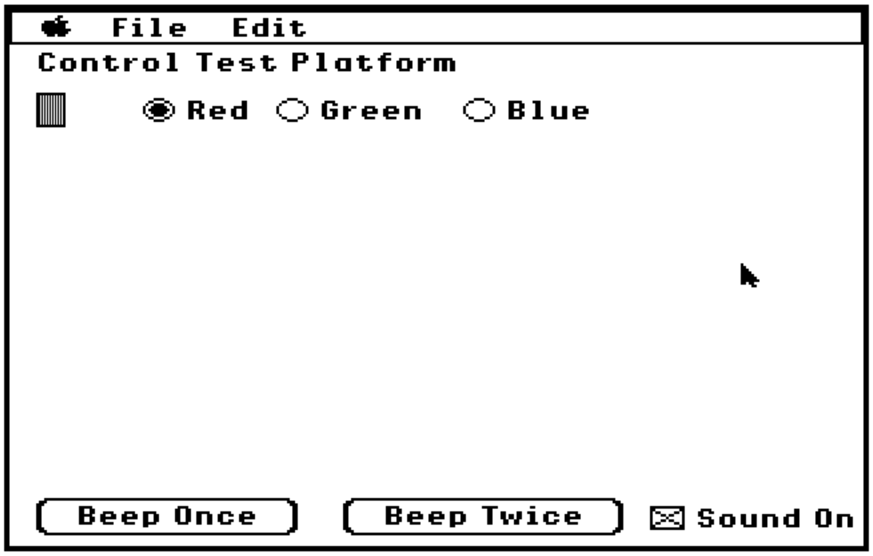


Figure 13-7: Control Sampler with Check Box

Scroll Bars

Scroll bars are most commonly used to scroll through a document, but that's not their only use. You can use a scroll bar any time you want to allow a smooth range of choices. The position of the window in a document is one case where you have a smooth range of choices, but so is selecting the intensity of a color or picking a scrap from a bunch of choices in a scrap book.

Setting up a scroll bar isn't that much different from setting up any other control. Here's the resource and flags for a typical scroll bar control:

```
resource rControlTemplate (1001) {
   10,                               /* control ID */
   {10,10,20,100},                   /* control rect */
   scrollControl {{
      $001C,                         /* flags */
      $1000,                         /* more flags */
      0,                             /* refcon */
      16,                            /* document size */
      1,                             /* view size */
      0,                             /* initial value */
      0                              /* color table */
      }};
   };
```

Listing 13-8: A Typical Scroll Bar Control

`flag` bits	use
15-8	Reserved; set to 0.
7	0 for a visible control, 1 for an invisible control.
6-5	Reserved; set to 0.
4	0 for a vertical scroll bar, 1 for a horizontal scroll bar.
3	Set to 1 if the scroll bar should have a right arrow, or zero if there is no right arrow. The Control Manager ignores this flag for vertical scroll bars.
2	Set to 1 if the scroll bar should have a left arrow, or zero if there is no left arrow. The Control Manager ignores this flag for vertical scroll bars.
1	Set to 1 if the scroll bar should have a down arrow, or zero if there is no down arrow. The Control Manager ignores this flag for horizontal scroll bars.
0	Set to 1 if the scroll bar should have an up arrow, or zero if there is no up arrow. The Control Manager ignores this flag for horizontal scroll bars.

`moreFlags` bits	use
15-13	Reserved; set to 0.
12	This bit tells the Control Manager that you are using a standard control. It must be set to 1.
11-4	Reserved; set to 0.
3-2	Defines the type of reference in the color table field. 00 Color table reference is a pointer. 01 Color table reference is a handle. 10 Color table reference is a resource ID.
1-0	Reserved; set to 0.

The color table is optional, so you can leave that parameter off if you like.

As you can see, you can define a scroll bar with or without arrows, and the size of the scroll bar is completely under your control. It's possible, for example, to create a short, fat vertical scroll bar, rather than the tall thin one you are used to.

There are three new parameters in this control that you haven't seen before; they are used to control the value, or position, of the scroll bar. To understand how the scroll bar handles positions, let's think of it as it's used to scroll through a document. There are three parameters that control what the scroll bar looks like. The first is the size of the document, which is set to 16 in Listing 13-8. That tells us that the scroll bar can represent up to 16 different values. You might use the number of lines in a document, the number of pages, or even some fixed value that's used like a percentage. There are tool calls to read and write this value, so you can change the value as the size of the document changes. We'll look at those a bit later.

The initial value tells the Control Manager where the thumb is at. As the scroll bar is used, or as the program scrolls a document for some other reason, this value is updated. As you'll see when we start looking at the calls that deal with the scroll bar, this value can be changed either by your program or by the Control Manager, so you will need to be able to read and write the value after the control exists.

Finally, scroll bars on the Apple IIGS change size based on the relative size of the document and the display area. If the window can show half of a document, the scroll bar's thumb fills half of the space between the scroll bar arrows. If the window can only display a quarter of the document, the thumb only fills a quarter of the scroll bar. At some point, the thumb reaches it's smallest size and doesn't shrink any more, but for a lot of applications, the variable size thumb is pretty cool. The view size is used to set the thumb size. The ratio of the view size to the document size should be the same as the ratio between the amount of information the window can display and the size of the entire document.

Creating a scroll bar is a lot like creating any other control, but using one is a very different. The reason for the difference is that scroll bars need to do five different things, depending on where the mouse is pressed. There are several ways to handle a scroll bar control, but the easiest is to set up a scroll bar action procedure, then tell the Control Manager to call your scroll bar action procedure whenever the user presses on the scroll bar. The `SetCtlAction` call is used to tell the Control Manager (in conjunction with `TaskMaster`) to call your action procedure; a typical call looks like this:

```
   SetCtlAction(@ScrollAction, GetCtlHandleFromID(wPtr, ctlID));
```

The first parameter is the address of the action procedure; we'll talk more about the action procedure in a moment. The second parameter is the handle for the control. Since you know the control ID, not the control handle, you need to call `GetCtlHandleFromID` to get the handle. You've seen that call once before, but to refresh your memory, you pass a pointer to the window containing the control and the control ID, and `GetCtlHandleFromID` returns the control handle.

Your action procedure looks and works a lot like a window update procedure. It's called from the tools, so you need to use the databank directive to switch to the program's data bank. The scroll bar action procedure has two parameters. The first is an integer part number, and the second is the control handle. You'll need to do different things based on what part of the scroll is being used; here's the part codes that can be returned for a scroll bar control. These definitions are straight out of the Control Manager interface files for ORCA/Pascal, so you can (and should) use the names instead of the numbers.

```
    UpArrow          =   $05;            (* up arrow on scroll bar   *)
    DownArrow        =   $06;            (* down arrow on scroll bar *)
    PageUp           =   $07;            (* page up                  *)
    PageDown         =   $08;            (* page down                *)
    Thumb            =   $81;            (* thumb                    *)
```

If you look at the part codes, you can see that there's a clear definition for up and down arrows. Of course, not all scroll bars are vertical. For horizontal scroll bars, "up" means left, and "down" means right.

Listing 13-9 shows a sample action procedure.

```
  {$databank+}

  procedure ScrollAction (part: integer; ctlHandle: ctlRecHndl);

  { Scroll bar action procedure                                }
  {                                                            }
  { Parameters:                                                }
  {    part - scroll bar part code                             }
  {    ctlHandle - scroll bar handle                           }

  const
     pageSize = 4;                      {size of a page}
     maxPos = 16;                       {max position for the scroll bar}

  var
     value, oldValue: integer;          {control value}

  begin {ScrollAction}
  value := GetCtlValue(ctlHandle);
  oldValue := value;
```

```
case part of
   upArrow:     value := value-1;
   downArrow:   value := value+1;
   pageUp:      value := value-pageSize;
   pageDown:    value := value+pageSize;
   thumb:       begin value := oldValue; oldValue := value-1; end;
   otherwise:   ;
   end; {case}
if value < 0 then
   value := 0
else if value > maxPos then
   value := maxPos;
if value <> oldValue then begin
   SetCtlValue(value, ctlHandle);
   {do any action to update the thing being scrolled here}
   end; {if}
end; {ScrollAction}

{$databank+}
```

Listing 13-9: A Sample Scroll Bar Action Procedure

As you can see, it's up to the action procedure to decide how far the scroll bar should be moved, and then to actually call `SetCtlValue` to move the scroll bar. You can (and should) also do anything you need to to update the document, or whatever it is you are scrolling, from inside of the action procedure. You should not just invalidate the effected rectangle, since the scroll bar can be used to scroll continuously by pressing the mouse button on an arrow or in the page area, and when that happens, your scroll action procedure is called continuously, too, without ever returning to the event loop.

In general, you'll use a single `ScrollAction` procedure to handle several different scroll bars. Since you get the control handle, but no the control ID, you need some way of switching between the two. `GetCtlID` takes the control handle as a parameter and returns the control ID. It's the inverse of a call you've already seen, `GetCtlHandleFromID`, which takes the control ID as a parameter and returns the control handle.

```
id := GetCtlID(ctlHandle);
```

Scroll bar controls have one other problem to deal with on occasion, too: they change size. If you are looking at the middle of a text document, then paste in a chunk of text that is half the size of the whole document, you're not at the middle anymore – you're 1/3 of the way down. The position of the thumb needs to change to reflect that. The size of the thumb may change, too, since the size of the page relative to the whole document has changed, and the thumb's size reflects that proportion until the document is so large that the thumb would have to shrink too much to stay in the correct proportion.

Changing the scroll bar control's value changes the position of the thumb. To change the size of a page or the size of the document, you need to use `SetCtlParams`. This call takes the document size (or data size) as the first parameter and the page size as the second. If you only want to change one of the values, you can pass -1 for the other. Here's a typical call, setting the view size to 10 and the data size to 100:

```
SetCtlParams(100, 10, ctlHandle);
```

If you need to read the value for some reason, use `GetCtlParams`.

```
parms := GetCtlParams(ctlHandle);
```

The two values are returned in a long integer, so you have to pull them apart, but you're probably getting pretty used to that by now. The view size is in the most significant word; the least significant word holds the data size.

Problem 13-5: In this problem, you'll add a color mixer to the control sampler we've been creating in this lesson.

Start with the solution to Problem 13-4. Add three horizontal scroll bars, and use static text controls to label them Red, Green and Blue. Set a document size of 16 for each scroll bar and a view size of 1. The Control Manager allows the scroll bar to scroll from a position of 0 to the data size minus the page size, so this gives positions ranging from 0 to 15. Assume a page has a size of 1, too. That's unusual, but since our "document" is basically the 16 possible values for the red, green, and blue color guns of the CRT, it doesn't make much sense to move by anything larger than 1.

Just to the left of the scroll bars, which should be stacked on top of each other, create a rectangle with a black outline and an interior that is filled with some unused color.

As part of initializing the program, set the color table entry for the color you've used to fill the rectangle to black – that would be $0 for all three colors. Set the position for the scroll bars to 0 using `SetCtlValue`, too.

Whenever the value of one of the scroll bars changes, adjust the color table entry appropriately. The result is a simple but effective color mixer.

Hint: You can use `SetColorTable` and `GetColorTable` to change the color table, but it would be easier to use `SetColorEntry` and `GetColorEntry`. We haven't discuss those calls in the course, and won't – but it would be good practice to find out about the calls by looking in the toolbox reference manuals, figuring out how to use them on your own. They are also documented in Appendix A.

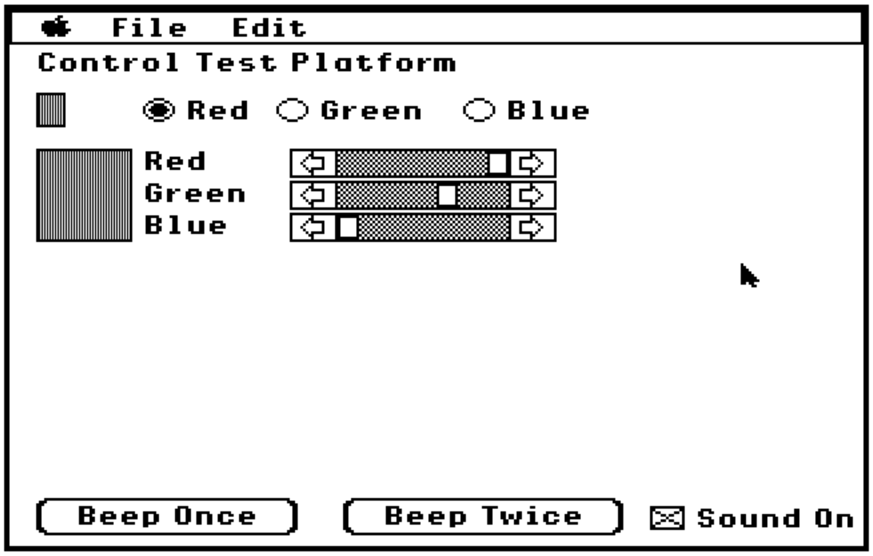


Figure 13-8: Control Sampler with Scroll Bars

Problem 13-6: In Problem 12-2 you created a scrapbook that used two menu items to scroll back and forth through a list of scraps. Get rid of the menu and add a scroll bar near the bottom of the window, using the scroll bar to move among the various scraps. Use `FrameRect` to draw a box in the window, and draw the scrap inside of the box.

Make sure your program keeps track of the data size and view size, setting the scroll bar's parameters accordingly. You should see the thumb size change as you cut or paste scraps, and you should be able to scroll through all of the scraps.

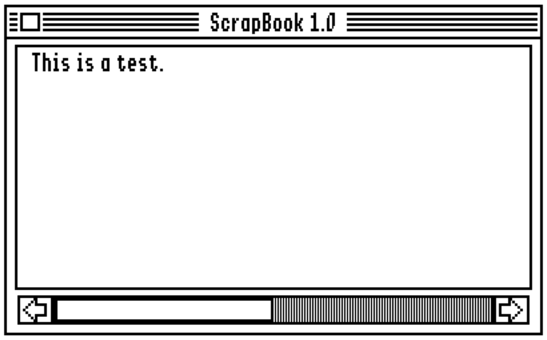


Figure 13-9: Scrapbook With Scroll Bar

The Grow Box

The grow box control is almost always used to change the size of a window, although it is technically possible to use it inside a window, just like any other control. Defining a grow box is very similar to defining the other simple controls. Here's a typical resource, followed by the definitions for the flag words:

```
resource rControlTemplate (1001) {
   10,                              /* control ID */
   {90,90,10,10},                   /* control rect */
   growControl {{
      $0001,                        /* flags */
      $1000,                        /* more flags */
      0,                            /* refcon */
      0                             /* color table */
      }};
   };
```

Listing 13-10: A Typical Scroll Bar Control

flag bits	use
15-8	Reserved; set to 0.
7	0 for a visible control, 1 for an invisible control.
6-1	Reserved; set to 0.
0	Set this bit to 1 if the grow control is being used to resize a window. This tells the Control Manager to draw a window outline as the user drags the control around, and to change the size of the window when the control is released. Set this bit to 0 if you will do all of the tracking manually.

`moreFlags` bits	use
15-13	Reserved; set to 0.
12	This bit tells the Control Manager that you are using a standard control. It must be set to 1.
11-4	Reserved; set to 0.
3-2	Defines the type of reference in the color table field. 00 Color table reference is a pointer. 01 Color table reference is a handle. 10 Color table reference is a resource ID.
1-0	Reserved; set to 0.

Setting bit 0 of the flags word tells the Control Manager to make all of the appropriate calls to handle changing the size of a window. An outline of the window with the new size is shown, and changes as the mouse moves. When the mouse is released, the size of the window is changed.

I'm not going to assign a problem involving the grow box, since it's not that different from other controls, and it's generally created and handled for you as a part of something else – like a set of `TaskMaster` controlled scroll bars and grow box, or as part of a TextEdit control. If you would like to experiment with the control, I'd suggest using a program with a single window, placing the control at the bottom right of the window and using it to change the window's size.

Summary

This lesson introduced controls, showing how to use some of the basic control types. You learned about simple buttons, radio buttons, check boxes, static text, scroll bars, and grow boxes. In addition to using the controls individually, you also learned how to set up and use your own scroll bars and grow box for a document window.

Tool calls used for the first time in this lesson:

```
GetColorEntry     GetCtlID          GetCtlParams      GetCtlValue
SetColorEntry     SetCtlAction      SetCtlParams      SetCtlValue
```

Resource types used for the first time in this lesson:

```
rControlList      rTextForLETextBox2
```

Lesson 14 – Controls, Part 2

Goals for This Lesson

This lesson is a continuation of the last one. In this lesson you will learn how to create and use several kinds of controls, including rectangles, thermometers, icon buttons, pictures, pop-up menus, lists, and edit line controls.

Rectangles

Rectangle controls are usually used as dividers or boxes in a complicated dialog. A thin rectangle can be used as a dividing line. You can also make it look like a control or text item is "poked through" the edge of the rectangle by putting the rectangle after the other controls in the control list. This takes advantage of the fact that controls are drawn in the opposite order of their appearance in the control list to draw some other control over the top of the rectangle control.

Here's a typical resource for a rectangle control, followed by the definitions for the flag words:

```
resource rControlTemplate (1001) {
   10,                            /* control ID */
   {8,8,22,102},                  /* control rect */
   rectangleControl {{
      $0002,                      /* flags */
      $1000,                      /* more flags */
      0,                          /* refcon */
      1;                          /* pen height */
      2;                          /* pen width */
      "FFFFFFFFFFFFFFFF",         /* penmask */
      "FFFFFFFFFFFFFFFF",         /* penpattern */
      "FFFFFFFFFFFFFFFF",
      "FFFFFFFFFFFFFFFF",
      "FFFFFFFFFFFFFFFF",
      }};
   };
```

Listing 14-1: A Typical Rectangle Control

flag bits	use
15-8	Reserved; set to 0.
7	0 for a visible control, 1 for an invisible control.
6-2	Reserved; set to 0.
1-0	Control pattern. 00 Transparent control. Nothing is drawn, but you can still test for hits in the rectangle. 01 Gray pattern. 10 Black pattern

moreFlags bits	use
15-13	Reserved; set to 0.
12	This bit tells the Control Manager that you are using a standard control. It must be set to 1.
11-0	Reserved; set to 0.

The last four parameters in the resource are optional. If you leave them out, you will get a pen that is one pixel high and two pixels wide. The pen width you specify is always divided by two if you are using 320 mode. While the pen width and pen height are both optional, you have to put both in or leave both out – you can't code the pen height and leave off the pen width.

The last two bits of the first flag word control the pen pattern. You can get a transparent control, which draws nothing but can be used to test for a mouse hit; a solid black outline, or a gray outline. You can also specify the pen pattern and pen mask, which gives you complete control over picking colored edges or complicated patterns. If you do include a pen pattern and pen mask, you should set the last two bits of the flag word to 10.

Problem 14-1: Start with your solution to Problem 13-5. Add a rectangle control around the three scroll bars and colored rectangle to group the color picker controls together, separating them from the other controls.

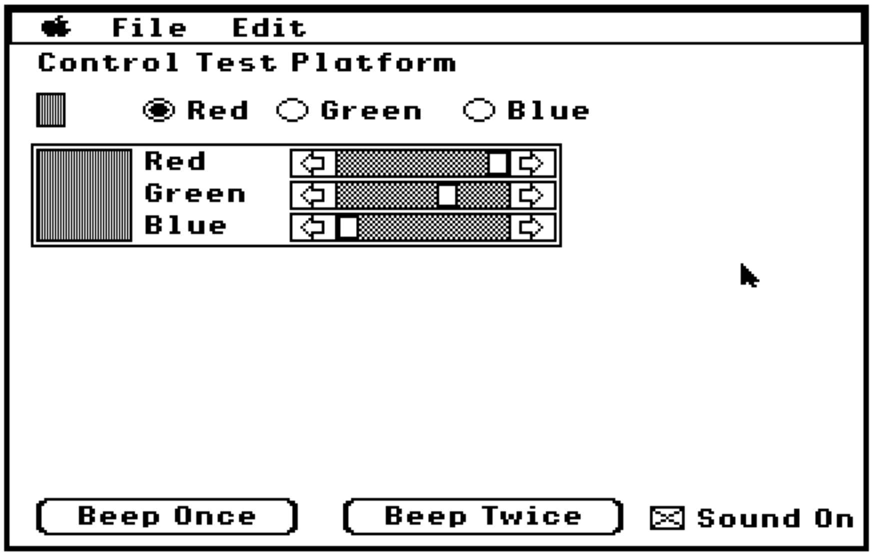


Figure 14-1: Rectangle Control

Thermometers

Thermometers are used to show how far something has progressed. The most familiar example of a thermometer is the one you see each time you boot your computer. It's on the boot screen, and fills up gradually as the computer boots. Here's a preview of the thermometer you'll create in Problem 14-2.

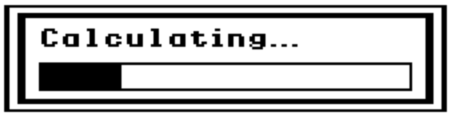


Figure 14-2: A Thermometer Control

The resource definition and flags words look like this:

```
resource rControlTemplate (1001) {
   10,                              /* control ID */
   {8,8,200,16},                    /* control rect */
   thermometerControl {{
      $0001,                        /* flags */
      $1000,                        /* more flags */
      0,                            /* refcon */
      1,                            /* value for pos of mercury */
      2,                            /* scale for mercury */
      $00000000,                    /* color table reference */
      }};
   };
```

Listing 14-2: A Typical Thermometer Control

`flag` bits	use
15-1	Reserved; set to 0.
0	0 for a vertical control, 1 for a horizontal control.

`moreFlags` bits	use
15-13	Reserved; set to 0.
12	This bit tells the Control Manager that you are using a standard control. It must be set to 1.
11-2	Reserved; set to 0.
1-0	Defines the type of reference in the color table field. 00 Color table reference is a pointer. 01 Color table reference is a handle. 10 Color table reference is a resource ID.

As usual, the color table is optional. The thermometer control is the only one that uses color by default, though. By default, the control is outlined with a black rectangle, and the "mercury" fills in with red in 640 mode, and blue in 320 mode.

The thermometer itself is controlled by two fields, labeled `/* value for pos of mercury */` and `/* scale for mercury */`. The last of these tells the Control Manager what scale to use, starting at 0. For example, to show a thermometer based on percent completion, you could use a scale value of 100. The first value sets the original position of the thermometer, so it is almost always set to 0. As you complete a task, your program should make calls to `SetCtlValue` to change the thermometer setting.

Let's look at how this would work in a simple example. We'll run the Savage floating-point benchmark, using a thermometer to show how far we've progressed. The Savage benchmark repeatedly adds one to a value, using one of the most complicated ways to increment a number around – but then, the purpose is to do a quick check of how fast floating-point calculations are performed. The main loop looks like this:

```
sum := 1.0;
for i := 1 to 250 do
   sum := tan(arctan(exp(ln(sqrt(sqr(sum))))))+1.0;
```

To use a thermometer to show the progress, we'd start by drawing a window with a static text item that said something like "Calculating..." and a thermometer control. Since the loop will go from 0 to 250, it makes sense to set the scale to 250, and just use the loop variable `i` to set the control value. Assuming the thermometer control has a control ID of 2, here's how to modify the loop to show the progress. (Setting up and removing the window is left as an exercise for the reader – literally. See Problem 14-2.)

```
ctlHandle := GetCtlHandleFromID(wPtr, 2);
sum := 1.0;
for i := 1 to 250 do begin
   SetCtlValue(i, ctlHandle);
   sum := tan(arctan(exp(ln(sqrt(sqr(sum))))))+1.0;
   end; {for}
```

There are some things to keep in mind when you are using a thermometer. The biggest is that updating the control takes time, too, and you don't want to add significantly to the time it takes to complete an operation by updating the thermometer too often. The other is that some tasks don't like to be interrupted. You may have to be a little creative in some circumstances. For example, there is no convenient way for the boot code to know how long it will take you to boot your computer, since you can customize the operating system, use a variety of disk drives, and even add accelerator cards, and all of these effect the boot time. Apple's programmer's took a rather pragmatic approach: the boot code times itself, and writes the time to the disk after you boot. Since you don't change the operating system very often, the boot time doesn't change often, either. The thermometer itself is updated with an interrupt handler that simply watches the system clock to see how much time has elapsed.

Problem 14-2: Start with the solution to Problem 14-1. Add a simple button called Savage. When this button is pressed, bring up a new window with two controls, a static text control with a title of "Calculating..." and a thermometer control, as described in the text. Update the thermometer as the benchmark runs.

Try running the benchmark with and without updating the thermometer. How much of the time is spent calculating, and how much time is spent showing the user how far things have progressed? Is it worth it?

You might immediately say, "No!" Keep in mind, though, that in a real program the actual wait isn't nearly as important as the psychological wait. An interesting example is the amount of time it takes to use the mouse as opposed to the keyboard. Apple did a study which compared the two, but also asked the people who were being timed which took longer. The result was interesting: the people being tested said the keyboard was faster, but the stopwatch disagreed.

Icon Buttons

An icon button is basically a simple button control with a picture inside the button instead of the text you are used to for a simple button. You can also put text under the icon, more or less like a title.

Icon controls really aren't very hard to create or use, but there are lots of options. You can change the type of border, or eliminate it altogether; use text or leave it out; and you have a lot of control over how the icon is drawn. Icons themselves are not just a picture, and there are a lot of little things you need to know about icons to use them effectively. We'll cover all of these options and concepts in this section, but I want to start with a typical resource definition for an icon button, and the bit-by-bit breakdown of the flag words in an icon control. It's a little complicated, but don't get bogged down in the details, yet. Just skim through the definitions to see where things are, and refer back to this example and the flag words as you read the rest of the section.

```
resource rControlTemplate (1001) {
   10,                                /* control ID */
   {8,8,42,64},                       /* control rect */
   IconButtonControl {{
      $0000,                          /* flags */
      $1022,                          /* more flags */
      0,                              /* refcon */
      1001,                           /* Icon Ref */
      1001,                           /* Title Ref */
      0,                              /* color table ref */
      $0F00,                          /* Display mode */
      {"P","p",0,0}                   /* key equivalents */
      }};
   };

resource rIcon (1001) {
   $8000,                        /* Icon Type bit 15  1 = color, 0 = mono */
   10,                                /* height of icon in pixels */
   16,                                /* width of icon in pixels */
   $"AAAAAAAAAAAAAAAA"                /* icon image */
   $"AAAAAAAAAAAAAAAA"
   $"AAAAAAAAAAAAAAAA"
   $"AAAAAAAAAAAAAAAA"
   $"AAAAAAAAAAAAAAAA"
   $"AAAAAAAAAAAAAAAA"
   $"AAAAAAAAAAAAAAAA"
   $"AAAAAAAAAAAAAAAA"
   $"AAAAAAAAAAAAAAAA"
   $"AAAAAAAAAAAAAAAA",
   $"FFFFFFFFFFFFFFFF"                /* icon mask */
   $"FFFFFFFFFFFFFFFF"
   $"FFFFFFFFFFFFFFFF"
   $"FFFFFFFFFFFFFFFF"
   $"FFFFFFFFFFFFFFFF"
   $"FFFFFFFFFFFFFFFF"
   $"FFFFFFFFFFFFFFFF"
   $"FFFFFFFFFFFFFFFF"
   $"FFFFFFFFFFFFFFFF"
   $"FFFFFFFFFFFFFFFF";
   };

resource rPString (1001) {"Icon"};
```

Listing 14-3: A Typical Icon Button Control

`flag` bits	use
15-8	These bits set the `ctlHilite` field for the control.
7	0 for a normal, visible control, 1 for an invisible control.
6-3	Reserved; set to 0.
2	0 for a border, 1 for no border. The border is an outline around the edge of the button, like the rounded rectangle used on standard simple button controls.
1-0	Assuming bit 2 is set, these bits determine the type of border that will be drawn. 00 Normal rounded rectangle. 01 A bold rounded rectangle. As with simple buttons, this is a double outline with the outer line a little thicker than the inner line. 10 Normal square outline. 11 Square outline with a drop shadow.

`moreFlags` bits	use
15-13	Reserved; set to 0.
12	This bit tells the Control Manager that you are using a standard control. It must be set to 1.
11-6	Reserved; set to 0.
5-4	Defines the type of reference for the field labeled `/* Icon Ref */`. 00 Icon reference is a pointer. 01 Icon reference is a handle. 10 Icon reference is a resource ID.
3-2	Defines the type of reference in the color table field. 00 Color table reference is a pointer. 01 Color table reference is a handle. 10 Color table reference is a resource ID.
1-0	Defines the type of reference for the field labeled `/* Title Ref */`. 00 Title reference is a pointer. 01 Title reference is a handle. 10 Title reference is a resource ID.

`displayMode` bits	use
15-12	This is the background color; it is used instead of black for black and white icons.
11-8	This is the foreground color; it replaces white in black and white icons.
7-3	Reserved; set to 0.
2	If this bit is set, the icon image is anded with a gray pattern as it is drawn, giving an "unselected" look. If the bit is clear, the image is not changed.
1	If this bit is set, a light gray pattern is used instead of the icon image.
0	If this bit is set, the icon image is inverted.

A Typical Icon Button

We'll start with a fairly typical, general case of the icon button. Our starting case is a picture with text underneath, surrounded by a rounded rectangle border. This is actually the same icon button that is defined in Listing 14-3; it's shown below in Figure 14-3.

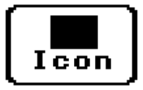


Figure 14-3: A Typical Icon Button

This icon button shows the three parts of an icon button: the border, the icon, and the text. Two of these parts are optional; you can leave out the border or the text. When the user clicks inside the icon rectangle, the icon button switches to a highlighted state, then switches back when the mouse is released, just like a simple button control. The hit is reported the same way as a simple button control, too.

Besides the three visible parts of the icon, you also need to specify the icon rectangle. If you are using a border, the border will be drawn right inside of the rectangle. The rectangle is important even if you aren't using a border, though, since it is used to position the icon and text. To do that, the Control Manager starts by positioning the text right below the icon, forming a "super icon." The resulting picture is centered in the rectangle. Of course, the icon and text can be larger than the rectangle, and in that case they will overflow. That causes all sorts of unsightly things to happen, though. The three most severe are that the entire control isn't erased if you hide the control, only the part inside the rectangle; if the user clicks on a part of the icon that is outside of the rectangle, the button doesn't work; and when the button is inverted, only the part inside the rectangle is changed. In short, it pays to make sure the control rectangle is large enough to hold the icon and text, even if you aren't using a border.

Controlling the Outline

The least significant three bits of the flags word controls the border. If bit 2 is clear (i.e. set to 0) there won't be a border at all, and bits 0 and 1 are ignored. If bit 2 is set, bits 0 and 1 control the style of the border. You get the same four styles that were available for a simple button, namely a rounded rectangle (the bits would be 100), a bold rounded rectangle (bits 101), a square button (bits 110) or a square button with a drop shadow (bits 111).

The Title

The title for the icon button is specified as a p-string. Using resources, the easiest way to create a title is with an `rPString` resource, like the one you saw in the icon button example at the start of this section.

The title becomes a part of the icon button, centered just below the icon itself. That's not always what you want, since text inside of the icon button can ruin the effect of a picture used as a button. To get rid of the title entirely, use 00 for bits 4 and 5 of the `moreFlags` field, then use 0 for the `/* Title Ref */` field. This tells the Control Manager that the title reference is a pointer, then passes nil as the pointer. The title itself will vanish, and the icon will be centered in the control rectangle. You can use a static text control right below the icon for a title-like effect where the text doesn't become a part of the button itself.

Icons and Icon Masks

If you look closely at Listing 14-3, you will see that there are actually two versions of the little picture that is drawn as the icon. The first is the actual picture that will be drawn, and that's the part that is called the icon. The second version of the picture is called the icon mask. The icon mask tells QuickDraw II Auxiliary (which is the tool that actually ends up drawing the icons) what bits actually make up an icon. The mask lets you create irregularly shaped pictures and drop them onto a background without wiping out the background itself. That's why icons in the Finder don't have to be square, and why the blue desktop shows around the edges of even the most complicated icon.

You can also create hollow icons using a hollow mask. For example, here's a doughnut shaped icon. The center is literally hollow, since the mask for the center is set to 0. Whatever was on the screen before this icon is drawn will show up in the middle of the doughnut.

```
resource rIcon (1001) {
   $8000,                       /* Icon Type bit 15  1 = color, 0 = mono */
   10,                              /* height of icon in pixels */
   16,                              /* width of icon in pixels */
   $"0000000AA0000000"              /* icon image */
   $"000AAAAAAAAAA000"
   $"00AAAAAAAAAAAA00"
   $"0AAAAA0000AAAAA0"
   $"AAAAA000000AAAAA"
   $"AAAAA000000AAAAA"
   $"0AAAAA0000AAAAA0"
   $"00AAAAAAAAAAAA00"
   $"000AAAAAAAAAA000"
   $"0000000AA0000000",
   $"0000000FF0000000"              /* icon mask */
   $"000FFFFFFFFFF000"
   $"00FFFFFFFFFFFF00"
   $"0FFFFF0000FFFFF0"
   $"FFFFF000000FFFFF"
   $"FFFFF000000FFFFF"
   $"0FFFFF0000FFFFF0"
   $"00FFFFFFFFFFFF00"
   $"000FFFFFFFFFF000"
   $"0000000FF0000000";
   };
```

Listing 14-4: A Doughnut Shaped Icon

There is also one restrictions on the size of the icon: all icons must be a multiple of eight pixels wide. The icon you see here, for example, is 16 pixels wide. Since icons can be irregularly shaped, though, this isn't really a severe limitation. If you need an icon that is 14 pixels wide, create it with 16 pixels in the definition and use the icon mask to mask out the left and right pixel on each side. The icon will end up being 16 pixels wide, but the figure drawn on the screen will be 14 pixels. Since the Control Manager centers the icon in the icon button control, you won't even be able to tell that the icon is a little wider than the button.

The mechanics of typing the icon definition are also sort of interesting. The resource compiler has a mechanism for creating long hexadecimal fields using strings, which is what we're doing with this icon definition. The icon itself is variable length, so the resource compiler needs some way of knowing where the end of the icon and the start of the icon mask is. That's what the lone comma you see on the last line of the icon is for; it marks the end of one hexadecimal field. The other lines must not have any punctuation after them at all – no comma and no semicolon. The resource compiler combines all of these lines into a single, long chunk, and fills in the values in the resource file with the appropriate hexadecimal values.

Display Mode Bits

The `displayMode` field is a flags word that the Control Manager passes along to QuickDraw II Auxiliary when the icon is drawn. It controls several obscure icon drawing options. The first is that icons can be either colored or black and white. The choice is actually made in the `rIcon` resource by setting or clearing the most significant bit of the first entry; setting the bit gives a colored icon, while clearing it gives a black and white icon. While you'll probably use colored icons more often than black and white icons, there are some interesting things you can do with a black and white icon. The most significant four bits of the `displayMode` field is a color; it replaces any white pixels in the icon. The next four bits are also a color, controlling the black bits. By setting these bits, you can quickly change the color of an icon without changing the icon image itself. For a good example of how this capability can be used, try setting the color of a folder in the Finder.

The last three bits if the displayMode field control some masking options. You can use these three bits in combination to get a variety of masks, or you can set just one of the bits. Most of the time, you'll leave all three bits clear, though.

Bit 2 ands a light gray pattern with the image, giving a deselected look. The Finder uses this mask when it draws the icon for a folder that is open. You can use it for an icon button that is inactive. Bit 1 copies a light gray pattern instead of drawing the picture; the Finder uses this for devices that are offline. Finally, bit 0 inverts the image before drawing it. The Finder uses this bit for icons that are selected.

Inactive Icon Buttons

Bits 15-8 of the flags word are passed on to the Control Manager as the ctlHilite field. ctlHilite is used to make a control inactive. For example, there may be situations when a control can't be used until something is selected in the dialog. In that case, you would draw the icon in it's inactive state, and the Control Manager would not report hits on the control.

I really have no idea why this field is available for an icon button but not for any of the other controls we've seen so far. There is another way to set the active state for a control if you need to, using Control Manager calls. I'd recommend setting these bits to 0 all of the time, creating active controls, then using Control Manager calls to activate and deactivate controls if you need to. That way, all of the Controls are handled the same way, and there is less chance that you will make a mistake writing a program.

Creating Icons in the Real World

It's certainly possible to create an icon by hand coding bit maps in a resource compiler source file, but it's very time consuming and pretty close to extremely boring. In real life, if you want to use icon buttons, you really need some sort of a tool to draw the pictures. A program like Design Master, which is very handy for laying out dialogs, is a good choice, since it can write the finished icon as an rIcon resource, and you can paste the result into your resource file for the program. In a pinch, you can also use an icon editor or paint program, then write a quick program to dump the result as hexadecimal strings. It's not hard to do, but of course a real programmer's CAD tool like Design Master saves you the trouble.

Problem 14-3: Start with the solution to Problem 14-2. Add two arrow-shaped icons, one to the left of the three color radio buttons, and one to the right, as shown below. Hook up the left arrow so clicking on it finds the current color radio button and sets the one to the left. The right arrow button should select the radio button to the right on the one that is selected when the button is pressed.

```
resource rIcon (1001) {
   $8000,                       /* Icon Type bit 15  1 = color, 0 = mono */
   9,                                /* height of icon in pixels */
   8,                                /* width of icon in pixels */
   $"FFFF0FFF"                       /* icon image */
   $"FFF00FFF"
   $"FF0F0000"
   $"F0FFFFF0"
   $"0FFFFFF0"
   $"F0FFFFF0"
   $"FF0F0000"
   $"FFF00FFF"
   $"FFFF0FFF",
   $"0000F000"                       /* icon mask */
```

```
     $"000FF000"
     $"00FFFFFF"
     $"0FFFFFFF"
     $"FFFFFFFF"
     $"0FFFFFFF"
     $"00FFFFFF"
     $"000FF000"
     $"0000F000";
     };
```

Listing 14-5: Left Arrow Icon

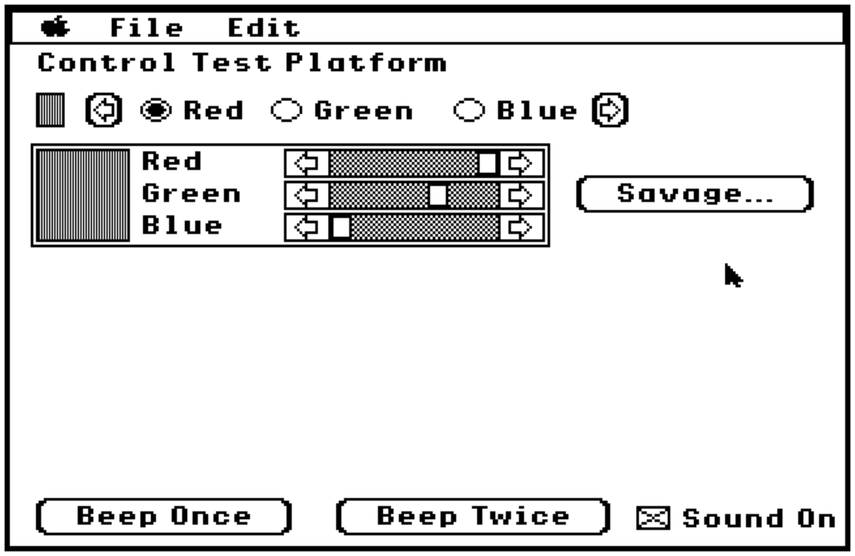


Figure 14-4: Control Sampler Window

Pictures

Picture controls are used to put large, rectangular pictures into a window. They are usually disabled, since they really aren't much good as buttons, and icon buttons do the job of creating a pictorial button better, anyway. The reason picture controls don't make good buttons is that they just sit there. Nothing visibly changes when you click on the picture. In addition, you can't have a border and you can't create text titles.

It's fair to ask why there is even such a thing as a picture control, since you could just draw whatever you wanted into the window with QuickDraw II commands. Using a picture, though, you can create a background for the controls, drawing the picture before the controls are drawn. (Controls are drawn in the reverse of the order in which they appear in the list of controls.) You can also use the picture control to draw pictures automatically, at the same time the controls are drawn, avoiding any chance of flicker if there happens to be a lot of time between when the controls are drawn and when the picture is drawn.

Here's a very simple picture control. It creates a small gray square on the screen.

```
resource rControlTemplate (1001) {
   10,                                /* control ID */
   {23,43,40,64},                     /* control rect */
   PictureControl {{
      $FF00,                          /* flags */
      $1002,                          /* more flags */
      0,                              /* refcon */
      1001                            /* Picture Ref */
      }};
   };

data rPicture (1001) {
   $"0000020002001300170011820100 0A0001C001C0FF3FFF3F9000000000000C00"
   $"2A003C003B0054002A003D003B00540002000200130019000000EEEEEEEEEEEE"
   $"EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE"
   $"EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE"
   $"EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE"
   $"EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE"
   $"EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE"
   $"EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE"
   $"EEEEEEEEEEEE"
   };
```

Listing 14-6: Typical Picture Control

`flag` bits	use
15-8	These bits set the `ctlHilite` field for the control. For a picture control, this will normally be $FFxx, giving an inactive control.
7	0 for a normal, visible control, 1 for an invisible control.
6-0	Reserved; set to 0.

`moreFlags` bits	use
15-13	Reserved; set to 0.
12	This bit tells the Control Manager that you are using a standard control. It must be set to 1.
11-2	Reserved; set to 0.
1-0	Defines the type of reference for the field labeled `/* Picture Ref */`. 00 Picture reference is a pointer. 01 Picture reference is a handle. 10 Picture reference is a resource ID.

The various fields in the resource for the control and the various flag bits are all things you've seen several times by now, so I won't go over them point by point. The only unusual feature of a picture control is the control rectangle. Pictures can start out as any size, although they do fit into a rectangle. If the control rectangle is not the same size as the picture, though, the picture is automatically scaled to fit into the rectangle, shrinking or enlarging the picture as necessary. If you want the tools to figure out how big the picture is on the fly, just leave the bottom right coordinates for the rectangle at 0. The Control Manager will fill in the proper values for a "normal" size picture before drawing the picture.

Creating Pictures

You might think that a picture would be a bit map, but that isn't actually true. Pictures are a recording of various QuickDraw drawing commands. In fact, one way to create the original data for a picture is to ask QuickDraw to start recording a picture, then draw something, and finally ask QuickDraw for the binary version of the picture.

It's probably obvious, but this isn't something you're likely to do by hand. By far the easiest and best way to create an `rPicture` resource definition is to use a program like Design Master, which can, among other things, read in a bit mapped drawing and convert it to a picture resource.

If you would like to learn more about pictures, or even take a crack at creating some by hand, you'll need to do some reading outside of this course. QuickDraw II Auxiliary is used to create and manipulate pictures; you can find a description of this tool in the *Apple IIGS Toolbox Reference: Volume 2*. The format for a picture is outlined in Apple II Technical Notes, Apple IIGS #46: DrawPicture Data Format. Appendix C tells you more about the technical notes and where to get them.

Since creating pictures by hand isn't really a viable option I won't be assigning a problem that deals with pictures. The picture control shown in Figure 14-6 will work, so if you want to try creating a picture control and don't have a program that will do it for you, you can give the control in Listing 14-6 a try.

EditLine Items

Edit line controls are the ones that let you enter a single line of text. A common example is a string search dialog, like the one shown here from PRIZM.

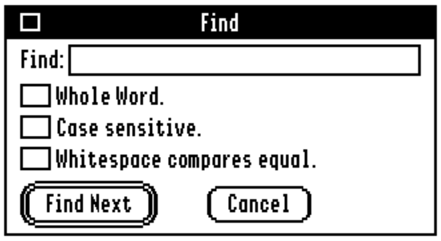


Figure 14-5: PRIZM's Find Dialog

Edit line controls are fairly straight-forward to define and use. In fact, probably the most complicated thing about this type of control is that it has two names. There seems to have been a subtle argument over whether this is an edit line control or a line edit control. Being strictly neutral, I use both terms.

Here's a typical edit line control, followed by the definition for the flags.

```
resource rControlTemplate (1001) {
   10,                                /* control ID */
   {8,8,19,208},                      /* control rect */
   editLineControl {{
      $0000,                          /* flags */
      $7000,                          /* more flags */
      0,                              /* refcon */
      20,                             /* Max Size */
      0,                              /* text Ref */
      $D7,                            /* password character - 6.0 */
      }};
   };
```

Listing 14-7: A Typical EditLine Control

`flag` bits	use
15-8	Reserved; set to 0.
7	0 for a normal, visible control, 1 for an invisible control.
6-0	Reserved; set to 0.

`moreFlags` bits	use
15	Reserved; set to 0.
14-12	Must be set to 1.
11-2	Reserved; set to 0.
1-0	Defines the type of reference for the text field. 00 Text reference is a pointer. 01 Text reference is a handle. 10 Text reference is a resource ID.

When the control is drawn, the control rectangle will be surrounded by a box, and the editable text appears inside of this box. Assuming you aren't doing anything really tricky, you'll be using the system font. In that case, make sure the rectangle is at least 11 pixels high – if it is any shorter, the Control Manager won't draw the text. The width of the rectangle doesn't matter a whole lot, since the text will scroll sideways to accommodate text that is too long for the rectangle.

You have the option of showing some default text in an edit line item when it is first drawn. For example, SFO shows the default name for a file in an edit line item that let's the user enter a file name. If you want to create some default text, set bits 0 and 1 of the `moreFlags` field to 10, then add an `rPString` resource with the default text, putting the resource ID for the text in the text reference field. If you don't want default text, set bits 0 and 1 to 00 and use 0 for the text reference field. Most of the time, any default text will change as the program runs (the search string will be the last one used, the file name will be the last one saved, and so on) so you'll set the default string in the resource to nil and set up any default string in from the program.

Accessing Text

Creating a line edit control is no harder than creating any other control. Using one is pretty easy, too, since the Control Manager and `TaskMaster` handle all of the details of flashing the cursor and selecting text. The major thing you have to do manually is read the text to see what the user typed. That's done with `GetLETextByID`:

```
GetLETextByID(wPtr, editLineID, textBuffer);
```

The three parameters are a pointer to the window containing the control, the control ID, and a buffer to put the text in. The buffer itself has to be two characters longer than the longest possible string the user can type. That maximum length is a value you set when the control is defined; it's the value in the field labeled `/* Max Size */`. If the maximum size is 255 characters or less, you can use the string as a Pascal string. That second extra character you had to allow for in the buffer, though, is a null character that will be stuffed at the end of the string. If you create a pointer to the byte right past the length byte at the start of the string, you have a pointer to a valid null-terminated string. ORCA/Pascal's string handling facilities will work just fine with a null terminated string, allowing edit line strings up to 32K characters in length. Of course, anything even approaching that length should be handled with a text edit control, like the one you used for your editor in Lesson 11.

You can also put text into a text edit control, generally to create a default. That's done with `SetLETextByID`:

```
SetLETextByID(wPtr, editLineID, str);
```

The first two parameters are the same as for `GetLETextByID`; the third is a p-string containing the default text.

Cut, Copy, Paste and Clear

Back when we created a TextEdit control, you saw that the Control Manager, `TaskMaster` and the TextEdit Tool Set worked together to handle the editing commands (cut, copy, paste and clear) for you. `TaskMaster` does so much, in fact, that you may be starting to expect it to do everything. Well, almost. As it turns out, the editing commands don't get handled automatically for edit line controls. If you want to support the editing commands in a window (and you should) you need to do a little extra work.

In some ways, handling the editing commands is pretty easy, and in some ways it's ridiculously hard. Actually handling the editing commands is pretty straight forward, at least on the surface. When the user uses one of the menu editing commands, and your window has an edit line control, you just make a single call, passing the handle for the edit line item. The LineEdit Tool Set takes things from there, deleting or adding text and redrawing the edit line control.

Menu Command	LineEdit Call
Copy	`LECopy(leHandle);`
Clear	`LEDelete(leHandle);`
Cut	`LECut(leHandle);`
Paste	`LEPaste(leHandle);`

So far, the hardest part is remembering that the LineEdit Tool Set calls Clear Delete.

The LineEdit Tool Set uses a private scrap to store the stuff it copies, and pastes from the private scrap, too. Sometimes that's what you want. For example, you might want to allow copying and pasting between two edit line items in a search and replace dialog without effecting the scrap being used by the text editor. In other cases, you do want to use the Scrap Manager. In general, unless there is a good reason for not using the Scrap Manager, you should tell the LineEdit Tool Set to use the Scrap Manager. To to that, call `LEToScrap` right after any Copy or Cut operation to move the LineEdit Tool Set scrap to the Scrap Manager, and call `LEFromScrap` just before calling `LEPaste` to copy the Scrap Manager's scrap into the LineEdit Tool Set's scrap. Neither call has any parameters. The only issue, and it's a minor one, is `LEFromScrap` will ignore the Scrap Manager's scrap if it has more than 255 characters. No harm is done, it just won't use a scrap that is longer than 255 characters.

So far, this looks pretty easy – and it is. The problem is figuring out just what the handle for the line edit item actually is, so you can pass it to the appropriate call. There are two parts to this problem. The first is figuring out which LineEdit control an editing command applies to (if any) and the second is figuring out what the line edit handle is – unfortunately, it's not the same as the control handle for the line edit control!

Multiple controls are handled using the idea of the target control. In loose terms, the target control is the one the user is typing in. Unless you prevent it, the user can use the tab key to move from one target control to another. (You would block the use of the tab key by, for example, allowing the user to use the tab key in a TextEdit control.) The user can also pick the target control by clicking on the control, something they might do naturally to place the insertion point at a specific spot or select text. You should only call the LineEdit Tool Set for an editing command if a line edit control is the target control. To find out which control is the target control, call `FindTargetCtl`. It works on the active window, returning a control handle.

```
ctl := FindTargetCtl;
```

You can pass the control handle to `GetCtlID` to find the control ID, then check to see if it matches the control IDs for any of your line edit controls.

If you dig through the toolbox reference manuals, you will find out that the LineEdit Tool Set is completely independent of the Control Manager. The Control Manager knows about the LineEdit Tool Set, and can set up a control that uses LineEdit Tool Set calls to function, but the LineEdit Tool Set has no idea what a control is. The reason this is important is that the calls you just learned about (`LECut` and it's palls) are LineEdit Tool Set calls, so you can't just pass the control handle to them. The line edit handle the LineEdit Tool Set calls expect is buried inside of the control record itself. To make matters worse, there is no call to fetch the line edit handle directly; you have to fish out the values yourself. Rather that make you suffer through a long discussion of how that's done, though, I'm just going to give you a subroutine that returns the correct field from the control record. The subroutine is called `GetCtlData`, since the field it retrieves is actually called the `ctlData` field. The subroutine is shown in Listing 14-8, along with a sample `DoCut` subroutine that implements all of the ideas we've talked about in this section.

`DoCut` should be called when the user picks Cut from the Edit menu. You can use this subroutine as a model to build subroutines to handle the other three editing commands.

```
function GetCtlData (ctl: ctlRecHndl): longint;

{ Returns the contents of a controls ctlData field              }
{                                                               }
{ Parameters:                                                   }
{    ctl - control                                              }
{                                                               }
{ Returns: ctlData field                                        }

var
   swap: integer;                    {used to swap words}
   val: long;                        {param/ctlData field}

begin {GetCtlData}
val.long := GetCtlParams(ctl);
swap := val.msw;
val.msw := val.lsw;
val.lsw := swap;
GetCtlData := val.long;
end; {GetCtlData}


procedure DoCut;

{ Handle a cut command                                          }

var
   ctl: ctlRecHndl;                  {target control handle}
   id: longint;                      {control ID}
   port: grafPortPtr;                {caller's grafPort}
```

```
begin {DoCut}
port := GetPort;
SetPort(wPtr);
ctl := FindTargetCtl;
id := GetCtlID(ctl);
if (id = ctlLine1) or (id = ctlLine2) then begin
   LECut(leRecHndl(GetCtlData(ctl)));
   LEToScrap;
   end; {if}
SetPort(port);
end; {DoCut}
```

Listing 14-8: Sample `DoCut` Subroutine

Passwords

Text edit controls are sometimes used to enter a password for online services or data encryption programs. One of the customary things to do when a user types a password is to hide the actual characters typed, displaying some bogus character. The last parameter in the text edit control resource definition is the character to display instead of the characters that are actually typed. If you leave the parameter off, the text edit control works like you would normally expect, displaying the characters the user actually types. If you put in a character, that is the character displayed.

$D7 is the traditional password character on Apple computers. This shows up as a hollow diamond character.

Problem 14-4: Start with the solution to problem 14-3. Add two edit line items with a maximum allowed length of 20 characters.

Make sure your controls support the Cut, Copy, Paste and Clear commands. In particular, you should be able to cut or copy text from one control and paste it into another.

The radio button controls in your program have key equivalents of R, G and B. Try typing these characters in your line edit control.

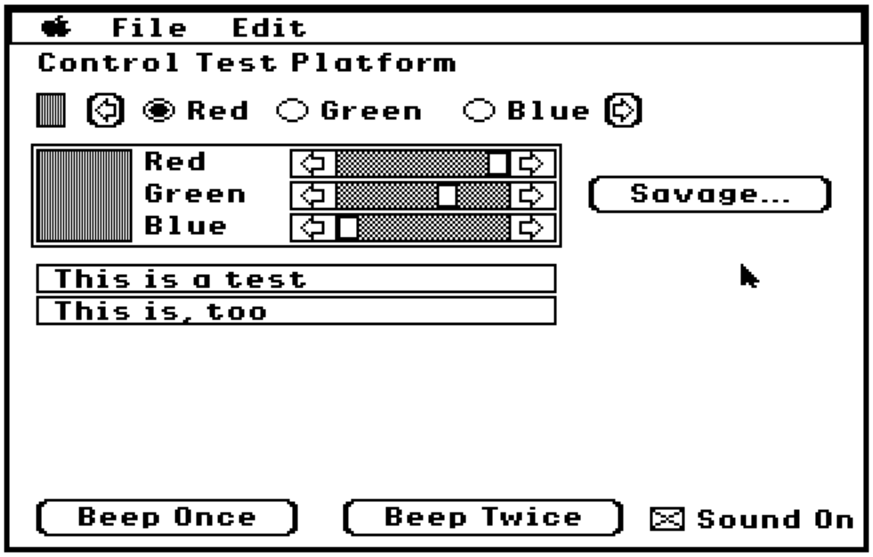


Figure 14-6: Control Sampler with Edit Line Controls

List Controls

List controls are used to display more than one choice in a box. It's possible, and even common, to have more things in the list than can be displayed in the box, so list controls come complete with their own scroll bars. The most common examples of list controls are the lists of file names in the SFO dialogs.

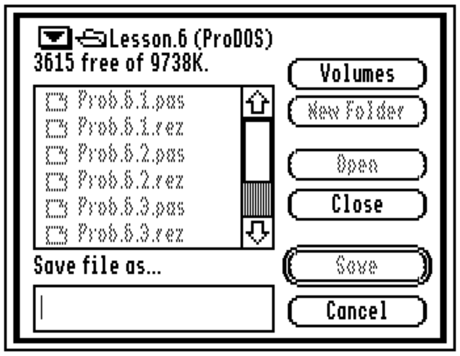


Figure 14-7: The SFO Save As Dialog Uses a List for Files

I've assumed throughout this course that you know how to *use* desktop programs, but list controls have a lot of options that you may not be aware of. I'm going to mention them briefly here; if you want to know more about the options, you can look up lists in either the user documentation that comes with the computer and system software, or you can dig the same information out of the toolbox reference manuals.

Most lists let you select more than one member by holding down the shift key or option key and clicking. You can tab from one control to another – if a list control is the target control, you can type a character key, and the List Manager will select the first element of the list that starts with the character you type. The arrow keys can also be used to move through a list, or you can use the old standby and click on the scroll bar, then click on the list member you want to select.

As usual, we'll start with a typical resource for a list control and the flags, then go over the details. As with the icon control, there are a lot of options, so don't get too bogged down in the details, yet.

```
resource rControlTemplate (1001) {
   10,                                /* control ID */
   {8,8,60,128},                      /* control rect */
   ListControl {{
      $0000,                          /* flags */
      $1400,                          /* more flags */
      0,                              /* refcon */
      0,                              /* list size */
      5,                              /* List View */
      $0000,                          /* List Type */
      0,                              /* List Start */
      10,                             /* ListMemHeight */
      5,                              /* List Mem Size */
      0,                              /* List Ref */
      0.                              /* Color Ref */
      }};
   };
```

Listing 14-9: A Typical Icon Button Control

`flag` bits	use
15-8	Reserved; set to 0.
7	0 for a normal, visible control, 1 for an invisible control.
6-0	Reserved; set to 0.

`moreFlags` bits	use
15-13	Reserved; set to 0.
12	Must be set to 1.
11	Reserved; set to 0.
10	Must be set to 1.
9-4	Reserved; set to 0.
3-2	Defines the type of reference for the color table field. 00 Color table reference is a pointer. 01 Color table reference is a handle. 10 Color table reference is a resource ID.
1-0	Defines the type of reference for the list field. 00 List reference is a pointer. 01 List reference is a handle. 10 List reference is a resource ID.

Creating the List Control

The list control itself is made up of a rectangle and a scroll bar. The scroll bar appears just to the right of the rectangle, and is always the same height as the rectangle.

The items in the list are displayed inside the rectangle. You have control over how many items are visible, and how high each line is, but you cannot control the width (it always matches the width of the rectangle), create sideways lists, or create scattered lists. In other words, lists are line oriented.

The fact that lists are line oriented doesn't mean they can only use text, though. While we won't go into details here, it's possible to create a list of pictures, or anything else you'd like to draw. The only restriction is that each item in the list must be the same height, and as I mentioned, one entire line will always be set aside for each member of the list.

With this in mind, it's pretty easy to understand the various flag settings and the fields in the list resource. Here's a field by field breakdown of the parts of the resource that are specific to the list control, along with the values from our example:

```
      0,                                     /* list size */
```

You can create a list with nothing in it and fill in the list under program control, or you can create a list with some items already there. Frankly, it's pretty tough to create the list from a resource, since the format for a list actually changes, and some lists need a custom drawing routine. In most cases, you'll want to create an empty list and fill it in from the program. That's how we'll do it in this section. For an empty list, the list size field is set to 0.

```
      5,                                     /* List View */
```

This field gives the number of items you can see at any one time. This has nothing to do with the number of items that are actually in the list; this is the number of lines that the user can see.

The height of the list is set in a separate field labeled `ListMemHeight`. The height of the control rectangle should be `ListMemHeight*ListView+2`; the extra two pixels give a small border at the top and bottom of the list.

The width is up to you. If something in the list is too long to display, the right side will be chopped off. The user can't scroll a list horizontally, so it's a good idea to make lists wide enough to see the entire line.

```
        $0000,                                  /* List Type */
```

The list type field is another flags word. Something happened here that would be convenient if it were done throughout the toolbox: to get the default list control, you set all of the bits to 0.

Bits 15 to 3 are reserved, and must be set to 0.

Bit 2 controls how the scroll bar is drawn. Normally, the scroll bar is drawn to the right of the control rectangle, so the area covered by the control is a little wider than the rectangle you code in the resource. If you set this bit, the size of the list area is reduced enough so the scroll bar lies entirely inside of the control rectangle. This hacks a little off of the end of each line, but it gives you better control over the space in the window, allowing very precise alignment of the right edge with other controls.

Bit 1 controls the type of selections allowed. Normally the user can select any number of items in the list by shift clicking or one of the other selection shortcuts. If this bit is set, only one item can be selected at a time.

Bit 0 controls the type of strings used in the list. Setting the bit to 0 gives the normal p-strings, with a leading length byte. If the bit is set to 1, the strings should be null terminated strings.

```
        0,                                      /* List Start */
```

When the list is first drawn, you can set things up so a particular item in the list appears at the top. Normally this value is set to 1 to show the first item in the list, or to 0 when you aren't supplying a default list in the resource.

```
        10,                                     /* ListMemHeight */
```

This value gives the height of one line of the list, in pixels. A value of 10 works well when the list contains text strings using the system font.

```
        5,                                      /* List Mem Size */
```

The list itself is defined as an array of list items, but strangely enough, the elements of the array don't have a fixed size. We'll go into this in gory detail in a moment, but for now, this is the field you use to specify the length of each element of the list array.

```
        0,                                      /* List Ref */
```

This field points to a list resource. Using the list resource, you can set up a default list, but there are a number of dicey technical issues related to doing so, especially if your list needs a special list drawing routine. If you want to try setting up a default list, refer to the *Apple IIGS Toolbox Reference: Volume 3*, pages 28-57 and E-51 to get started.

Building a List

With a list control in place, the next step is to create the list. The list itself is an array with two fixed fields in front, and any number of other fields afterward. As a general rule, it's best to define your own list type instead of using the one from the interface file. After all, the size of each element in the list and the size of the list itself need to be customized for each list.

In the simplest, and normal, case, each list element is a five byte long record.

```
type
   listElement = record
      memPtr: pStringPtr;
      memFlag: byte;
      end;
```

`memPtr` is a pointer to a p-string; this is the text the List Manager will draw on each line. If the first thing in your list isn't a p-string, you have to supply your own custom drawing routine to draw the list elements. `memFlag` is a flag byte. (Since it is a byte, not a word, there are only eight bits, not the sixteen bits you are used to in flag words.) Bits 6 and 7, in combination, let you tell the List Manager if the list member can be selected, and let the List Manager tell you if the memory actually has been selected. This flag byte has to be in this location for the List Manager to work correctly with your list.

Bits 7-6	Meaning
00	The list member can be selected, but it isn't currently selected.
01	The list member is disabled, and always shows up as dimmed.
10	The list member can be selected, and at the moment, it is selected.
11	The list member is disabled, but it is currently selected.

The list itself is an array of these records. You can create the list dynamically if it will change size, or you can just use a fixed size array. Using a fixed size array, you might have something like this:

```
const
   listSize = 8;

type
   listArray = array[1..listSize] of listElement;

var
   list: listArray;
```

The list itself should be filled in before you pass the list to the List Manager. Be sure and fill in both the pointer to the string and the flag byte. (The flag byte is normally initialized to 0.)

Once you have a list and a control, the next step is to connect the two. That's done with the List Manager's `NewList2` tool call:

```
NewList2(nil, 1, ord4(@listArray), 0, listSize, cHandle);
```

There are several parameters, most of which are actually overriding values from the resource definition. The first parameter is the address of the procedure that draws lines in the list. Passing nil, as we're doing here, tells the List Manager to do that for us. Passing `pointer(-1)` tells the List Manager to keep on using whatever drawing routine it was using before. You can also pass the address of a custom drawing routine; that's something we'll talk about a little later.

The second parameter, which is a 1 in our sample call, is the element in the list to put at the top. The first element in the list is numbered 1.

The next two parameters are the list itself. The first is a pointer to the first element of the list, but there's a catch: you pass the parameter as a long integer. The reason is that it doesn't have to be a pointer; it could also be a handle or a resource ID. The parameter right after, which is a 0 in our sample call, tells the List Manager what sort of value we are passing for the list. The 0 we are passing tells the List Manager to expect a pointer, 1 is used for a handle, and 2 is used if the list is given as a resource number.

The next element is the size of the list; that's the number of elements that are actually in the list at the current time. In this example, we're passing the size of the array, so each element of the array needs to point to a valid string, and needs to have the flag byte initialized.

The last parameter is the control handle, which you can get by calling `GetCtlHandleFromID`.

Sorting Lists

Sometimes it makes sense to sort a list, say alphabetically, before you display the list, but it's easier to build the list out of order. The most common list of all, a list of file names, is a good example. Making calls to GS/OS, it's pretty easy to get all of the file names, but GS/OS passes them back in the order that the files appear on disk, which isn't usually alphabetical order. People are used to seeing the file names in alphabetical order, though, and some of the keyboard shortcuts the List Manager supports only work when the list is sorted alphabetically.

The List Manager can sort the list for you. In fact, it's pretty easy to sort the list:

```
   SortList2(nil, cHandle);
```

This call sorts the list, doing all sorts of intelligent things, like ignoring the case of letters and handling foreign characters, like ü, in a reasonable way.

You can get tricky with this call, passing a custom sort procedure for the first parameter. Unfortunately, the person who designed the mechanism for custom sort routines either didn't know or didn't care how high-level languages work, and as a result, you have to define the sort routine from assembly language. If you would like to see the details, refer to the *Apple IIGS Toolbox Reference: Volume 1*, page 11-24.

The last parameter, of course, is the list control handle.

If you sort the list after it has already been displayed, or if you add, remove, or change a member of the list, you need to tell the List Manager to redraw all or part of the list. After all, the List Manager really doesn't know what you are doing to the list while your program has control. To redraw the list, make a call to `DrawMember2`, like this:

```
   DrawMember2(item, cHandle);
```

`DrawMember2` draws the entire list if you pass 0, and a specific item from the list if you pass some other value.

Custom Draw Procedures

Fortunately, custom draw procedures don't suffer the same limitation as custom sort procedures – you can define one in Pascal! Here's a framework for a custom list draw procedure:

```
{$databank+}

procedure CustomDraw (r: rect; member: listElementPtr;
   ctlHandle: ctlRecHndl);

{ Custom List Drawing Procedure                              }
{                                                            }
{ Parameters:                                                }
{    r - rectangle enclosing the area where the list element is }
{        to be drawn                                         }
{    member - pointer to the list element to draw            }
{    ctlHandle - scroll bar handle                           }

begin {CustomDraw}
end; {CustomDraw}

{$databank+}
```

Listing 14-10: Custom Draw Procedure Framework

To use this procedure, pass the address of the procedure (`@CustomDraw`) as the first parameter to `NewList2`. As with any procedure that will be called by the tools, you must remember to tell the compiler to reset the data bank register with the `{$databank+}` directive.

The first of the parameters is a rectangle enclosing the area where the list element is to be drawn. You're on the honor system – drawing outside of the lines leads to a messy window. The next parameter is a pointer to the list element to draw. If you need to know the relative position, you can count it off by scanning your array, or calculate it based on the address of the array. Be sure to check the flag byte to see if the list member is disabled; you generally need to draw the item in a dimmed state if it is. You also need to check to see if the item is selected; selected items are generally drawn in inverse.

Problem 14-5: Start with the solution to problem 14-4. Add a list control that is tall enough for 4 items and 130 pixels wide, as seen in Figure 14-8. From your program, create a list with ten states:

California	Colorado	Connecticut	New Hampshire	New York
New Mexico	New Jersey	Alaska	Alabama	Arkansas

Create the list with the states in this order, then call `SortList2` to sort the items before calling `NewList2` to display the list.

You should be able to select any of the states, or even a range of states, and you should be able to use keyboard characters to select a particular member. Make sure all of that works for your list.

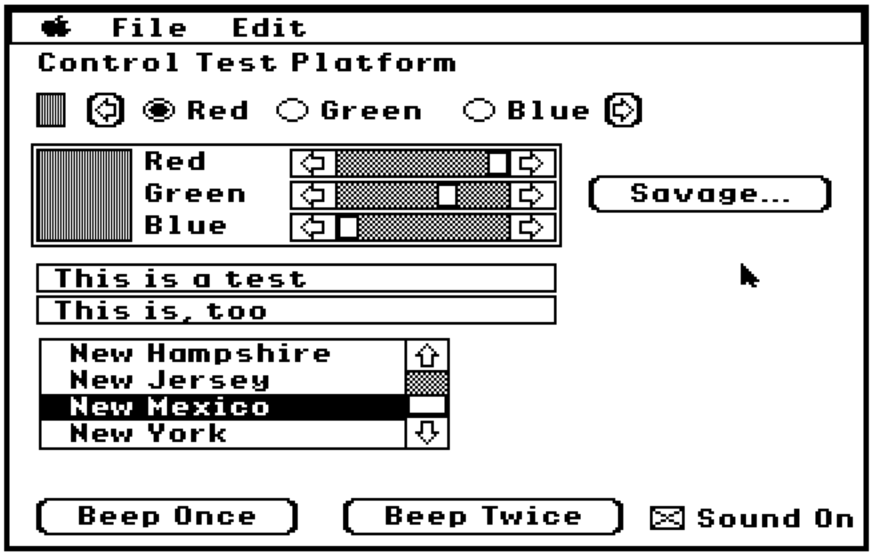


Figure 14-8: Control Sampler with List Control

Pop-up Menus

Pop-up menus are a cross between the concept of a radio button and the user interface of a menu. The control itself has two parts, a title (on the left in Figure 14-9) and the result, or pop-up rectangle (on the right). Figure 14-9 shows a window with a typical pop-up control; this one is from Design Master.

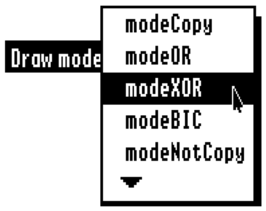


Figure 14-9: A Typical Pop-up Control

Figure 14-9 shows the control in its "popped" state, when the user is pressing on the control. In this state, the menu associated with the control is visible. If there are more menu items than will fit in the available space, the uppermost or bottom-most menu item will be a scroll arrow, so there is no theoretical limit to the number of choices you can offer, other than the number of menu items you can specify. In the "unpopped" state, the menu vanishes, and the currently selected choice is shown in the pop-up rectangle. The title is just there for information, but if you click on the title, the menu pops up.

I mentioned that the pop-up control implements the concept of a radio button, and that's perfectly true. In both cases, the idea is to offer the user one choice out of a lot of possible choices. The advantage of pop-up controls over radio buttons is that they don't use much space, even when you need to offer a lot of choices. The advantage of radio buttons is that the user can see all of the available choices easily, without popping the control.

Technically, you can use a pop-up control to select multiple choices from a list by checking the active choices, but from a user interface viewpoint, this stinks. A pop-up menu can only show one choice, not all of the ones that are selected. You could also do the same thing with radio buttons by putting each radio button in a separate family. That's bad, too. If you need to offer multiple on-off choices, use check boxes.

While pop-up controls are pretty complicated, there isn't nearly as much to learn before you can use them as you might think. The reason is that a pop-up control really is a pop-up menu; the menu itself looks just like any other menu you might create for your main menu bar. You still have all of the same options and choices, and you even use the same resource types to define the menu. In fact, most of this section deals with the choices you can make that effect the way the control itself is drawn.

Here's a typical pop-up control resource, along with its associated menu and menu item resources. The flags are defined below, and the various fields and flags are discussed in detail below the flag definitions.

```
resource rControlTemplate (1001) {
   10,                                /* control ID */
   {8,8,20,108},                      /* control rect */
   PopUpControl {{
      $0000,                          /* flags */
      $3002,                          /* more flags */
      0,                              /* refcon */
      50,                             /* Title Width */
      1001,                           /* menu Ref */
      300,                            /* Initial Value */
      0,                              /* Color Ref */
      }};
   };

resource rMenu (1001) {               /* the Color menu */
   1001,                              /* menu ID */
   refIsResource*menuTitleRefShift    /* flags */
      + refIsResource*itemRefShift
      + fAllowCache,
   1001,                              /* menu title resource ID */
   {300,301,302};                     /* menu item resource IDs */
   };

resource rMenuItem (300) {
   300,                               /* menu item ID */
   "","",                             /* key equivalents */
   0,                                 /* check character */
   refIsResource*itemTitleRefShift,   /* flags */
   300                                /* menu item title resource ID */
   };

resource rMenuItem (301) {
   301,                               /* menu item ID */
   "","",                             /* key equivalents */
   0,                                 /* check character */
   refIsResource*itemTitleRefShift,   /* flags */
   301                                /* menu item title resource ID */
   };

resource rMenuItem (302) {
   302,                               /* menu item ID */
   "","",                             /* key equivalents */
   0,                                 /* check character */
   refIsResource*itemTitleRefShift,   /* flags */
   302                                /* menu item title resource ID */
   };
```

```
resource rPString (1001, noCrossBank) {"Color"};
resource rPString (300, noCrossBank) {"Red"};
resource rPString (301, noCrossBank) {"Green"};
resource rPString (302, noCrossBank) {"Blue"};
```

Listing 14-11: A Typical Pop-up Control

`flag` bits	use
15-8	These bits set the `ctlHilite` field for the control. For a pop-up control, this will normally be $00xx, giving an active control. If you want to create an inactive control, use $FFxx. (You can change the control from active to inactive using Control Manager calls, but they aren't covered in this course.)
7	0 for a normal, visible control, 1 for an invisible control.
6	If this bit is set, the pop-up control will have extra white space for scrolling. In the examples we'll use here, the bit will always be clear. Refer back to the toolbox reference manual if you want to explore this option.
5	Setting this bit highlights the menu title; clearing it draws the title in the normal way.
4	This bit is used to get rid of the title. (For a pop-up control, the title is the name that appears in the rectangle at the left of the pop-up part of the control.) If this bit is set, no title will be shown. Even if you don't want a title, though, you still have to put something valid in the title field, and the title width field is still used.
3	The current selection is generally shown in the box that pops up when the control is selected. If this bit is set, the box is left empty.
2	Pop-up controls can be quite long, expanding well beyond the edge of a window. Setting this bit tells the Control Manager not to allow the pop-up control to extend beyond the edge of the window, even if there are enough menu items that all of the choices can't be shown.
1	The title is normally left justified. If this bit is set, the title is right justified, and the left edge of the title rectangle and the title width field are adjusted to eliminate unused pixels on the left side of the control. (See the description of the title width field for some pictures that explain this better.)
0	The selection rectangle normally extends from title width pixels past the left edge of the control rectangle to the right edge of the control rectangle, with the selected text left justified in the available space. Setting this bit right justifies the selected item, adjusting the size of the selection rectangle to fit near the left edge of the text.

`moreFlags` bits	use
15-14	Reserved; set to 0.
13-12	Must be set to 1.
11-5	Reserved; set to 0.
4-3	Defines the type of reference for the color table field. 00 Color table reference is a pointer. 01 Color table reference is a handle. 10 Color table reference is a resource ID.
2	Defines the type of reference in the menu reference field. A value of zero tells the Control Manager that the menu is given as a menu template, just like you are used to laying out menus for the program. A value of one tells the Control Manager to expect a menu string. We'll use a value of zero, since it matches what you are already used to doing, and works just as well

	as the other method. For details about menu text strings, see the toolbox reference manual.
1-0	Defines the type of reference for the menu field. 00 Menu reference is a pointer. 01 Menu reference is a handle. 10 Menu reference is a resource ID.

The Size of the Control and Text Justification

The size of the pop-up control is determined by two things: the control rectangle and the title width field. The control rectangle sets the overall size of the control, extending from the left edge of the title to the right edge of the pop-up rectangle, and from the top to bottom of both fields. The text you use in the title and in the various menu items will govern the width of the control. The height should generally be 12 pixels, which gives enough room to display text in the system font, but doesn't take up any extra screen space.

The title width field determines where the boundary is between the title and the pop-up rectangle. It's measured from the left edge of the control.

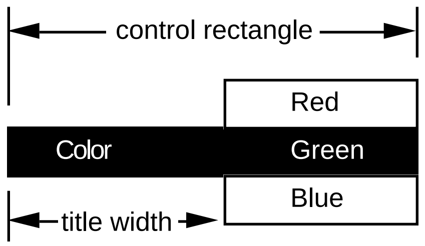


Figure 14-10: The Rectangle and Title Width Fields

The title text is normally left justified in the part of the control rectangle to the left of title width. If you set bit 1 of the `flag` word, the title text is right justified. The rectangle itself also shrinks, pulling even with the left edge of the title text (with a small margin to make things look neat). This doesn't change the overall width of the control, it just doesn't use all of the available space.

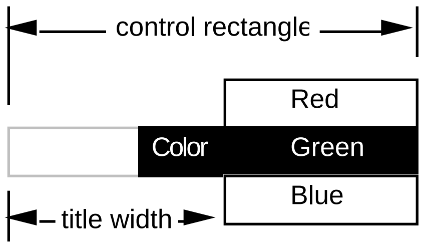


Figure 14-11: Right Justified Title

The selection is normally left justified, too. You can right-justify the text in the pop-up rectangle by setting bit 0 of the flag word. The box still gets shrunk, so the net effect is to shrink the visible part of the pop-up rectangle so it is even with the right edge of the text.

Display Options

The `flag` word has a number of options to control the overall appearance of the pop-up control. You can tell the Control Manager to print the title normally or in a highlighted state, to skip printing either the control title or the currently selected value, to add some extra whitespace to make scrolling easier, or to restrict the control's popped state so it doesn't draw past the edge of the window. Once you finish Problem 14-6 you can experiment with these flags to see exactly what they do.

Detecting the Control State

When the user selects something from a pop-up menu, the Control Manager will report a hit on the control. Once you know something has happened to the control, you can use `GetCtlValue` to read the state of the control. The value returned will be the menu item id for the menu item the user selected.

As far as when you look at the menu item, you have two choices. In most dialogs, you really don't care what the user picks until the dialog is closed. In that case, you can ignore hits on the control, and just wait until the dialog is closed and read the control value once. If you need to do something in the window or dialog as soon as the user picks a new value, though, you will need to track the hits on the control a little more carefully, reading the control value after each hit.

Problem 14-6: Add the color picker pop-up control we used as an example in this section to the solution to Problem 14-5. Add a colored rectangle above the pop-up control, coloring it appropriately as the user picks colors using the control.

This completes the control sampler, which now has one example of nearly every control type available in System 6.0. (We left out TextEdit and Picture controls.)

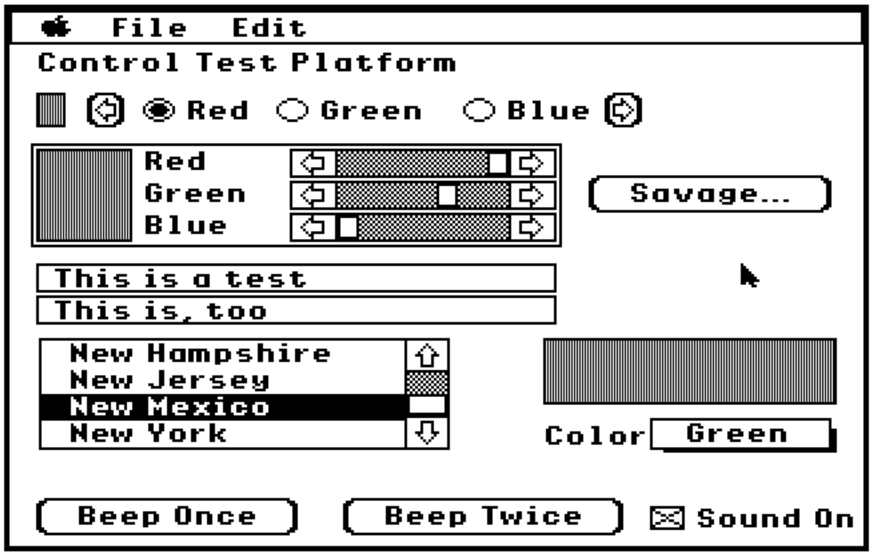


Figure 14-12: The Completed Control Sampler Window

Summary

This lesson and the last one have covered all of the controls available as of Apple IIGS System Disk 6.0 except for the TextEdit control, which was covered separately in Lesson 11. In each case, you learned how to create the control, how to track the control manually in a window, and how to read the setting of the control.

Tool calls used for the first time in this lesson:

```
DrawMember2        GetLETextByID      FindTargetCtl      LECopy
LECut              LEDelete           LEFromScrap        LEPaste
LEToScrap          NewList2           SetLETextByID      SortList2
```

Resource types used for the first time in this lesson:

`rIcon`	`rPicture`

Lesson 15 – Meaningful Dialogs

Goals for This Lesson

This lesson covers dialogs. By the end of the lesson, you will know the difference between modal and modeless dialogs, and how to create and use each kind of dialog. You will learn when to use a modal dialog and when to use a modeless dialog, and you'll find out about some of the human interface issues that apply to dialogs and menu commands that bring up dialogs.

Death of a Tool

Before you flip open your toolbox reference manual to the chapter on the Dialog Manager, I want to take a moment to warn you that the Dialog Manager is obsolete. Back when the Apple IIGS was first released, the Dialog Manager was a great tool. It handled alerts, modal dialogs and modeless dialogs, making it a lot easier to handle the various controls. In a very real sense, the Dialog Manager did for control events what `TaskMaster` does for the main event loop: it packaged the process in a few neat tool calls so you didn't have to reinvent the wheel every time you used a dialog.

As time moved on, though, Apple's engineers extended the controls available and added resources, both of which strained the design of the original Dialog Manager. Today, the Window Manager call `AlertWindow` has replaced the original Dialog Manager calls to handle alerts, and resources and the Window Manager's `DoModalWindow` have replaced the old Dialog Manager calls that handled modal dialogs and modeless dialogs.

If you troll the online services for source code samples, you will run into a lot of old programs (and a few new ones written by programmers that haven't caught up with the new tool calls) that use the Dialog Manager. After reading this lesson, you shouldn't have any trouble figuring out what those calls do by reading the descriptions in the toolbox reference manual. The concepts haven't changed, just the calls and which tool the calls are found in.

Dialogs are Simple Windows with Controls

The last two lessons spent a lot of time covering controls, but if you think about it, most programs don't use many controls in their document window. Whether you are using a text editor, spread sheet, CAD program, or whatever, most of the windows that show data have scroll bars, a size box, and not much else in the way of controls. (The big exceptions are the hyper-environments, like HyperCard GS and HyperStudio.) Most of the controls appear in supplementary windows that pop up in response to picking a menu command. These supplementary windows are, of course, dialogs.

From a user's viewpoint, dialogs are special windows that let the user pick options, conduct searches, and so forth. From the programmer's standpoint, dialogs are windows with controls.

There are two major kinds of dialogs. Modal dialogs lock the user in a mode – all the user can do is operate the various controls in the dialog. An alert is actually a special, restricted form of modal dialog. Another common example of modal dialogs are the various dialogs in SFO.

Modeless dialogs work pretty much the same way as a modal dialog, but they are true windows. The user can click on a document window or another modeless dialog to bring the other window to front and work on it, and the modeless dialog is still available. String search and string search and replace dialogs are usually modeless dialogs.

Visually, a modal dialog should generally use an alert frame, while a modeless dialog generally looks like any other window.

Creating a Dialog

Whether you are creating a modal dialog or a modeless dialog, you create the window the same way you created the control sampler window in lessons 13 and 14. In short, you use an `rWindParam1` resource, with all of the controls attached via an `rControlList` resource. About the only difference between a standard window with controls and a dialog window is that it's customary to use an alert frame for a modal dialog window. You do that by setting bit 13 of the `wFrameBits` flag word in the `rWindParam1` resource. Some modal dialogs can be moved. In that case, you need the title bar. That does cause some problems, since the Window Manager doesn't like to create windows with alert frames and title bars, so you'll have to draw the alert frame for yourself – or use another tool call that will. (Stay tuned! We'll get back to this point later.)

Handling Events in a Modal Dialog

There are a lot of problems with modal dialogs that aren't very apparent at first glance. Here are a few of the major ones:

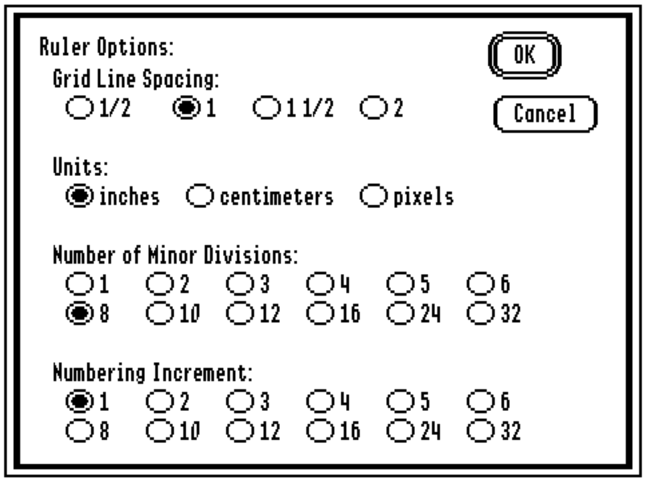
- Modal dialogs have to be the front-most window from the time they first appear until the user does something to make the dialog go away. You've seen this with alert windows, which don't let the user select any other window until one of the alert window buttons is clicked. Our main event loop has always been set up to allow the user to select any window at any time. While there are ways to prevent this, it messes up the main event loop, making the program more complicated.

- A modal dialog should grab the user's attention for the program they are running, preventing them from selecting some other program window, but it's handy to be able to use desk accessories while a modal dialog is in use. After all, the user may want to do some calculating with a calculator desk accessory, or scan a disk with some disk utilities, to figure out what to do about the dialog. This is possible from the main event loop, but it is very messy.

- Inside your program, you usually use a modal dialog in a situation where you need to get some information, then move on to process it, finally returning to the main event loop. Ideally, your subroutine would consist of three main parts:

  1. Draw the dialog.
  2. Handle any user events to set the various controls.
  3. Act on the user's choices.

  To see what I mean, think about this ruler dialog from Quick Click Draw:



Ruler Options:

Grid Line Spacing:
1/2 [✓ 1] 1 1/2 2

Units:
[✓ inches] centimeters pixels

Number of Minor Divisions:
1 2 3 4 5 6
[✓ 8] 10 12 16 24 32

Numbering Increment:
[✓ 1] 2 3 4 5 6
8 10 12 16 24 32

Figure 15-1: Ruler Dialog from Quick Click Draw

This dialog appears when the user picks the "Custom Rulers..." command from the Layout menu. The natural way to write the program is:

1. Draw the dialog.
2. Handle any user events.
3. Make the appropriate changes to the internal tables that control the ruler in a drawing.
4. Redraw the front-most user window with the new ruler settings.
5. Return control to the main event loop.

This leaves you between a rock and a hard place concerning the program design, though. The first choice is to mess up this clean subroutine design (and your main program loop) by bailing out to the main program loop after drawing the dialog, letting the main event loop handle user events, then call some other subroutine to handle the ruler changes. The other choice is to create a second event loop to handle the dialog. That is bad for two reasons: it makes the program larger, and it creates a second event loop, making the program harder to modify and making errors more likely.

Well, I wouldn't have spent all that time telling you about all of these problems if there wasn't a solution, and there is. The Window Manager has a call called `DoModalWindow`. It's sort of like `TaskMaster` for modal dialogs. You call `DoModalWindow` repeatedly, just like you call `TaskMaster` in your main event loop. `DoModalWindow` returns the control that was hit (if any), just like `TaskMaster` returns the event code, and even returns 0 if no control was hit, just as `TaskMaster` returns null events. You just call this subroutine until you see a hit on one of the buttons that gets rid of the dialog, like an OK button or a Cancel button, then drop out of your loop and do whatever else you need to do.

`DoModalWindow` is defined like this:

```
function DoModalWindow (event: eventRecord;
   updateProc, eventHook, beepProc: procPtr; flags: integer): longint;
```

Using the call isn't very complicated, but there are a lot of options. I'm going to go into detail on a couple, and gloss over some others. The reason is to keep this discussion simple, and concentrate on the concepts rather than getting bogged down in details.

The first parameter is an event record; `DoModalWindow` uses this event record for calls to `GetNextEvent`. You can pass the same event record that you use in your main event loop, but I'd recommend against it. A local event record will keep the two loops separate, and just might prevent some hard to track down bugs. Keeping separate code separate is one of the fundamental principles of modularized coding, and that's really what you are doing when you use a local event record.

The next three parameters are addresses to procedures that will replace standard procedures inside `DoModalWindow`, or that give you a chance to get into the act, doing something while `DoModalWindow` processes an event. The ideas behind writing these procedures are the same as the ideas behind other procedures you have already learned about. I'm not going to go into details about these procedures here. If you are curious, you can refer to the toolbox reference manuals for details. All of these procedures have default procedures; you can pass nil for all three parameters to get the defaults.

The last parameter is a flag word. Here's a bit-by-bit breakdown:

bit 15	If this bit is set, the dialog can be moved. If the bit is clear, the dialog can't be moved.
	Setting this bit is a start, but you still have to have something to click on with the mouse. If you want a moveable dialog, you have to set this bit *and* make sure the dialog has a title bar.
bit 14	Some windows may need to be updated while `DoModalWindow` has control. If the dialog is moveable, just moving it might uncover part of another window. Even if the dialog can't be moved, a desk accessory or interrupt handler (like a message from the AppleTalk network software) could wipe out part of a window. If this bit is set, `DoModalWindow` will try to call the update procedure for any window that needs to be updated; that will work fine the way we've defined windows and update procedures in this course. If this bit is not set, windows other than the dialog itself and desk accessory windows won't get updated unless you do it manually in the loop that calls `DoModalWindow`.
bits 13-6	These bits are reserved, and must be set to 0.
bit 5	If this bit is set, `DoModalWindow` returns any time it handles an activate event, giving your program a chance to do any clean-up tasks it might want to do after a system window appears or vanishes. If this bit is clear, `DoModalWindow` doesn't return after an activate event; instead, it loops and checks for another event.
bit 4	If this bit is set, `DoModalWindow` allows desk accessories to operate, and does everything appropriate to handle them.
bit 3	If this bit is set, `DoModalWindow` will switch the cursor from the normal arrow to an I-beam cursor when the cursor is over an edit line or text edit control. That's what you would normally want. Be careful, though – if this bit is set, and if your dialog actually has an edit line or text edit control, the cursor might still be an I-beam cursor when the dialog closes. It's up to you to call `InitCursor` to make sure the cursor is reset to an arrow.
bit 2	If this bit is set, `DoModalWindow` calls `MenuKey` to see if keys pressed while the open-apple key is pressed are menu command equivalents. If this bit is clear, `DoModalWindow` returns all key events except the standard keyboard equivalents for cut (X), copy (C), paste (V), and undo (Z). Those four key equivalents are always handled by `DoModalWindow`.

bit 1 If this bit is set, `DoModalWindow` will allow the user to pull down menus and use menu commands. `DoModalWindow` will handle any of the standard editing commands, but your program will have to handle any other menu commands. Of course, you could disable all of the menu commands except the ones you want to handle while you are in the dialog.

bit 0 If this bit is clear, `DoModalWindow` uses the Scrap Manager for cut, copy and paste. If this bit is set, `DoModalWindow` doesn't use the Scrap Manager.

Since `DoModalWindow` returns the part control for any controls that are hit, you can handle the controls the way you did in lessons 13 and 14.

Putting these ideas together, here's how you would use `DoModalWindow` in a typical loop:

```
dPtr := NewWindow2(@'Test', 0, nil, nil, $02, 1001, $800E);
repeat
   part := DoModalWindow(myEvent, nil, nil, nil, $C01E);
   case part of
      {handle control hits here}
      end; {case}
until part in [{put a list of the part codes for exit buttons here}];
CloseWindow(dPtr);
InitCursor;
{take action based on control settings here}
```

Problem 15-1: Start with the solution to Problem 12-1; that's the latest version of your text editor. In this problem, you'll add margin adjustment.

Create a menu command called "Margins...", adding it to the Style menu. (The "..." after the menu command name is a convention – all menu commands that bring up a dialog end with "…") When this menu command is used, bring up a modal dialog like the one shown below.

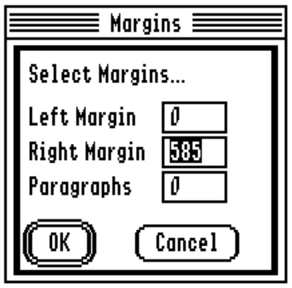


Figure 15-2: Margin Adjustment Dialog

The user should be able to move this dialog, use desk accessories, and use menu commands.

Once the OK or Cancel button is selected, drop out of the `DoModalWindow` loop. If the Cancel button was hit, don't do anything. If the OK button was hit, change the ruler and redraw the window.

Hint: You will need to call `TESetRuler` to implement this command, but you're also dealing with more then one window. That makes it tough to use nil for the last parameter, telling TextEdit to use the current text record. The program will be a lot easier to write if you use `GetCtlHandleFromID` to fetch the actual TextEdit control handle, and pass the result as the last parameter to `TESetRuler`.

Modeless Dialogs

A modeless dialog is just another window with a lot of controls instead of data. Since modeless dialogs are just windows, you need to handle them from the main event loop. One way to handle this situation is to implement two subroutines, one to see if the front window is a dialog, and the other to actually handle any event that might occur in the dialog. The possible events are usually limited to control events, so you don't usually have to make the check in every position in the event loop. Here's a typical event loop, modified to check for modeless dialog events:

```
repeat
   event := TaskMaster($076E, lastEvent);
   case event of
      wInSpecial,
      wInMenuBar:  HandleMenu;

      wInGoAway:   DoClose(grafPortPtr(myEvent.taskData));

      wInControl:  if IsDialog(FrontWindow) then
                      HandleDialog(FrontWindow);

      otherwise:   ;
      end; {case}
until done;
```

If there is more than one dialog, `HandleDialog` should probably call a separate subroutine for each one. Any duplication of effort this creates will be well paid for with a program that is easy to modify and debug.

When you actually use a modeless dialog, it's quite common to select a document window, bringing it to front, and covering up the dialog in the process. The natural reaction when you want the dialog and don't see it is to pull down the menu and use the command to bring up the dialog. The best way to handle this in your program is to check to see if the dialog is already open before you call `NewWindow2`. If the dialog is already open, just bring the dialog window to front by calling `BringToFront`, like this:

```
BringToFront(dialogWPtr);
```

Problem 15-2: Start with the solution to Problem 15-1, and add a new menu called Find. This menu should have one menu item, "Find..." (In a complete editor, it would also have a search and replace dialog, and perhaps some other movement commands.) Use a dialog like this for the command:

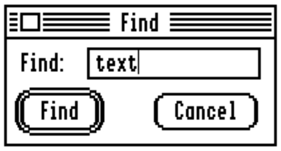


Figure 15-3: Sample Find Dialog

Add appropriate subroutines to create this dialog, detect and handle hits in the dialog from the main event loop, and to make the dialog go away when the appropriate button is hit. Use a close box with the dialog, and support closing the dialog when the cancel button is hit, when the close box is hit, or when Close is selected from the File menu and the dialog is the front window.

The point of this problem is to handle a modeless dialog, not to delve into the data structures and algorithms for searching a text edit buffer. If you would like to actually handle the Find and

Replace commands, go for it: it's a great programming exercise, but it won't teach you much about using dialogs. The solution does implement the commands, though.

If you do decide to implement the Find command, there are two problems you'll have to deal with. The first is finding the window to search – it's not the front window, since the Find dialog is active. You can use `GetNextWindow` to scan the open windows, starting with the Find window, to find the topmost document. The other problem is that you'll have to find the current selection point (so you know where to start the search) and set the selection point (to select the string, if you find it). `TEGetSelection` and `TESetSelection` will find and set the selection point. See Appendix A or the toolbox reference manuals for details about these calls.

Summary

This lesson introduced dialogs, which are the main kind of window used with controls. The lesson covered the difference between modal and modeless dialogs. It showed you that creating a dialog is no different than creating a standard window with controls. The lesson also covered the calls and ideas needed to handle events in both modal and modeless dialogs without destroying the structure of your program's main event loop.

Tool calls used for the first time in this lesson:

```
BringToFront      DoModalWindow     GetNextWindow     TEGetSelection
TESetSelection
```

Lesson 16 – Sound Off!

Goals for This Lesson

The 'S' in Apple IIGS stands for sound. This lesson introduces the various sound tools. It would be easy to fill a book this size with nothing but information about sound, and still not cover everything completely, so this lesson doesn't even attempt to cover all of the things you can do with sound on the Apple IIGS. Instead, this lesson shows you how to handle two of the most common and useful kinds of sound on the Apple IIGS: playback of digitally recorded sound and using the Apple IIGS to play notes using an instrument.

Along the way, the lesson tells you enough about what sound is for you to understand what the sound tools are doing. It also reviews the various tools, hardware devices, and software packages that you can explore to create and play back various kinds of sound.

The Physics of Sound

It's easier to see how the sound tools work, and a lot easier to design instrument files, if you have some idea what sound really is. You've probably heard that sound is a wave, but what does that mean? Most people think of a wave of water crashing onto a beach, or ripples in a pond. That's certainly one kind of wave, and there are a lot of similarities between ripples in a pond and sound waves. There are a lot of differences, too, though.

You see, there are three kinds of waves. Ripples on a pond are transverse waves; they happen when something moves back and forth (or up and down; in this case, the surface of the water). Sound waves are a compression wave, or longitudinal wave; sound is a compression of the air. The third kind of wave is a torsional wave, like a kid twisting the ropes on a swing, turning first to the left, and then to the right. Each of these waves behaves a little differently.

Sound gets started when something vibrates in the air. A really simple example is a guitar string. When you pluck the string, the string vibrates back and forth (a transverse wave). As it vibrates, the string moves air back and forth, too, first compressing it and then pulling back, forming a mild vacuum, which pulls on the air. All of those little air molecules get busy, banging into each other. As they do, a wave forms, which spreads out at the speed of sound. Imagining sound at the molecular level, with all of those little molecules bumping into each other, you can see why the speed of sound depends on a lot of things, like what those air molecules are made of (generally how much moisture is in the air), how many of them there are in a given amount of space (the density, measured as barometric pressure) and how fast the molecules were moving on their own (the temperature). In any case, the sound moves out from the string.

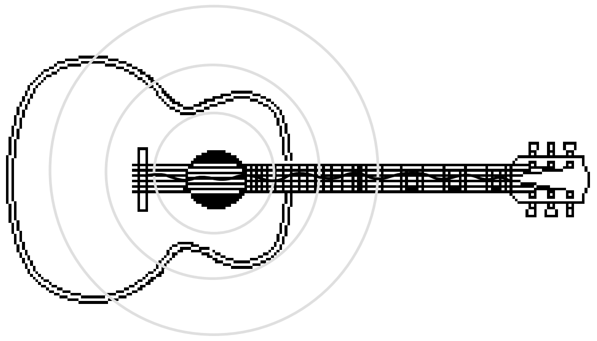


Figure 16-1: A Vibrating Guitar String Causes Sound

One of the big differences between a transverse wave, like ripples in a pond, and a longitudinal wave, like sound, is what happens when the wave hits something. When a water wave hits the side of a pool, for example, the water rises, then sinks, and the wave bounces back towards the middle of the pool. Sound does that, too; that's what causes an echo. The water doesn't jerk the side of the pool up and down, though, sending the wave through the surrounding concrete, soil, and so forth. On the other hand, sound does keep right on going. When sound hits a window, for example, it vibrates the window back and forth. Some of the energy in the sound wave is wasted, bouncing back towards the source. Some of the energy gets absorbed by the molecules in the glass, heating them up a bit. Some of the sound reappears on the other side, though, as the glass vibrates the air and sends the sound on it's way on the other side. With all of the energy that gets absorbed, it's easy to see why it's hard to hear someone on the other side of a window – but you can certainly hear them.

Eventually, sound arrives at your ear. The sound waves vibrate your eardrum, just like they vibrated the window. Your eardrum separates out the various frequencies of sound in a tuba shaped organ called the cochlea – each frequency ends up vibrating a little better in one part of the tube compared to the other parts. Nerves in the tube determine whether there is any vibration at a particular frequency, and how loud it is.

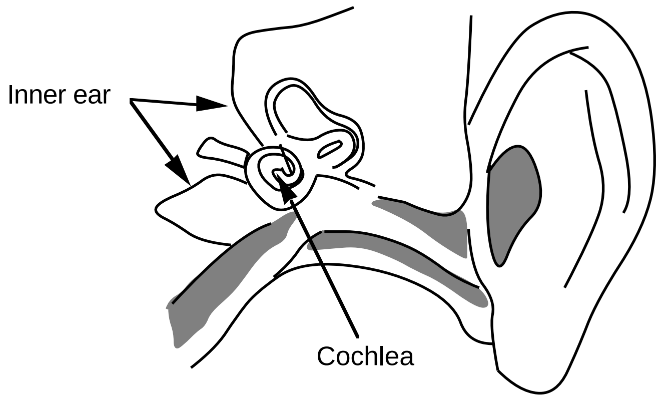


Figure 16-2: The Inner Ear

Your brain has a lot more to do with sound than most people think (pun intended). Sound doesn't stop at the cochlea, to be replaced by some nerve impulse that says, "That's just Mildred, again." The neural network of your brain starts filtering the sound right away, ignoring some parts and "paying more attention" to others. That's why you can "hear" someone talking from across the room, when a dozen other people's voices are at least as loud. As you turn your attention from one person to the next, you hear the one you are listening to, but the others are just a distraction.

To see how all of this impacts computerized sound, let's put the computer in the middle. This time, the sound hits a microphone. While there are a lot of different kinds of microphones, the original, and still one of the simplest, uses a cone to catch the sound. As sound waves hit the cone, the cone vibrates, and this in turn compresses a pack of carbon powder, a crystal, or something else that will change its electrical properties when it's compressed.. Current running through the carbon, for example, goes through easier when the carbon is compressed (less resistance) and has a harder time when the carbon is not compressed (higher resistance). The microphone, then, turns the sound into a changing electric current.

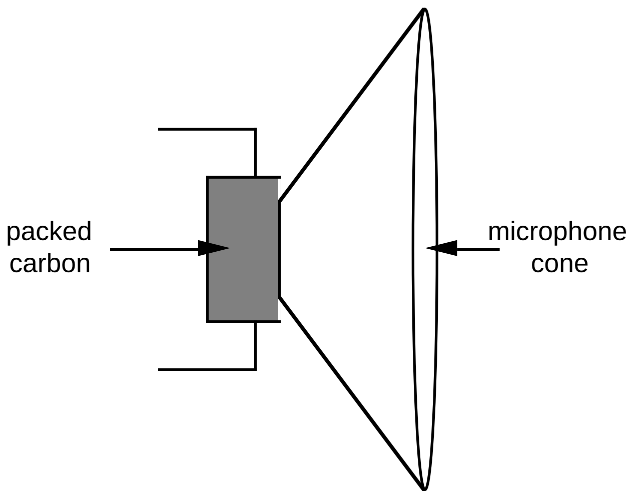


Figure 16-3: A Simple Microphone

There are a lot of ways to record the electric impulses created by the microphone. With a few intervening steps, though, the computer ends up converting the electric impulses into digital data. Ignoring all of the problems that might distort the sound, ideally the resulting digital data, when plotted, should be a plot of the pressure from the original sound wave.

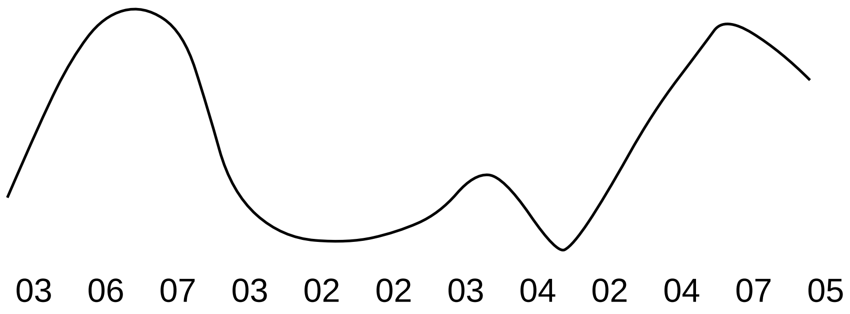


Figure 16-4: Sound Becomes Digital Data

What you see in Figure 16-4 is a digital recording of sound. This is the idea behind Compact Disc (CD) players, and it is also the idea behind the Sound Tool Set. While there are some details, the Sound Tool Set's basic function is to take a recorded stream of bytes and convert them back into sound. Through some very sophisticated electronic wizardry, the bytes of data are used to vibrate the cone of a speaker, which in turn vibrates the air, sending out a pretty good copy of the original sound wave.

As a quick aside, pop the cover off of your computer some time and take a look at the speaker that's built into the Apple IIGS. First off, it's not a particularly good speaker. Second, no speaker, no matter how good it is, is going to give very good sound sitting underneath a metal power supply box and sending the sound straight onto your table top. If you want good sound, take a few dollars down to Radio Shack and buy a really cheap external speaker with the proper cables to hook it into your Apple IIGS. Even with the cheapest speaker, the difference is amazing. In fact, you'll get quite an improvement if you just take your speaker out of the computer and lay it on a paper towel or thick pad of paper in front of the computer.

Digitally Recorded Sound

The type of sound we talked about in the last section is digitally recorded sound. Disregarding distortion, it's an exact copy of the original sound wave, converted to digital data. Even with sound values stored in a single byte and the rather unsophisticated speaker built into the Apple IIGS, you get some pretty amazing sound. It's also easy to play with the sound in this form – making certain parts louder, cutting and pasting sounds, blending one sound into another, playing old Beetles records backwards, and so forth. The disadvantage of digitally recorded sound is the amount of space it takes. Think about it this way: the CD ROM format was invented to hold the sound from a single music album, which is generally about 45 minutes long. A single CD ROM holds about 250M of data. Even a decked out Apple IIGS rarely has more than 5M of memory, and most hard disks range from 40M to 100M.

Another way to think about the amount of information is to figure out the number of bytes it takes to hold one second of sound. It's important to realize that the more bytes you use for each second of sound, the better the reproduction will be. After all, more samples let you create a better copy of the original sound wave. Of course, it also takes more memory to hold all of that sound.

Let's assume you want to get about the same sound quality as an AM radio station. That means you need to record at a rate of 16KHz, or 16,000 samples per second. That's about 16K of memory per second of sound. A little time with a calculator shows you can hold 4 seconds of sound in 1K of memory, or about one hour of sound per megabyte.

This also tells you a little about the sound quality of a CD! AM radio station quality is fine for speech, but is a bit on the light side for high quality music.

The Sound Tool Set

The Sound Tool Set is designed to give you more or less direct access to the Ensoniq synthesizer chip that is built into the Apple IIGS. This chip is literally a small music synthesizer, with 64K of it's own memory and up to 16 channels of sound. These sound channels are used in pairs, and one is reserved, but you can still play up to 7 distinct tracks of sound simultaneously. And, because the sound chip is really a separate sound-specific CPU, playing sound has very little or no impact on the main 65816 CPU. (For longer sound sequences, there is some impact as the sound data is moved back and forth from main memory. Some features of the sound tool also use the CPU, as you'll see. For a short, digitally recorded sound played over and over, the CPU isn't effected at all.)

We're going to use a few of the Sound Tool Set calls to play a digitally recorded sound. As you write the sample program, though, keep in mind that there's a lot more you can do with the Sound Tool Set. Not only that, but if the Sound Tool Set is getting in your way, it's even possible to bypass the tools entirely and program the Ensoniq chip directly. You'll need to drop into assembly language to do some things, of course. If you want to program the Ensoniq chip directly, you'll also need the *Apple IIGS Hardware Reference*.

The Basic Sound Calls

The basic tool call for playing a digitally recorded sound in `FFStartSound`. Here's a typical call:

```
FFStartSound($0201, soundBlock);
```

That looks simple enough. The first parameter controls the channel and synthesizer we'll be using, as well as identifying what sort of sound we're playing. I'll let you refer to the detailed description of the `FFStartSound` call to see how this parameter is used to play multiple sounds, but the $0201 value will work find for playing a single sound at a time. The second parameter is a record, and it's a pretty complicated one. Here's the record declaration, from the SoundMgr.pas file in the ORCA/Pascal header files:

```
    soundParamBlock = record
        waveStart:     ptr;                (* starting address of wave    *)
        waveSize:      integer;            (* waveform size in pages      *)
        freqOffset:    integer;            (* waveform playback frequency *)
        DOCBuffer:     integer;            (* DOC buffer starting address *)
        DOCBufferSize: integer;            (* DOC buffer size code        *)
        nextWAddr:     soundPBPtr;         (* ptr to next waveform block  *)
        volSetting:    integer;            (* DOC volume setting          *)
        end;
```

The various fields contain very specific information, and we'll need to step through them carefully.

`waveStart` Pointer to the first byte of the digitally recorded sound wave. The wave itself consists of a series of bytes.

The way you allocate memory for the wave itself is very critical. As you'll see in a moment, the memory has to be allocated in one page chunks, and the number of pages must be a power of two. The buffer also has to be aligned in memory on the same boundary as the size of the buffer. For example, since the smallest buffer is one page (256, or $100 bytes) the sound for a one-page buffer has to be aligned to a 256 byte boundary. That means that memory starting at $030400 would be fine, but memory starting at $030480 would not work. For a bigger buffer, the alignment is more critical. For a 4K buffer ($1000) you would need to align the memory buffer to a 4K boundary, so a buffer starting at $030400 won't work – you would need one starting at, say, $031000.

In practice, the easiest way to make sure the buffer is aligned properly is to get memory with `NewHandle` calls rather than with Pascal's new procedure, and ask the Memory Manager to align the memory to a bank boundary. That will work for sound buffers up to 256 pages long. If the sound you need to play is longer than 256 pages (64K), you can break it up into smaller pieces. From a memory management standpoint, that's probably a good idea, anyway.

Here's a sample `NewHandle` call you could use to allocate a 64K sound buffer. I'll let you verify the various parameters by checking with the description of the `NewHandle` call on your own.

```
soundHandle := NewHandle($010000, userID, $C014, nil);
```

This buffer is too big for the largest single chunk of sound, which is 32K long, but allocating a full bank is the only way to guarantee that the memory is aligned to a 32K boundary. The memory allocated by this call will actually be aligned to a 64K boundary. Of course, you can use the memory for two 32K sound buffers.

The sound bytes themselves range from 1 to 255 for each byte. A zero byte marks the end of the sound wave, so you can stop the sound before the end of a page boundary.

`waveSize` Size of the wave buffer, in pages. For a 32K sound (128 pages) the value is 128. If you need to calculate this value on the fly, use the equation

```
(size_in_bytes + 255) div 256
```

`freqOffset` This value is related to the playing frequency for the sound. Basically, you have to tell the Sound Manager what the sampling rate for the sound is, in hertz. (As used here, hertz is samples per second, not the normal cycles per second.)

The relationship isn't straight-forward, though. This parameter should be

```
round(32.0*hertz/1645.0)
```

where hertz is the frequency of the sample.

This simplification will work fine for our introduction, but I'm skipping over some details. You should refer to the complete documentation for the `FFStartSound` call before trying anything beyond playing back a prerecorded sound.

`DOCBuffer` The Ensoniq sound chip has 64K of it's own memory. The sound memory isn't handled by the Memory Manager, so it's up to you to make sure your sounds don't step on each other. This parameter is the location in that 64K of memory where the sound data will be placed. It should always be a multiple of 256 ($100), since the sound data is assumed to be in one-page chunks. For a single sound, you can use 0.

`DOCBufferSize` The smallest sound buffer is one page, or 256 bytes. Each higher size is twice the size of the next smallest buffer. This parameter represents the sound buffer size; it ranges from 0 to 7:

parameter	sound buffer size
0	256
1	512
2	1024
3	2048
4	4096
5	8192
6	16384
7	32768

`nextWAddr` This is a pointer to the next waveform to be played. By pointing one record to another, you can set up a sequence of sounds that will automatically be played one after the other. If the value is nil, nothing else is played. To play a single sound over and over, set this pointer to point to the current sound record.

`volSetting`	This is the volume for the sound. You can use any value from 0 (no sound) to 255 (the loudest).

Besides starting a sound, there are two other basic operations you may need to perform. The first is checking to see if a sound has finished playing. You do that by checking a specific sound generator; we're using generator 2, so the check would look like this:

```
while not FFSoundDoneStatus(2) do {nothing};
```

The parameter for `FFSoundDoneStatus` is the generator you want to check; the call returns true if the sound has finished, and false if it hasn't. The example will wait until the sound has finished; that's something we'll use in one of the sample programs a little later in the lesson.

The other thing you might want to do is to stop a sound. That's particularly useful if you've looped a sound back on itself by setting the `nextWAddr` parameter of the record to point back to the original record. That gives you a continuous background sound without taking up any CPU time, so your program runs just as fast as it would without sound. Before your program finishes, though, it should turn off the sound. `FFStopSound` does the job.

```
FFStopSound($7FFF);
```

With the parameter set to $7FFF, as it is in the example, `FFStopSound` turns off all sound, not just the sound for generator 0.

The only other thing to keep in mind is that you're using a new tool, so you need to add it to your tool start up table. The Sound Tool Set is tool 8. For System 6.0, you should use a version number of 3.3, coding $0303 in your `rToolStartup` resource.

Creating and Playing a Sound

For our first sound example, we'll create a sound from scratch. A "pure" sound, created by a perfect oscillator, would just be a sine wave. Middle C is 256 hertz. (For those of you who are about as musically knowledgeable as I am, Middle C is a musical note.) We can create a digitized sound that will play a pure middle C by creating a single sine wave as a series of bytes, then playing the sound over and over by pointing the sound packed back on itself. Rather than creating a full desktop program to do something this simple, we'll just use a simple text program that plays the sound until a key is pressed.

```
{----------------------------------------------------------------}
{                                                                }
{  Sine Wave Generator                                           }
{                                                                }
{----------------------------------------------------------------}

program SineWave (input, output);

uses Common, SoundMgr, ResourceMgr, MemoryMgr, ToolLocator, MscToolSet;

label 98, 99;

const
   twoPi = 6.28318531;                  {2*pi}
```

```
var
   i: integer;                          {loop/index variable}
   soundBlock: soundParamBlock;         {sound parameter block}
   soundHandle: handle;                 {sound block handle}
   soundPtr: ptr;                       {sound block pointer}
   startStopParm: longint;              {tool start/shutdown parameter}


begin {SineWave}
startStopParm :=                        {start up the tools}
   StartUpTools(userID, 2, 1);
if ToolError <> 0 then begin
   writeln('Could not start tools: ', ToolError:1);
   goto 99;
   end; {if}

                                        {create the sound buffer}
soundHandle := NewHandle($010000, userID, $C014, nil);
if ToolError <> 0 then begin
   writeln('Could not allocate a sound buffer: ', ToolError:1);
   goto 98;
   end; {if}
writeln('Setting up the sound...');     {put a sine wave in the buffer}
soundPtr := soundHandle^;
for i := 1 to 256 do begin
   soundPtr^ := round(sin(i/256.0*twoPi)*120 + 128);
   soundPtr := pointer(ord4(soundPtr)+1);
   end; {for}
with soundBlock do begin                {set up the sound parameter block}
   waveStart := soundHandle^;
   waveSize := 1;
   freqOffset := round(32.0*65536.0/1645.0);
   DOCBuffer := 0;
   DOCBufferSize := 0;
   nextWaddr := @soundBlock;
   volSetting := 250;
   end; {with}
writeln('Playing the sound...');        {start the sound}
FFStartSound($0201, soundBlock);
writeln('Press return to stop the sound.');
readln;                                 {wait for a keypress}
FFStopSound($7FFF);                     {stop the sound}

98:
ShutDownTools(1, startStopParm);        {shut down the tools}
99:
end. {SineWave}
```

Listing 16-1A: A Program to Play Pure Middle C

```
/*--------------------------------------------------------------*/
/*                                                              */
/*  Resources for Sine Wave                                     */
/*                                                              */
/*--------------------------------------------------------------*/

#include "types.rez"

resource rToolStartup(1) {
   mode640,
   {
       3, $0302,                        /* Misc Tool */
       8, $0303,                        /* Sound Tools */
      }
   };
```

Listing 16-1B: Resource Description File for the Sound Program

Problem 16-1: Games often use a ray-gun blast as a sound effect. One way to create a simple sound that will do is to create a series of sine waves that gradually diminish in volume, then repeat the sound several times.

Create a program that makes a sound like this, repeating the sound ten times. After the sound stops, the program should quit. The entire sound cycle should last for one second, and should be based on a 4096 hertz sine wave. Each 1/10 second sound burst should start at full volume and decrease evenly to zero volume.

Hint: Create ten separate sound records, each pointing to the same 1/10 second sound buffer.

Loading Sounds from Disk

In most cases, you won't create a sound from scratch. Instead, you'll load a prerecorded sound from disk, and play the recording. There are a lot of formats used for sound, mostly depending on how the sound was created. Apple's file type notes (see Appendix C) list several formats, and the documentation for various sound cards list even more.

You'll find a file called Hello.Sound on the disk that comes with this course. It's a recording of Fred greeting a guest. The sound was recorded on an Applied Engineering Sonic Blaster card using a microphone from Radio Shack. The software that comes with the Sonic Blaster records the sound as a header file followed by the actual sound bytes. Here's the format for the header:

bytes	information
0-3	The characters 'AEPM', as an identifier. The actual values are $41 $45 $50 $4D. You can skip over these bytes.
4-5	This is the kind of sound information. The Hello.Sound file has a value of 0, which means that the sound data is just a sequence of bytes. A value of 1 is used for sound that has been compressed to half its original size by the ACE Tool Set, while 2 means the sound has been compressed to 3/8 of its original size using ACE.
6-7	This word will be 0 for mono, and 1 for stereo. It's zero in the Hello.Sound file.
8-11	This is the length of the sound sample, in bytes.
12-13	This is the sampling frequency, in hertz. You need to set the `freqOffset` field of the sound record based on this value.

There are two differences between playing a prerecorded sound as opposed to playing a synthesized sound, like we did in the last section. First, the sound bytes are read from disk rather than being created by an equation. Reading the file should not be a big problem. Since speed isn't an issue, you can just declare a file of byte and let Pascal handle the file input.

The second difference is that you don't really know what frequency the sound was recorded at, so you need to read the frequency from the sound header and calculate a value for the `freqOffset` field of the sound record. This isn't hard, but it is the easiest place to make a mistake. Assuming you are reading the file as a series of bytes, and have read the frequency bytes into variables called byte12 and byte13, you can calculate the `freqOffset` value like this:

```
hertz := byte12 + byte13*256;
soundBlock.freqOffset := round(32.0*hertz/1645.0);
```

Problem 16-2: Write a program that loads the Hello.Sound file, plays the sound once, and exits. You can find the file in the Lesson.16 folder on the course disks. Be sure to wait until the sound finishes playing before exiting!

Note Synthesizer

Recorded sound has it's place, but when it comes to musical instruments, you expect something a little more flexible. If you want to play middle C with an oboe, it doesn't make much sense to load a prerecorded sound! That's when the Note Synthesizer tool comes in. With this tool, you can actually say (in effect) that you want to play middle C on an oboe, and the computer takes over and creates the sound.

Instruments

The obvious problem, of course, is that the Apple IIGS has even less of an idea as to what an oboe sounds like than I do. Instrument files are used to teach the computer what a particular instrument should sound like.

There are two issues to deal with with instruments, and once those are out of the way, using the instrument is pretty easy. The first is what the Apple IIGS thinks an instrument is. The second issue is where you get one.

The Instrument Record

The Note Synthesizer calls that play a note ask for a pointer to an instrument record. Here's the definition for an instrument record, straight from ORCA/Pascal's tool interface files:

```
instrument = record
  envelope:          array [1..24] of byte;
  releaseSegment:    byte;
  priorityIncrement: byte;
  pitchBendRange:    byte;
  vibratoDepth:      byte;
  vibratoSpeed:      byte;
  spare:             byte;
  aWaveCount:        byte;
  bWaveCount:        byte;
  aWaveList:         array [1..1] of waveForm; (* aWaveCount * 6 bytes *)
  bWaveList:         array [1..1] of waveForm; (* bWaveCount * 6 bytes *)
  end;
```

There are actually two parts to an instrument definition. The instrument record is the first part; it deals with things like how quickly an instrument starts making sound (a drum starts pretty quickly, while a flute builds up gradually), how the sound dies off, and so forth. These facts are stored in the envelope, and to a lesser extent, the various other fields. There's a lot more that's different between two instruments than the way the sound builds up and dies, though. Let's think about another familiar source of sound to see what I mean: your voice. Even if you do your very best to say something just like another person, it's usually pretty easy to tell the two of you apart. That's because the actual sound waves have a different shape. The same thing is true with instruments. The second part of the instrument definition deals with this issue; it's essentially a recording of the shape of the sound wave produced by an instrument. These wave forms are actually placed in the 64K of sound memory. The instrument record has some pointers which point to the start of the waves, and a field that tells how long the wave is. This information is stored in `waveForm` records:

```
waveForm = record
  topKey:      byte;
  waveAddress: byte;
  waveSize:    byte;
  DOCMode:     byte;
  relPitch:    integer;
  end;
```

The critical field is the `waveAddress` field; this changes depending on where you put the wave in the sound RAM. You won't have to worry about these fields for the sample programs in this lesson, since the samples will only use one instrument. Because of that, we can load the wave forms for the instruments right at the start of the sound RAM. Since the address field in the `waveForm` record in an instrument file is a displacement from 0, the `waveAddress` values will be correct. If you are using more than one instrument, though, you would have to figure out where to put each wave form, and adjust the `waveAddress` fields appropriately.

ASIF Instrument Files

There are a lot of different ways to store instruments on a disk file. Apple recommends one particular file format for people using the Note Synthesizer. The files are called ASIF files; they have a file type of $D8, with an auxiliary file type of $0002. The most popular program that uses this file format is Sound Smith. You can find lots of ASIF instrument files by using key words instrument, Sound Smith, or ASIF in the music special interest forums on national online services. In fact, the ones you'll be using in a moment were downloaded from America Online.

The ASIF file format is very flexible, with lots of optional fields. We're not going to go into all of the possibilities here, nor are we going to go into technical details on what all of the fields mean. This is a very nuts and bolts section. All the theory will be tossed to the wind, and we'll concentrate on the mechanics of actually loading an instrument from an ASIF file. If you want to design your own instruments, deal with ASIF files, or write programs that use multiple instruments, you'll need to learn a few more details than we'll cover here. In that case, you should get a copy of the Apple IIGS File Type Notes, particularly the one for file type $D8, auxiliary file type $0002. There's also a lot of good information about the fields in the description of the Note Synthesizer tool, which you'll find in Chapter 41 of the *Apple IIGS Toolbox Reference: Volume 3*. And now, back to our regularly scheduled nuts and bolts section...

ASIF files are made up of a series of data blocks called chunks, all imbedded in a super-chunk. There are a lot of different kinds of chunks with different kinds of information and different lengths, but all of the chunks start the same way: a four-character block-type identifier and a four byte length. There's nothing special about the four-character type; it's just four ASCII characters. The length is a little odd, though – the bytes are in the opposite order of what they would normally be in a file. (The first byte is the most significant byte, instead of the least significant byte.)

Because of the regular structure of the chunks, you don't have to know what all of them are to use an ASIF instrument file. In fact, the file is designed that way on purpose, so new kinds of chunks can be added without breaking old programs. When you come across a chunk in the file, you check the four-character type. If it's a chunk you need, you process the information. If it's not a chunk that you need, you add the length in the second four bytes plus 8 to the address of the start of the chunk, and the result is a pointer to the start of the next chunk.

Let's assume you already have a pointer to the first of the normal chunks, and you know what four-character type you are looking for. In addition, you know the total length of all of the chunks (that's the key piece of information imbedded in the super-chunk, which we'll talk about in a moment). Here's a subroutine that will scan the chunks, returning either a pointer to the first byte of data in the correct chunk, or nil if the chunk you are looking for doesn't exist.

```
type
   chunkType = packed array[1..4] of char; {used to identify chunk types}

function FindChunk (p: ptr; chunk: chunkType; len: longint): ptr;

{ Locate a chunk in an ASIF file                                  }
{                                                                 }
{ Parameters:                                                     }
{    p - pointer to the start of the first chunk                  }
{    chunk - character type for the chunk to find                }
{    len - number of bytes left in the file                       }
{                                                                 }
{ Returns: Pointer to the chunk data; nil if not found.          }

var
   disp: longint;                       {length of the data in the chunk}
   i: integer;                          {loop variable}
   id: chunkType;                       {chunk type for the current chunk}


   function GetByte: integer;

   { Read a byte from the file buffer                             }
   {                                                              }
   { Returns: Byte from the file                                  }

   begin {GetByte}
   GetByte := p^;
   p := pointer(ord4(p)+1);
   end; {GetByte}


begin {FindChunk}
repeat
   for i := 1 to 4 do
      id[i] := chr(GetByte);
   for i := 1 to 4 do
      disp := (disp << 8) | GetByte;
   FindChunk := p;
   p := pointer(ord4(p)+disp);
   len := len - disp;
   if len <= 0 then
      if id <> chunk then begin
         p := nil;
         FindChunk := nil;
         end; {if}
until (p = nil) or (chunk = id);
end; {FindChunk}
```

Listing 16-2: Function for Locating ASIF Chunks

I've mentioned the super-chunk a couple of times. Basically, the entire ASIF file is one huge chunk, and the data for the chunk is a series of smaller chunks. Each ASIF file is only supposed to have one of these super-chunks, but it's probably a good idea to allow for other information after the super-chunks. That way, if someone starts creating files with multiple super-chunks, your program will still work.

The super-chunk itself starts off with a type of 'FORM'. The next four bytes are the length. The length works just like the length for the regular chunks, so adding 8 to the length should give the length of the entire file. The next four bytes are the type of the file, and should be 'ASIF'. This is followed by the various normal chunks in the file.

Let's put all of this together, now, and sketch the overall form for handling these files. The program should read the entire file into memory, using the same ideas we used in Lesson 6. Next, check the two four-character type fields in the super-chunk to make sure the file is the right type of file. Finally, the pointer to the first normal chunk is formed by adding 12 to the pointer to the start of the file, and the length of all of the normal chunks is formed by reading the four-byte length field and subtracting 4 from the length. These two values are the values we need to pass to the `FindChunk` subroutine to locate a particular chunk in the file.

While an ASIF file can have all sorts of chunks, there are only two we need to worry about, and both of them are supposed to be in any ASIF file. The two chunks we need to find are the 'WAVE' chunk, which contains the wave form we need to copy into the sound RAM, and the 'INST' chunk, which contains the instrument record. (Actually, an ASIF file can have more than one 'INST' chunk. The way we'll write our program, well use the first instrument, and ignore any others, if there are any.)

The 'WAVE' Chunk

Let's start of with the 'WAVE' chunk. There's a lot of information in this chunk, but all we really need for our program is the waveform itself, which comes at the very end. I'll summarize the various fields we skip over, but the only part of the information you really need to use the instrument is how to skip the various fields. The names of the fields are taken straight from the file type notes, so you can cross reference what I'm telling you here with the more detailed information in the file type notes, if you like.

name	length	use
`ckID`	4 bytes	This is the chunk type, 'WAVE'. `FindChunk` skips this field for you.
`chSize`	4 bytes	This is the length of the chunk, minus the 8 bytes used by the name and this field. `FindChunk` also skips this field.
`waveName`	variable	This is a name field for the wave itself. It's a p-string, so you can skip this field by reading the first byte, which is a length byte, and adding the length plus one (the one is for the length byte itself) to the pointer.
`waveSize`	2 bytes	This is the size of the wave form. You need to read this value and save it so you know how many bytes have to be copied into the sound RAM. Unlike the length field in the header, this value is in the normal number order for the 65816, so the actual value is the value of the first byte plus 256 times the value of the second byte.
`numSamples`	2 bytes	This is the number of samples imbedded in the wave form. This isn't important when we're using the wave, but you still need to read the value, since there are `numSamples` copies of the next field.
`sampleTable`	12 bytes	These sample tables give some information about the way the wave form was originally sampled. Each of the sample tables is

		12 bytes long, but there can be more than one of them. To skip these fields, add `12*numSamples` to the pointer.
`waveData`	`waveSize+1`	Finally! Here are the actual bytes in the wave form. These are the bytes that need to be moved to the sound RAM.

The documentation for the ASIF file format says that `waveSize` is a zero based counter, so a value of 0 means the wave contains one byte, and so forth. In all of the files I downloaded and looked at, though, the size of the wave was exactly `waveSize` bytes. I'd suggest treating the wave form as if it is exactly `waveSize` bytes long. If that turns out to be wrong, you loose a single byte – and that isn't likely to be something you'll miss when the wave form is used. If you want to be extra careful, though, you could calculate the actual wave form size on the fly by subtracting the location of the chunk after the 'WAVE' chunk from the start of the `waveData` field. Either way, the program won't loose track of the data, since the program keeps track of things based on the original chunk length.

Now that you know how to find the wave, the next step is copying it into the sound RAM. You do that with the Sound Tool Set call `WriteRamBlock`:

```
WriteRamBlock(p, 0, waveSize);
```

The first parameter is a pointer to the first byte of the wave itself. The next parameter is the starting location in the sound RAM; we want to put the wave form right at the start of the sound RAM, so we pass a 0. The next parameter is the number of bytes in the wave, which we read from the chunk.

This particular call has two very strange requirements. The fist is that the wave itself can't be in banks $00, $01, $E0 or $E1. As it turns out, all four of those banks have some memory that is reserved by the Memory Manager, so one cheap way to fulfil this requirement is to make sure you allocate at least 64K of memory when you read the instrument file. The second requirement is that you have to disable interrupts before you make the call. There is no practical way to disable interrupts from Pascal, so I've included two assembly language subroutines that will get the job done. The subroutines are called `DisableInterrupts` and `EnableInterrupts`; you can think of them and use them as if they were new tool calls. You'll find these files in the folder for Lesson 16:

Interrupt.asm	Assembly language source files for the two subroutines. You don't need this for the course, but if you have the ORCA/M assembler, you can look at and modify the source files.
Interrupt.pas	The header file source code for the module. This is the equivalent of the tool header files in the TOOL.INTERFACE folder on your ORCA/Pascal disks. You can look at this file to find out the proper parameters. As it turns out, there aren't any.
Interrupt.int	The interface file. Copy this file to the ORCAPascalDefs folder of your ORCA libraries folder.
Interrupt	This is a library containing the executable code for the two subroutines. Copy this file to the ORCA libraries folder. As you may know, ORCA is fairly sensitive to the order of the library files, but this library can appear anywhere in your libraries folder without causing any conflicts.

Once you've copied these files into the proper folders, be sure to put a

```
uses Interrupt;
```

statement at the start of your program, just like you would for a tool header file. Then, when you are ready to set up the instrument wave form, make the calls like this:

```
   DisableInterrupts;
   WriteRamBlock(p, 0, waveSize);
   EnableInterrupts;
```

The 'INST' Chunk

The instrument record is the main component of the 'INST' chunk. Like I mentioned earlier, there has to be at least one 'INST' chunk, but it's possible to have more than one, too. As long as we're just trying to use a single instrument from a file, it will work fine to just scan for the first 'INST' chunk and use it.

The 'INST' chunk starts off with the characters 'INST' and a four-byte length, just like all of the other chunks. These eight bytes are skipped by `FindChunk`. There are two more fields we need to skip before we get to the instrument record itself. The first is the name of the instrument; this is a Pascal string, so you can skip this field by adding the one-byte length of the string plus one to the pointer. The next field is a sample number; this is a bookkeeping field that helps keep track of waves in 'WAVE' chunks that have several sub-waves imbedded in a single wave packet. The field is two bytes long. We don't need it for anything, so just add two to the pointer.

At this point, you have a pointer to the start of the instrument record. Since we're only copying one wave into the sound RAM at a time, the pointer is all you need. Save the pointer for later, when we actually play some notes.

Some Practical Comments About Instrument Files

Now that you've seen the file format for an instrument, you can see that there are two distinct parts for an instrument definition. The 'WAVE' chunk is literally a digital recording of the instrument, playing a note at a constant volume. When you download an instrument file, this is the important part that you're after – this recording of the instrument is something you can't fake easily.

The other part of the instrument definition is all of the entries in the instrument record itself. If you think about it for a moment, instruments don't really play a single note at a constant volume. The process of converting a sound into another note is something the Note Synthesizer does pretty well, but there is another issue we have to be concerned with, and it's one that the Note Synthesizer can't face, and that the folks who upload instruments don't worry much about. That's the issue of how the volume of a note changes over time.

Let's start looking at this issue with a couple of examples. Let's say you hit a tight drum head with a drumstick. The sound you hear starts very quickly, and stops almost as quickly as it starts. Now compare that to a gentle note on a violin. The note starts slowly, builds up to a maximum volume, then fades away slowly, too. The point is that the shape of the sound wave as a function of volume verses time is very different for these two instruments. Other instruments are different, too. Plucked string and percussion instruments, as a general rule, build up volume very quickly, then the volume drops off. In some instruments, like a guitar or loose base drum, the sound may hang in there for quite a while. Wind instruments and stroked string instruments vary a lot, but as a group, they hold their maximum volume longer than the plucked string instruments or percussion instruments.

The Note Synthesizer has a pretty good way of handling this situation, as well as another we'll talk about in a moment. You literally define the wave form as volume verses time. The problem is that all of the instrument files I found used a constant wave envelope that built up and dropped off very quickly – so quickly that even the percussion instruments were hard to recognize. As a result, you can't just load a sound file and play it if you expect to get good results. You also have to modify the instrument definition.

In the next two sections, I'm going to do two things. First, I'm going to tell you how the `envelope`, `releaseSegment`, `vibratoDepth` and `vibratoSpeed` fields of the instrument file work, so you can create your own sound profiles for your instruments. Next, I'm going to give you a canned set of parameters that will make most of the instruments on the disks that comes with this course sound pretty good. If you're interested in getting the program to work, but don't care about the details, skip the next section.

Changing the Sound Quality for an Instrument

The major task in creating an instrument profile is to tell the Note Synthesizer how the volume should vary over a period of time. The `envelope` parameter in the instrument record is basically just a plot of volume verses time, coded in a rather obscure way. In the instrument record, you see a 24 byte array, but in fact, the envelope is really a series of eight three byte fields. In each case, the first byte is a volume, and the next two bytes are an integer telling the Note Synthesizer how fast it should change from the old volume to the new one. Let's work through an example to see how these fields are used.

When the note starts, it starts at a volume level of zero. The highest volume for the note is 127, and for the most part, we want to jump up to that peak volume fairly quickly. Let's say we want to get there in 0.01 seconds. To figure out the second parameter, we use this formula:

```
   rise := round(deltaVolume*0.128/time);
```

where `deltaVolume` is the difference between the original volume and the new one (127 in this case) and `time` is the amount of time it should take to make the change, in seconds (0.01). If this is what we want to do for the first part of the note envelope, we would set the first byte of envelope to 127, and the next word to `rise`, like this:

```
   envelope[1] := 127;
   envelope[2] := rise & $00FF;
   envelope[3] := rise >> 8;
```

This rise time parameter works in either direction. The Note Synthesizer looks at the two volumes to figure out if it should move up or down; you always put the rise time in as a positive integer.

The units for time are pretty obvious, but the units for volume are something you may not be as familiar with as the units for time. Volume is measured in decibels, which is a logarithmic scale. Changing the volume by 16 is equivalent to changing the sound level by 6 decibels. Increasing the volume parameter by 37 roughly doubles the actual sound, while cutting the volume parameter by 37 cuts the volume in half.

You may not need all eight parts of the volume profile. Later on, in fact, we'll only use four entries for our canned waveform. If you don't need all of the entries, set the extra ones to zero. The last entry should always have a volume of zero, too.

Obviously I'm skipping over some background here, like where the equation comes from, what a decibel really is, and so forth. Most of that theory is covered in the toolbox reference manuals. (The definition for a decibel isn't – one decibel is a ratio between two sound intensities such that `20*ln(volume1/volume2)` is one, if you're curious.) It's also not important when it comes to designing the instrument profile. What you really need to do is sketch the waveform out as volume verses time, then use this equation to generate the numbers for the waveform table. As you create the table, keep in mind that changing the volume parameter by one unit has a much bigger effect when the volume is high than when the volume is low. There's a lot of art in defining the instrument profile, and you're going to have to play with it a bit to get one right. The details I'm skipping over wouldn't help design the instrument envelope any better.

For a lot of instruments, you can start to play, then hold a note for quite a long time. That's something the Note Synthesizer can do, too. Once you start a note, you can either wait until it fades out, or you can stop the note early. If you stop the note early, the Note Synthesizer skips from wherever it is in the note sequence to the release segment, and picks up from there. The `releaseSegment` parameter in the instrument record is an index into the envelope array, from 0 to 7. Generally you point it to the last entry you used – the one that finally decays to a volume of 0.

Stop ant try to imagine what it really sounds like as a violin, trumpet, or even a human voice tries to hold a constant note. It usually isn't quite constant, and isn't supposed to be. Instead, it wobbles back and forth. That's called vibrato; it's a little volume shift that's in almost every sound. You can use the `vibratoDepth` and `vibratoSpeed` fields to create vibrato in an instrument. The `vibratoDepth` field varies from 0 to 127; it's the amount of change you're introducing. The `vibratoSpeed` field is the speed of the change, which ranges from 0 to 255. The units aren't outlined exactly in the toolbox reference manuals, but it wouldn't help much even if they were. In practice, you have to play with these numbers until you get something that sounds good.

Unlike most of the other sound effects, vibrato is implemented in software. If you use anything but 0 for `vibratoDepth`, you will have some effect on the speed of the program you're running while the sound is played. If you're writing a game, you might want to be sure `vibratoDepth` is set to zero, even though vibrato adds a lot to the quality of the sound.

Canned Parameters

In the last section, I went into enough of the details about how the instrument record is used for you to customize the volume envelope for an instrument. That's something you need to do for all of the instruments I've found on the online services, since the authors seemed to spend more time collecting good sound samples than developing a proper volume envelope. There is no single proper volume envelope, but here's one that will get you close enough to recognize most of the instruments on the solutions disk. (They're in the Instruments folder, by the way.) This sample code fragment assumes you are working from a pointer to the instrument record, which is what you'll have if you follow the outline for loading the files from a few pages back.

```
with instrumentPtr^ do begin
   envelope[1] := 127;
   envelope[2] := $80;
   envelope[3] := $3F;
   envelope[4] := 100;
   envelope[5] := $B3;
   envelope[6] := $00;
   envelope[7] := 90;
   envelope[8] := $06;
   envelope[9] := $00;
   envelope[10] := 0;
   envelope[11] := $26;
   envelope[12] := $01;
   releaseSegment := 3;
   vibratoDepth := 80;
   vibratoSpeed := 40;
   end; {with}
```

Listing 16-3: A General Purpose Volume Envelope

Using the Note Synthesizer

With a little work, you can turn the information in the last few sections into a program that can load an ASIF sound file and install the first instrument from that file. I'll let you put that information to use in a problem in a moment, but first we need to cover what to *do* with the instrument! Fortunately, loading and installing the instrument is the hard part.

The Note Synthesizer has a number of calls that let you play notes. The first one you should call, though, is `AllNotesOff`. It doesn't take any parameters. `AllNotesOff` will turn off any notes that were already playing; there shouldn't be any, but this makes sure. You should call `AllNotesOff` again before you quit.

Before you can play a note, you need to allocate a generator. That's done with `AllocGen`, like this:

```
generator := AllocGen(100);
```

Earlier in this lesson, I mentioned that the Ensoniq sound chip could play more than one sound at a time. The Note Sequencer makes use of this fact to let you play more than one note at a time. If there is more than one instrument in the sound RAM, you can even play notes from different instruments; that's something that is very important for playing real songs. The `AllocGen` call is the mechanism that lets all of this happen smoothly. Before playing a note, you ask for one of the available generators, and get back a number from 0 to 13. That's your generator number, and you use it for the rest of the calls associated with playing the note.

Of course, it's possible for more than 14 notes to be playing at any one time. In that case, only 14 can be played, so you need some way of telling the Note Synthesizer which notes are most important. The parameter to `AllocGen` is a priority; if it's higher than the priority for some note that's already playing, the Note Synthesizer will stop that note and give you it's generator. If all of the notes that are playing have a higher priority than the new note, `AllocGen` returns an error, and you should skip playing the note.

Once you have a generator, the next step is to start playing a note. That's done with the `NoteOn` call:

```
NoteOn(generator, note, volume, instrument);
```

`generator` is the generator number returned by `AllocGen`. `note` is a note number. Note numbers are assigned sequentially, starting with 1, which is a low note, and ranging up to 127, which is a high frequency note. Middle C is 60. There are 12 notes (7 major notes and 5 sharps) per octave, so 72 and 48 are also C notes. `volume` is the volume for the note, which can vary from note to note; in general, you can also think of this as the force or speed for the note, like how hard a piano key is struck. This parameter can also range from 1 to 127. A good starting value is 64. The final parameter is a pointer to the instrument record; that's the value we spent so much time getting ready when the instrument file was loaded.

The next step is a little touchy; you need to wait for a while. The problem is that the right amount of time to wait is different for different instruments. What you're waiting for is the proper time for the note to start fading away. For a piano, that's right away – the note starts to fade as soon as the piano wire is struck. For a flute, the note doesn't start to fade until the flutist stops playing the note.

```
NoteOff(generator, note);
```

Once the note fades away to nothingness, the Note Synthesizer shuts off the generator and makes it available for other notes. If the user stops the program before a note is quite finished, the `AllNotesOff` call I mentioned at the start of the section will shut things down properly.

Problem 16-3: This problem puts the Note Synthesizer to work, creating a simple test bed program that let's you try out ASIF instruments. There are two main parts to the program.

The first part is the user interface. Start with the Frame program, setting it up so there is always one open window. Use the 640 mode version of frame, and set the window size to 377 by 50 pixels. This is divided into 21 equal piano keys with vertical black lines, with sharp keys formed with black rectangles, as in the picture. You're main event loop should check for mouse presses on the key; set it up so it starts the note when the mouse is pressed, and calls `NoteOff` when the mouse is released. While you're waiting for the mouse up event, check to see if the mouse moves to another key; if it does, turn off the current note and start the new one. That let's the user "strum" the keys.

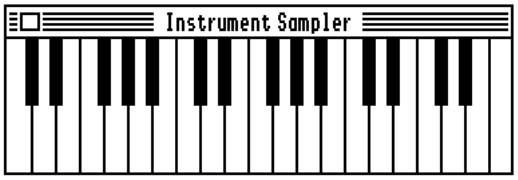


Figure 16-8: The Keyboard

Start with a volume value of 120, and set the keyboard up so the leftmost note has a note number of 36. You can change those values if you like, but they're good starting places.

Your program should include an Open call in the File menu, but New, Close, Save and Save As should all be missing. Naturally, you can't play a note until an instrument is loaded, so if the user plays a note right away, display an appropriate dialog.

The second main part of the program is the part that loads an instrument. We covered loading the file in the text of the session. Besides implementing those mechanics, you also need to hook up an SFO Open dialog that only allows the user to open files with a file type of $D8 and auxiliary file type of $0002. Once the ASIF file has been loaded, patch the instrument file with the code from Listing 16-3.

Your program will be using calls from both the Sound Tool Set and the Note Sequencer, so be sure both tools are started in your `StartUpTools` call.

Getting the Most from Instruments

At first blush, you might think that all of the talk about the Note Synthesizer is only useful if you want to write a music program. Wrong!

Try your instrument sampler on some of the instruments from the solutions disk. Play a single note from the instrument, trying them at various notes. I'd particularly recommend listening carefully to the instruments Again, Parrot, Phasor and Pistola – none of which are an instrument in the traditional, musical sense. With the proper instruments and a little imagination, you can use the Note Synthesizer for almost all of the sound effects in a typical game or education program.

The Sound Tools

Ultimately, all sounds produced by the computer end up as vibrations of a speaker cone, but there are a lot of ways to get that final result. Depending on what you are trying to do, you have a wide variety of sound tools available, too. And, of course, there's the whole topic of getting sounds into the computer in the first place!

You've seen a little of what the Apple IIGS can do with the Sound Tool Set and the Note Synthesizer. In the rest of this section, we'll take a look at some of the other sound related tools. Some of these are true tool sets, either built into the Apple IIGS or in the Tools folder of your system disk, but I'm not going to limit the information to just the things that come with the system. After all, that wouldn't give you any way to get sound into the computer!

The Audio Compression and Expansion Tool Set

In most cases, the biggest drawback to digitally recorded sound is the sheer size of the sound files. The Audio Compression and Expansion Tool Set (ACE) is used to compress some of those huge chunks of data. ACE uses a clever technique of compression that doesn't always give you back exactly what you put in. That sounds like it would cause problems (sorry about the pun!) but in practice the results are very impressive. ACE can compress a sound to either 3/8 or 5/8 of it's original size, then expand the sound back out to something that is close enough to the original sound that you generally can't hear the difference.

ACE is a RAM based tool. It's on your System Disk (Tool029 in the Tools folder). ACE is documented in Chapter 27 of *Apple IIGS Toolbox Reference: Volume 3*.

The Note Sequencer

We used the Note Synthesizer to play individual notes using a variety of instruments. The Note Synthesizer is designed to string those notes together into multi-instrument songs. Once you get a song going, it can even play the song in the background while the main CPU of the Apple IIGS keeps right on working. That's real handy in a game, for example. You could even create a desk accessory to play background music while you use other programs.

The Note Sequencer is a RAM based tool. It's on your System Disk (Tool026 in the Tools folder). The Note Sequencer is documented in Chapter 40 of *Apple IIGS Toolbox Reference: Volume 3*.

The MIDI Tool Set

If you walk into a store that sells musical instruments mumbling "MIDI," they will know exactly what you're talking about. The Musical Instrument Digital Interface (MIDI) is a common hardware and software interface that let's musical devices talk to each other. You'll need a MIDI card and one or more MIDI compatible musical instruments to get going, but with those installed, you can turn your Apple IIGS into an awesome music box. For a very simple example, you could record what you play on a MIDI keyboard (most MIDI keyboards look more or less like a piano or organ keyboard), edit the results in the computer to clean up timing, and play the edited song back through the MIDI device.

The MIDI Tool Set was created to make it easier for you to deal with all of the different MIDI instruments and cards. It gives you a standard interface with a well defined set of tool calls that let you talk to virtually any MIDI instrument.

The MIDI Tool Set is a RAM based tool. It's on your System Disk (Tool032 in the Tools folder). The MIDI Tool Set is documented in Chapter 38 of *Apple IIGS Toolbox Reference: Volume 3*. Several commercially available music programs can use the MIDI tools to talk to MIDI instruments, too.

synthLAB

synthLAB is actually an application, created by Apple Computer and distributed with System 6.0. synthLAB is designed as a MIDI editing tool, but it's also a great way to play with the instrument and music capabilities of the Apple IIGS.

synthLAB is a self contained package, with documentation right on the disk. Just catalog the synthLAB disk from System 6.0 and start exploring!

The Media Control Tool Set

The Media Control Tool Set does for stereo systems what the MIDI Tool Set does for MIDI instruments: it gives you a single, well defined tool that can talk to virtually any modern stereo component. You can control CDs, Laser Discs and VCRs, all with a single tool. In theory, you can turn your Apple IIGS into a rather powerful computerized video or sound DJ.

The Media Control Tool Set is new with System Disk 6.0. It is documented in Chapter 11 of *Programmer's Reference for System 6.0*.

Talking Tools

When we looked at the Note Synthesizer, you saw how the computer could take an instrument file and a computerized version of sheet music and play a song. The computer can also take some computerized knowledge of the English language and straight ASCII text and create sound from the text. The Talking Tools package does exactly that.

In practice, the rules for speaking are a lot more complicated than the rules for playing an instrument. You would never mistake the computer for a real person. The voice is easy to understand, though, and you do have some control over it – like picking a male or female voice, controlling the speed, and controlling the volume.

There are two big reasons for picking text to speech conversion over a digitally recorded voice. The first, of course, is space. You can easily store an entire novel on an 800K floppy disk using ASCII text, then read the novel with Talking Tools. There's no way you could get a digital recording of an entire book on a floppy disk, though.

The other reason for picking text to speech conversion is flexibility. With a digitally recorded voice, you're stuck with a specific sequence of words. With text to speech conversion, you can have the computer say anything you like – even, for example, asking for the name of the user and fitting it into future spoken sentences.

The Talking Tools package comes with the tools themselves, demonstration programs, documentation for the tools, and interfaces for a variety of languages, including ORCA/Pascal. It is published by the Byte Works.

Sound Input

The Apple IIGS doesn't have a built-in microphone or any other sound input device, so if you want to create digital recordings of your own, you'll need some sort of sound input device. There are a lot of them out there, ranging in price from about $30 to a couple of hundred dollars. The cheaper devices plug right into your motherboard, and usually include a small card. These are pretty good for playing around, but electrical interference from the computer itself limits the quality of the sound. The more expensive cards can approach CD quality sound.

All of the various cards have standard jacks, so you can plug a lot of different things into the card. Of course, the obvious choice is a microphone, but you can also connect the computer to a VCR or stereo to record songs or sound effects. Some of the cards also have stereo output – the built-in output on the Apple IIGS can only play one channel, even thought the sound chip can easily handle stereo.

One of the inexpensive sound input systems comes bundled with HyperStudio, published by Roger Wagner Publishing. For the more expensive systems, get a recent catalog from a mail order house, or check out the cards offered by Applied Engineering.

Summary

This lesson has reviewed the major tools, hardware, and software used to make sounds on the Apple IIGS. You learned to use some of the features of two of these tools: the Sound Tool Set for playing digitally recorded sound, and the Note Sequencer for creating musical notes using instruments. You also learned to load instruments from ASIF instrument files.

Tool calls used for the first time in this lesson:

```
AllNotesOff        AllocGen           FFSoundDoneStatus FFStartSound
FFStopSound        NoteOff            NoteOn            WriteRamBlock
```

Lesson 17 – Professional Polish

Goals for This Lesson

At this point, you can crank out a pretty complete desktop program, but it still doesn't have that polished look of a professional program. One of the big differences between your programs and the professional programs is that the professional programs interact with the operating system through icons and messages. This lesson shows you how to make your own programs work smoothly with the Apple IIGS operating system, and especially with the Finder. Topics covered include icons, Finder information, and using the message center.

Finder Resources

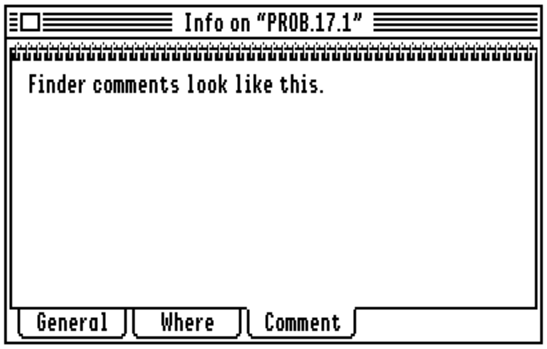
The `rComment(1)` Resource

When you're in the Finder, you can ask for information about a program. When the Finder is asked for information, it looks in the resource fork of the file for an `rComment` resource with a resource ID of 1. If it finds one, it displays the text in the resource. This comment resource is generally reserved for the user, who can write comments about the program or data files, but you can also use the `rComment` resource to imbed information you think the user might need, like the amount of memory and minimum hardware or software your program needs. There's another place for you to record copyright information, so you normally won't put that here.

Listing 17-1 shows a sample resource for an `rComment` resource. Figure 17-1 shows the information as it's displayed by the Finder when you use the Icon Info command under the Finder's Special menu.

```
resource rComment (1) {
   "Finder comments look like this."
   };
```

Listing 17-1: `rComment` Resource for a Typical Program



Finder comments look like this.

Figure 17-1: The Finder's Comment Box

The `rComment(2)` Resource

Another resource the Finder recognizes is an `rComment` resource with a resource ID of 2. The Finder shows the text from this resource whenever the user clicks on the icon for a file that can't be opened. That might happen if your program is used to create a data file, but someone is trying to open the data file on another computer. This resource is a good place to put some information about your program, like the name and where to get a copy! To do that, though, you need to create a resource under program control. Creating a resource fork and adding a resource is a lot like writing data to a file, but there are some differences, especially in the particular calls used.

The first step is to open the resource fork on the data file. `OpenResourceFile` does that; here's a sample call:

```
id := OpenResourceFile(2, nil, fileName);
```

The first parameter tells the Resource Manager whether you want have read access (use a value of 1) write access (use a value of 2) or both (use a value of 3). In this case, we're supposedly creating a new resource fork and adding a new resource, so all we want is write access. The next parameter is a pointer to the resource map for the file; this is almost always nil, especially for a new resource file. The last parameter is a GS/OS path name (the parameter type is `gsosInString`), just like the one you use to open the data fork for the file. `OpenResourceFile` returns a file ID for the resource fork and makes the resource fork the current one. In practical terms, this means you could cause problems by doing anything with resources before the file is closed, so keep this section of code tight! You will need the file ID to close the resource fork.

The next step is to add the new resource. That's done with an `AddResource` call:

```
AddResource(dataHandle, $0300, $802A, 2);
```

This call adds the resource to the current resource file. `OpenResourceFile` made your new resource file the current resource file, so the resource will get added to the resource fork of your data file.

The `dataHandle` parameter is a handle containing the information you actually want to put in the resource. The handle needs to be the right size, so you need to allocate the handle specifically for this resource. Once you make this call, the memory belongs to the Resource Manager, and will be disposed of at the proper time by the Resource manager. In short, after this call, forget the handle exists! The information you stuff in this particular resource should be straight ASCII text.

The $0300 parameter is a flags word with attributes for the resource. The ones I'm giving here work fine for our purpose, but if you want to do anything else with `AddResource`, be sure to read the complete description of the attributes.

The next parameter is the type number for an `rComment` resource. It would be good style to create a constant called `rComment`, and use `rComment` as the parameter, in any real program.

The last parameter is the resource ID. We're defining an `rComment(2)` resource, so this parameter should be 2.

Finally, you need to close the resource file.

```
CloseResourceFile(id);
```

The `id` parameter is the same one returned by `OpenResourceFile`. By now, I'm assuming you're tuned in to details of error handling, and wouldn't even thing of making the `AddResource` or `CloseResourceFile` calls unless the `OpenResourceFile` call succeeded.

One of the side effects of the `CloseResourceFile` call is that it will change the current resource fork back to its setting before the `OpenResourceFile` call. (This is a simplification, but it's true the way we've used the calls here.) That means that after this call is made, any resource file activity will point back to your program's resource fork, which is where it should be.

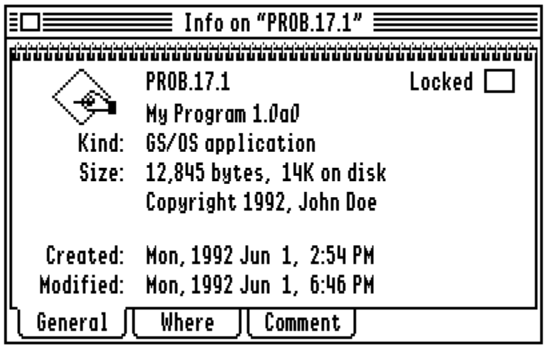
The `rVersion` Resource

The `rComment` resource is optional, but your program must have an `rVersion` resource and an `rBundle` resource to work properly with the Finder. These two resources are used by the Finder to identify your program and decide how some advanced features should be handled.

```
resource rVersion(1) {
   {
      1,                                   /* Major revision */
      0,                                   /* Minor revision */
      0,                                   /* Bug version */
      final,                               /* Release stage */
      0,                                   /* Non-final release # */
      },
   verUS,                                  /* Region code */
   "My Program",                           /* Short version number */
   "Copyright 1992, John Doe",             /* Long version number */
   };
```

Listing 17-2: A Sample `rVersion` Resource

As you can see, the `rVersion` resource contains version information. The Finder shows the user some of this information when the user uses the Icon Info command.


Info on "PROB.17.1"

PROB.17.1 Locked
My Program 1.0a0
Kind: GS/OS application
Size: 12,845 bytes, 14K on disk
Copyright 1992, John Doe

Created: Mon, 1992 Jun 1, 2:54 PM
Modified: Mon, 1992 Jun 1, 6:46 PM

Figure 17-2: What the Finder Shows

The Finder also uses this resource to keep track of some icon information. For example, if you change the icons and release a new version of your program, and keep track of the version numbers properly, the Finder automatically updates some internal files containing your icons.

As for the resource itself, the first number is the major release number. From the time you start working on your program until the first release of the program, this should be 1. Technically, it's up to you when you change this to a 2. Personally, I only change the major revision number when a program changes so much that it writes data in a way that older versions cannot handle, or when I feel like I have to release a whole new book to describe all of the changes. A lot of other people are pretty free with major version numbers, though, changing them for almost any new release.

The second number is the minor release number. This should be 0 up until the first release. I generally change this number whenever the program changes in any way the user can see. For example, when I added support for CDevs in ORCA/Pascal, the version number went from 1.3 to 1.4.

The next number is the bug fix level. This number should start at 0 and be increased for any release that changes anything, unless the version number or minor version number is changed. For example, ORCA/Pascal 1.4 had some bug fixes, but since I was changing the version number from 1.3, this bug fix digit stayed at 0. (The full version number for 1.4 would be 1.4.0, but I don't usually type the bug fix digit if it is zero.) Later, when some bugs were found in the 1.4 version, I released a new version of the compiler with a version number of 1.4.1.

The next field tells the Finder what release stage you're at. I'll go over the various legal values, and give you one interpretation of how to use them.

`development`	The development stage is the point when you're writing the program.
`alpha`	The alpha stage starts when the program is far enough along in the development cycle to be useful. By this point, all of the features you planned to put in should be there, and most or all of the bugs should be gone. In a large company, a good way to think about alpha is that it's when the programmer thinks he's done. This is when testers and upper management starts getting involved in a big way. (Some testers are generally involved from the very beginning, but this is the point when the testing starts getting serious. On a large project, this is when more testers are added.)
`beta`	Beta is the stage when the managers agree with the programmer. By this time, some changes have been made based on the comments from people inside the company, and a few trusted outside testers have also tried the program and made some comments. The beta release is generally sent to quite a few testers outside the company. This is when the program is put through its paces by some carefully selected "real users." There should be no known bugs by the time this stage is released. The documentation should be finished. If the development team has done it's job well, there won't be many bugs found, and the beta testers won't ask for many changes to the program.
`final` or `release`	Either of these values are used for the final, commercial release of the program.

Of course, you might not need all of these stages. If you're writing a freeware game, for example, you might use the development stage until you finish, then bump the designation to final when you post the program online.

During the development cycle, the main version number should be 1.0.0. The field after the release stage is used to keep track of the release number for a particular stage. It goes up by one each time you release a new version of the program at the same stage, and is set back to zero when the stage advances. This field gets set back to 0 permanently when the program is released.

The region code tells the Finder which country or area the program is intended for. This helps keep track of special versions that are used with special character sets, languages, or keyboards. Here are the values that are available as of System Disk 6.0:

`verUS`	`verFrance`	`verBritain`	`verGermany`
`verItaly`	`verNetherlands`	`verBelgiumLux`	`verFrBelgiumLux`
`verSweden`	`verSpain`	`verDenmark`	`verPortugal`
`verFrCanada`	`verNorway`	`verIsrael`	`verJapan verAustralia`
`verArabia`	`verArabic`	`verFinland`	`verFrSwiss`
`verGrSwiss`	`verGreece`	`verIceland`	`verMalta verCyprus`
`verTurkey`	`verYugoslavia`	`verYugoCroatian`	`verIndia`
`verIndiaHindi`	`verPakistan`	`verLithuania`	`verPoland`
`verHungary`	`verEstonia`	`verLatvia`	`verLapland`
`verFaeroeIsl`	`verIran`		`verRussia`
`verIreland`	`verKorea`	`verChina`	`verTaiwan`
`verThailand`			

Frankly, I doubt if some of those places even have an Apple IIGS, but you never know!

The last two fields are text strings. You can put anything you want there, or even leave them blank. The Finder itself uses these fields for the name of the program and the copyright for the program.

The `rBundle` Resource

The Finder also looks for an `rBundle` resource in your application's resource fork. The `rBundle` resource serves two different purposes. First, it tells the Finder what icon to show then the user sees the application itself in a folder or on the desktop. Second, it tells the Finder what kind of application files the program can handle, and exactly how they are handled.

When the Finder runs you application for the first time, it looks for this resource. Assuming it finds one, it collects the `rBundle` resource and puts it in a local data table. This local data table is in a file, so the Finder doesn't normal ever look at the `rBundle` resource again – it uses it's own, internal copy. That's why the `rVersion` resource is so important: it tells the Finder that the application has changed, so the Finder will reread your `rBundle` resource.

Once the Finder has read the `rBundle` resource, it uses the icon you give it when it displays your program in a folder or on the desktop.

The other thing that's in the `rBundle` resource is one or more structures called `oneDoc` structures. Each `oneDoc` structure tells the Finder about one kind of data file your program can handle. You can get pretty detailed here, right down to telling the Finder that you can handle a specific file type, auxiliary file type, file name, a specific creation date and so forth. You can also tell the Finder whether your program created the document, or is just able to handle it. Finally, you can give the Finder an icon to use when the document is displayed on the desktop. The Finder uses all of this information for two things. The first, obviously, is when it shows the file on the desktop, and uses the icon you give it. The other use is when the user double-clicks on a document. When that happens, the Finder scans all of the `oneDoc` structures it knows about, trying to find some application that can deal with the document. If it finds one, it runs the application and passes a message. (We'll see how your program can use that message later in the lesson.) If there aren't any applications that can deal with the document, the Finder lets the user look for one.

Listing 17-3 shows a sample `rBundle` resource. We'll spend the rest of this section going over this sample in detail, talking about the various options and how the information is actually used by the Finder.

```
resource rBundle (1, preload, nospecialmemory) {
   1001,                        /* rIcon ID for application or NIL */
   1,                           /* rBundle ID of this rBundle */
   {                            /* first oneDoc structure */
      {
         $0081,                 /* bit 0 = launch this = boolean */
                                /* bits 4-7 = voting clout */
         {0},                   /* rFinderPath ID for this document */
         {1002},                /* rIcon ID for large icon for document */
         {1003},                /* rIcon ID for small icon for document */
         "",                /* string to describe this type of document */
         },
      $00000003,                /* match field bits*/
      MatchFileType {{$D8}},    /* file type to match */
      MatchAuxType {            /* auxiliary file type to match */
         {$FFFFFFFF,$00000002}
         },
      empty {},
      empty {},
      empty {},
      empty {},
      empty {},
      empty {},
      empty {},
      empty {},
      empty {},
      empty {}
      }
   {                           /* second oneDoc structure */
      {
         $0021                 /* bit 0 = launch this = boolean */
                               /* bits 4-7 = voting clout */
         {0},                  /* rFinderPath ID for this document */
         {1004},               /* rIcon ID for large icon for document */
         {1005},               /* rIcon ID for small icon for document */
         "",               /* string to describe this type of document */
         },
      $00000001,               /* match field bits*/
      MatchFileType {{$D6}},   /* file type to match */
      empty {},
      empty {},
      empty {},
      empty {},
      empty {},
      empty {},
      empty {},
      empty {},
      empty {},
      empty {},
      empty {}
      }
   };
```

Listing 17-3: A Sample `rBundle` Resource

There's a lot to talk about in this resource, so we'll go over everything through the first `oneDoc` structure line by line.

```
1001,                                /* rIcon ID for application or NIL */
```

This is the resource ID for an `rIcon` resource. The Finder will use this icon when it shows your application. You can use a 0 instead of a resource ID for an icon; in that case, the Finder uses the default application icon.

There's nothing special about the `rIcon` resource itself. It's defined and used just like the icon in an icon button control. The only thing to keep in mind is that the Finder can align icons on a grid, and if you make your program's icon too big, it can hang off of the edge of the window or overlap with other icons. To avoid that, make sure the icon is no more than 22 pixels high. The width isn't usually a problem.

As I write this course, there aren't any icon editors around that are designed specifically for creating icons for `rBundle` resources. On the other hand, you can always create an icon button with a program like Design Master, have it write out the result as a Rez source file, and use just the `rIcon` part for your program's icon.

(As of Finder 6.0, this field is always set to nil, and the icon is ignored.)

```
1,                                   /* rBundle ID of this rBundle */
```

This is the resource id for this resource. The Finder uses this field for its own purposes. You should always set it to 1.

```
 {                                      /* first oneDoc structure */
```

As I said earlier, the `oneDoc` structures tell the Finder what sort of document files your program can deal with. You can have as many of these as you want, and may need more than one. For example, if you are adding a `oneDoc` structure to your text editor, it's pretty obvious that it can handle your own file type, with a minor change or two, it can handle standard text files,and source code files. To tell the Finder about both possibilities, you need two `oneDoc` structures. To create two `oneDoc` structures, you just pop in another one right after the first, just like you see in the sample.

```
    {
       $0081                             /* bit 0 = launch this = boolean */
                                         /* bits 4-7 = voting clout */
```

The first entry in the `oneDoc` structure is a flag word. The least significant bit is 1 if the Finder should launch your program when the user double-clicks on the document, and 0 if you are just creating an icon. For example, you might want to create an icon for a preferences file, but not actually launch the program if the user picks the file. In that case, the least significant bit should be 0.

Bits 4-7 are the voting clout for the program. This is where you tell the Finder what you can do with the file.

bit	use
7	This is the highest priority of all. If you set bit 7, you are telling the Finder that you are the owner of the file type. Normally you would only set this bit if you reserve a specific file type for the files for your program. For example, AppleWorks GS should lay claim to all of the AppleWorks GS specific file types.
6	This bit says your program knows exactly what to do with the file, and can read and write the file. When the user double-clicks on a document, the Finder will try to run the program that claims ownership with bit 7, but failing that, the Finder will try to find some program that sets bit 6, claiming that it can read and write the file. For example, if you updated your word processor to handle text files, it should set this bit for text files and source files.
5	This bit says you can read the file, and may be able to write it – but not necessarily using the same format. The Finder will pick you in a pinch.
4	Setting this bit says you can read the file, but that's about it. If the Finder is really desperate, and can't find anyone else to deal with the file, it will use your program. You might use this bit, for example, with your instrument player from the last lesson, telling the Finder you can play the file. If a music editor is around, though, it will have a higher priority, since it can read and write the file, and the Finder will pick it first.

(Voting isn't actually implemented in Finder 6.0, but the voting bits should be set anyway, since voting may be implemented at some point in the future.)

```
       {0},                        /* rFinderPath ID for this document */
```

The Finder uses this field to keep track of your program once the `rBundle` resource is moved to the Finder's internal table. You should set it to 0 in your resource definition.

```
       {1002},              /* rIcon ID for large icon for document */
```

```
        {{1003}},          /* rIcon ID for small icon for document */
```

You can define a resource for the document here, giving the Finder two `rIcon` resource numbers. The first is used when the document is shown using a full-size icon, and the second is used when the Finder is displaying small icons. You can also use 0 for either or both entries, telling the Finder to use the default icon.

Normally you will only set up an icon if you own the file, letting the real owner define the icon if you have a voting priority less than 7. If you do define the icons, the large icon should generally be 22 pixels high or less, while the small icon should be 8 pixels high and 16 to 80 pixels wide.

```
        "",               /* string to describe this type of document */
```

The Finder has a file it can read to find a short text description for various file types; the result is displayed in the Kind entry when you ask for information about the icon. If you can give a better description than the one the Finder will normally use, put it here. If you can't use a null string, like you see in this example.

```
    $00000003,                          /* match field bits*/
```

So far, all of the fields in the `oneDoc` structure have been used to tell the Finder what to do with a particular file. The rest of the `oneDoc` structure tells the Finder what file we're talking about.

There are a lot of different ways you can tell the Finder what files you are referring to, and I'm not going to go over all of them here. After all, it's not likely that most of you are going to try to match a particular creation date right away. Instead, I'll show you how to use the two most common file selectors, then I'll list the others that are available as of Finder 6.0. If you want to use one of the other file selectors you can find all of the technical information you need in *Programmer's Reference for System 6.0*.

The first of the file matching fields is the flag long word you see above. Each of the bits is used to tell the Finder that you will use one particular kind of file selector. In this case, bits 0 and 1 are set. Bit 0 is set when you are telling the Finder to pick a particular file type, and bit 1 is set to tell the Finder you will also be selecting the file based on the auxiliary file type. The Finder will only consider a file a match to this `oneDoc` structure if both the file type and auxiliary file type match.

Of course, all you've done here is to tell the Finder what to expect next. The actual values to use come right after this flag long word.

```
    MatchFileType {{$D8}},         /* file type to match */
```

This field tells the Finder which file type the current `oneDoc` record refers to. You don't actually have to select the file by the file type, but in practice the file type is almost always used.

```
    MatchAuxType {                  /* auxiliary file type to match */
       {$FFFFFFFF,$00000002}
      },
```

This field matches an auxiliary file type, but it does it in sort of an odd way. You might want to handle some, but not all, auxiliary file types, so the Finder lets you mask out certain bits. When the Finder is trying to decide if this `oneDoc` structure refers to a file that is being opened, it starts by anding the file's auxiliary file type with the first value in this entry. In this case, the mask has all of the bits set, so the result is just the original value. The Finder compares the result to the second entry; if they match, the Finder passes the file. Assuming all of the other tests pass, too, the Finder applies this `oneDoc` structure to the file.

```
        empty {}
```

These lines tell the resource compiler that you aren't defining the rest of the tests. You do have to have the right number of entries, so for each test you leave out, there must be one line like this one.

Here's a short summary of the other tests that are available in Finder 6.0. Again, for details, see *Programmer's Reference for System 6.0*.

`matchFilename`	Match a particular file name. You can use wild cards.
`matchCreateDateTime`	Match a particular creation date. This doesn't have to be an exact match; you can match files that are more recent than a certain date, or files that are older than a certain date.
`matchModDateTime`	Match a particular modification date. Again, you can check for files that are more recent than a certain date, or files that are older than a certain date.
`matchLocalAccess`	Match files with specific access bits set. For example, you could match only files that need to be backed up.
`matchNetworkAccess`	Files on a network can have different access bits set. This field lets you match the access bits for the network, as opposed to the local access bits.
`matchExtended`	Match files that have a resource fork.
`matchHFSFileType`	Match files based on the HFS file type.
`matchHFSCreator`	Match files based on the HFS creator. (This is sort of like the ProDOS FST's auxiliary file type.)
`matchOptionList`	Match based on fields in the option list for a file.
`matchEOF`	Match files of a certain size. As with dates, you can do comparisons. For example, you can pick all files that are longer than some value, or all files that are shorter than some value.

The Desktop File

As of System 6.0, when the Finder runs a program with an `rBundle` and `rVersion` resource, it actually saves some information about the program in a file named DeskTop, located in a folder called Icons on the same disk as the application. As you develop, delete, and change applications, this file will eventually fill up with old information, not all of which you need. Especially on your development disks or partitions, you should delete this folder from time to time to free up all of the old stuff. (Hopefully someday some utilities will be available to clean up this file – that's a hint!) You might also want to delete this file if you've made some changes that just don't seem to be registering when you use your program.

Deleting the DeskTop file wipes out all of the things the Finder knows about programs on the disk or partition, so you will have to run all applications again before the Finder will realize they exist and save the appropriate information in a new desktop file.

Problem 17-1: Add an `rBundle` and `rVersion` resource to the instrument sampler you created in Problem 16-3. Tell the Finder your program can handle files with a file type of $D8 and an auxiliary file type of $0002, but set your voting status to 4.

Use the following icons. Icon 1001 is for your program, icon 1002 is the large icon for file type $D8, and icon 1003 is the small icon for file type $D8.

```
 resource rIcon (1001, preload, nospecialmemory) {
    $8000,                      /* Icon Type bit 15  1 = color, 0 = mono */
    21,                                 /* height of icon in pixels */
    24,                                 /* width of icon in pixels */
    $"FFFFFFFFFF00FFFFFF0000FFF"        /* icon image */
    $"FFFFFFFFF0FF00000FFFFFFF"
    $"FFFFFFFF0FFF00FFFFFFFFFF"
    $"FFFFFFF0FFFF000FFF0000FF"
    $"FFFFFF0FFFFF000000FFFFFF"
    $"FFFFF0FFFFFF00FF0FFFFFFF"
    $"FFFF0FFFFFFF00FFF0FFFFFF"
    $"FFF0FFFFFFFF00FFFF0FFFFF"
    $"FF0FFFFFFFFF00FFFFF0FFFF"
    $"F0FFFFFFFFFF00FFFFFF0FFF"
    $"0FFFFFFFFFFF00FFFFFFF0FF"
    $"F0FFFFFFFFFF00FFFFFF0FFF"
    $"FF0FFFFFFFFF00FFFFF0FFFF"
    $"FFF0FFFF000000FFFF0FFFFF"
    $"FFFF0FF0000000FFF0FFFFFF"
    $"FFFFF0FF00000FFF0FFFFFFF"
    $"FFFFFF0FFFFFFFF0FFFFFFFF"
    $"FFFFFFF0FFFFFF0FFFFFFFFF"
    $"FFFFFFFF0FFFF0FFFFFFFFFF"
    $"FFFFFFFFF0FF0FFFFFFFFFFF"
    $"FFFFFFFFFF00FFFFFFFFFFFF",
    $"0000000000FF00000FFFF000"          /* icon mask */
    $"000000000FFFFFFFF0000000"
    $"00000000FFFFFF0000000000"
    $"0000000FFFFFFFF000FFFF00"
    $"000000FFFFFFFFFFF000000"
    $"00000FFFFFFFFFFF0000000"
    $"0000FFFFFFFFFFFFF000000"
    $"000FFFFFFFFFFFFFFF00000"
    $"00FFFFFFFFFFFFFFFFF0000"
    $"0FFFFFFFFFFFFFFFFFFF000"
    $"FFFFFFFFFFFFFFFFFFFFF00"
    $"0FFFFFFFFFFFFFFFFFFF000"
    $"00FFFFFFFFFFFFFFFFF0000"
    $"000FFFFFFFFFFFFFFF00000"
    $"0000FFFFFFFFFFFFF000000"
    $"00000FFFFFFFFFFF0000000"
    $"000000FFFFFFFFF00000000"
    $"0000000FFFFFFFF000000000"
    $"00000000FFFFFF0000000000"
    $"000000000FFFF00000000000"
    $"0000000000FF000000000000"
    };

  resource rIcon (1002, preload, nospecialmemory) {
    $8000,                         /* Icon Type bit 15  1 = color, 0 = mono */
    16,                                 /* height of icon in pixels */
    16,                                 /* width of icon in pixels */
```

```
   $"00000000000000FF"               /* icon image */
   $"0FFFFFFFFFFF0F0F"
   $"0FFFFFFFFFFF0FF0"
   $"0FFFFFFF000F0000"
   $"0FFFFF000FFFFFF0"
   $"0FFFFF00F000FFF0"
   $"0FFFFF0000FFFFF0"
   $"0FFFFF00FFFFFFF0"
   $"0FFFFF00FFFFFFF0"
   $"0FFFFF00FFFFFFF0"
   $"0FFFFF00FFFFFFF0"
   $"0FFF0000FFFFFFF0"
   $"0FF00000FFFFFFF0"
   $"0FFF000FFFFFFFF0"
   $"0FFFFFFFFFFFFFF0"
   $"0000000000000000",
   $"FFFFFFFFFFFFFF00"                 /* icon mask */
   $"FFFFFFFFFFFFFFF0"
   $"FFFFFFFFFFFFFFFF"
   $"FFFFFFFFFFFFFFFF"
   $"FFFFFFFFFFFFFFFF"
   $"FFFFFFFFFFFFFFFF"
   $"FFFFFFFFFFFFFFFF"
   $"FFFFFFFFFFFFFFFF"
   $"FFFFFFFFFFFFFFFF"
   $"FFFFFFFFFFFFFFFF"
   $"FFFFFFFFFFFFFFFF"
   $"FFFFFFFFFFFFFFFF"
   $"FFFFFFFFFFFFFFFF"
   $"FFFFFFFFFFFFFFFF"
   $"FFFFFFFFFFFFFFFF"
   $"FFFFFFFFFFFFFFFF"
   };

resource rIcon (1003, preload, nospecialmemory) {
   $8000,                       /* Icon Type bit 15  1 = color, 0 = mono */
   8,                                /* height of icon in pixels */
   8,                                /* width of icon in pixels */
   $"0000000F"                       /* icon image */
   $"0FFF00F0"
   $"0FFF0000"
   $"0FFF0FF0"
   $"0F000FF0"
   $"0FF00FF0"
   $"0FFFFFF0"
   $"00000000",
   $"FFFFFFF0"                         /* icon mask */
   $"FFFFFFFF"
   $"FFFFFFFF"
   $"FFFFFFFF"
   $"FFFFFFFF"
   $"FFFFFFFF"
   $"FFFFFFFF"
   $"FFFFFFFF"
   };
```

Listing 17-4: `rIcon` Resources for Problem 17-1

GS/OS Aware Bits

There are a lot of programs around for the Apple IIGS that were written before System Disk 4.0, when Apple introduced GS/OS. GS/OS handles prefixes differently than the older ProDOS 16 file system, and allows much longer path names. Your programs can handle the longer prefixes, but so far don't – that's because you haven't told GS/OS that you are a modern program. You do that by setting the auxiliary file type for your program. The auxiliary file type is a flag long word; here are the bit definitions:

bits 31-16	Reserved; set to 0.
bits 15-8	These bits should be set to $DB, telling GS/OS that the next 7 bits are valid.
bits 7-3	Reserved; set to 0.
bit 2	Set this bit if your program handles message center messages. It doesn't, yet, but will in a moment!
bit 1	Set this bit to 1 for all desktop applications. This bit should be clear for text applications.
bit 0	Set this bit if your program handles the long, GS/OS style path names. All of the programs from this course do.

You can set the auxiliary file type several different ways, but the method varies with the various versions of the programming languages. See your reference manual for details.

Formatting Disks

Let's say you're using a program, and run out of space on your work disk. The natural thing to do is to pop in a new disk, right? Of course, that doesn't do much good unless the program will recognize the disk and give you a chance to format the new disk. That's what `HandleDiskInsert` is for. If you put this call:

```
device := HandleDiskInsert($C000, 0);
```

in your main event loop, the Window Manager will check the various disk drives for new disks, letting the user format any that need to be formatted. `device` is a long integer; we don't need this result for anything, so you can put it in a junk variable and ignore the value.

There are some other options for this call, but they aren't used in most programs. For details, see *Programmer's Reference for System 6.0*.

Opening and Printing Files on Entry

Assuming you've set bit 2 of the auxiliary file type for your program, like we talked about in the last section, the Finder will pass a message to your program when the user runs the program by opening a document. If the user just runs your program, there won't be a message. The message itself is a series of path names, along with a flag telling you if you should open the documents or print them.

To get this message, create a handle for a moveable chunk of memory (it doesn't matter how big it is, so just create one with one byte) and call `MessageCenter`, like this:

```
MessageCenter(getMessage, 1, msgHandle);
```

The Tool Locator scans the message center to see if the Finder posted a message with a message type of 1. If so, `MessageCenter` expands the handle you passed to the proper size and stuffs the message into the handle. If there is no message, `MessageCenter` returns an error code of $0111.

Assuming `MessageCenter` returns with no error, the next step is to get rid of the message, like this:

```
   MessageCenter(deleteMessage, 1, msgHandle);
```

This removes the message from the messages in the message center, so the message won't pop up again later. The last parameter isn't actually used when you ask `MessageCenter` to delete a message, so the message you just read isn't damaged. Incidentally, even through `MessageCenter` changed the size of this handle, it belongs to your program, so you need to dispose of it when you've finished.

The message you get back looks like this:

displacement	length (bytes)	use
0	4	Handle of the next message. This is used by the message center; you should ignore it.
4	2	Message type. This will be 1, since you asked for message type 1.
6	2	This value will be 0 if you are supposed to open the files, and 1 if the files should be printed.
8	varies	This is a string with a leading length byte. The string is a full or partial path name for the file you are supposed to open or print. There can be more than one of these strings! Right after the last one, there will be a zero byte. You can take this to mean a string of length 0, and that's a convenient way of dealing with it in the program, or you can take this to mean a separate field, which is how it's defined in the toolbox reference manual.

There are two issues we need to discuss about this record: the mechanics of using it, and what you're actually supposed to do with the information.

First, the mechanics. The easiest way to deal with the record is to create two pointers, one defined as a pointer to a boolean, and the other defined as a pointer to a string, like this:

```
   var
      bPtr: ^boolean;
      sPtr: pStringPtr;
```

Start by locking the handle, then use pointer math to set the boolean pointer to point to the print flag, and the string pointer to the first path name, like this:

```
   HLock(msgHandle);
   bPtr := pointer(ord4(msgHandle^)+6);
   sPtr := pointer(ord4(bPtr)+2);
```

Now you have a convenient way to use the information in the message. To see if you are supposed to print the file, just dereference the first pointer. When it comes time to open or print a file, check the length of the string pointed to by the second pointer. If it's 0, you're done. If it isn't, do whatever you are supposed to do with the file, then advance the string pointer to point to the next string by adding the length of the current string plus one to the pointer.

If you are supposed to open the files, you should treat each of the file names as if the user had launched the program and immediately used the Open command to open the files. There is one big difference you need to keep in mind, though: the message center is giving you a p-string, while SFO gives you a GS/OS path name. You'll probably want to convert the p-string into a GS/OS string to handle this difference. Once all of the files are open, drop into your main event loop.

Printing files is a little different. It's probably easiest to do a normal start-up on your program, the do an open-print-close sequence on each of the files that you are supposed to print. You should not leave a file open after it is printed, for two reasons. The first is that you could run out of memory trying to open all of the files. The second is that you're program is supposed to quit as soon as all of the files are printed anyway, so there's no reason to run the risk of running out of memory.

Whether you are supposed to open the files or print the files, don't forget to dispose of the message when you finish!

Problem 17-2: Start with the result of problem 17-1, and add message center support. Your instrument sampler can only load one instrument at a time, so if there is more than one file in the message, warn the user. It doesn't make much sense to print an instrument, either, so if the Finder asks you to print an instrument, put up an appropriate error dialog.

Be sure to set the proper GS/OS aware bits so the Finder knows your program can use the message center.

Also, add a call to `HandleDiskInsert` so your program will recognize new disks and give the user a chance to format them.

Summary

In this lesson, you've learned to put a professional touch on your programs. You've learned how to tell the Finder what documents your program can deal with, and you've learned how to create programs that will open or print documents when the user picks those options from the Finder. Your programs will recognize new disks and give the user a chance to format one. You can also create icons, both for your program and for the document files it handles.

Tool calls used for the first time in this lesson:

```
AddResource       CloseResourceFile HandleDiskInsert  MessageCenter
OpenResourceFile
```

Resource types used for the first time in this lesson:

```
rBundle           rComment          rVersion
```

Lesson 18 – New Desk Accessories

Goals for This Lesson

New desk accessories are the mini-programs that show up under the Apple menu in most desktop programs. This lesson shows you how to write a new desk accessory.

The Anatomy of an NDA

New Desk Accessories are a rather interesting hybrid: they are sort of an application, and sort of not an application. Desk accessories are available from any desktop program that is written in the normal way and doesn't go out of it's way to keep you from using them. Your program doesn't have to shut down when you use a desk accessory, either.

There's really nothing to stop you from writing full-fledged applications that are desk accessories, but generally, desk accessories are small utility programs that are used while some other main application is running.

The desk accessory itself isn't handled like a normal application at all. The operating system knows about desk accessories right away, letting them do things as the computer boots. Desk accessories get called as your program starts, too, so they can do things almost right away. They also have to be able to shut down when the user picks Quit from the main application, and they get called then, too.

There are a lot of ways that these different calls could have been handled, but as it turns out, the Apple IIGS handles them by making desk accessories a lot different from standard applications. Instead of a single entry point at the main body of the program, desk accessories have several entry points, one for each purpose. The Pascal compiler has to do several things behind your back to make all of that work, so you need to tell it right up front that you want to create a desk accessory, and not a standard application. You do that using the `NewDeskAcc` directive. It looks like this:

```
{$NewDeskAcc NDAOpen NDAClose NDAAction NDAInit 60 $03FF '  Sample\H**'}
```

This directive goes before the program statement, right at the top of your program. The first four parameters are the various entry points; each of these will end up being a function or procedure in the program. The main part of the program doesn't actually get used, and should be empty. We'll talk about all four of these entry points in detail in a moment, but I'd like to mention that there is nothing special about the names you see here. You can pick any four names you like; I picked these because the names are pretty close to the names used in the toolbox reference manual.

Right after the names of the four entry points for the NDA are three other parameters. The first, 60, is the update period. One of the things the Desk Manager can do is call your NDA every so often. That makes it easy to implement something like a clock NDA. If you use a value of -1, the Desk Manager will only call you if an event occurs while your desk accessory has control. Use a 0, and the Desk Manager will call your NDA as often as possible. That's generally a bad idea, though, since your NDA could slow down the main application if it gets called too often. Any other value is the number of ticks to wait between calls. Since one tick is 1/60th of a second, the value of 60 you see in the sample tells the Desk Manager to call the NDA once every second – a good value for a clock.

The next parameter is an event mask. It works just like the one you pass to `TaskMaster`, telling the Desk Manager what events the NDA will handle.

The last parameter is a menu string; it's used to form the menu item that the user will see in the Apple menu. It must start with two spaces and end with the characters '\H**'. The rest of the name is what the user will see.

Before we get bogged down in the details, let's step back and get an overview of how the various subroutines in the NDA are used. While there are some ways to get around it, an NDA should have exactly one window – no more and no less. When the user picks your NDA from the Apple menu, the Desk Manager calls your open routine. The open routine is a function. You are supposed to open your one window, returning a pointer to the window to the Desk Manager as the function result, or nil if there is some problem and you can't start. When the user clicks on the close box of your NDA, or picks close from the File menu while your NDA is active, the Desk Manager calls your close routine. That, of course, is your cue to close your window. The action routine gets called based on what events are posted and how you've set the period in the `NewDeskAcc` directive, but it will only be called while your window is open. Finally, the Init routine is called both when the tools are started and when they are shut down.

As we go through the four entry points, I'll give you an example of a simple but complete desk accessory, a Clock NDA. In fact, the `NewDeskAcc` sample directive from a few paragraphs back is the first part of the Clock NDA. Assembling the pieces will be left for a problem.

The Init Routine

A moment ago, I mentioned that the Init routine is called as the tools are started and shut down. Actually, to be precise, the Init routine is called when `DeskStartUp` and `DeskShutDown` are called. `DeskStartUp` is the actual tool call made by `StartDesk` or `StartUpTools` when the Desk Manager is started, and `DeskShutDown` is called when the Desk Manager is shut down. Of course, it's sort of important to know whether the tools are being started or shut down! That's why the Init routine has a parameter. It's a single integer, set to 1 if the tools are being started, and 0 if they are being shut down.

Naturally, you can do any sort of initialization or clean-up you like in your Init routine, but there are two things you should always do. The first is to close your window if it is open, and the second is to close any tools you started. Of course, you have to know if the window is open and whether you actually did start any tools, so you should create some global variables to tell you whether you did either of these things, and set them to false at start up time.

We'll talk more about the issue of tools later. Here's a sample Init routine that deals with the issue of whether the window has been opened or not, though:

```
procedure NDAInit (code: integer);

{ Init entry point                                              }
{                                                               }
{ Parameters:                                                   }
{    code - 1 for tool start up time, 0 for tool shut down time }

begin {NDAInit}
if code = 1 then
   clockActive := false
else
   if clockActive then
      NDAClose;
end; {NDAInit}
```

Listing 18-1: `NDAInit` Subroutine

Of course, the open and close routines will have to get in the act, too, setting and clearing the `clockActive` variable as the window is opened and closed.

The Open Routine

The main job for the open routine is to open the main NDA window. This subroutine is actually a function returning a window pointer to the newly opened window. You can create and open the window itself using a resource, just like you normally do, but there is one housekeeping chore you have to take care of that's done automatically in a standard application: you have to open your own resource fork. We'll talk about that in the next section. In Listing 18-2A, the problem is swept under the rug by calling subroutines to do the work – which isn't a bad idea anyway.

Here's the `NDAOpen` subroutine for the Clock NDA.

```
function NDAOpen: grafPortPtr;

{ Open entry point                                             }
{                                                              }
{ Returns: Pointer to the new NDA window                       }

const
   rWindParam1 = $800E;                {window resource type}
   wrNum = 1001;                       {window resource number}

begin {NDAOpen}
NDAOpen := nil;
if not clockActive then
   if OpenResourceFork then begin
      clockWinPtr :=
         NewWindow2(@' Clock ', 0, nil, nil, $02, wrNum, rWindParam1);
      if ToolError = 0 then begin
         SetSysWindow(clockWinPtr);
         clockActive := true;
         NDAOpen := clockWinPtr;
         end; {if}
      CloseResourceFork;
      end; {if}
end; {NDAOpen}
```

Listing 18-2A: `NDAOpen` Function

```
resource rWindParam1 (1001) {
   $C0A5,                      /* wFrameBits */
   nil,                        /* wTitle */
   0,                          /* wRefCon */
   {0,0,0,0},                  /* ZoomRect */
   linedColors,                /* wColor ID */
   {0,0},                      /* Origin */
   {0,0},                      /* data size */
   {0,0},                      /* max height-width */
   {0,0},                      /* scroll ver hors */
   {0,0},                      /* page ver horiz */
   0,                          /* winfoRefcon */
   0,                          /* wInfoHeight */
   {50,50,62,200},               /* wposition */
   infront,                    /* wPlane */
   nil,                        /* wStorage */
   $0800                       /* wInVerb */
   };

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

Listing 18-2B: Resources for the Clock Window

There's one extra step in NDAOpen besides the housekeeping you were probably expecting. The extra step is the call to SetSysWindow, which tells the Window Manager that the window you just created belongs to an NDA, and not to an application. This is a very important step! Without it, the toolbox won't realize when it should give control to your NDA instead of the main program.

Using Resources in an NDA

In all of your programs up to this point, the resource fork for your program is opened and made the current resource fork automatically. This isn't done for you in an NDA, so you need to open your resource fork manually before you use any tool calls that will need to load a resource from your resource fork. You also have to remember to close your resource fork before you return to the Desk Manager, since leaving it open can interfere with the application.

Opening and closing the resource fork isn't really that hard. In fact, you've already seen how to do it – in the last lesson, you learned how to use OpenResourceFile and CloseResourceFile to write an rComment resource. The only difference between what you saw then and what you need to do now is that you open the resource fork for input, not for output.

Strangely enough, the real problem is figuring out what file name to use. After all, an NDA will be moved to any number of disks, all of which have a different name. The user might rename the program, too, so your program can't even count on knowing it's own file name, let alone it's full path name. Fortunately, your program can call the System Loader and ask what it's name is, like this:

```
fPtr := LGetPathname2(id, 1);
```

LGetPathname2 returns a pointer to a GS/OS input string, which is exactly what you want to pass to OpenResourceFile. The first parameter is the user ID for your program. The second parameter is your program's file number, which is always 1.

LGetPathname2 comes with two of it's own problems, though. The first is that ORCA/Pascal doesn't have a header file for the System Loader, most of which has to be called from assembly language to be useful. To get around that problem, put this declaration right after all of your global variable declarations:

```
   function LGetPathname2 (userID, fileNum: integer): gsosInStringPtr;
      tool ($11, $22);
```

The second problem is that you need the original user ID used by the loader, which is not the same one returned by ORCA/Pascal's UserID function. You can get the original user ID with a couple of new GS/OS calls:

```
  id := SetHandleID(0, FindHandle(@OpenResourceFork));
```

FindHandle takes a pointer to any location in memory and returns the handle for the location. In this case, we're giving FindHandle the address of a subroutine in the program. SetHandleID is generally used to change the user ID of a handle, changing the owner. In this case, though, we're passing 0 for the new user ID, which tells SetHandleID not to change the user ID. SetHandleID always returns the user ID for the handle, so we get what we need.

Here's a pair of subroutines that implement these ideas. These are the same ones called by NDAOpen in Listing 18-2B.

```
  function OpenResourceFork: boolean;

  { Open the resource fork for this DA                            }

  var
     fPtr: gsosInStringPtr;               {GS/OS file name pointer}
     id: integer;                         {user ID for our executable chunk}

  begin {OpenResourceFork}
  id := SetHandleID(0, FindHandle(@OpenResourceFork));
  fPtr := LGetPathname2(id, 1);
  if ToolError = 0 then
     rID := OpenResourceFile(1, nil, fPtr^);
  OpenResourceFork := ToolError = 0;
  end; {OpenResourceFork}


  procedure CloseResourceFork;

  { Close the resource fork opened by OpenResourceFork            }

  begin {CloseResourceFork}
  CloseResourceFile(rID);
  end; {CloseResourceFork}
```

Listing 18-3: Opening and Closing Our Resource Fork

The Close Routine

The close routine doesn't have any parameters. It is called when the user wants to close the window, so naturally, that's what you should do. Since the window pointer isn't passed, though, you have to get it from the global variable, which is why the open subroutine saved the window pointer. (One of the reasons, anyway.)

There is one subtle point you have to be careful of in your close routine. It can be called even if the open routine is never called at all, and it can be called more than once. Because of that, you have to write it carefully so you don't try to close a window that doesn't exist. If you forget, the best thing you can expect is for the Window Manager to crash.

```
procedure NDAClose;

{ Close entry point                                            }

begin {NDAClose}
if clockActive then begin
   CloseWindow(clockWinPtr);
   clockActive := false;
   end; {if}
end; {NDAClose}
```

Listing 18-4: `NDAClose` Subroutine

The Action Routine

The action routine is the one that gets called when your NDA is actually supposed to do something. There are two parameters. The first is a code that tells the action routine why the call is being made. The second parameter is only used for one of the action codes: for `eventAction`, the second parameter is the event record for the event your program is supposed to handle. For all other action codes, the second parameter is not used, and should be ignored. The action routine is also a function, but only a few of the action codes need to return a value.

Here's the various action codes, along with what they mean and what the second parameter is. The names you see here are declared as constants in the ORCA/Pascal interface files, and are also listed in the toolbox reference manual and Appendix A with the numeric values for the names.

code	use
`eventAction`	This code is used when your NDA needs to handle an event. in this case, the second parameter is a pointer to an event record containing the event you are supposed to handle. Some events aren't passed to NDAs. If the event mask in the `NewDeskAcc` directive has the proper bits enabled, the NDA will get button down events, button up events, and key down events. No matter how the bits are set, the NDA will always get its own activate events and update events. All other events are always passed to the application.
`runAction`	This is the code used when the time period you set up in the `NewDeskAcc` directive expires.
`cursorAction`	This action code is used when the NDA window is the front window when `SystemTask` is called. (`SystemTask` is called by `TaskMaster`; you don't have to worry about calling it in your program if you use `TaskMaster`.) The intent is to give you a chance to change the cursor if it's over the NDA window.
`undoAction`, `cutAction`, `copyAction`, `pasteAction`, `clearAction`	These five action codes are used when the user uses one of these standard editing operations. If your NDA handles the editing operation, you should

return true; if your NDA did not handle the operation, you should return false.

While the action routine is always a function, you only have to return a value for these five action codes. Returning a value for one of the other action codes doesn't hurt, but it isn't required, either.

Handling an event in an NDA isn't really any harder than handling an event in an application, but it isn't any easier, either. There's a `TaskMasterDA` call that's designed for NDAs. It looks and works just like `TaskMaster`, except that the event mask (the first parameter) isn't really used for anything; it's just there to make `TaskMasterDA` look like `TaskMaster`.

Here's the action routine for the Clock.

```
function NDAAction (code: integer; myEvent: eventRecord): boolean;

{ Action entry point                                            }
{                                                               }
{ Parameters:                                                   }
{    code - action code; tells why the call is being made       }
{    event - if the code is eventAction, this is an event       }
{       record; the parameter is unused for all other codes     }
{                                                               }
{ Returns: True if one of the five editing codes is handled,    }
{    false if not.                                              }

var
   event: integer;                     {TaskMaster event code}
   port: grafPortPtr;                  {caller's grafPort}

begin {NDAAction}
NDAAction := false;
case code of
   eventAction:
      event := TaskMasterDA(0, myEvent);

   runAction: begin
      port := GetPort;
      SetPort(clockWinPtr);
      DrawTime;
      SetPort(port);
      end;

   otherwise: ;
   end; {case}
end; {NDAAction}
```

Listing 18-5: `NDAAction` Subroutine

For completeness sake, here's the `DrawTime` subroutine used by the Clock NDA to actually read the time and draw it in the window.

```
procedure DrawTime;

{ Draw the time in the window                               }

var
   i: integer;                         {loop variable}
   timeString: packed array[1..21] of char; {string to hold time}

begin {DrawTime}
ReadAsciiTime(@timeString);
for i := 1 to 20 do
   timeString[i] := chr(ord(timeString[i])&$7F);
timeString[21] := chr(0);
MoveTo(7, 10);
DrawCString(pointer(@timeString));
end; {DrawTime}
```

Listing 18-6: `DrawTime` Subroutine

There is one new call here, `ReadAsciiTime`. This call gets the current time from the Miscellaneous Tool Set, returning the time more or less as a 20 character ASCII string. In the `DrawTime` subroutine the characters are turned into a true ASCII string by anding out the extra bit `ReadAsciiTime` sets in each character. After that, it's a simple matter to add a terminating null character and draw the string in the window.

Starting and Shutting Down Tools

One of the points I glossed over and said I would come back to is the issue of starting and stopping tools. This is a really touchy point in an NDA, and there really isn't any completely safe solution.

Let's start by looking at the tools that are always started when an NDA is running. They are:

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

So how do you know that these tools are started? Believe it or not, you have to trust the application. An application isn't supposed to allow NDAs unless all of these tools are started. Fortunately, with the exception of the last three tools, you really can't write much of a program unless all of these tools are started.

If your NDA needs some tool other that one of the ones listed here, there's a chance it won't be started. The natural reaction is to start the tools you need in your Init procedure, then shut them down in the same place. Unfortunately, that doesn't work. The application will be starting the Desk Manager, and thus calling your NDA, pretty early in the tool start up sequence, and you don't want to interfere with the application by starting a tool before it does! The next possibility is to start any tools you need in your open routine, and that works, to a point. You just need to be sure the tool hasn't already been started before you try, and be sure you can handle the situation gracefully (i.e. without interfering with the application!) if you fail.

The next problem is shutting the tools down. Since you started your tools in the open routine, the natural solution would be to shut the tools down in the close routine. That can cause problems, though. Think about this sequence of events:

1. Your NDA is opened, and needs the Print Manager. The application didn't start the Print Manager, so you do.
2. The user opens another NDA, which also uses the Print Manager. This NDA sees that the Print Manager has been started, so it doesn't start the Print Manager.
3. The user closes your NDA, so you shut down the Print Manager.
4. The user then tries to print something with the second NDA, which promptly crashes because the Print Manager isn't active.

Well, that's a problem, all right. The best you can do is to take two other steps. First, always check to be sure a tool is started just before you use it, and start the tool if it hasn't been started already. Do that even if you think you've already started the tool. Second, shut down tools when the Init routine is called at tool shutdown time, and not before. In addition, always check to make sure a tool is still active before you shut it down.

This system still isn't perfect. Most applications start all of the tools they need at start up time, and shut down all of the tools they started just before exiting, but it's possible for the application itself to start a tool and shut it down at any time. You can protect yourself from the application by always double checking to be sure all of the tools you need are going, but the application may get upset if it tries to start a tool and it's already going. You can't do anything practical to avoid this situation, so it's fortunate that it doesn't come up very often.

I've talked a lot about checking to see if a tool is active in this section. While I'm not going to go into details on how to do that, you can follow up on the idea if you like. Every tool has a status call, and that status call will tell you if the tool is installed and active. The status call works even if the tool is a RAM based tool and hasn't been loaded, so it's always safe. It's this status call that lets you check to see if the tool is active so you can decide if you need to start the tool.

Installing an NDA

There are two other steps you have to take to create an NDA. The first is to be sure and set the file type to $B8 (NDA) instead of leaving it as $B5 (EXE) or setting it to $B3 (S16) like you've done for your other programs. After all, the NDA isn't called like other programs. Trying to call an NDA like any other application would cause a crash.

The other step is to put the NDA in the right place. All desk accessories have to be in the System:Desk.Accs folder on your boot disk; that's the only place the Desk Manager will look. There are several utilities around that can bypass this restriction, including PRIZM, but in general you need to copy your NDA to the desk accessories folder, then reboot the computer.

Problem 18-1: Put all of the pieces together to create a Clock NDA. Install the NDA and test it.

Summary

This final lesson of the course covered NDAs – how to create them and how to install them.

Tool calls used for the first time in this lesson:

```
FindHandle        LGetPathname2     ReadAsciiTime     SetHandleID
SetSysWindow      TaskMasterDA
```

Appendix A – Abridged Toolbox Reference Manual

Control Manager

In ORCA/Pascal the header file for this tool is named ControlMgr. This header file makes use of the Common unit. Both of these units must be listed in your `uses` statement to make calls listed in this section.

DrawControls

```
procedure DrawControls (theWindow: grafPortPtr);
```

Possible Errors: None

Draws all of the controls in a window. If your window defines controls other than the scroll bars and grow box used by `TaskMaster`, you should include a call to DrawControls in your update procedure.

FindTargetCtl

```
function FindTargetCtl: ctlRecHndl;
```

Possible Errors:			
	`$1004`	`noCtlError`	No controls in the window
	`$1005`	`noExtendedCtlError`	No extended controls in the window
	`$1006`	`noCtlTargetError`	No extended control is currently the target control
	`$100C`	`noFrontWindowError`	There is no front window

Returns the control handle for the current target control.

GetCtlHandleFromID

```
function GetCtlHandleFromID (theWindow: grafPortPtr; ctlID: univ longint):
   ctlRecHndl;
```

Possible Errors:			
	`$1004`	`noCtlError`	No controls in the window
	`$1005`	`noExtendedCtlError`	No extended controls in the window
	`$1009`	`noSuchIDError`	The specified control ID cannot be found
	`$100C`	`noFrontWindowError`	There is no front window

`GetCtlHandleFromID` returns the control handle for a control, given the window containing the control and the control ID.

`GetCtlHandleFromID` only works with extended controls. All of the controls we've defined and used in this course are extended controls, so you can always used `GetCtlHandleFromID` safely for solutions to the problems. If you are defining controls without resources, though, you should check the detailed description of this command in *Apple IIGS Toolbox Reference Manual: Volume 3*, page 28-27 to make sure it will work.

GetCtlID

```
function GetCtlID (theControl: ctlRecHndl): longint;
```

Possible Errors:			
	`$1004`	`noCtlError`	No controls in the window
	`$1007`	`notExtendedCtlError`	The action is valid only for extended controls

Returns the control ID for the given control handle.

`GetCtlID` only works with extended controls. All of the controls we've defined and used in this course are extended controls, so you can always used `GetCtlID` safely for solutions to the problems. If you are defining controls without resources, though, you should check the detailed description of this command in *Apple IIGS Toolbox Reference Manual: Volume 3*, page 28-28 to make sure it will work.

GetCtlParams

```
function GetCtlParams (theControl: ctlRecHndl): longint;
```

Possible Errors: None

`GetCtlParams` returns the control parameters.

In one place in this course `GetCtlParams` is used to fetch the current document size and page size for a scroll bar control. The data size is returned in the least significant word of the long result, while the view size is returned in the most significant word.

`GetCtlParams` is also used in the `GetCtlData` subroutine from Lesson 14.

GetCtlValue

```
function GetCtlValue (theControl: ctlRecHndl): integer;
```

Possible Errors: None

Returns the current value for a control. For a scroll bar control, this will be the position of the thumb. For a radio button or check box, the value will be zero if the button is off, and some non-zero value if the button is on.

GetLETextByID

```
procedure GetLETextByID (theWindow: grafPortPtr; controlID: longint;
   var text: pString);
```

Possible Errors: Control Manager errors are returned unchanged

`GetLETextByID` returns the text from a line edit control.

The text returned starts with a length byte, which is set to 0 if the text is longer than 255 characters. This is followed by the text itself, then by a terminating null character (`chr(0)`);

Because of this format, you must make sure the string buffer the text is placed in is at least two characters longer than the longest allowed string for the line edit control. You can use the result directly as a p-string, or skip the first byte and use the remainder of the entry as a c-string. (ORCA/Pascal will handle a c-string as a Standard Pascal string.)

SetCtlAction

```
procedure SetCtlAction (newAction: procPtr; theControl: ctlRecHndl);
```

Possible Errors: None

Changes the action procedure for the given control to `newAction`. In this course, `SetCtlAction` is used to define a procedure to handle continuous scrolling for scroll bar controls.

SetCtlParams

```
procedure SetCtlParams (param2, param1: integer; theControl: ctlRecHndl);
```

Possible Errors: None

Sets the parameters for the control.

In this course, this call is used to change the document size and data size for a scroll bar. For a scroll bar control, the first parameter (`param2`) is the document size, while the next parameter (`param1`) is the view size.

You can pass a -1 for either parameter to tell the Control Manager to leave that value alone.

SetLETextByID

```
procedure SetLETextByID (windPtr: grafPortPtr; leCtlID: longint;
   text: pString);
```

Possible Errors: Control Manager errors are returned unchanged

`SetLETextByID` sets the text in a line edit control to the contents of the given string.

SetCtlValue

```
procedure SetCtlValue (curValue: integer; theControl: ctlRecHndl);
```

Possible Errors: None

Sets the current value for a control and redraws the control with the new value. For a scroll bar control, changing the value changes the position of the thumb. For a radio button or check box, setting the value will to zero turns the button is off, while setting the value to 1 turns the button on (and, for radio buttons, turns off any other buttons in the same family).

Desk Manager

In ORCA/Pascal the header file for this tool is named DeskMgr. This header file makes use of the Common unit. Both of these units must be listed in your `uses` statement to make calls listed in this section.

FixAppleMenu

```
procedure FixAppleMenu (menuID: integer);
```

Possible Errors: None

The names of all of the currently installed NDAs are added to the menu specified by `menuID`. NDAs are traditionally added to the apple menu, which should already contain an About menu item. The menu items added by this call will appear after any items that are already in the menu.

The menu items are given menu ID numbers numbered sequentially, starting with a menu ID of `1`.

Event Manager

In ORCA/Pascal the header file for this tool is named EventMgr. This header file makes use of the Common unit. Both of these units must be listed in your `uses` statement to make calls listed in this section.

GetMouse

```
procedure GetMouse (var mouseLocPtr: point);
```

Possible Errors: None

Returns the current location of the mouse using the local coordinates of the current `grafPort`.

GetNextEvent

```
function GetNextEvent (eventMask: integer; var theEvent: eventRecord):
   boolean;
```

Possible Errors: None

`GetNextEvent` checks to see if any events have occurred, returning true if there are events the program needs to handle and false if there are no pending events.

It's possible for more than one event to occur before the program has time to call `GetNextEvent`. To handle this situation, the Event Manager places events in an event queue, which is a prioritized first in, first out list. (Some kinds of events have a higher priority than others, and are always reported first, but events with the same priority are treated in a true first in, first out basis.) The length of the list is specified when you start the Event Manager, and is normally set to 20. When you call `GetNextEvent`, it removes the next event from the queue and returns the information about that event.

You can use the `eventMask` parameter to tell `GetNextEvent` what kind of events you would like to see. If you ask for a certain kind of event, say a keyboard event, but only mouse events are in the event queue, `GetNextEvent` returns false, and the unwanted events remain in the event queue. The `eventMask` parameter is a series of bit fields telling what events to report. Figure A-1 shows the bit fields and what they are used for. See the definitions section for a series of predefined names that can be added together to get various bit mask combinations.

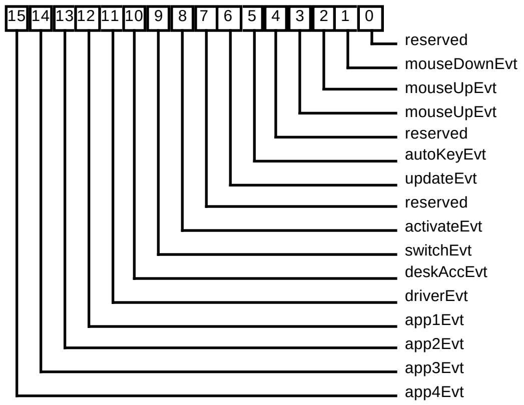


Figure A-1: Event Masks

Information is returned in the event record any time a call is made to `GetNextEvent`, whether or not an event needs to be handled. There are five fields in the event record:

`eventWhat`	The event that occurred.
`eventMessage`	Information about the specific event.
`eventWhen`	When the event occurred. When the tools are started, an internal clock begins counting 1/60ths of a second; this time is the number of 1/60ths of a second that have elapsed from when the tools were started and when the event was detected.
`eventWhere`	Position of the cursor (mouse) when the event was detected.
`eventModifiers`	Extra information about the state of the computer when the event was detected.

The `eventMessage` field contains a variety of information based on the kind of event that occurred. Table A-1 shows the kinds of events by the name used in the Pascal toolbox interface files, along with what the `eventMessage` field contains. For the numeric value of the events, see the definitions section for the Event Manager.

`eventWhat`	`eventMessage`
`nullEvt`	Undefined.
`mouseDownEvt`	Button number (1 or 0) in low-order word, high-order word undefined.
`mouseUpEvt`	Button number (1 or 0) in low-order word, high-order word undefined.
`keyDownEvt`	ASCII character code in low-order byte, other bytes undefined.
`autoKeyEvt`	ASCII character code in low-order byte, other bytes undefined.
`updateEvt`	Pointer to the window to update.
`activateEvt`	Pointer to the window that was activated.
`switchEvt`	Undefined.
`deskAccEvt`	Undefined.
`driverEvt`	Defined by the device driver.
`app1Evt`	Defined by the application.
`app2Evt`	Defined by the application.
`app3Evt`	Defined by the application.
`app4Evt`	Defined by the application.

Table A-1: Event Messages by Event Type

The `eventModifiers` field is a series of bit flags that give specific information about the state of the computer when the event was detected. Figure A-2 shows the bit fields and what they are used for. See the definitions section for a set of predefined constants that can be used to form bit masks for the event modifiers field.

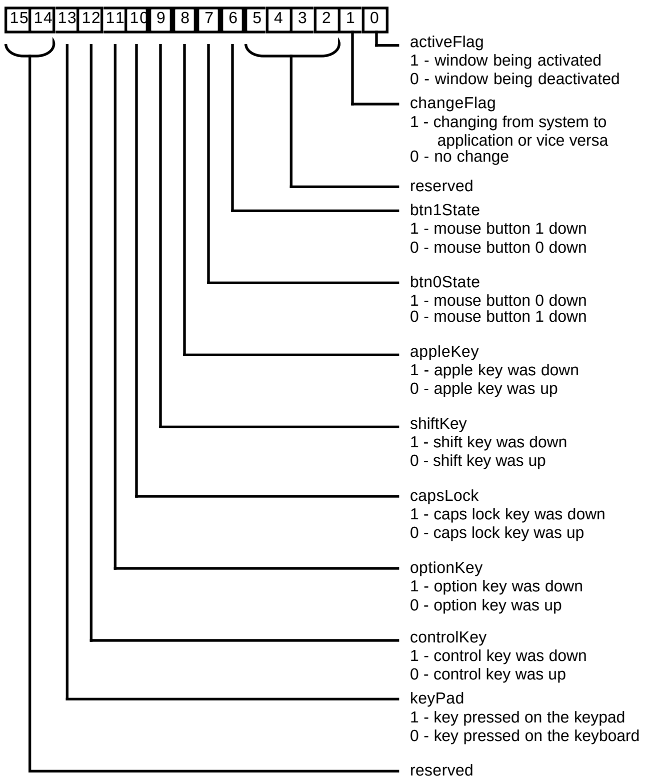


Figure A-2: Modifier Flags

StillDown

```
function StillDown (buttonNumber: integer): boolean;
```

Possible Errors: `$0605` `emBadBttnNoErr` The button number given was not 0 or 1

This call is used after the Event Manager detects and returns a mouse down event. `StillDown` will return true until a mouse up event occurs; it will then return false, whether or not the application has made a call to see the event. `StillDown` also handles multiple presses correctly; if the mouse button has been released since the last time the application detected a mouse down event, `StillDown` will return false, even if the mouse has been pressed again.

The button number for a single button mouse is 0.

Event Manager Definitions

```
const
   {eventWhat values}
   nullEvt         =   $0000;
   mouseDownEvt    =   $0001;
   mouseUpEvt      =   $0002;
   keyDownEvt      =   $0003;
   autoKeyEvt      =   $0005;
   updateEvt       =   $0006;
   activateEvt     =   $0008;
   switchEvt       =   $0009;
   deskAccEvt      =   $000A;
   driverEvt       =   $000B;
   app1Evt         =   $000C;
   app2Evt         =   $000D;
   app3Evt         =   $000E;
   app4Evt         =   $000F;

   {event masks}
   mDownMask       =   $0002;      {call applies to mouse-down events}
   mUpMask         =   $0004;      {call applies to mouse-up events}
   keyDownMask     =   $0008;      {call applies to key-down events}
   autoKeyMask     =   $0020;      {call applies to auto-key events}
   updateMask      =   $0040;      {call applies to update events}
   activeMask      =   $0100;      {call applies to activate events}
   switchMask      =   $0200;      {call applies to switch events}
   deskAccMask     =   $0400;      {call applies to desk accessory events}
   driverMask      =   $0800;      {call applies to device driver events}
   app1Mask        =   $1000;      {call applies to application-1 events}
   app2Mask        =   $2000;      {call applies to application-2 events}
   app3Mask        =   $4000;      {call applies to application-3 events}
   app4Mask        =   $8000;      {call applies to application-4 events}
   everyEvent      =   $FFFF;      {call applies to all events}

   {Modifier flags}
   activeFlag    = $0001;          {set if window was activated}
   changeFlag    = $0002;          {set if active window changed state}
   btn1State     = $0040;          {set if button 1 was up}
   btn0State     = $0080;          {set if button 0 was up}
   appleKey      = $0100;          {set if Apple key was down}
   shiftKey      = $0200;          {set if Shift key was down}
   capsLock      = $0400;          {set if Caps Lock key was down}
   optionKey     = $0800;          {set if Option key was down}
   controlKey    = $1000;          {set if Control key was down}
   keyPad        = $2000;          {set if keypress was from key pad}

type
   {event record (interface file uses the TaskMaster version)}
   eventRecord = record
       eventWhat:      integer;
       eventMessage:   longint;
       eventWhen:      longint;
       eventWhere:     point;
       eventModifiers: integer;
       end;
```

Font Manager

In ORCA/Pascal the header file for this tool is named FontMgr. This header file makes use of the Common unit and the QuickDrawII. All of these units must be listed in your `uses` statement to make calls listed in this section.

ChooseFont

```
function ChooseFont (currentID: fontID; famSpecs: integer): longint;
```

Possible Errors:	`$1B08`	`fmBadFamNumErr`	Illegal family number
	`$1B09`	`fmBadSizeErr`	Illegal font size
	Memory Manager errors are returned unchanged		

`ChooseFont` brings up a dialog that allows the user to choose a font family, font size, and font style, then returns the values selected in a longint with the same format as a `fontID`. The result can be type cast to `fontID` and stored in a `fontID` variable.

`currentID` is a starting font ID; it is used to create defaults for the font family, font size, and font style.

`famSpecs` is a flags word. If bit 5 (`baseOnlyBit`) is set, `ChooseFont` will only display base font families. (Roughly, that means it will only display the fonts that actually exist as bit maps.) If the bit is clear, the user can pick any combination of font families, sizes and styles.

CountFonts

```
function CountFonts (desiredID: fontID; specs: integer): integer;
```

Possible Errors:	`$1B08`	`fmBadFamNumErr`	Illegal family number
	`$1B09`	`fmBadSizeErr`	Illegal font size

`CountFonts` returns the number of fonts that fit a given specification. While there are many uses for this call, in this course we use it to find out if a given font size exists as a bit mapped font, or if the Font Manager will have to scale the font; this information is used to highlight appropriate font sizes in the font size menu. For this purpose, pass the `fontID` for the font with the font style byte set to $FF, and the font family and size set normally. The `specs` flags word should be $0A.

For other ways to use this call, and a complete description of the various bit flags, see the *Apple IIGS Toolbox Reference Manual: Volume 1*, page 8-29.

FamNum2ItemID

```
function FamNum2ItemID (familyNum: integer): integer;
```

Possible Errors:	`$1B04`	`fmFamNotFndErr`	Family not found
	`$1B0B`	`fmMenuErr`	FixFontMenu never called

`FamNum2ItemID` takes a font family number as input and returns the menu item ID used by `FixFontMenu` when the font family was placed in the menu bar. An error is generated if the font family is not in the menu, or if `FixFontMenu` was never called.

FixFontMenu

```
procedure FixFontMenu (menuID, startingID, famSpecs: integer);
```

Possible Errors: Memory Manager errors are returned unchanged

Attaches the names of all of the font families to a specified menu.

`menuID` is the menu number to attach the menu items to. `startingID` is the first menu item ID to use; the remaining menu item IDs are assigned sequentially until all of the font families have been added to the menu. `famSpecs` is a flags word. In this course, the `famSpecs` parameter set to 0; see the *Apple IIGS Toolbox Reference Manual: Volume 1*, page 8-36 for other possibilities.

You must call `FixMenuBar` before the menu is actually displayed. In most programs, the call to `FixFontMenu` and `FixAppleMenu` will be made in the same place that the menu bar is set up.

FMGetCurFID

```
function FMGetCurFID: longint;
```

Possible Errors: None

Returns the font ID for the current font. Functions cannot return records, so the value is returned as a long integer. Font ID records, though, are four bytes long, so you can use type casting to store the result in a font ID variable.

ItemID2FamNum

```
function ItemID2FamNum (itemID: integer): integer;
```

Possible Errors:	`$1B04`	`fmFamNotFndErr`	Family not found
	`$1B0B`	`fmMenuErr`	FixFontMenu never called

Takes a menu item ID for a menu item created by `FixFontMenu` and returns the corresponding font family number. An error is generated if the menu ID was not created by `FixFontMenu`, or if `FixFontMenu` was never called.

Font Manager Definitions

```
const
   (* Family Numbers *)
   newYork        =   $0002;
   geneva         =   $0003;
   monaco         =   $0004;
   venice         =   $0005;
   london         =   $0006;
   athens         =   $0007;
   sanFran        =   $0008;
   toronto        =   $0009;
   cairo          =   $000B;
   losAngeles     =   $000C;
   times          =   $0014;
   helvetica      =   $0015;
   courier        =   $0016;
   symbol         =   $0017;
   taliesin       =   $0018;
   shaston        =   $FFFE;
```

```
type
   (* Font ID *)
   fontID = record
        famNum: integer;
        fontStyle, fontSize: byte;
        end;
```

GS/OS

In ORCA/Pascal the header file for GS/OS is named GSOS. This header file makes use of the Common unit. Both of these units must be listed in your `uses` statement to make calls listed in this section.

Path Names

Path names are used to uniquely identify files on a disk. (They are also used for other things; see *Apple IIGS GS/OS Reference* for details.) The full path name consists of the name of the diskette, followed by the names of any folders you would have to open to get to the file using the Finer, followed by the name of the file itself. These names are separated from each other with either : characters or / characters.

For example, let's say you are trying to identify a file named MyFile. This file is located on a disk called MyDisk, inside a folder called MyFolder. The full path name for the file is

```
:MyDisk:MyFolder:MyFile
```

When you are writing toolbox programs, you generally don't have to worry about issued like partial path names, device names, or just what the rules are for creating a path name, since SFO makes sure the path names are legal before it returns the name to your program. The only special situation you should be aware of is that * is used for the boot disk. So, for example, if you want to store a file called ScrapBook in the System folder of the boot disk, you could use the name

```
*:System:ScrapBook
```

File Types

Table A-2 shows the file types and auxiliary file types used in this course, as well as a few of the other common file types you will see as you program. For a complete list of the file types and auxiliary file types (and in many cases, descriptions of the file formats) see the Apple IIGS File Type Notes.

In the table, Code refers to the three letter file type abbreviation you see when you use a text shell to catalog a disk. If the Auxiliary file type is not listed, it generally means that it can vary without effecting the internal format for the file. In those cases, you will have to refer to Apple's file type notes for details.

File Type	Auxiliary File Type	Code	Description
$04		TXT	ASCII text file
$06		BIN	General-purpose binary file
$0F		DIR	Folder (directory)
$B0	$0005	SRC	ORCA/Pascal source code
$B0	$0006	SRC	Script (exec) file source code
$B0	$0015	SRC	Rez source code
$B1		OBJ	Object file (compiler output, linker input)

$B2		LIB	Object code library
$B3		S16	GS/OS application
$B5		EXE	Shell application
$B8		NDA	New Desk Accessory
$BA		TOL	Tool file
$C0	$0000	PNT	Paintworks packed picture
$C0	$0001	PNT	Packed super high resolution screen
$C0	$0002	PNT	Apple preferred format image
$C1	$0000	PIC	Super high resolution screen dump
$C1	$0001	PIC	QuickDraw II Pic file
$C8	$0000	FON	Standard font
$D8	$0002	SND	ASIF instrument

Table A-2: Some Common File Types

CloseGS

```
type
   closeOSDCB = record
      pcount: integer;
      refNum: integer;
      end;

procedure CloseGS (var parms: closeOSDCB);
```

Possible Errors:	`$27`	`drvrIOError`	I/O error
	`$2B`	`drvrWrtProt`	Device is write protected
	`$2E`	`drvrDiskSwitch`	The disk has been switched
	`$43`	`invalidRefNum`	Invalid reference number
	`$48`	`volumeFull`	The volume is full
	`$5A`	`damagedBitMap`	Block number too large

This call closes an open file. Any information GS/OS has buffered in memory will be written to the disk in the process.

`pcount`	Must be 1.
`refNum`	This is the reference number (returned by `OpenGS`) for the file to close. A `refNum` of 0 tells GS/OS to close all open files.

CreateGS

```
type
   createOSDCB = record
       pcount:        integer;
       pathName:      gsosInStringPtr;
       access:        integer;
       fileType:      integer;
       auxType:       longint;
       storageType:   integer;
       dataEOF:       longint;
       resourceEOF:   longint;
       end;

procedure CreateGS (var parms: createOSDCB);
```

Possible Errors:	`$10`	`devNotFound`	Device not found
	`$27`	`drvrIOError`	I/O error
	`$2B`	`drvrWrtProt`	Device is write protected
	`$2F`	`drvrOffLine`	Device offline or no media present
	`$40`	`badPathSyntax`	Invalid path name syntax
	`$44`	`pathNotFound`	Subdirectory does not exist
	`$45`	`volNotFound`	Volume not found
	`$46`	`fileNotFound`	File not found
	`$47`	`dupPathname`	Create or rename attempted with a name that already exists
	`$48`	`volumeFull`	The volume is full
	`$49`	`volDirFull`	The volume directory is full
	`$4B`	`badStoreType`	Unsupported or incorrect storage type
	`$52`	`unknownVol`	Unknown volume type
	`$58`	`notBlockDev`	Not a block device
	`$5A`	`damagedBitMap`	Block number too large
	`$70`	`resExistsErr`	Cannot expand file; resource already exists
	`$71`	`resAddErr`	Cannot add a resource fork to this kind of file

Creates a new file. For the purpose of this call, a directory is also a file; it's just a file with a file type of $0F.

`pcount` Any value from 1 to 7. This tells how many of the following parameters to use. In most cases, you should use a value of 5.

`pathName` Pointer to a GS/OS input string of type `gsosInString`. This parameter is the name of the file to create. See "Path Names" at the start of this section for details about the names.

`access` This flag word determines how the file can be accessed. These are the bits you can set or clear from shells and the Finder to lock and unlock files. This parameter is normally set to $C3.

While all of these bits are available with the ProDOS FST, some of the bits will be unavailable with other FSTs.

bits 15-8	Reserved; set to 0.
bit 7	Set this bit if the file can be deleted.

bit 6	Set this bit if the file can be renamed.
bit 5	Set this bit if the file needs to be backed up. (GS/OS sets this bit anytime the file is changed.)
bits 4-3	Reserved; set to 0.
bit 2	Set this bit if the file should be invisible. Invisible files normally can't be seen with shells or the Finder unless you specifically ask to see them.
bit 1	Set this bit if the file can be written to.
bit 0	Set this bit if the file can be read from.

`fileType` This is the file type for the file. See "File Types" at the start of this section for some common file types.

`auxType` This is the auxiliary file type for the file.

`storageType` This entry determines what sort of file you are creating.

0-3	Creates a normal data file with a data fork. Some of these values were used for special purposes in older versions of the operating system; they are all converted to 1 in GS/OS, so that's what you would normally use.
5	Creates a file with both a data fork and a resource fork. These files are also called extended files.
13	Create a directory. If you use this parameter, make sure the file type is $0F.
$8005	Adds a resource fork to an existing file.

`dataEOF` This value is used to allocate a specific amount of disk space for the data fork of the file. As a general rule, don't use it. See *Apple IIGS GS/OS Reference* for details.

`resourceEOF` This value is used to allocate a specific amount of disk space for the resource fork of the file. As a general rule, don't use it. See *Apple IIGS GS/OS Reference* for details.

DestroyGS

```
type
   destroyOSDCB = record
       pcount:   integer;
       pathName: gsosInStringPtr;
       end;

procedure DestroyGS (var parms: destroyOSDCB);
```

Possible Errors:	$10	devNotFound	Device not found
	$27	drvrIOError	I/O error
	$2B	drvrWrtProt	Device is write protected
	$40	badPathSyntax	Invalid path name syntax
	$44	pathNotFound	Subdirectory does not exist
	$45	volNotFound	Volume not found
	$46	fileNotFound	File not found
	$4B	badStoreType	Unsupported or incorrect storage type
	$4E	invalidAccess	Access not allowed
	$50	fileBusy	File is already open
	$52	unknownVol	Unknown volume type
	$58	notBlockDev	Not a block device
	$5A	damagedBitMap	Block number too large

This command deletes a file. If the file is an extended file, both the resource fork and the data fork are deleted.

It is possible to delete directories with this command, but only if all of the files in the directory have already been deleted.

This command will fail if the disk is write protected or if the delete access bit for the file is not enabled.

`pcount`	Must be 1.
`pathName`	Pointer to a GS/OS input string of type `gsosInString`. This parameter is the name of the file to create. See "Path Names" at the start of this section for details about the names.

OpenGS

```
type
   openOSDCB = record
       pcount:          integer;
       refNum:          integer;
       pathName:        gsosInStringPtr;
       requestAccess:   integer;
       resourceNumber:  integer;
       access:          integer;
       fileType:        integer;
       auxType:         longint;
       storageType:     integer;
       createDateTime:  timeField;
       modDateTime:     timeField;
       optionList:      optionListPtr;
       dataEOF:         longint;
       blocksUsed:      longint;
       resourceEOF:     longint;
       resourceBlocks:  longint;
       end;

procedure OpenGS (var parms: openOSDCB);
```

Possible Errors:	`$27`	`drvrIOError`	I/O error
	`$28`	`drvrNoDevice`	No device connected
	`$2E`	`drvrDiskSwitch`	The disk has been switched
	`$40`	`badPathSyntax`	Invalid path name syntax
	`$44`	`pathNotFound`	Subdirectory does not exist
	`$45`	`volNotFound`	Volume not found
	`$46`	`fileNotFound`	File not found
	`$4A`	`badFileFormat`	Version error (incompatible file type)
	`$4E`	`invalidAccess`	Access not allowed
	`$4F`	`buffTooSmall`	Buffer too small
	`$50`	`fileBusy`	File is already open
	`$52`	`unknownVol`	Unknown volume type
	`$58`	`notBlockDev`	Not a block device

This command opens a file. The open command returns a file reference number (`refNum`) which is used with many of the other GS/OS commands. You must open a file using this command and get a valid `refNum` before using any of the commands that require a `refNum` instead of a path name. In general, that's the commands that manipulate the file itself, like `ReadGS` and `WriteGS`.

In the process of opening the file, `OpenGS` can tell you a great deal about the file itself.

`pcount`	Any value from 2 to 15. This tells how many of the following parameters to use.
`refNum`	This value is returned; it is set to the reference number to use with subsequent GS/OS calls.
`pathName`	Pointer to a GS/OS input string of type `gsosInString`. This parameter is the name of the file to open. See "Path Names" at the start of this section for details about the names.

`requestAccess` This value must be set to 1, 2 or 3 before the call. A value of 1 opens the file with read only access. A value of 2 opens the file with write only access. A value of 3 opens the file for both reading and writing. If `pcount` is 2, GS/OS opens the file for both input and output.

`resourceNumber` If the file is an extended file and this parameter is set to 1, GS/OS opens the resource fork instead of the data fork. If the file is not an extended file or if this value is set to 0, the data fork is opened.

Resource forks should only be manipulated with Resource Manager calls, so this parameter should always be set to 0.

`access` This value is returned; it is the current setting for the access bits. See `CreateGS` for a description of these bits.

`fileType` This value is returned; it is the file type for the file.

`auxType` This value is returned; it is the auxiliary file type for the file.

`storageType` This value is returned; it is the storage type for the file. For a standard file (a file with just a data fork) this will be 1; for an extended file (a file with a resource fork, whether or not the data fork is empty) this will be 5; for a directory, this will be 13.

`createDateTime` This value is returned; it is the creation date and time for the file. See *Apple IIGS GS/OS Reference* for details.

`modDateTime` This value is returned; it is the date and time when the file was last modified. See *Apple IIGS GS/OS Reference* for details.

`optionList` You can pass a pointer to a GS/OS output string buffer or nil for this parameter. If you pass the address of a GS/OS output string, `OpenGS` returns some FST specific information in the buffer. See *Apple IIGS GS/OS Reference* for details.

`dataEOF` This value is returned; it is the number of bytes of information stored in the data fork. GS/OS can do some rudimentary file compression (it doesn't actually store blocks of zeros on the disk), so this value should not be used to see how much space is used on the disk. Instead, this is the amount of memory you need to reserve to load the entire data fork into memory.

`blocksUsed` This value is returned; it is the number of blocks of disk space needed to store the data fork of the file. This includes the data fork and blocks used to keep track of the file. On most devices used on the Apple IIGS, one block is 512 bytes.

Add this value to `resourceBlocks` to get the total number of disk blocks used by the file.

`resourceEOF` This value is returned; it is the number of bytes of information stored in the resource fork.

`resourceBlocks` This value is returned; it is the number of disk blocks needed to store the resource fork of the file.

Add this value to `blocksUsed` to get the total number of disk blocks used by the file.

ReadGS

```
type
   readWriteOSDCB = record
       pcount:         integer;
       refNum:         integer;
       dataBuffer:     ptr;
       requestCount:   longint;
       transferCount:  longint;
       cachePriority:  integer;
       end;

procedure ReadGS (var parms: readWriteOSDCB);
```

Possible Errors:			
	`$27`	`drvrIOError`	I/O error
	`$2E`	`drvrDiskSwitch`	The disk has been switched
	`$43`	`invalidRefNum`	Invalid reference number
	`$4C`	`eofEncountered`	End of file encountered
	`$4E`	`invalidAccess`	Access not allowed

`ReadGS` reads bytes from a file opened by `OpenGS`.

There are a number of options and strange ways you can use this command that are not covered here; see *Apple IIGS GS/OS Reference Manual* for details.

`pcount` This can be 4 or 5.

`refNum` Set this value to the `refNum` returned by the `OpenGS` call.

`dataBuffer` The bytes read from disk are placed in memory starting at this address.

`requestCount` Set this parameter to the number of bytes you wish to read.

It is possible to read the file in pieces. Opening the file sets things up so you start reading from the start of the file. If you only read part of the file, the next read command will pick up where the previous read command left off. There are also GS/OS commands (not covered in this appendix) that will let you start reading from a specific spot in the file.

`transferCount` This value is returned; it is the number of bytes actually read. It can be smaller than `requestCount` if the end of file was reached, if a disk error occurs, or when some GS/OS options not discussed here are in effect. If you are reading disk files as described in this course, though, and being careful not to ask for more bytes than are in the file, `transferCount` will always equal `requestCount` unless an error occurs.

`cachePriority` If you will be reading a file several times, you can use this parameter to speed up the process. If this value is 1, it instructs GS/OS to try to cache blocks from the file. Assuming there is enough memory available in the GS/OS cache buffer, the blocks from the file will still be in memory when you read them the next time. Use a value of 0 to tell GS/OS not to cache the file.

WriteGS

```
type
   readWriteOSDCB = record
       pcount:         integer;
       refNum:         integer;
       dataBuffer:     ptr;
       requestCount:   longint;
       transferCount:  longint;
       cachePriority:  integer;
       end;

procedure WriteGS (var parms: readWriteOSDCB);
```

Possible Errors:			
	`$27`	`drvrIOError`	I/O error
	`$28`	`drvrNoDevice`	No device connected
	`$2E`	`drvrDiskSwitch`	The disk has been switched
	`$40`	`badPathSyntax`	Invalid path name syntax
	`$43`	`invalidRefNum`	Invalid reference number
	`$48`	`volumeFull`	The volume is full
	`$4E`	`invalidAccess`	Access not allowed
	`$5A`	`damagedBitMap`	Block number too large

`WriteGS` writes to a file, transferring bytes from memory to the disk file. The parameters – and in fact, the type of the record passed – are identical to those used for the `ReadGS` call; see `ReadGS` for a complete description.

GS/OS Definitions

```
  type
     (* GS/OS class 1 input string *)
     gsosInString = record
         size:  integer;
     (* Change the array size as needed for your application *)
         theString:  packed array [1..254] of char;
         end;
     gsosInStringPtr = ^gsosInString;

     (* GS/OS class 1 output string *)
     gsosOutString = record
         maxSize:    integer;
         theString:  gsosInString;
         end;
     gsosOutStringPtr = ^gsosOutString;
```

LineEdit Tool Set

In ORCA/Pascal the header file for this tool is named LineEdit. This header file makes use of the Common unit. Both of these units must be listed in your `uses` statement to make calls listed in this section.

LECopy

```
procedure LECopy (LEHandle: leRecHndl);
```

Possible Errors: Memory Manager errors returned unchanged
QuickDraw II errors returned unchanged

Copies the selected text from the given line edit record into the line edit scrap. If no text is selected, the line edit scrap is emptied.

LECut

```
procedure LECut (LEHandle: leRecHndl);
```

Possible Errors: Memory Manager errors returned unchanged
QuickDraw II errors returned unchanged

Copies the selected text from the given line edit record into the line edit scrap, then deletes the characters from the line edit record and redraws the contents. If no text is selected, the line edit scrap is emptied, but the text in the control is not changed.

LEDelete

```
procedure LEDelete (LEHandle: leRecHndl);
```

Possible Errors: Memory Manager errors returned unchanged
QuickDraw II errors returned unchanged

Removes any selected characters from the given line edit record. If no text is selected, the call does nothing.

LEFromScrap

```
procedure LEFromScrap;
```

Possible Errors: `$1404` `leScrapErr` The desk scrap is too big to copy
Scrap Manager errors returned unchanged

Copies the text scrap from the Scrap Manager into the line edit scrap buffer.
The text edit scrap is not changed if the text scrap in the Scrap Manager's scrap is longer than 255 characters.

LEPaste

```
procedure LEPaste (LEHandle: leRecHndl);
```

Possible Errors: Memory Manager errors returned unchanged
QuickDraw II errors returned unchanged

If any text is selected, it is deleted. The text in the line edit scrap is then inserted at the insertion point.

LEToScrap

```
procedure LEToScrap;
```

Possible Errors: Scrap Manager errors returned unchanged

The text in the line edit scrap is copied into the Scrap Manager's scrap.

List Manager

In ORCA/Pascal the header file for this tool is named ListMgr. This header file makes use of the Common unit. Both of these units must be listed in your `uses` statement to make calls listed in this section.

DrawMember2

```
procedure DrawMember2 (itemnum: integer; theListCtl: ctlRecHndl);
```

Possible Errors: None

After your application changes a list that is being displayed in a list control, you should call `DrawMember2`. To draw a single list entry, pass the relative number of the entry (i.e., the first item in the list is 1, the next item is 2, and so on) for `itemnum`. To redraw the entire list, pass 0 for `itemnum`.

NewList2

```
procedure NewList2 (drawRtn: procPtr; listStart: integer; listRef: longint;
   listRefDesc, listSize: integer; theListCtl: ctlRecHndl);
```

Possible Errors: None

`NewList2` changes the list in a given list control. If there was already a list being displayed, the control forgets about the original list. It is up to the application to dispose of the original list, if that is appropriate.

For details about how to use resources for the list itself, see *Apple IIGS Toolbox Reference: Volume 3*, page 35-7.

`drawRtn` This is the address of a subroutine that will draw the list. This subroutine can be defined in Pascal, so long as the databank directive is used to reset the data bank register. The procedure should be declared with three parameters:

```
procedure DrawListEntry (r: rect; entry: listEntry;
   h: ctlRecHndl);
```

The first parameter is a rectangle surrounding the area where the list element should be drawn. `entry` is the list element to redraw; it can be declared as you see it, as a var parameter, or as a pointer to a list entry. `h` is the control handle for the list control.

The draw routine should take the highlight and selection state for the list entry into account.

	Pass nil to use the default list draw subroutine. Pass -1 to use the drawing routine that is already in place.
`listStart`	This is the number of the list element to display at the top of the list. Pass 1 to display the first member of the list, 2 to place the second list item at the top, and so forth.
`listRef`	A pointer, handle, or resource ID for the list to display.
`listRefDesc`	Controls the type of parameter passed for `listRef`. Pass 0 for a pointer, 1 for a handle, 2 for a resource ID, and -1 to leave the current list in place.
`listSize`	Number of elements in the list.
`theListCtl`	Control handle for the list control.

SortList2

```
procedure SortList2 (compareRtn: procPtr; theListCtl: ctlRecHndl);
```

Possible Errors: None

Sorts the elements in a list. Call `DrawMember2` after making this call to redraw the list.

`compareRtn` is a pointer to a compare procedure. Pass nil to use the default compare procedure, which sorts the list based on the leading string in each list element. For details about how to write your own compare subroutine, see *Apple IIGS Toolbox Reference Manual: Volume 1*, page 11-24. For information about exactly how characters are compared, see *Programmer's Reference to System 6.0*.

Loader

The loader is a tool documented in Apple IIGS GS/OS Reference; it is used to load programs. Most of the tool calls in this tool can't be used effectively from a high level language, so ORCA/Pascal does not have an interface file for this tool.

LGetPathname2

```
   function LGetPathname2 (userID, fileNum: integer): gsosInStringPtr;
      tool ($11, $22);
```

Possible Errors:	`$1101`	Entry not found
	`$1103`	Path name error

`LGetPathname2` returns a pointer to the full path name for an executing program. The path name is an internal name in a loader table, so it should not be modified in any way.

`userID` is the user ID for the program.

`fileNum` is the load file number for the program. This will always be 1 for an application running from a single executable file.

There is no interface file for the loader, so you must declare this call in your program. To declare the call, include the definition exactly as you see it, with the tool directive, in your program.

Memory Manager

In ORCA/Pascal the header file for this tool is named MemoryMgr. This header file makes use of the Common unit. Both of these units must be listed in your `uses` statement to make calls listed in this section.

DisposeHandle

```
procedure DisposeHandle (theHandle: handle);
```

Possible Errors:	`$0206`	`handleErr`	Invalid handle

Deallocates the memory associated with the handle, allowing the memory to be reused.

FindHandle

```
function FindHandle (memLocation: ptr): handle;
```

Possible Errors: None

`FindHandle` returns the handle for the block of memory that contains the byte pointed to by `memLocation`. `FindHandle` returns nil if the byte isn't in an allocated memory handle.

FreeMem

```
function FreeMem: longint;
```

Possible Errors: None

Returns the total amount of free memory, in bytes. This does not include memory that could be made free by purging.

The fact that this call says a certain number of bytes are free does not indicate that all of the memory can be allocated with a single `NewHandle` call, since the free memory is not all at one location.

HLock

```
procedure HLock (theHandle: handle);
```

Possible Errors:	`$0206`	`handleErr`	Invalid handle

Locks the handle. Once locked, the memory pointed to by the handle cannot be moved or purged.

HUnlock

```
procedure HUnLock (theHandle: handle);
```

Possible Errors:	`$0206`	`handleErr`	Invalid handle

Unlocks the handle. If the handle is moveable, unlocking the handle allows it to be moved. If the purge level for the handle is greater than zero, unlocking the handle allows the memory to be purged.

MaxBlock

```
function MaxBlock: longint;
```

Possible Errors: None

Returns the size, in bytes, of the largest block of free memory. Purging or compacting memory could increase this value.

NewHandle

```
function NewHandle (blockSize: longint; userID, memAttributes: integer;
   memLocation: univ ptr): handle;
```

Possible Errors:	`$0201`	`memErr`	Unable to allocate memory
	`$0204`	`lockErr`	Illegal operation on a locked or immovable block
	`$0207`	`idErr`	Invalid user ID

Allocates a blocks of memory and returns a handle to that block of memory.

`blockSize`	This is the amount of memory to set aside, in bytes.
`userID`	This is the user ID for the memory. In most cases, this is the program's memory ID, as returned by Pascal's `UserID` function.
`memAttributes`	This flags word controls the way the memory is allocated – where it comes from, whether it can be moved, and whether it can be purged.

15	`attrLocked`	If this bit is set, the handle will be initially locked; if this bit is clear, the handle will be unlocked.
14	`attrFixed`	This bit is set for fixed memory, and clear for moveable memory. Both bits 14 and 15 must be clear before the Memory Manager will actually move the memory.
13-10		Reserved; set to 0.
9-8	`attrPurge`	These two bits define the initial purge level, which can be 0 (00), 1 (01), 2 (10) or 3 (11).
7-5		Reserved; set to 0.
4	`attrNoCross`	If this bit is set, the Memory Manager will make sure that all of the memory is in a single bank of memory.
3	`attrNoSpec`	If this bit is set, the Memory Manager won't give you any special memory. Special memory includes most of the memory in banks $00, $01, $E0 and $E1.

2	`attrPage`	If this bit is set, the Memory Manager will make sure the chunk of memory it returns starts on a page boundary.
1	`attrAddr`	This flag forces the memory to be allocated at a particular location; specifically, wherever the last parameter, `location`, points.
0	`attrBank`	If this flag is set, the memory will come from a specific bank, but it can come from anywhere in the bank.

`memLocation`	If the `attrAddr` bit is set, this location is the address of the first byte of memory to be allocated; the Memory Manager will return an out of memory error if the memory is not available. If the `attrBank` bit is set, the memory will be allocated from the same bank as this location, but can be allocated from any position within that bank.

RealFreeMem

```
function RealFreeMem: longint;
```

Possible Errors: None

Returns the total amount of memory that is currently free, plus the amount of memory that could be freed by purging.

SetHandleID

```
function SetHandleID (newID: integer; theHandle: handle): integer;
```

Possible Errors: None

`SetHandleID` changes the user ID for the specified handle to `newID`, returning the old user ID in the process. Pass 0 for `newID` to read the user ID for a handle without changing the handle's current user ID.

SetPurge

```
procedure SetPurge (purgeLevel: integer; theHandle: handle);
```

Possible Errors: `$0206` `handleErr` Invalid handle

Set the purge level for the given handle. The purge level should be a value from 0 to 3. Handles with a purge level of 0 cannot be purged. Handles with any other purge level may be deallocated if the handle is unlocked and the Memory Manager can't allocate enough memory for a `NewHandle` call without purging.

If memory is purged, handles with a purge level of 3 are purged first, then handles with a purge level of 2, and finally those with a purge level of 1.

A purge level of 3 is reserved for use by the System Loader.

Memory Manager Definitions

```
 type
    ptr        = ^byte;
    handle     = ^ptr;
```

Menu Manager

In ORCA/Pascal the header file for this tool is named MenuMgr. This header file makes use of the Common unit. Both of these units must be listed in your `uses` statement to make calls listed in this section.

CheckMItem

```
procedure CheckMItem (checkedFlag: boolean; itemNum: integer);
```

Possible Errors: None

If `checkedFlag` is true, a check mark is placed beside the menu item specified by `itemNum`. If `checkedFlag` is false, any character to the left of menu item `itemNum` is erased.

DisableMItem

```
procedure DisableMItem (itemNum: integer);
```

Possible Errors: None

Menu item `itemNum` is dimmed. The item can't be selected from the menu until it has been enabled by a call to `EnableMItem`.

DrawMenuBar

```
procedure DrawMenuBar;
```

Possible Errors: None

Draws the current menu bar, along with the titles for any menus in the menu bar.

EnableMItem

```
procedure EnableMItem (itemNum: integer);
```

Possible Errors: None

Menu item `itemNum` is enabled. The menu item is drawn normally the next time the menu is pulled down, and the menu item can be selected.

See also `DisableMItem`.

FixMenuBar

```
function FixMenuBar: integer;
```

Possible Errors: None

Computes the sizes of the menu bar and the menus in the menu bar. The height of the menu bar depends on the fonts used in the titles of the various menus in the menu bar. The widths of each menu title are also calculated. Finally, `CalcMenuSize` is called for each of the menus in the menu bar.

If you don't make this call, it is quite possible that your program will crash the first time you use the menu bar.

`FixMenuBar` returns the height of the menu bar in pixels.

GetMenuFlag

```
function GetMenuFlag (menuNum: integer): integer;
```

Possible Errors: None

Returns the menu flags for menu `menuNum`. See `SetMenuFlag` for a description of the menu flags returned by this call.

HiliteMenu

```
procedure HiliteMenu (hiliteFlag: boolean; menuNum: integer);
```

Possible Errors: None

Highlights or unhighlights menu `menuNum`. Set `hiliteFlag` to true to highlight the menu, or to false to unhighlight the menu.

When a menu item is selected by the user, either though a keyboard command identified by `MenuKey` or through a mouse down handled by `MenuSelect`, the Menu Manager highlights the menu containing the selected command. According to the Apple human interface guidelines, the menu should stay highlighted until your program finishes executing the command. Once your program finishes executing the command, you should call `HiliteMenu` with the appropriate menu number and with `hiliteFlag` set to false.

InsertMenu

```
procedure InsertMenu (addMenu: menuHandle; insertAfter: integer);
```

Possible Errors: None

Inserts the menu `addMenu` into the current menu bar. If `insertAfter` is 0, the menu is placed to the left of all of the menus currently in the menu bar. If `insertAfter` is not zero, the new menu is inserted to the right of the menu whose ID is specified by `insertAfter`.

After making this call, you need to call `FixMenuBar` and `DrawMenuBar` before you enter your event loop.

MenuKey

```
procedure MenuKey (var theTask: eventRecord; theMenuBar: ctlRecHndl);
```

Possible Errors: None

When you find a key down event in your event loop and the command key is being held down, make a call to `MenuKey` to see if the keypress is a menu command key equivalent. When you call `MenuKey`, you pass an event record; this is generally the same event record you used when you called `GetNextEvent`. You also pass the handle of the menu bar you want `MenuKey` to check; in general, you pass nil to indicate the system menu bar.

If the keystroke is a menu command key equivalent, the menu bar containing the correct menu item becomes the current menu bar, the menu containing the menu item selected is highlighted, and the task record is modified. The `taskData` field of the event record is set so the least significant word is menu item ID, while the most significant word holds the menu ID for the menu containing the selected menu item.

If the keystroke is not a menu command key equivalent, the least significant word of the `taskData` field is set to 0.

If there is more than one menu item with the same keyboard equivalent, `MenuKey` returns the menu item for the first one it finds. Of course, you shouldn't have two menu items with the same keyboard equivalent.

MenuSelect

```
procedure MenuSelect (var theTask: eventRecord; theMenuBar: ctlRecHndl);
```

Possible Errors: None

When you detect a mouse down event in the menu bar, you should call `MenuSelect` to handle the event. (Mouse down events in the menu bar are generally detected by a call to `FindWindow`.) `MenuSelect` tracks the mouse's movements, pulling down menus and highlighting menus and menu items as appropriate. When the mouse is released, `MenuSelect` figures out if a menu command has been selected. If so, the menu item number for the selected menu command is returned in the least significant word of the `taskData` field of `theTask`, and the menu number for the menu that contains the menu item is returned in the most significant word of the `taskData` field. If no command was selected, the least significant word of the `taskData` field is set to zero.

When you call `MenuSelect`, you pass an event record and a menu bar handle. To track a mouse down in the system menu bar, pass nil for `theMenuBar`. The event record you pass should generally be the same one returned by `GetNextEvent`. `MenuSelect` uses some of the fields in the record as inputs, and returns information to you in the `taskData` field.

NewMenu

```
function NewMenu (newMenuString: textPtr): menuHandle;
```

Possible Errors: None

You pass a string defining a menu and the items in a menu, and `NewMenu` returns a handle to a menu that you can then use as an input to `InsertMenu`.

The menu string consists of a series of lines, separated from one another by null characters (chr(0)) or return characters (chr(13)). The first line defines the menu title. This is followed by one or more lines defining menu items. The last line is used solely to mark the end of the menu string.

Each line starts with one or two control characters. The initial menu line starts with two characters, generally `'>>'`. While both of these characters must be the same, you can use any character you like. Menu items also start with two characters, generally `'--'`. Once again, you can pick any character you like, so long as you use the same two characters before each menu item. The last line consists of a single character, generally `'.'`. This character cannot be the same character used to start the menu title line or the menu item lines. This single character is the only character on the last line of the menu string.

For menu items and menus, the initial two characters are followed by the characters making up the title of the menu or menu item. All characters up to the end of the line or the first `'\'` character become a part of the menu title or item string.

A menu name of `'@'` has a special meaning. When this character is the only character in a menu title, the Menu Manager draws a rainbow apple instead of printing the @ character.

A menu item with a title of `'-'` has special meaning. Instead of writing the – character as the name of the menu item, the Menu Manager draws a line across the entire menu. Lines of this type are generally disabled, and are used as separators to visually divide the commands in a complex menu into logical groups.

The menu title or menu item title can be followed by a backslash character (\), which is followed by one or more control codes. The allowed control codes are shown in Table A-3.

The Menu string must remain in a fixed memory location, and must not be changed by the application after the call to `NewMenu`.

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 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 chr(18).
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. From Pascal, it's generally best to use N, instead.
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.
V	Place a dividing line under the item.
X	Use color replace mode to avoid green-apple menu sickness.

Table A-3: Menu String Control Characters

NewMenuBar2

```
function NewMenuBar2 (refDescriptor: integer; menuBarTRef: longint;
   theWindow: grafPortPtr): menuBarHandle;
```

Possible Errors: None

`NewMenuBar2` creates a menu bar in a single step using a menu template.

`refDescriptor` defines the type of parameter that will be used for `menuBarTRef`. The possible values for `refDescriptor` are:

value	name	meaning
0	`refIsPointer`	`menuBarTRef` is a pointer to a menu bar template. In this course, we always create the menu bar template using resources, so the format for the template is not covered. See *Apple IIGS Toolbox Reference: Volume 3* for complete information about menu templates.
1	`refIsHandle`	`menuBarTRef` is a handle to a menu bar template.
2	`refIsResource`	`menuBarTRef` is the resource ID for an `rMenuBar` resource. See Appendix B for a description of the `rMenuBar` resource.

`theWindow` is a pointer to the window where the menu bar will appear. The menu bar will be lined up with the top left corner of the window, and will be as wide as the screen. Pass nil for this parameter if you a creating a system menu bar.

While this call can create a new system menu bar, it doesn't make the menu bar the current system menu bar. To create a new system menu bar, follow this call with calls to `SetSysBar` and `SetMenuBar`, like this:

```
    menuBarHand := NewMenuBar2(refIsResource, menuID, nil);
    SetSysBar(menuBarHand);
    SetMenuBar(nil);
```

SetMenuBar

```
procedure SetMenuBar (theBarHandle: ctlRecHndl);
```

Possible Errors: None

`SetMenuBar` makes the menu bar whose handle is `theBarHandle` the current menu bar. If you want to make the system menu bar the current menu bar, pass nil.

SetMenuFlag

```
procedure SetMenuFlag (newValue, menuNum: integer);
```

Possible Errors: None

Sets the flags that control the state of the menu title. Call `DrawMenuBar` after this call to redraw the menu bar in the new state.

Table A-4 shows the flags you can set with `SetMenuFlag`, along with the names reserved for the flags. To change a flag, start by calling `GetMenuFlag` to get the current flags. For values that start with a zero, you should or the value with the value returned by `GetMenuFlag` and pass the result as the `newValue` parameter. This sets the flag in the flags word. Values that start with an F are anded with the original value, turning the flag off.

The `menuNum` parameter is the menu ID number of the menu you want to change.

Name	Value	Description
`enableMenu`	`$FF7F`	The menu will be selectable, and will not be dimmed.
`disableMenu`	`$0080`	The menu and all items in the menu will be dimmed and cannot be selected.
`colorReplace`	`$FFDF`	Draw the title and background in the normal highlighted state.
`xorTitleHilite`	`$0020`	Highlight the title and background using XOR color replace.
`standardMenu`	`$FFEF`	The menu is a standard menu.
`customMenu`	`$0010`	The menu is a custom menu.

Table A-4: Menu Flags

SetMItem

```
procedure SetMItem (newItem: cStringPtr; itemNum: integer);
```

Possible Errors: None

Change the title of menu item `itemNum` to the title string `newItem`. The title string must remain in a fixed location in memory. Like a menu item string as passed to `NewMenu`, this menu string must start with two characters that are not a part of the menu item title, and the string must end in either a null character (chr(0)) or a return character (chr(13)). Unlike the menu item string passed to `NewMenu`, the new name of the menu item is not followed by control characters. All of the information normally set by control characters is copied from the existing menu item.

SetMItemStyle

```
procedure SetMItemStyle (textStyle, itemNum: integer);
```

Possible Errors: None

Sets the text style for a menu item. The `textStyle` parameter is a flags word with the font styles layed out as follows:

bits 15-5	Reserved; set to 0.
bit 4	Set for shadowed text.
bit 3	Set for outlined text.
bit 2	Set for underlined text.
bit 1	Set for italicized text.
bit 0	Set for bold text.

SetSysBar

```
procedure SetSysBar (theBarHandle: ctlRecHndl);
```

Possible Errors: None

`SetSysBar` makes the menu bar whose handle is `theBarHandle` the current, system menu bar.

Menu Manager Definitions

```
const
   {Inputs to SetMenuFlag routine}
   customMenu      =   $0010;      {menu is a custom menu}
   xorTitleHilite  =   $0020;      {menu title will be XORed to highlighted}
```

```
                                   {  state                          }
   disableMenu     =   $0080;      {menu will be dimmed and not selectable}
   enableMenu      =   $FF7F;      {menu will not be dimmed; is selectable}
   colorReplace    =   $FFDF;      {menu title and background will be}
                                   {  redrawn and highlighted       }
   standardMenu    =   $FFEF;      {menu considered a standard menu}

type
   {Null-terminated string}
   cString = packed array [1..256] of char;
   cStringPtr = ^cString;

   {Menu strings.  The size of the array can be changed for long menus.}
   textBlock = packed array [1..300] of char;
   textPtr = ^textBlock;
```

Miscellaneous Tool Set

In ORCA/Pascal the header file for this tool is named MscToolSet. This header file makes use of the Common unit. Both of these units must be listed in your `uses` statement to make calls listed in this section.

ReadAsciiTime

```
procedure ReadASCIITime (bufferAddress: ptr);
```

Possible Errors: None

The current time is returned in the buffer, which should be at least 20 characters long. The time is returned as a series of 20 characters with the most significant bit set. There is no length word or trailing null character.

In most cases, you will have `ReadASCIITime` stuff the characters into a string, then set a length byte (for a p-string) or a trailing null character (for a C-string) manually. Anding all 20 characters with $7F will clear the most significant bit. These steps, combined, create a true ASCII string.

There are several time formats available. The various time formats are selected from the control panel, so they aren't under the control of this call. For details, see *Apple IIGS Toolbox Reference: Volume 1*, page 14-16.

SysBeep

```
procedure SysBeep;
```

Possible Errors: None

Beeps the speaker.

SysFailMgr

```
procedure SysFailMgr (errorCode: integer; failString: univ pStringPtr);
```

Possible Errors: None

Stops the computer cold, forcing a reboot. The screen switches to a text display with a sliding apple. The first parameter is printed as a four digit hexadecimal value; this error number follows the error message. The second parameter is either nil, which displays a default message ("FATAL

SYSTEM ERROR") or a pointer to a p-string; this string is displayed on the screen. Because of the way the parameter is defined in the ORCA/Pascal interfaces, you can pass a pointer to a string, the address of a string constant, the address of a string variable, or the name of a string variable.

Note Synthesizer

In ORCA/Pascal the header file for this tool is named Synthesizer. This header file makes use of the Common unit. Both of these units must be listed in your `uses` statement to make calls listed in this section.

AllNotesOff

```
procedure AllNotesOff;
```

Possible Errors: None

Turns off all notes started using the Note Synthesizer.

AllocGen

```
function AllocGen (requestPriority: integer): integer;
```

Possible Errors:			
	`$1921`	`nsNotAvail`	No generators are available
	`$1923`	`nsNotInit`	The Note Synthesizer has not been started

`AllocGen` looks for an available sound generator, or, of all sound generators are in used, one with a lower priority than `requestPriority`. If it finds a sound generator, it returns the generator number, which will be in the range 0 to 13. If no generators are available, `AllocGen` returns an error.

NoteOff

```
procedure NoteOff (genNum, semitone: integer);
```

Possible Errors: None

Switches a note started with `NoteOn` to it's release mode, where it will decay quickly to a zero volume. At that point, the priority for the generator is set to zero so the generator can be reallocated by `AllocGen`.

The `genNum` and `semitone` parameters should be the same ones passed earlier to `NoteOn`.

NoteOn

```
procedure NoteOn (genNum, semitone, volume: integer;
   var theInstrument: instrument);
```

Possible Errors:			
	`$1924`	`nsGenAlreadyOn`	The specified note is already being played

Starts playing a note.

`genNum`	This is the sound generator number to use. It should be allocated by a call to `AllocGen` before calling `NoteOn`.

`semitone`	This is the note number, ranging from 0 to 127. Middle C is 60. Increasing or decreasing the note by 12 changes it by a full octave. (Physically, that changes the frequency by a factor of two.)
`volume`	The volume, which can range from 0 to 127. Changing the volume by 16 is equivalent to a change in the sound level of 6 decibels.
`theInstrument`	This is the instrument record. See Lesson 16 for a complete description of this record and its use.

Note Synthesizer Definitions

```
type
  waveForm = record
    topKey:      byte;
    waveAddress: byte;
    waveSize:    byte;
    DOCMode:     byte;
    relPitch:    integer;
    end;

  instrument = record
    envelope:          array [1..24] of byte;
    releaseSegment:    byte;
    priorityIncrement: byte;
    pitchBendRange:    byte;
    vibratoDepth:      byte;
    vibratoSpeed:      byte;
    spare:             byte;
    aWaveCount:        byte;
    bWaveCount:        byte;
    aWaveList:         array [1..1] of waveForm; (* aWaveCount * 6 bytes *)
    bWaveList:         array [1..1] of waveForm; (* bWaveCount * 6 bytes *)
    end;
```

Print Manager

In ORCA/Pascal the header file for this tool is named PrintMgr. This header file makes use of the Common unit. Both of these units must be listed in your `uses` statement to make calls listed in this section.

PrCloseDoc

```
procedure PrCloseDoc (printerPort: grafPortPtr);
```

Possible Errors:	`$1302`	`portNotOn`	The specified port is not selected in the control panel

Closes the printer's grafPort. This call is normally used at the end of a print loop.

PrClosePage

```
procedure PrClosePage (printerPort: grafPortPtr);
```

Possible Errors:	`$1302`	`portNotOn`	The specified port is not selected in the control panel

Closes the printing of the current page. This call is normally used after every page is printed. `PrClosePage` and `PrOpenPage` are used in pairs inside the print loop.

PrDefault

```
procedure PrDefault (thePrintRecord: prHandle);
```

Possible Errors:	`$1303`	`noPrintRecord`	No print record was specified

Fills in the print record with default information for the current printer. The current printer is selected by the user using the printer chooser control panel device. This call is normally used to fill in a new print record.

PrError

```
function PrError: integer;
```

Possible Errors: None

Checks for errors flagged during the print loop. This call is normally made right after a print loop, and just before spooled printing.

PrJobDialog

```
function PrJobDialog (thePrintRecord: prHandle): integer;
```

Possible Errors: Memory Manager errors returned unchanged

Brings up a dialog that lets the user pick options related to a single printing of a document, like the number of copies to print and the print quality to use. This call is normally made when the user selects the Print command, just before entering the print loop.

The value returned will be 0 if the user canceled the print operation, and 1 if the user confirmed the print operation.

PrOpenDoc

```
function PrOpenDoc (thePrintRecord: prHandle; printerPort: grafPortPtr):
   grafPortPtr;
```

Possible Errors:	`$1302`	`portNotOn`	The specified port is not selected in the control panel
	`$1304`	`badLaserPrep`	The version of LaserPrep in the drivers folder is not compatible with this version of the Print Manager
	`$1305`	`badLPFile`	The version of LaserPrep in the drivers folder is not compatible with this version of the Print Manager
	`$1306`	`papConnNotOpen`	Connection can't be established with the LaserWriter
	`$1307`	`papReadWriteErr`	Read-write error on the LaserWriter
	Memory Manager errors returned unchanged		
	GS/OS errors returned unchanged		

This call is used at the start of a print loop to allocate a `grafPort` for use by the Print Manager. The first parameter is the print record, and the second is normally set to nil, telling the Print Manager to create a new grafPort. For other possibilities, see *Apple IIGS Toolbox Reference: Volume 1*, page 15-36.

PrOpenPage

```
procedure PrOpenPage (printerPort: grafPortPtr; pageFrame: rectPtr);
```

Possible Errors:	`$1302`	`portNotOn`	The specified port is not selected in the control panel
	Memory Manager errors returned unchanged		

This call is used in the print loop to set up a new output page. The first parameter is the `grafPort` set up by `PrOpenDoc`; the second is normally nil, but can be used as a scaling rectangle. For details on scaling rectangles, see *Apple IIGS Toolbox Reference Manual: Volume 1*, page 15-38.

PrPicFile

```
procedure PrPicFile (thePrintRecord: prHandle; printerPort: grafPortPtr;
   statusRecPtr: PrStatusPtr);
```

Possible Errors:	`$1302`	`portNotOn`	The specified port is not selected in the control panel
	Memory Manager errors returned unchanged		

Prints a spooled document. This call is made after the print loop. The call does not cause problems, even if the document is not spooled, so it is normally used routinely, without actually checking to see if the document is spooled.

The first parameter is the print record used throughout the print loop.

The second parameter is a `grafPort` to use while doing the spooled printing. This is not the same grafPort that was allocated by `PrOpenDoc`; that `grafPort` was disposed of when `PrCloseDoc` was called to finish the print loop. Normally, you will pass nil, telling the Print

Manager to allocate a `grafPort` specifically for this call. For other possibilities, see *Apple IIGS Toolbox Reference Manual: Volume 1*, page 15-40.

The last parameter is a status record that is constantly updated to reflect the print loop status. Other than creating a variable and passing it to `PrPicFile`, you can ignore this record. For details on the format and use of the record, see see *Apple IIGS Toolbox Reference Manual: Volume 1*, page 15-40.

PrStlDialog

```
function PrStlDialog (thePrintRecord: prHandle): boolean;
```

Possible Errors: Memory Manager errors returned unchanged

`PrStlDialog` brings up a dialog that lets the user pick document-related printer options, like the page layout. This call is normally made when the user picks the Page Setup command from the File menu. You should pass a print record that has already been filled in by either `PrDefault` or `PrValidate`. If possible, the print record should be saved with the document, and each document should have it's own private copy of the print record.

If the user confirms the dialog, `PrStlDialog` returns true; if the user picks cancel, `PrStlDialog` returns false and does not modify the print record in any way.

PrValidate

```
function PrValidate (thePrintRecord: prHandle): boolean;
```

Possible Errors: Memory Manager errors returned unchanged

`PrValidate` checks an existing print record to make sure it is consistent with the current printer, making appropriate changes if not. This call is normally used to validate a print record that is loaded with a document. It returns true if changes were made to the print record, and false if no change was needed.

Print Manager Definitions

```
type
   (* Printer information subrecord *)
   prInfoRec = record
       iDev:  integer;
       iVRes: integer;
       iHRes: integer;
       rPage: rect;
       end;

   (* Printer style subrecord *)
   prStyleRec = record
       wDev:      integer;
       internA:   array [0..2] of integer;
       feed:      integer;
       paperType: integer;
       case boolean of
           true:  (crWidth:   integer;);
           false: (vSizing:   integer;
                  reduction: integer;
                  internB:   integer;);
           end;
```

```
(* Job information subrecord *)
prJobRec = record
    iFstPage:  integer;
    iLstPage:  integer;
    iCopies:   integer;
    bJDocLoop: byte;
    fFromUser: byte;
    pIdleProc: procPtr;
    pFileName: pathPtr;
    iFileVol:  integer;
    bFileVers: byte;
    bJobX:     byte;
    end;

(* Print record *)
PrRec = record
    prVersion: integer;
    prInfo:    prInfoRec;
    rPaper:    rect;
    prStl:     prStyleRec;
    prInfoPT:  array [0..13] of byte;
    prXInfo:   array [0..23] of byte;
    prJob:     PrJobRec;
    printX:    array [0..37] of byte;
    iReserved: integer;
    end;
PrRecPtr = ^PrRec;
PrHandle = ^PrRecPtr;

(* Printer status subrecord *)
PrStatusRec = record
    iTotPages:  integer;
    iCurPage:   integer;
    iTotCopies: integer;
    iCurCopy:   integer;
    iTotBands:  integer;
    iCurBand:   integer;
    fPgDirty:   boolean;
    fImaging:   integer;
    hPrint:     prHandle;
    pPrPort:    grafPortPtr;
    hPic:       longint;
    end;
PrStatusPtr = ^PrStatusRec;
```

QuickDraw II

In ORCA/Pascal the header file for this tool is named QuickDrawII. This header file makes use of the Common unit. Both of these units must be listed in your `uses` statement to make calls listed in this section.

LocInfo Records

```
type
    locInfo = record
        portSCB:          integer;
        ptrToPixelImage:  ptr;
        width:            integer;
        boundsRect:       rect;
        end;
```

The `locInfo` record contains information about a bit mapped drawing area. In this course, this record is used for drawing pictures with the `PPToPort` call, but the record is used widely in the toolbox and window manager for other purposes, too.

`portSCB`	The scan line control byte for the image.
`ptrToPixelImage`	Points to the picture. The picture is organized as a series of lines, each line being made up of 2 bit pixels or 4 bit pixels, depending on `portSCB`.
`width`	Width in bytes of each line in the image. The number of pixels in the image must be an even multiple of 8.
`boundsRect`	A rectangle enclosing the entire image.

Scan Line Control Byte

The scan line control byte (SCB) defines the screen resolution for a scan line or pixel image. Only one bit is used in this course. If the most significant bit is set, the SCB refers to 640 mode graphics; if the bit is clear, the SCB refers to 320 mode graphics. All other bits should be set to 0.

For information about the other bits in the SCB, see *Apple IIGS Toolbox Reference: Volume 2*, page 16-34.

CharBounds

```
procedure CharBounds (theChar: char; var result: rect);
```

Possible Errors: None

`CharBounds` sets the rectangle `result` to the character bounds rectangle for the character `theChar`.

The character bounds rectangle is basically the rectangle that extends from the ascent to descent lines, and from the character origin on the left to the new character origin on the right. If kerning is used, though, the pixels in the character could extend past the character origin on either the left or right, and in that case, the bounds rectangle is extended to include all of the pixels in the character.

CharWidth

```
function CharWidth (theChar: char): integer;
```

Possible Errors: None

`CharWidth` returns with width of the character in pixels.

ClipRect

```
procedure ClipRect (var theRect: rect);
```

Possible Errors: Memory Manager errors returned unchanged

Sets the clip region for the current `grafPort` to the rectangle given.

ClosePoly

```
procedure ClosePoly;
```

Possible Errors:	`$0441`	`polyNotOpen`	No polygon is open
	Memory Manager errors returned unchanged		

Finishes the definition of the polygon started with `OpenPoly`.

CStringBounds

```
procedure CStringBounds (theCString: univ CStringPtr; var result: rect);
```

Possible Errors: None

Returns the bounds rectangle for the given string of characters. See `CharBounds` for a definition of the bounds rectangle.

CStringWidth

```
function CStringWidth (theCString: univ CStringPtr): integer;
```

Possible Errors: None

Returns the width of the string in pixels. This is equivalent to the pen displacement if the string is drawn.

DrawChar

```
procedure DrawChar (theChar: char);
```

Possible Errors: None

Draws the character, using the current font, text mode, foreground color and background color. The character is drawn to the current `grafPort` at the current pen location; the pen is then moved to the right by the width of the character.

See `CharWidth` for one way to find the distance the pen will be advanced.

DrawCString

```
procedure DrawCString (theString: univ CStringPtr);
```

Possible Errors: None

Draws the string, using the current font, text mode, foreground color and background color. The string is drawn to the current `grafPort` at the current pen location; the pen is then moved to the right by the width of the string.

See `CStringWidth` for one way to find the distance the pen will be advanced.

DrawString

```
procedure DrawString (theString: univ pStringPtr);
```

Possible Errors: None

Draws the string, using the current font, text mode, foreground color and background color. The string is drawn to the current `grafPort` at the current pen location; the pen is then moved to the right by the width of the string.

See `StringWidth` for one way to find the distance the pen will be advanced.

DrawText

```
procedure DrawText (theText: univ textPtr; textLen: integer);
```

Possible Errors: None

Draws the block of text, using the current font, text mode, foreground color and background color. The text is drawn to the current `grafPort` at the current pen location; the pen is then moved to the right by the width of the text.

See `TextWidth` for one way to find the distance the pen will be advanced.

EraseArc

```
procedure EraseArc (var theRect: rect; startAngle, arcAngle: integer);
```

Possible Errors: None

Erases the interior of the arc, painting the area with the background pattern.

EraseOval

```
procedure EraseOval (var theRect: rect);
```

Possible Errors: None

Erases the interior of the oval, painting the area with the background pattern.

ErasePoly

```
procedure ErasePoly (thePolyHandle: polyHandle);
```

Possible Errors: Memory Manager errors returned unchanged

Erases the interior of the polygon, painting the area with the background pattern.

EraseRect

```
procedure EraseRect (var theRect: rect);
```

Possible Errors: None

Erases the interior of the rectangle, painting the area with the background pattern.

EraseRRect

```
procedure EraseRRect (var theRect: rect; ovalWidth, ovalHeight: integer);
```

Possible Errors: None

Erases the interior of the rounded rectangle, painting the area with the background pattern.

FillArc

```
procedure FillArc (var theRect: rect; startAngle, arcAngle: integer;
   var thePattern: pattern);
```

Possible Errors: None

Fills the interior of the arc with the given pattern.

FillOval

```
procedure FillOval (var theRect: rectPtr; var thePattern: pattern);
```

Possible Errors: None

Fills the interior of the oval with the given pattern.

FillPoly

```
procedure FillPoly (thePolyHandle: polyHandle; var thePattern: pattern);
```

Possible Errors: Memory Manager errors returned unchanged

Fills the interior of the polygon with the given pattern.

FillRect

```
procedure FillRect (var theRect: rect; var thePattern: pattern);
```

Possible Errors: None

Fills the interior of the rectangle with the given pattern.

FillRRect

```
procedure FillRRect (var theRect: rect; ovalWidth, ovalHeight: integer;
   var thePattern: pattern);
```

Possible Errors: None

Fills the interior of the rounded rectangle with the given pattern.

FrameArc

```
procedure FrameArc (var theRect: rect; startAngle, arcAngle: integer);
```

Possible Errors: None

Outlines the rounded portion of the given arc using the current pen mode, pattern and size.

FrameOval

```
procedure FrameOval (var theRect: rect);
```

Possible Errors: Memory Manager errors returned unchanged

Outlines the given oval using the current pen mode, pattern and size.

FramePoly

```
procedure FramePoly (thePolyHandle: polyHandle);
```

Possible Errors: Memory Manager errors returned unchanged

Outlines the given polygon using the current pen mode, pattern and size.

FrameRect

```
procedure FrameRect (var theRect: rect);
```

Possible Errors: Memory Manager errors returned unchanged

Outlines the given rectangle using the current pen mode, pattern and size.

FrameRRect

```
procedure FrameRRect (var theRect: rect; ovalWidth, ovalHeight: integer);
```

Possible Errors: Memory Manager errors returned unchanged

Outlines the given rounded rectangle using the current pen mode, pattern and size.

GetColorEntry

```
function GetColorEntry (tableNumber, entryNumber: integer): integer;
```

Possible Errors:			
	`$0450`	`badTableNum`	Invalid color table number
	`$0451`	`badColorNum`	Invalid color number

Returns the color table entry for the specified color table number and color table entry number. There are 16 color tables, numbered 0 to 15, and 16 color entries in each color table, also numbered 0 to 15.

GetColorTable

```
procedure GetColorTable (tableNumber: integer; var saveTable: colorTable);
```

Possible Errors:			
	`$0450`	`badTableNum`	Invalid color table number

Fills in the color table with the colors in one of the screen color tables. There are 16 screen color tables, numbered 0 to 15.

GetFontInfo

```
procedure GetFontInfo (var theFIRec: fontInfoRecord);
```

Possible Errors: None

Returns information about the current font in the given font information record.

GetPort

```
function GetPort: grafPortPtr;
```

Possible Errors: None

Returns a pointer to the current `grafPort`.

GetPortRect

```
procedure GetPortRect (var theRect: rect);
```

Possible Errors: None

Returns the current `grafPort`'s port rectangle.

GetVisHandle

```
function GetVisHandle: rgnHandle;
```

Possible Errors: None

Returns the handle for the visible region of the current `grafPort`. The handle belongs to the system, and should not be modified in any way.

GlobalToLocal

```
procedure GlobalToLocal (var thePoint: point);
```

Possible Errors: None

Converts the specified point from global coordinates to local coordinates.

InitCursor

```
procedure InitCursor;
```

Possible Errors: None

Sets the mouse cursor to an arrow and makes the cursor visible. This call is used at the start of a program to make the mouse cursor visible, and to change back to the arrow cursor in a program after you have switched to some other cursor.

InvertArc

```
procedure InvertArc (var theRect: rect; startAngle, arcAngle: integer);
```

Possible Errors: None

Inverts the interior of the specified arc.

InvertOval

```
procedure InvertOval (var theRect: rect);
```

Possible Errors: None

Inverts the interior of the specified oval.

InvertPoly

```
procedure InvertPoly (thePolyHandle: polyHandle);
```

Possible Errors: Memory Manager errors returned unchanged

Inverts the interior of the specified polygon.

InvertRect

```
procedure InvertRect (var theRect: rect);
```

Possible Errors: None

Inverts the interior of the specified rectangle.

InvertRRect

```
procedure InvertRRect (var theRect: rect; ovalWidth, ovalHeight: integer);
```

Possible Errors: None

Inverts the interior of the specified rounded rectangle.

KillPoly

```
procedure KillPoly (thePolyHandle: polyHandle);
```

Possible Errors: Memory Manager errors returned unchanged

Destroys the given polygon, disposing of all memory associated with the polygon.

LineTo

```
procedure LineTo (h, v: integer);
```

Possible Errors: Memory Manager errors returned unchanged

Draws a line from the current pen location to the point passed as parameters to `LineTo`. The line is drawn with the current pen pattern, mode and size.

LocalToGlobal

```
procedure LocalToGlobal (var thePoint: point);
```

Possible Errors: None

Converts the specified point from local coordinates to global coordinates.

MoveTo

```
procedure MoveTo (h, v: integer);
```

Possible Errors: None

Moves the drawing pen to the specified point. Nothing is drawn on the screen.

OffsetPoly

```
procedure OffsetPoly (thePolyHandle: polyHandle; dH, dV: integer);
```

Possible Errors: Memory Manager errors returned unchanged

Offsets a polygon, effectively moving the entire shape. `dH` is the distance to move in the horizontal direction, while `dV` is the distance to move in the vertical direction. Positive values move the polygon down and to the right, while negative values shift the polygon up and to the left.

OpenPoly

```
function OpenPoly: polyHandle;
```

Possible Errors:	`$0440`	`polyAlreadyOpen`	Polygon is already open
	Memory Manager errors returned unchanged		

`OpenPoly` starts the process of defining a polygon. It returns the handle for a new polygon. All subsequent calls to `LineTo` will add a line to the edge of the polygon, up until the matching `ClosePoly` call, which ends the definition of the polygon.

If the last point in the polygon does not match the original starting point, an extra line is automatically added to close the shape.

When you are finished with a polygon, call `KillPoly` to dispose of the memory associated with this structure.

PaintArc

```
procedure PaintArc (var theRect: rect; startAngle, arcAngle: integer);
```

Possible Errors: None

Paints all points inside the specified arc using the current pen pattern and pen mode.

PaintOval

```
procedure PaintOval (var theRect: rect);
```

Possible Errors: None

Paints all points inside the specified oval using the current pen pattern and pen mode.

PaintPoly

```
procedure PaintPoly (thePolyHandle: polyHandle);
```

Possible Errors: Memory Manager errors returned unchanged

Paints all points inside the specified polygon using the current pen pattern and pen mode.

PaintRect

```
procedure PaintRect (var theRect: rect);
```

Paints all points inside the specified rectangle using the current pen pattern and pen mode.

PaintRRect

```
procedure PaintRRect (var theRect: rect; ovalWidth, ovalHeight: integer);
```

Possible Errors: None

Paints all points inside the specified rounded rectangle using the current pen pattern and pen mode.

PenNormal

```
procedure PenNormal;
```

Possible Errors: None

Sets the current drawing pen do "normal" mode. Specifically:

1. The pen size is set to 1 pixel wide and 1 pixel high.
2. The pen mode is set to `modeCopy`.
3. The pen pattern is set to solid black.
4. The pen mask is set to allow all bits to be drawn.

PPToPort

```
procedure PPToPort (srcLoc: locInfoPtr; var srcRect: rect;
   destX, destY, transferMode: integer);
```

Possible Errors:	`$0420`	`notEqualChunkiness`	Chunkiness is not equal

`PPToPort` transfers a bit map to a port.

`srcLoc`	This is a pointer to a `locInfo` record that defines the image to draw. See "`LocInfo` Records" at the start of this section for more information about the `locInfo` record.
`srcRect`	This is the source rectangle. Using this parameter, you can clip a specific part of the image from the source image. The most common option, though, is to pass the rectangle that is imbedded in the `locInfo` record for the image. Doing so paints the entire source image.
`destX`	Horizontal location for the destination image. The left edge of the rectangle will be aligned to this coordinate.
`destY`	Vertical location for the destination image. The top edge of the rectangle will be aligned to this coordinate.
`transferMode`	This is the pen mode that will be used to draw the picture. Any of the pen modes that are valid for `SetPenMode` can be used.

PtInRect

```
function PtInRect (var thePoint: point; var theRect: rect): boolean;
```

Possible Errors: None

This function tests to see if `thePoint` lies inside of `theRect`. If so, the function returns true; otherwise, it returns false.

RectInRgn

```
function RectInRgn (var theRect: rect; theRgnHandle: rgnHandle): boolean;
```

Possible Errors: Memory Manager errors returned unchanged

`RectInRgn` tests to see if any part of the given rectangle intersects the given region. If so, `RectInRgn` returns true; if not, it returns false.

SetAllSCBs

```
procedure SetAllSCBs (newSCB: integer);
```

Possible Errors: None

Sets all scan line control bytes to the specified value.

SetBackColor

```
procedure SetBackColor (backColor: integer);
```

Possible Errors: None

Sets the background color for the current port to `backColor`. The background color should be in the range 0 to 3 if the 640 graphics mode is in use, and 0 to 15 if the 320 graphics mode is in use. If the value is outside of the allowed range, the value is reduced to the correct range automatically.

The background color is used when text is painted on the screen. The region behind the text is filled in with the background color.

SetColorEntry

```
procedure SetColorEntry (tableNumber, entryNumber, newColor: integer);
```

Possible Errors:	`$0450`	`badTableNum`	Invalid color table number
	`$0451`	`badColorNum`	Invalid color number

Sets the color table entry for the specified color table number and color table entry number to the given color value. There are 16 color tables, numbered 0 to 15, and 16 color entries in each color table, also numbered 0 to 15.

SetColorTable

```
procedure SetColorTable (tableNumber: integer; var newTable: colorTable);
```

Possible Errors:	`$0450`	`badTableNum`	Invalid color table number

Fills in one of the active color tables with the colors from the given color table array. There are 16 active color tables, numbered 0 to 15.

SetForeColor

```
procedure SetForeColor (foreColor: integer);
```

Possible Errors: None

Sets the foreground color for the current port to `foreColor`. The foreground color should be in the range 0 to 3 if the 640 graphics mode is in use, and 0 to 15 if the 320 graphics mode is in use. If the value is outside of the allowed range, the value is reduced to the correct range automatically.

When text is drawn to the screen, the foreground color is used to draw the text.

SetPenMask

```
procedure SetPenMask (var theMask: mask);
```

Possible Errors: None

Sets the pen mask for the current `grafPort` to the given mask.

SetPenMode

```
procedure SetPenMode (penMode: integer);
```

Possible Errors: None

Sets the pen mode. The pen mode controls how pixels are drawn on the screen.

Table A-5 shows the various pen modes and what they do. For the numeric equivalent of the constants, see the QuickDraw II definitions section.

Most of the pen modes perform some binary logic operation between the source and destination pixels. This is a bitwise operation. For example, `modeXOR` is frequently used in rubber-banding routines to draw an image, then erase it by drawing the same image in the same spot. When color 2 is drawn across a color 3 background in 640 mode, each pixel is drawn using an exclusive or operation. The source pixel in binary is 10, while the destination pixel is 11. The exclusive or operation results in a value of 01. When the line is drawn a second time, the source is again 10, while the destination is 01, and the screen is set to 11.

Mode	Description
`modeCopy`	Copy each source pixel exactly. For lines or pictures, the pen pattern is copied to the screen, while for text, each letter is painted using the foreground color, and the background is filled in using the background color. This is the typical drawing mode for most operations.
`notCopy`	Each source pixel is reversed in value, changing all 0 bits to 1 and all 1 bits to 0. The resulting image is drawn to the screen.
`modeOR`	The source bits are ored with the destination bits.
`notOR`	The source bits are reversed, and the result is ored with the destination bits.
`modeXOR`	The source bits are exclusive ored with the destination bits. If the bit is on in both the source and destination, or if it is off in both the source and destination, the result will be a bit that is off. If either the source or destination bit is on, but not both, the bit will be on in the final drawing.
`notXOR`	This mode works like `modeXOR`, but the bits are reversed in the final drawing.
`modeBIC`	Pixels in the source are reversed, then anded with the destination pixels. This mode is generally used to turn off pixels in an area in preparation for drawing a new image in the same area.
`notBIC`	The source bits are anded with the destination bits.

Table A-5: Pen Modes

SetPenPat

```
procedure SetPenPat (var thePattern: pattern);
```

Possible Errors: None

Sets the pen pattern for the current `grafPort` to the given pattern.

SetPenSize

```
procedure SetPenSize (width, height: integer);
```

Possible Errors: None

Sets the pen size for the current `grafPort` to the given size.

SetPort

```
procedure SetPort (thePort: grafPortPtr);
```

Possible Errors: None

Makes the specified port the current `grafPort`.

SetSCB

```
procedure SetSCB (scanLine, newSCB: integer);
```

Possible Errors: `$0452` `badScanLine` Invalid scan line number

Sets the scan line control byte for scan line `scanLine` to `newSCB`.

SetSolidPenPat

```
procedure SetSolidPenPat (colorNum: integer);
```

Possible Errors:	None

The current pen pattern is set to the solid color passed as a parameter. In 320 mode, the pen color can be any number from 0 to 15, while in 640 mode, the pen color can be any color from 0 to 3. If you pass a number outside of the allowed range, the value is reduced to the correct range using a mod operation. In 640 mode, the operation is `colorNum mod 4`, while in 320 mode, the operation is `colorNum mod 16`.

SetTextMode

```
procedure SetTextMode (textMode: integer);
```

Possible Errors:	None

Sets the text mode to the specified value. The text mode is equivalent to the pen mode (see `SetPenMode`), but the text mode applies only to text, while the pen mode applied to all drawing except text.

All of the pen modes that are valid for `SetPenMode` are also valid for `SetTextMode`, and all do exactly the same thing.

In addition, there are eight new pen modes. The reason for these modes is that a character normally prints as a full rectangle, with the part normally though of as the character drawn in the foreground color, and the rest of the rectangle filled in with the background color. The additional eight pen modes match the normal eight pen modes, but they only draw the text itself. The background of the rectangle is not filled in.

Here are the eight foreground only modes, along with the standard modes they correspond to.

standard mode	foreground mode
`modeCopy`	`modeForeCopy`
`notCopy`	`notForeCopy`
`modeOR`	`modeForeOR`
`notOR`	`notForeOR`
`modeXOR`	`modeForeXOR`
`notXOR`	`notForeXOR`
`modeBIC`	`modeForeBIC`
`notBIC`	`notForeBIC`

StringBounds

```
procedure StringBounds (theString: univ pStringPtr; var theRect: rect);
```

Possible Errors:	None

Returns the bounds rectangle for the given string of characters. See `CharBounds` for a definition of the bounds rectangle.

StringWidth

```
function StringWidth (theString: univ pStringPtr): integer;
```

Possible Errors: None

Returns the width of the string in pixels. This is equivalent to the pen displacement if the string is drawn.

TextBounds

```
procedure TextBounds (theText: univ textPtr; textLen: integer;
   var theRect: rect);
```

Possible Errors: None

Returns the bounds rectangle for the given block of characters. See `CharBounds` for a definition of the bounds rectangle.

TextWidth

```
function TextWidth (theText: univ textPtr; textLen: integer): integer;
```

Possible Errors: None

Returns the width of the block of text in pixels. This is equivalent to the pen displacement if the text is drawn.

QuickDraw II Definitions

```
type
   point = record
      v: integer;
      h: integer;
      end;

   rect = record
      case rectKinds of
         normal:  (v1: integer;
                   h1: integer;
                   v2: integer;
                   h2: integer);

         mac:     (top:    integer;
                   left:   integer;
                   bottom: integer;
                   right:  integer);

         points:  (topLeft:  point;
                   botRight: point);
         end;
```

```
    locInfo = record
        portSCB:          integer;
        ptrToPixelImage:  ptr;
        width:            integer;
        boundsRect:       rect;
        end;
    locInfoPtr = ^locInfo;

   (* FontInfo record *)
   fontInfoRecord = record
      ascent:  integer;
      descent: integer;
      widMax:  integer;
      leading: integer;
      end;
```

QuickDraw II Auxiliary

In ORCA/Pascal the header information for this tool is combined with QuickDraw II. If your program uses the tool calls listed in this section, you must include both QuickDrawII and Common in your `uses` statement.

DrawPicture

```
procedure DrawPicture (picHandle: handle; var destRect: rect);
```

Possible Errors: None

`DrawPicture` draws a picture. `picHandle` is the handle of a picture created by QuickDraw II Auxiliary. In this course, it comes into the program from a scrap. `destRect` is the destination rectangle for the picture; the picture is expanded or shrunk to fit into the available space.

Resource Manager

In ORCA/Pascal the header file for this tool is named ResourceMgr. This header file makes use of the Common unit. Both of these units must be listed in your `uses` statement to make calls listed in this section.

AddResource

```
procedure AddResource (resourceHandle: handle; resourceAttr: integer;
   resourceType: integer; resourceID: longint);
```

Possible Errors:	`$1E03`	`resNoConverter`	No converter routine for the resource type
	`$1E04`	`resNoCurFile`	No current resource file
	`$1E05`	`resDupID`	The specified resource ID is already in use
	`$1E06`	`resNotFound`	The specified resource was not found
	`$1E0E`	`resDiskFull`	Volume full
	Memory Manager errors returned unchanged		
	GS/OS errors returned unchanged		

Adds a new resource to a resource fork.

`resourceHandle` This is the handle for the new resource. Once you call `AddResource`, the handle belongs to the Resource Manager.

`resourceAttr`	This is a resource attributes flag word. In this course, this flags word should be set to $0300. For a complete discussion of this parameter, see *Apple IIGS Toolbox Reference: Volume 3,* page 45-35.
`resourceType`	This is the resource type for the new resource.
`resourceID`	This is the resource ID for the new resource. The resource must not already exist.

CloseResourceFile

```
procedure CloseResourceFile (fileID: integer);
```

Possible Errors:	`$1E03`	`resNoConverter`	No converter routine for the resource type
	`$1E06`	`resNotFound`	The specified resource was not found
	`$1E0E`	`resDiskFull`	Volume full
	GS/OS errors returned unchanged		

Closes a resource fork previously opened by `OpenResourceFile`. The `fileID` parameter should be the same one returned by `OpenResourceFile`.

If the resource file being closed is the current resource file, the next most current resource file becomes the new current resource file.

For further details about this call, see *Apple IIGS Toolbox Reference: Volume 3,* page 45-37.

LoadResource

```
function LoadResource (resourceType: resType; resourceID: resID): handle;
```

Possible Errors:	`$1E03`	`resNoConverter`	No converter routine for the resource type
	`$1E06`	`resNotFound`	The specified resource was not found
	Memory Manager errors returned unchanged		
	GS/OS errors returned unchanged		

`LoadResource` loads a resource based on a resource type and resource ID and returns a handle to the resource.

`LoadResource` works with `ReleaseResource` to manage the memory used by the various resources. In some cases, you may want to load a resource and keep it in memory from that time on, but in most cases, you will load a resource, use it for a while, then call `ReleaseResource` to release the resource. If you need the resource again, you call `LoadResource` again. In many cases, the resource is still in memory, since it is simply marked purgeable by `ReleaseResource`. The effect is that `LoadResource` can grab the resource from memory without reading it in from disk again, so long as the memory wasn't needed for anything else between uses.

You should never dispose of a handle returned by `LoadResource`.

There are some capabilities of `LoadResource` that are not used in this course. For a complete description, see page 45-56 of *Apple IIGS Toolbox Reference Manual: Volume 3*.

OpenResourceFile

```
function OpenResourceFile (openAccess: integer; mapAddress: resMapPtr;
   var fileName: gsosInString): integer;
```

Possible Errors:	`$1E06`	`resNotFound`	The specified resource was not found
	`$1E09`	`resNoUniqueID`	No more resource IDs are available
	`$1E0B`	`resSysIsOpen`	The system resource file is already open
	Memory Manager errors returned unchanged		
	GS/OS errors returned unchanged		

Opens a resource file, making it the current resource file. `OpenResourceFile` returns a resource file ID; this should be passed to `CloseResourceFile` when it is time to close the resource fork.

`openAccess`	This parameter tells `OpenResourceFile` whether you want read-only access (1), write-only access (2), or access to both read and write resources (3).
`mapAddress`	This is a pointer to a resource map in memory, or nil to retrieve the resource map from disk. In this course, the parameter should always be nil. For other possibilities, see *Apple IIGS Toolbox Reference: Volume 3*, page 45-61.
`fileName`	This is the GS/OS path name for the file to open.

ReleaseResource

```
procedure ReleaseResource (purgeLevel: integer; resourceType: resType;
   resourceID: resID);
```

Possible Errors:	`$1E06`	`resNotFound`	The specified resource was not found
	`$1E0C`	`resHasChanged`	The resource has been changed and has not been updated

`ReleaseResource` sets the purge level for the handle used by a resource. You pass the purge level (0-3), the resource type, and the resource ID of a resource that has been loaded (usually by `LoadResource`) and `ReleaseResource` sets the purge level to the value you specify.

Purge levels range from 0-3, and exist for all handles. A handle with a purge level of 1, 2 or 3 is purgeable. When a program asks the Memory Manager for memory, the Memory Manager checks to see if enough memory is available. If not, the Memory Manager will (among other things) dispose of enough purgeable memory to allocate the new memory block. If the handle has not been purged before the contents are needed again, the purge level can be set back to 0 and the information in the handle will still be valid.

If the Memory Manager needs to dispose of some purgeable handles, it will start with handles with a purge level of 3, then take handles with a purge level of 2, and finally take handles with a purge level of 1.

Scrap Manager

In ORCA/Pascal the header file for this tool is named ScrapMgr. This header file makes use of the Common unit. Both of these units must be listed in your `uses` statement to make calls listed in this section.

GetIndScrap

```
procedure GetIndScrap (index: integer; buffer: scrapBuffer);
```

Possible Errors:	`$1610`	`basCrapType`	No scrap of this type

Gets a scrap by scrap index. This call is used to read the current scrap when you don't know the scrap type for the scrap, something that is generally only necessary in a scrapbook program. The first call is made with `index` set to 1, and information about the first scrap is returned in `buffer`. For each subsequent call, `index` is incremented by 1, until `GetIndScrap` returns with an error.

The scrap buffer has this format:

```
scrapBuffer = record
   scrapType: integer;
   scrapSize: longint;
   scrapHandle: handle;
   end;
```

The handle is a copy of the handle owned by the Scrap Manager. Your program can copy the information, but should not delete the handle or change the contents of the handle.

GetScrapHandle

```
function GetScrapHandle (scrapType: integer): handle;
```

Possible Errors:	`$1610`	`basCrapType`	No scrap of this type
	Memory Manager errors returned unchanged		
	GS/OS errors returned unchanged		

Returns the handle for the current scrap with the given scrap type. The handle is a copy of the handle owned by the Scrap Manager. Your program can copy the information, but should not delete the handle or change the contents of the handle.

GetScrapSize

```
function GetScrapSize (scrapType: integer): longint;
```

Possible Errors:	`$1610`	`basCrapType`	No scrap of this type
	Memory Manager errors returned unchanged		
	GS/OS errors returned unchanged		

Returns the size of the scrap for the given scrap type. The size is given in bytes.

LoadScrap

```
procedure LoadScrap;
```

Possible Errors:	Memory Manager errors returned unchanged
	GS/OS errors returned unchanged

This call should be made by any program that uses the Scrap Manager. The call is made right after initializing the tools. It tells the Scrap Manager to look for a scrap file, reading the scrap and making it the current scrap if a scrap file is found. See also `UnloadScrap`.

PutScrap

```
procedure PutScrap (numBytes: longint; scrapType: integer; srcPtr: ptr);
```

Possible Errors: Memory Manager errors returned unchanged
GS/OS errors returned unchanged

Adds `numBytes` of information to the current scrap with a scrap type of `scrapType`. The bytes are copied from memory starting at `srcPtr`.

This call will not effect the contents of scraps with a scrap type other than the one given. If more that one call is made to `PutScrap` with the same scrap type, the new information is appended to the old information.

See also `ZeroScrap`.

UnloadScrap

```
procedure UnloadScrap;
```

Possible Errors: Memory Manager errors returned unchanged
GS/OS errors returned unchanged

This call should be made by any program that supports the Scrap Manager. The call should be made just before the program shuts down the tools. It tells the Scrap Manager to write the current scrap to a private, system scrap file. The scrap file is loaded with `LoadScrap`.

ZeroScrap

```
procedure ZeroScrap;
```

Possible Errors: Memory Manager errors returned unchanged
GS/OS errors returned unchanged

`ZeroScrap` deletes all of the information for all scrap types in the current scrap. `ZeroScrap` is generally called just before creating a new scrap with calls to `PutScrap`.

Sound Tool Set

In ORCA/Pascal the header file for this tool is named SoundMgr. This header file makes use of the Common unit. Both of these units must be listed in your `uses` statement to make calls listed in this section.

FFSoundDoneStatus

```
function FFSoundDoneStatus (genNumber: integer): boolean;
```

Possible Errors:	`$0813`	`invalGenNumErr`	Invalid generator number

Returns true if the specified generator is busy playing a sound, and false if it is free.

FFStartSound

```
procedure FFStartSound (genNumFFSynth: integer;
   var PBlockPtr: soundParamBlock);
```

Possible Errors:	`$0812`	`noSAddrInitErr`	The Sound Tool Set is not active
	`$0813`	`invalGenNumErr`	Invalid generator number
	`$0814`	`synthModeErr`	Synthesizer mode error
	`$0815`	`genBusyErr`	The generator is already in use

Starts playing a sound.

`genNumFFSynth` This value is broken down into several individual fields, as follows:

bits 15-12	DOC channel number, from 0 to 15. This value is always zero in this course; for a complete description, see *Apple IIGS Toolbox Reference: Volume 3*, page 47-13.
bits 11-8	The sound generator to use, in the range 0 to 14.
bits 7-4	Reserved; set to 0.
bits 3-0	In this course, this call is used to play a digitized sound; for that use, these bits should be set to 0001. For other uses, see *Apple IIGS Toolbox Reference: Volume 2*, page 21-16.

`PBlockPtr` Sound parameter block. See Lesson 16 for a simple description of this record and its parameters, or *Apple IIGS Toolbox Reference: Volume 3*, page 47-3 for a detailed description.

FFStopSound

```
procedure FFStopSound (genMask: integer);
```

Possible Errors: None

Stops the sound in one or more of the sound generators. Each bit corresponds to one of the 15 available sound generators, with the generator number matching the bit. Setting a bit in `genMask` turns off the sound in the corresponding sound generator.

For example, setting `genMask` to $7FFF turns off all of the user sound generators, while setting `genMask` to $0004 would turn off generator 2.

Generator 15 should never be turned off. It is used by the operating system as a timer; turning it off can cause all sorts of problems.

WriteRamBlock

```
procedure WriteRamBlock (srcPtr: ptr; DOCStart, byteCount: integer);
```

Possible Errors:	`$0810`	`noDOCFndErr`	The DOC or RAM was not found
	`$0811`	`docAddrRngErr`	DOC address range error

`WriteRamBlock` transfers a series of bytes from main memory to the sound memory. (The 64K sound memory is called DOC RAM in the toolbox reference manuals.)

The bytes to move must not lie in banks $00, $01, $E0 or $E1. Interrupts must be disabled while the bytes are moved.

`srcPtr`	Points to the first byte to transfer.
`DOCStart`	Location in DOC RAM to place the bytes.
`byteCount`	Number of bytes to transfer.

Sound Tool Set Definitions

```
type
   soundPBPtr = ^soundParamBlock;
   soundParamBlock = record
       waveStart:     ptr;                (* starting address of wave    *)
       waveSize:      integer;            (* waveform size in pages      *)
       freqOffset:    integer;            (* waveform playback frequency *)
       DOCBuffer:     integer;            (* DOC buffer starting address *)
       DOCBufferSize: integer;            (* DOC buffer size code        *)
       nextWAddr:     soundPBPtr;         (* ptr to next waveform block  *)
       volSetting:    integer;            (* DOC volume setting          *)
       end;
```

Standard File Operations Tool Set

In ORCA/Pascal the header file for this tool is named SFToolSet. This header file makes use of the Common unit and the DialogMgr unit. All three of these units must be listed in your `uses` statement to make calls listed in this section.

Reply Records

```
replyRecord5_0 = record
    good:         integer;
    fileType:     integer;
    auxFileType:  longint;
    nameVerb:     integer;
    nameRef:      longint;
    pathVerb:     integer;
    pathRef:      longint;
    end;
```

The Standard File Manager calls described here pass back a reply record to tell you what file the user picked.

`good`	This value will be 0 if the user canceled the operation, and non-zero if the user picked a file.
`fileType`	For `SFGetFile2`, this is the file type of the file the user picked. This field is not used for `SFPutFile2`.
`auxFileType`	For `SFGetFile2`, this is the auxiliary file type of the file the user picked. This field is not used for `SFPutFile2`.
`nameVerb`	You fill this parameter in before the call. It tells SFO what format to use when it returns `nameRef`. This parameter should almost always be 3, telling SFO to allocate an appropriate amount of memory and return a handle to the file name. For other possibilities, see *Apple IIGS Toolbox Reference Manual: Volume 3*, page 48-7.

`nameRef`	This is the name of the file selected by the user. It is a GS/OS output string, type `gsosOutString`.
`pathVerb`	You fill this parameter in before the call. It tells SFO what format to use when it returns `pathRef`. This parameter should almost always be 3, telling SFO to allocate an appropriate amount of memory and return a handle to the path name. For other possibilities, see *Apple IIGS Toolbox Reference Manual: Volume 3*, page 48-7.
`pathRef`	This is the full path name of the file selected by the user. It is a GS/OS output string, type `gsosOutString`.

SFGetFile2

```
procedure SFGetFile2 (whereX, whereY, promptVerb: integer;
   promptRef: univ longint; filterProcPtr: procPtr;
   var theTypeList: typeList5_0; var theReply: replyRecord5_0);
```

Possible Errors:	`$1701`	`badPromptDesc`	Invalid `promptRefDesc` value
	`$1704`	`badReplyNameDesc`	Invalid `nameRefDesc` value in the reply record
	`$1705`	`badReplyPathDesc`	Invalid `pathRefDesc` value in the reply record
	GS/OS errors returned unchanged		

`SFGetFile2` displays a standard dialog used to open files. All interaction with the user is handled by `SFGetFile2`, which returns a reply record indicating what file, if any, the user selected.

See "Reply Records" at the start of this section for details about the reply record.

`whereX`	The horizontal position for the top left corner of the dialog.
`whereY`	The vertical position for the top left corner of the dialog.
`promptVerb`	This is the type of reference for `promptRef`. 0 `promptRef` is a pointer to a p-string. 1 `promptRef` is the handle of a p-string. 2 `promptRef` is the resource ID for an `rPString` resource.
`promptRef`	This is a p-string used as a prompt at the top of the dialog.
`filterProcPtr`	This is a pointer to a filter procedure. In this course, this parameter is always nil, telling `SFGetFile2` to use the standard filter procedure. For alternatives, see *Apple IIGS Toolbox Reference Manual: Volume 3*, page 48-4.
`theTypeList`	This is a record containing the types of files to show in the file list. This is described fully below.
`theReply`	This is the reply record. See "Reply Records" at the start of this section for details.

The dialog shows a list of the files a user can open. This list always includes folders, allowing the user to navigate through the tree structured directories on a typical disk. In general, your application will also want to display all of the files the program can open, but not any files that the user really can't select. The `theTypeList` record is used to tell `SFGetFile2` which file types (other than directories) it should show the user.

The record itself is a count followed by one or more file type entries:

```
typeList5_0 = record
    numEntries:       integer;
    fileAndAuxTypes:  array [1..10] of typeRec;
    end;
```

While the ORCA/Pascal interfaces put a limit of 10 on the number of array entries, this is simply an artificial limit imposed by the need to pick some value. You can change this value if you need a longer type list.

Each type list entry is a `typeRec`:

```
typeRec = record
    flags:     integer;
    fileType:  integer;
    auxType:   longint;
    end;
```

The `fileType` and `auxType` entries specify one particular file type and auxiliary file type combination to allow. If a file's file type and auxiliary file type match one of these entries it will show up in the list of files the user can select. The `flags` word controls how the `fileType` and `auxType` entries are used. The bit flags are:

bit 15	If this bit is clear, a file must match the auxiliary file type exactly to show up in the list of files. If this bit is set, the `auxType` field is ignored.
bit 14	If this bit is clear, a file must match the file type exactly to show up in the list of files. If this bit is set, the `fileType` field is ignored.
bit 13	If this bit is set, matching files will be displayed in the file list, but they will be dimmed and unselectable.
bits 12-0	Reserved; set to 0.

SFPutFile2

```
procedure SFPutFile2 (whereX, whereY, promptVerb: integer;
   promptRef: univ longint; origNameVerb: integer;
   origNameRef: univ longint; var theReply: replyRecord5_0);
```

Possible Errors:			
	`$1701`	`badPromptDesc`	Invalid `promptRefDesc` value
	`$1702`	`badOrigNameDesc`	Invalid `origNameRefDesc` value
	`$1704`	`badReplyNameDesc`	Invalid `nameRefDesc` value in the reply record
	`$1705`	`badReplyPathDesc`	Invalid `pathRefDesc` value in the reply record
	GS/OS errors returned unchanged		

`SFPutFile2` displays a standard dialog used to pick a new file name. This is generally used by a program's Save As command to select a new file name, or by the Save command for a file that has never been saved to disk. All interaction with the user is handled by `SFPutFile2`, which returns a reply record indicating what file name, if any, the user selected.

See "Reply Records" at the start of this section for details about the reply record.

`whereX`	The horizontal position for the top left corner of the dialog.
`whereY`	The vertical position for the top left corner of the dialog.
`promptVerb`	This is the type of reference for `promptRef`.

0 `promptRef` is a pointer to a p-string.
1 `promptRef` is the handle of a p-string.
2 `promptRef` is the resource ID for an `rPString` resource.

`promptRef`	This is a p-string used as a prompt above the file name entry box.
`origNameVerb`	This is the type of reference for `origNameRef`. 0 `origNameRef` is a pointer to a GS/OS input string. 1 `origNameRef` is the handle of a GS/OS input string. 2 `origNameRef` is the resource ID for an `rC1InputString` resource. (This resource type is not used in this course. See *Apple IIGS Toolbox Reference Manual: Volume 3*, page E-4 for details.)
`origNameRef`	This is a GS/OS input string (type `gsosInString`). It is used as a default file name in the file name entry box.
`theReply`	This is the reply record. See "Reply Records" at the start of this section for details.

Standard File Operations Tool Set Definitions

```
type
   replyRecord5_0 = record
       good:         integer;
       fileType:     integer;
       auxFileType:  longint;
       nameVerb:     integer;
       nameRef:      longint;
       pathVerb:     integer;
       pathRef:      longint;
       end;

   typeRec = record
       flags:     integer;
       fileType:  integer;
       auxType:   longint;
       end;

   typeList5_0 = record
       numEntries:       integer;
       fileAndAuxTypes:  array [1..10] of typeRec;
       end;
```

TextEdit Tool Set

In ORCA/Pascal the header file for this tool is named TextEdit. This header file makes use of the Common unit and the ControlMgr unit. All three of these units must be listed in your `uses` statement to make calls listed in this section.

TEGetRuler

```
procedure TEGetRuler (rulerDescriptor: integer; rulerRef: univ longint;
   theTERecord: teHandle);
```

Possible Errors:			
	`$2202`	`teNotStarted`	The TextEdit Tool Set has not been started
	`$2203`	`teInvalidHandle`	The `teH` does not refer to a valid `TERecord`
	Resource Manager errors returned unchanged		

`TEGetRuler` returns the ruler for a TextEdit control.

In this course, `rulerDescriptor` is always 3, which tells `TEGetRuler` to create a handle for the ruler and store the handle at the location passed as `rulerRef`. See *Apple IIGS Toolbox Programming: Volume 3*, page 49-80 for other possibilities.

`theTERecord` is the handle for the text edit control, or nil for the current control in the current window.

TEGetSelection

```
procedure TEGetSelection (selectionStart, selectionEnd: univ ptr;
   theTERecord: teHandle);
```

Possible Errors:	`$2202`	`teNotStarted`	The TextEdit Tool Set has not been started
	`$2203`	`teInvalidHandle`	The `teH` does not refer to a valid `TERecord`

`TEGetSelection` reads the location of the current selection point, returning it as a character displacement from the start of the text. If no text is selected, both `selectionStart` and `selectionEnd` are set to the location of the insertion point.

Pass nil for `theTERecord` to update the active TextEdit control.

TEGetSelectionStyle

```
function TEGetSelectionStyle (var commonStyle: teStyle;
   styleHandle: TEStyleGroupHndl; theTERecord: teHandle): integer;
```

Possible Errors:	`$2202`	`teNotStarted`	The TextEdit Tool Set has not been started
	`$2203`	`teInvalidHandle`	The `teH` does not refer to a valid `TERecord`

`TEGetSelectionStyle` returns information about the style of the currently selected text. If no text is selected, the null style is returned; the null style is the style that will be applied to newly typed text.

`commonStyle` is a style record; all of the style information that applies to all of the selected text will be placed in this style record. The integer returned as the function result is a flags word that tells which of the style record entries are valid:

bits 15-6	Reserved; these will be 0.
bit 5	If this bit is set, the font family is valid.
bit 4	If this bit is set, the font size is valid.
bit 3	If this bit is set, the foreground color is valid.
bit 2	If this bit is set, the background color is valid.
bit 1	If this bit is set, the user data field is valid.
bit 0	If this bit is set, the attributes valid.

`TEGetSelectionStyle` places all of the style records that pertain to the selection in the handle passed as `styleHandle`, resizing the handle if necessary. We did not use this ability in this course. For more information on the style records returned, see *Apple IIGS Toolbox Programming: Volume 3,* page 49-84.

`theTERecord` is the handle for the text edit control, or nil for the current control in the current window.

TEGetText

```
function TEGetText (bufferDescriptor: integer; bufferRef: univ longint;
   bufferLength: longint; styleDescriptor: integer; styleRef: univ longint;
   theTERecord: teHandle): longint;
```

Possible Errors:			
	$2202	teNotStarted	The TextEdit Tool Set has not been started
	$2203	teInvalidHandle	The `teH` does not refer to a valid `TERecord`
	$2204	teInvalidDescriptor	Invalid descriptor value specified
	$2208	teBufferOverflow	The output buffer was too small to accept all data
	Memory Manager errors returned unchanged		
	Resource Manager errors returned unchanged		

`TEGetText` returns the text in a TextEdit control.

`bufferDescriptor` tells what kind of parameter is passed for `bufferRef`; in this course we use $0019, telling `TEGetText` to create a handle for the text, saving the handle at the location passed as the `bufferRef` parameter. The function returns the length of the text in bytes as the function result.

For the purposes of this course, `bufferLength` is always set to 0.

`styleDescriptor` tells `TEGetText` what sort of parameter to expect in `styleRef`; in this course, we used 3, telling `TEGetText` to return a handle. `styleRef` is the address of the handle variable. The information returned is a `teFormat` record; the length can be calculated by adding several values together from the record. (This is described in Lesson 11.)

`theTERecord` is the handle for the text edit control, or nil for the current control in the current window.

For information about other options available with this call see *Apple IIGS Toolbox Programming: Volume 3*, page 49-86.

TEPaintText

```
function TEPaintText (thePort: grafPortPtr; startingLine: longint;
   var destRect: rect; flags: integer; theTERecord: teHandle): longint;
```

Possible Errors:			
	$2202	teNotStarted	The TextEdit Tool Set has not been started
	$2203	teInvalidHandle	The `teH` does not refer to a valid `TERecord`
	$2209	teInvalidLine	The starting line value is greater than the number of lines in the text (can be interpreted as an end of file indication in some cases)

`TEPaintText` draws text in a `grafPort`. This is generally used to print text.

`thePort` is the `grafPort` where the text will be drawn. `destRect` is the size of one page in the `grafPort`.

`startingLine` is the line number at which to start printing. After filling one page, `TEPaintText` returns, returning the value to pass for `startingLine` on the next call. If the entire document has been printed, `TEPaintText` returns -1.

If `flags` is $4000, `TEPaintText` doesn't actually draw the text; instead, it calculates the correct line number for the next page and returns. This capability can be used to rapidly skip pages. Pass 0 for `flags` for normal printing.

`theTERecord` is the handle for the text edit control, or nil for the current control in the current window.

TESetRuler

```
procedure TESetRuler (rulerDescriptor: integer; rulerRef: univ longint;
   theTERecord: teHandle);
```

Possible Errors:	`$2202`	`teNotStarted`	The TextEdit Tool Set has not been started
	`$2203`	`teInvalidHandle`	The `teH` does not refer to a valid `TERecord`

`TESetRuler` changes the ruler information for a TextEdit control, then forces a screen update by calling `InvalRect`.

`rulerDescriptor` tells `TESetRuler` what sort of parameter to expect for `rulerRef`; in this course, we use 1 for `rulerDescriptor` and pass a ruler handle for `rulerRef`. See *Apple IIGS Toolbox Programming: Volume 3*, page 49-115 for other possibilities.

`theTERecord` is the handle for the text edit control, or nil for the current control in the current window.

TESetSelection

```
procedure TESetSelection (selectionStart, selectionEnd: longint;
   theTERecord: teHandle);
```

Possible Errors:	`$2202`	`teNotStarted`	The TextEdit Tool Set has not been started
	`$2203`	`teInvalidHandle`	The `teH` does not refer to a valid `TERecord`

`TESetSelection` sets the selection point to the values given, updating the control as appropriate. If `selectionStart` is the same as `selectionEnd`, the insertion point is placed `selectionEnd` characters into the document. If `selectionStart` is greater than `selectionEnd`, the values are reversed. If `selectionEnd` is past the end of the document, it is automatically reduced to a valid value.

Pass nil for `theTERecord` to update the active TextEdit control.

TESetText

```
procedure TESetText (textDescriptor: integer; textRef: teTextRef;
   textLength: longint; styleDescriptor: integer; styleRef: teStyleRef;
   theTERecord: teHandle);
```

Possible Errors:	`$2202`	`teNotStarted`	The TextEdit Tool Set has not been started
	`$2203`	`teInvalidHandle`	The `teH` does not refer to a valid `TERecord`
	`$2204`	`teInvalidDescriptor`	Invalid descriptor value specified
	Memory Manager errors returned unchanged		

`TESetText` adds text and style information to a TextEdit control.

`textDescriptor` tells `TESetText` what sort of parameter to expect for `textRef`. In this course, we always pass 5 for `textDescriptor` and a pointer to a text buffer for `textRef`. `textLength` is the number of characters in the text buffer.

`styleDescriptor` tells `TESetText` what sort of parameter to expect for `styleRef`. In this course, we always pass 0, indicating that `styleRef` is a pointer to a style record.

`theTERecord` is the handle for the text edit control, or nil for the current control in the current window.

For information about other options available with this call see *Apple IIGS Toolbox Programming: Volume 3*, page 49-117.

TEStyleChange

```
procedure TEStyleChange (flags: integer; var newStyle: teStyle;
   theTERecord: teHandle);
```

Possible Errors:	$2202	`teNotStarted`	The TextEdit Tool Set has not been started
	$2203	`teInvalidHandle`	The `teH` does not refer to a valid `TERecord`
	$2205	`teInvalidFlag`	The specified flag word is invalid

`TEStyleChange` changes the style information for the current text selection. If there is no current selection, the style is applied to the null style record; the null style record is the style that will be used if new text is typed.

`flags` is a set of bit flags indicating what style information should actually be changed.

bits 15-7	Reserved; set to 0.
6	If this bit is set, the font family will be changed.
5	If this bit is set, the font size will be changed.
4	If this bit is set, the foreground color will be changed.
3	If this bit is set, the background color will be changed.
2	If this bit is set, the user data field will be changed.
1	If this bit is set, the style attributes will be changed. You should not set this bit and bit 0.
0	If this bit is set, the style attributes will be updated, but the style isn't just blindly applied. Instead, TextEdit checks to see if the style is the current style for the entire selection. If so, the style is turned off. If not, the style is added to the style information already selected for the text.

`newStyle` is a style record containing the new style information.

`theTERecord` is the handle for the text edit control, or nil for the current control in the current window.

TextEdit Tool Set Definitions

```
type
   teStyle = record
       teFont:     fontID;
       foreColor:  integer;
       backColor:  integer;
       userData:   longint;
       end;

   teRuler = record
       leftMargin:     integer;
       leftIndent:     integer;
       rightMargin:    integer;
       just:           integer;
       extraLS:        integer;
       flags:          integer;
       userData:       longint;
       tabType:        integer;
   (* Change size of array for application. *)
       tabs:           array [1..1] of teTabItem;
       tabTerminator:  integer;
       end;
   teStyleGroupHndl = ^teStyleGroupPtr;
   teStyleGroupPtr  = ^teStyleGroup;
   teStyleGroup = record
       count:   integer;
   (* Change array size for application. *)
       styles:  array [1..1] of teStyle;
       end;

   teStyleItem = record
       length:  longint;
       offset:  longint;
       end;

   teFormatHndl = ^teFormatPtr;
   teFormatPtr  = ^teFormat;
   teFormat = record
       version:          integer;
       rulerListLength:  longint;
   (* Change array size for application. *)
       theRulerList:     array [1..1] of teRuler;
       styleListLength:  longint;
   (* Change array size for application. *)
       theStyleList:     array [1..1] of teStyle;
       numberOfStyles:   longint;
   (* Change array size for application. *)
       theStyles:        array [1..1] of teStyleItem;
       end;
```

ToolLocator

In ORCA/Pascal the header file for this tool is named ToolLocator. This header file makes use of the Common unit. Both of these units must be listed in your `uses` statement to make calls listed in this section.

MessageCenter

```
procedure MessageCenter (action, msgID: integer; messageHandle: handle);
```

Possible Errors:	`$0111`	`messNotFoundErr`	The specified message was not found

`MessageCenter` is used for communications between programs.

The `msgID` parameter tells `MessageCenter` what kind of message you are interested in. These message numbers are assigned by Apple Computer. The only one used in this course is message 1, which is a message created by the Finder. It contains lists of documents to open or print. See Lesson 17 for details on the format and use of this message type.

You can read, write or delete messages; the action parameter tells `MessageCenter` which you want to do. The table below shows the actions as a named action (the names are in the interface file for the Tool Locator) and as numeric values, along with a description of the action.

`addMessage`	1	Adds a new message. Any old message with the same message ID is deleted. `MessageCenter` makes a copy of the handle you pass.
`getMessage`	2	Reads a message. You must create and pass a handle, which should be moveable. `MessageCenter` will resize the handle and copy the message into the handle. If there is no message with the given message ID `MessageCenter` returns an error and doesn't change the handle.

`deleteMessage`	3	Removes a message from the message center. While you must pass a handle, it isn't used or modified in any way.

ShutDownTools

```
procedure ShutDownTools (startStopVerb: integer;
   startStopRecRef: univ longint);
```

Possible Errors: None

Shuts down the tools started by `StartUpTools`. The `startStopRecRef` is the value returned by `StartUpTools`; `startStopVerb` should match the `startStopVerb` parameter used in the `StartUpTools` call.

StartUpTools

```
function StartUpTools (myID, startStopVerb: integer;
   startStopRecRef: univ longint): longint;
```

Possible Errors:	`$0103`	`TLBadRecFlag`	The StartStop record is invalid
	`$0104`	`TLCantLoad`	A tool cannot be loaded
	System Loader errors returned unchanged		
	Memory Manager errors returned unchanged		
	GS/OS errors returned unchanged		

Starts one or more tools, doing all of the appropriate things (like reserving memory) and starting the tools in the correct order.

Some tools require other tools to work properly; it is up to you to make sure that all of the needed tools are actually started.

`StartUpTools` returns a `startStopRecRef` value that must be saved and passed to `ShutDownTools` when the tools are shut down.

See Table 7-1 for a list of the tools used in this course, along with their dependencies as of System Disk 6.0.

`myID`	The user ID to use for allocating memory. This is normally the application's user ID, as returned by Pascal's `UserID` function.
`startStopVerb`	This parameter tells `StartUpTools` what sort of value is being passed for `startStopRecRef`. In this course, we always use a resource to list the tools to start, so this parameter is 2. See *Apple IIGS Toolbox Reference: Volume 3*, page 51-19 for other possibilities.
`startStopRecRef`	This parameter is the resource ID for an `rToolStartup` resource.

TextEdit Tool Set Definitions

```
const
   (* MessageCenter action codes *)
   addMessage  =   1;         (* add message to msg center data *)
   getMessage  =   2;         (* return message from msg center *)
   deleteMessage = 3;         (* delete message from msg center *)
```

```
messageRecord = record
    blockLength: integer;
    IDstring:    pString;   (* may be a max of 64 chars long *)
(* Change length of array to suit application. *)
    dataBlock:   packed array [1..1] of byte;
    end;
```

Window Manager

In ORCA/Pascal the header file for this tool is named WindowMgr. This header file makes use of the Common unit. Both of these units must be listed in your `uses` statement to make calls listed in this section.

Window Parameters

When a window is defined, there are a number of parameters that are passed to the Window Manager to describe the type of the window. In this course, the parameters are passed via an `rWindParam1` resource or via a `paramList` record. The parameters described below apply to either, although some of the parameters are set to constants in an `rWindParam1` resource, and do not appear in the resource description file.

See "Window Manager Definitions," later in this section, for the definition of a `paramList` record. See Appendix B for a description of the `rWindParam1` resource.

`paramLength` The length of a `paramList` record; 78.

`wFrameBits` Flags word, defined as follows:

`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 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.
`fInfo`	4	If this bit is set, the window will have an info bar. 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 it's 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. If there is no title bar, this field can be set to nil.
`wRefCon`	Reserved for use by the application.
`wZoom`	This field is a rectangle defining 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 reference to a color table. It is a pointer in a `paramList` record, and can be a pointer, handle or resource ID in an `rWindParam1` resource.
`wYOrigin`, `wXOrigin`	These two fields define the starting origin for the window. 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. 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. Because of the way dithered colors are created in 640 mode, you should set `wScrollHor` to some multiple of 8.

	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. For details, see the *Apple IIGS Toolbox Reference Manual*.
`wInfoDefProc`	This pointer points to the subroutine to call when the information bar needs to be drawn. For details, see the *Apple IIGS Toolbox Reference Manual*.
`wContDefProc`	This field points to your update procedure.
`wPosition`	This is a rect record. 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.
`wPlane`	To make a new window the front most 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`	Used to allocate storage for the window. This parameter is generally set to nil, forcing the Window Manager to allocate the memory for the window record. For details, see the *Apple IIGS Toolbox Reference Manual*.

Window Color Tables

Window color tables are used to define the colors used to draw the various parts of the window, as well as the style for the title bar. Window color tables can be created with either an `rWindColor` resource or a `wColorTbl` record. In either case, the window colors consist of five flag words, as defined below.

All of the color entries are displacements into the current color table. Dithered colors are used in 640 mode, so all four color bits are used in both 320 mode and 640 mode.

`frameColor`	bits 15-8	Reserved; set to 0.
	bits 7-4	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.
	bits 3-0	Reserved; set to 0.
`titleColor`	bits 15-12	Reserved; set to 0.
	bits 11-8	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.
	bits 7-4	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, but you could set this to a different color, say to get gray text when a window is inactive.
	bits 3-0	This is the color of the text when the window is active.
`tBarColor`	bits 15-8	These eight bits define the type of title bar. There are three types of title bars: $00 Solid title bars are the normal kind, by default a boring black. The color is actually set by bits 3-0 of this word. $01 Dithered title bars use a checkerboard pattern, alternating between the foreground and background colors set by bits 7-4 and 3-0, respectively. $02 Lines title bars are the sort you see most often, with lines running across the title bar. Bits 7-4 and 3-0 define the two colors used.
	bits 7-4	This is one of the colors for a dithered title bar, or the color of the lines on a lined title bar.
	bits 3-0	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.
`growColor`	bits 15-12	These bits are only used on alert frames, where they define the color between the outside line around the window and the heavy line just inside the main outline.
	bits 11-8	Reserved; set to 0.
	bits 7-4	This is the interior color for the grow box when the window is not selected.
	bits 3-0	This is the interior color for the grow box when the window is selected.
`infoColor`	bits 15-12	These bits are only used on alert frames, where they define the color of the heavy box that runs inside of the main window outline.
	bits 11-8	Reserved; set to 0.
	bits 7-4	This is the interior color for an info bar.
	bits 3-0	Reserved; set to 0.

AlertWindow

```
function AlertWindow (alertFlags: integer; subStrPtr, alertStrPtr: ptr):
   integer;
```

Possible Errors: None

`AlertWindow` creates an alert window. The alert window can display an alert icon, a text message, and one or more buttons; these are all controlled with an alert string. `AlertWindow` maintains control until the user selects a button, then returns the button number. See "Alert Strings," below, for details on defining the buttons.

`alertFlags`	Flags word.	
	bits 15-3	Reserved; set to 0.
	bits 2-1	Indicates the type of reference for the `alertStrPtr` parameter: 00 `alertStrPtr` is a pointer to an alert string. 01 `alertStrPtr` is a handle to an alert string. 10 `alertStrPtr` is the resource ID of an `rAlertString` resource.
	bit 0	If this bit is 0, the strings referred to by `subStrPtr` must be c-strings. If this bit is 1, the strings should be p-strings.
`subStrPtr`	This parameter is a pointer to an array of pointers, each of which points to a string. The strings can be either c-strings or p-strings; this is determined by bit 0 of `alertFlags`. For complete details, see "Substitution Strings," below.	
`alertStrPtr`	This parameter refers to the alert string used to create the alert window. It can be a pointer, handle, or resource ID, as controlled by bits 2-1 of `alertFlags`. For complete details, see "Alert Strings," below.	

Alert Strings

An alert string consists of a series of characters. We'll break these characters down into named fields for convenience, but the actual alert string is still just a simple sequence of characters.

The fields in the alert string are:

size A single character or sequence of characters defining the size of the alert. The size is generally given by a single character from '1' to '9'. `AlertWindow` creates a window large enough to hold a message with a certain number of characters, adjusting the size for 320 mode or 640 mode screens.

size	characters	320 height	320 width	640 height	640 width
'1'	30	46	152	46	200
'2'	60	62	176	54	228
'3'	110	62	252	62	300
'4'	175	90	252	72	352
'5'	110	54	252	46	400
'6'	150	62	300	54	452
'7'	200	80	300	62	500
'8'	250	108	300	72	552
'9'	300	134	300	80	600

A size of '0' has a special meaning. In this case, the size character is followed by 8 other bytes, laid out in memory exactly as a `rect` record would appear in memory. This rectangle record gives the exact size and location for the alert window.

icon A single character or series of characters that specify what icon will appear at the top left corner of the alert. The icon is generally given as one of the following single characters:

character	icon
'0'	(no icon)

`'2'`	stop
`'3'`	note
`'4'`	caution
`'5'`	disk
`'6'`	disk swap

An icon character of '1' has a special meaning. In this case, the icon is followed in memory by a record consisting of a pointer to an icon image, an integer specifying the width of the icon in bytes, and an integer specifying the height of the icon in pixels. This facility can be used to create custom icons for `AlertWindow`.

separator A separator character. You can use any character you like for a separator, so long as you use the same character here and in the next separator field. It is customary to use a / character for the separator character.

message This is the text that will be displayed in the alert. The message cannot exceed 1000 characters. If the message is too long to fit on one line in the window, the message will be automatically broken at a word boundary, and the remaining part of the message displayed on a new line. You can also force a new line by imbedding the characters `\n` in the text.

separator This separator character signals the end of the message and the start of the button definitions. This character must match the character used for the first separator.

buttons This field contains the titles for up to three buttons. If more than one button is used, separate the titles with additional separator characters. The buttons will all be placed at the bottom of the window. They will be centered, and all of the buttons will have the same width. The width will be determined by the longest sequence of text in any of the button titles. The total length of the button text must not exceed 80 characters.

One of the buttons may be preceded with a ^ character. This character does not appear in the button text; instead, it signals that the button is the default button. The default button will be bold, and will use the return key as a key equivalent. Because of this special use, to use a ^ character as an actual character anywhere in the alert string, you must code two ^ characters.

terminator The end of the alert string is marked with a null character; you can code this character by placing \$00 at the end of the string.

Substitution Strings

Substitution strings let you specify where a string should appear in the alert string without actually coding the string itself as a fixed value. Substitution strings come in two flavors.

The first type of substitution string is a # character followed by a numeric character from '0' to '6'. These two characters will be replaced by one of these standard strings:

#0	OK
#1	Cancel
#2	Yes
#3	No
#4	Try again
#5	Quit
#6	Continue

The other type of substitution string is an asterisk (*) followed by a numeric character from `'0'` to '9'. In this case, the characters are still replaced by a string, but the string is set using a parameter to the `AlertWindow` call. This lets you set up the strings at run time, rather than compile time. The strings themselves are taken from the array of strings passed as the `subStrPtr` in the call to `AlertWindow`.

BeginUpdate

```
procedure BeginUpdate (theWindow: grafPortPtr);
```

Possible Errors: None

`BeginUpdate` is used when you find an update event for a window. At that point, you call `BeginUpdate`, which replaces the visible region for the window with the intersection of the visible region and update region – in other words, all of the parts of the window that have to be redrawn.

The next step is to call the procedure that draws the contents of the window. During this update cycle, the origin must not be changed, and `SetPort` must not be called.

Once the window's contents have been drawn, call `EndUpdate`. Every call to `BeginUpdate` must be matched by exactly one call to `EndUpdate`.

BringToFront

```
procedure BringToFront (theWindow: grafPortPtr);
```

Possible Errors: None

Makes the specified window the front window, bringing it to the front of all other windows and posting appropriate update events to force the windows involved to be updated.

CloseWindow

```
procedure CloseWindow (theWindow: grafPortPtr);
```

Possible Errors: None

`CloseWindow` gets rid of a window. It disposes of all of the controls in the window, disposes of the memory used by the Window Manager for the window's window record, and erases the window on the desktop.

DoModalWindow

```
function DoModalWindow (event: eventRecord;
   updateProc, eventHook, beepProc: procPtr; flags: integer): longint;
```

Possible Errors: None

`DoModalWindow` handles interactions between the user and a modal dialog. The window representing the modal dialog should be opened before this call.

The value returns is the control ID for the control selected.

Normally, you will open a dialog window, then drop into a loop that calls `DoModalWindow` repeatedly, stopping when the user selects some control that closes the dialog. At that time, you would check the controls to see how they are set, taking whatever action is appropriate. You can, of course, check the controls and take action in the loop that calls `DoModalWindow`.

If the dialog has any controls that allow text to be edited, and if you have set the proper flags bit to allow an I beam cursor, `DoModalWindow` may exit with the cursor still set to an I beam. For that reason, you should always call `InitCursor` after leaving the `DoModalWindow` loop.

`event`	An event record that will be used by `DoModalWindow` to call the Event Manager.
`updateProc`	Address of a custom update procedure. For the purposes of this course, you should pass nil. For alternatives, see *Programmer's Reference for System 6.0*.
`eventHook`	Address of a subroutine that will be called right after the Event Manager. For the purposes of this course, you can pass nil. For alternatives, see *Programmer's Reference for System 6.0*.
`beepProc`	Address of a custom beep procedure, called when the user clicks outside of the dialog. For the purposes of this course, you can pass nil. For alternatives, see *Programmer's Reference for System 6.0*.
`flags`	Flags word, controlling options for `DoModalWindow`:

bit 15	If this bit is set, the dialog can be moved. If the bit is clear, the dialog can't be moved. If you want a moveable dialog, you have to set this bit *and* make sure the dialog has a title bar.
bit 14	If this bit is set, `DoModalWindow` will try to call the update procedure for any window that needs to be updated. If this bit is not set, windows other than the dialog itself and desk accessory windows won't get updated unless you do it manually in the loop that calls `DoModalWindow`.
bits 13-6	These bits are reserved, and must be set to 0.
bit 5	If this bit is set, `DoModalWindow` returns any time it handles an activate event. If this bit is clear, `DoModalWindow` doesn't return after an activate event; instead, it loops and checks for another event.
bit 4	If this bit is set, `DoModalWindow` allows desk accessories to operate, and does everything appropriate to handle them.
bit 3	If this bit is set, `DoModalWindow` will switch the cursor from the normal arrow to an I-beam cursor when the cursor is over an edit line or text edit control.
bit 2	If this bit is set, `DoModalWindow` calls `MenuKey` to see if keys pressed while the open-apple key is pressed are menu command equivalents. If this bit is clear, `DoModalWindow` returns all key events except the standard keyboard equivalents for cut (X), copy (C), paste (V), and undo (Z). Those four key equivalents are always handled by `DoModalWindow`.
bit 1	If this bit is set, `DoModalWindow` will allow the user to pull down menus and use menu commands. `DoModalWindow` will handle any of the standard editing commands, but your program will have to handle any other menu commands.

bit 0	If this bit is clear, `DoModalWindow` uses the Scrap Manager for cut, copy and paste. If this bit is set, `DoModalWindow` doesn't use the Scrap Manager.

EndUpdate

```
procedure EndUpdate (theWindow: grafPortPtr);
```

Possible Errors: None

`EndUpdate` restores the visible region for a window. Use `EndUpdate` to balance `BeginUpdate` calls.

FindWindow

```
function FindWindow (var whichWindow: grafPortPtr; pointX, pointY: integer):
   integer;
```

Possible Errors: None

`FindWindow` checks the location of the cursor and returns a value indicating where the cursor was located. If the cursor was located in a window, `whichWindow` is set to the window pointer; otherwise, `whichWindow` is set to nil.

The various values `FindWindow` can return, and what they mean, are shown in Table A-6.

number	name	description
0	`wNoHit`	Not in anything.
16	`wInDesk`	Somewhere on the desktop.
17	`wInMenuBar`	In the system menu bar.
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.
>32767	`wInSysWindow`	In a system window.

Table A-6: `FindWindow` Return Codes

FrontWindow

```
function FrontWindow: grafPortPtr;
```

Possible Errors: None

Returns a pointer to the first visible window in the window list. If there are no visible windows, `FrontWindow` returns nil.

GetContentOrigin

```
function GetContentOrigin (theWindow: grafPortPtr): longint;
```

Possible Errors: None

`GetContentOrigin` returns the origin for a window. The origin is used by `TaskMaster` to handle scroll bars.

The two integers in the origin are returned as a single long integer. The vertical origin is returned in the least significant word, while the horizontal origin is returned in the most significant word.

GetNextWindow

```
function GetNextWindow (theWindow: grafPortPtr): grafPortPtr;
```

Possible Errors: None

Returns a pointer to the window that is just behind `theWindow`, or nil if `theWindow` is the rearmost window.

HandleDiskInsert

```
function HandleDiskInsert (flags, devNum: integer): longint;
```

Possible Errors: GS/OS errors are returned unchanged

In this course, `HandleDiskInsert` is used to check the various disk drives for an unformatted disk. If an unformatted disk is found, the user is presented with a series of dialogs that will result in either formatting the disk or ejecting the disk. For this use, `flags` should be $C000, and `devNum` should be 0. The value returned by `HandleDiskInsert` is not used, and should be discarded.

For details about this call, as well as some other uses you can make of the call, see *Programmer's Reference for System 6.0.*

HideWindow

```
procedure HideWindow (theWindow: grafPortPtr);
```

Possible Errors: None

`HideWindow` makes a window invisible. If the window is already invisible, `HideWindow` does nothing. If the window is the front window, `HideWindow` brings the next visible window in the window list to front.

InvalRect

```
procedure InvalRect (var badRect: rect);
```

Possible Errors: None

The specified rectangle is added to the update region. This causes the area to be redrawn on the window's next update cycle.

This call has one nasty side effect. The values in the rectangle will be changed, so you should not use the values in the rectangle after calling `InvalRect`.

NewWindow

```
function NewWindow (paramListPtr: paramList): grafPortPtr;
```

Possible Errors:	`$0E01`	`paramLenErr`	The first word of the parameter list is the wrong size
	`$0E02`	`allocateErr`	Unable to allocate the window record

Creates and draws a new window. `NewWindow` returns nil if an error occurs.

See "Window Manager Definitions," later in this appendix, for the definition of the parameter list record. See "Window Parameters," earlier in this section, for a description of the various fields in the record.

NewWindow2

```
function NewWindow2 (titlePtr: pStringPtr; refCon: univ longint;
   contentDrawPtr, defProcPtr: procPtr; paramTableDescriptor: integer;
   paramTableRef:  univ longint; resourceType: integer): grafPortPtr;
```

Possible Errors:	`$0E01`	`paramLenErr`	The first word of the parameter list is the wrong size
	`$0E02`	`allocateErr`	Unable to allocate the window record
	Control Manager errors returned unchanged		
	Memory Manager errors returned unchanged		
	Resource Manager errors returned unchanged		

Creates and draws a new window. `NewWindow2` returns nil if an error occurs.

There are several aspects of `NewWindow2` that are not used in this course, and will not be discussed here. See *Apple IIGS Toolbox Reference: Volume 3* for a complete description of `NewWindow2`.

`titlePtr`	This is a pointer to the window title, given as a p-string. The title must remain in a fixed location in memory. If a lined or dithered window is used, the pattern runs right up to the edge of the window title. For that reason, it is customary to pad the window title on the left and right with spaces.
`refCon`	This value will be used in place of the `wRefCon` in the window definition. The `wRefCon` value is for your application's use; the Window Manager does not use it for any reason.
`contentDrawPtr`	This is a pointer to a subroutine that will draw the content region of the window. This subroutine is called when the window needs to be updated. `BeginUpdate` has already been called, and `EndUpdate` will be called after return. Since this subroutine is called during an update, you must not change the origin or call `SetPort` in this subroutine.

As with all subroutines called from the tools, the data bank register will not be valid. Use the `DataBank` directive to correct the data bank register.

`defProcPtr`	This parameter points to a window definition procedure. This parameter is always set to nil in this course. See the *Apple IIGS Toolbox Reference Manual* for details.
`paramTableDescriptor`	This parameter tells `NewWindow2` what is being passed in `paramTableRef`. The possibilities are: 0 `paramTableRef` is a pointer to a window template. 1 `paramTableRef` is a handle to a window template. 2 `paramTableRef` is the resource ID of a window template. (This is the only option used in this course.)
`paramTableRef`	This is a pointer, handle, or resource ID for a window template.
`resourceType`	This parameter is the resource type, used when `paramTableRef` is a resource ID. The only resource type used in this course is $800E, `rWindParam1`.

SetContentOrigin

```
procedure SetContentOrigin (xOrigin, yOrigin: integer;
   theWindow: grafPortPtr);
```

Possible Errors: None

Sets the origin for the window and generates an update event. If `TaskMaster` is handling scrolling in the window, this call has the effect of moving the document under program control.

SetDataSize

```
procedure SetDataSize (dataWidth, dataHeight: integer;
   theWindow: grafPortPtr);
```

Possible Errors: None

Changes the height and width of the data area (the size of the entire document). You still need to change the scroll bars and redraw the window (or force an update with `InvalRect`).

SetPage

```
procedure SetPage (hPage, vPage: integer; theWindow: grafPortPtr);
```

Possible Errors: None

Assuming `TaskMaster` is being used to manage scroll bars, `SetPage` sets the number of pixels the window will scroll when the user clicks in the page region of one of the scroll bars. If this value is set to 0, `TaskMaster` will scroll ten pixels less than the size of the window.

SetScroll

```
procedure SetScroll (hScroll, vScroll: integer; theWindow: grafPortPtr);
```

Possible Errors: None

Assuming `TaskMaster` is being used to manage scroll bars, `SetScroll` sets the number of pixels the window will scroll when the user clicks on an arrow on one of the scroll bars.

SetSysWindow

```
procedure SetSysWindow (theWindow: grafPortPtr);
```

Possible Errors: None

Makes the window a system window. You should do this for any window opened by a desk accessory. You would not normally make this call for an application window.

SetWTitle

```
procedure SetWTitle (title: univ pStringPtr; theWindow: grafPortPtr);
```

Possible Errors: None

`title` should be a pointer to a p-string. Because of the way the parameter is defined, you can pass an actual pointer, the address of a variable, the address of a string constant, or a variable of type `pString`. The title for the window is changed to the one you pass, and the window is redrawn.

The title string you pass must remain at a fixed location in memory, and must not be changed unless the change is followed by another call to `SetWTitle`.

ShowWindow

```
procedure ShowWindow (theWindow: grafPortPtr);
```

Possible Errors: None

`ShowWindow` makes an invisible window visible and draws the window. If the window is already visible, `ShowWindow` does nothing.

TaskMaster

```
function TaskMaster (taskMask: integer; var theTaskRec: eventRecord): integer;
```

Possible Errors:			
	`$0E03`	`taskMaskErr`	Some reserved bits were not clear in the `wmTaskMask` field of the `WmTaskRec` record

`TaskMaster` performs the normal housekeeping functions associated with an event loop. It gets an event, does some standard processing on the event that must be done in virtually any desktop program, then returns information about the event in an expanded event record.

The `taskMask` parameter is passed to `GetNextEvent` as the event mask. The event record is also passed to `GetNextEvent`, but it can be modified by `TaskMaster` as `TaskMaster` performs the various housekeeping tasks.

While `TaskMaster` can do a great deal, it is possible for the programmer to ask `TaskMaster` to do some things but not others. The exact list of tasks performed by `TaskMaster` are controlled by the `taskMask` field in the event record, which should be set before `TaskMaster` is called. The bit flags that control task selection are shown in Table A-7.

`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 A-7: `TaskMaster` `taskMask` Codes

TaskMasterDA

```
function TaskMasterDA (eventMask: integer; var taskRecPtr: eventRecord):
    integer;
```

Possible Errors: None

`TaskMasterDA` is an alternate entry point for `TaskMaster`; it is used in New Desk Accessories.

eventMask is not used; it is included here to make `TaskMasterDA` calls look more like `TaskMaster` calls.

`taskRecPtr` is a pointer to an event record. This is both an input and an output. The NDA should pass the task record that was passed to the NDA; `TaskMasterDA` will act on the event, modifying fields and returning an appropriate event code. From there, the event is handled exactly as it would be handled in an application.

Window Manager Definitions

```
type
   (* Document and alert window color table *)
   wColorTbl = record
      frameColor: integer;
      titleColor: integer;
      tBarColor:  integer;
      growColor:  integer;
      infoColor:  integer;
      end;
   wColorPtr = ^wColorTbl;

   {Window parameter list}
   paramList = record
      paramLength:   integer;
      wFrameBits:    integer;
      wTitle:        pStringPtr;
      wRefCon:       longint;
      wZoom:         rect;
      wColor:        wColorPtr;
      wYOrigin:      integer;
      wXOrigin:      integer;
      wDataH:        integer;
      wDataW:        integer;
      wMaxH:         integer;
      wMaxW:         integer;
      wScrollVer:    integer;
      wScrollHor:    integer;
      wPageVer:      integer;
      wPageHor:      integer;
      wInfoRefCon:   longint;
      wInfoHeight:   integer;
      wFrameDefProc: procPtr;
      wInfoDefProc:  procPtr;
      wContDefProc:  procPtr;
      wPosition:     rect;
      wPlane:        grafPortPtr;
      wStorage :     windRecPtr;
      end;
   paramListPtr = ^paramList;
```

Tool and GS/OS Errors

System Failure Errors

$0001	pdosUnClmdIntErr	Unclaimed interrupt
$0004	divByZeroErr	Division by zero
$000A	pdosVCBErr	Volume control block is not useable
$000B	pdosFCBErr	File control block is not useable
$000C	pdosBlk0Err	Block zero allocated illegally
$000D	pdosIntShdwErr	Interrupt with I/O shadowing off
$0015	segLoader1Err	Segment loader error
$0017	sPackage0Err	Can't load a package
$0018	package1Err	Can't load a package
$0019	package2Err	Can't load a package
$001A	package3Err	Can't load a package
$001B	package4Err	Can't load a package
$001C	package5Err	Can't load a package
$001D	package6Err	Can't load a package
$001E	package7Err	Can't load a package
$0020	package8Err	Can't load a package
$0021	package9Err	Can't load a package
$0022	package10Err	Can't load a package
$0023	package11Err	Can't load a package
$0024	package12Err	Can't load a package
$0025	putOfMemErr	Out of memory
$0026	segLoader2Err	Segment loader error
$0027	fMapTrshdErr	File map destroyed
$0028	stkOvrFlwErr	Stack overflow
$0030	psInstDiskErr	Insert disk alert
$0032-$0053		Memory Manager errors
$0100	stupVolMntErr	Can't mount system startup volume

GS/OS

$01	badSystemCall	Bad GS/OS call number
$04	invalidPcount	The parameter count is out of range
$07	gsosActive	GS/OS is busy
$10	devNotFound	Device not found
$11	invalidDevNum	Invalid device number
$20	drvrBadReq	Invalid request
$21	drvrBadCode	Invalid control or status code
$22	drvrBadParm	Bad call parameter
$23	drvrNotOpen	Character device not open
$24	drvrPriorOpen	Character device is already open
$25	irqTableFull	Interrupt table full
$26	drvrNoResrc	Resources are not available
$27	drvrIOError	I/O error
$28	drvrNoDevice	No device connected
$29	drvrBusy	Driver is busy
$2B	drvrWrtProt	Device is write protected
$2C	drvrBadCount	Invalid byte count
$2D	drvrBadBlock	Invalid block address
$2E	drvrDiskSwitch	The disk has been switched

$2F	drvrOffLine	Device offline or no media present
$40	badPathSyntax	Invalid path name syntax
$43	invalidRefNum	Invalid reference number
$44	pathNotFound	Subdirectory does not exist
$45	volNotFound	Volume not found
$46	fileNotFound	File not found
$47	dupPathname	Create or rename attempted with a name that already exists
$48	volumeFull	The volume is full
$49	volDirFull	The volume directory is full
$4A	badFileFormat	Version error (incompatible file type)
$4B	badStoreType	Unsupported or incorrect storage type
$4C	eofEncountered	End of file encountered
$4D	outOfRange	Position out of range
$4E	invalidAccess	Access not allowed
$4F	buffTooSmall	Buffer too small
$50	fileBusy	File is already open
$51	dirError	Directory error
$52	unknownVol	Unknown volume type
$53	paramRangeError	Parameter out of range
$54	outOfMem	Out of memory
$57	dupVolume	Duplicate volume name
$58	notBlockDev	Not a block device
$59	invalidLevel	Invalid file level
$5A	damagedBitMap	Block number too large
$5B	badPathNames	Invalid path names for `ChangePath`
$5C	notSystemFile	Not an executable file
$5D	osUnsupported	Operating system not supported
$5F	stackOverflow	Too many applications on stack
$60	dataUnavail	Data unavailable
$61	endOfDir	End of directory has been reached
$62	invalidClass	Invalid FST call class
$63	resForkNotFound	The file does not contain a required resource
$64	invalidFSTID	Invalid FST number
$65	invalidFSTop	Invalid FST operation
$67	devNameErr	A device exists with the same name as the replacement name
$70	resExistsErr	Cannot expand file; resource already exists
$71	resAddErr	Cannot add a resource fork to this kind of file

Tool Locator

$0001	toolNotFoundErr	The tool was not found
$0002	funcNotFoundErr	The tool function was not found
$0103	TLBadRecFlag	The StartStop record is invalid
$0104	TLCantLoad	A tool cannot be loaded
$0110	toolVersionErr	The requested minimum tool version was not available
$0111	messNotFoundErr	The specified message was not found
$0112	messageOvfl	No message numbers are available
$0113	nameTooLong	The message name is too long
$0120	reqNotAccepted	Nobody accepted the request
$0121	srqDuplicateName	Duplicate name
$0122	invalidSendRequest	Bad combination of `reqCode` and `target`

Memory Manager

`$0201`	`memErr`	Unable to allocate memory
`$0202`	`emptyErr`	Illegal operation on an empty handle
`$0203`	`notEmptyErr`	Illegal operation on a handle that is not empty
`$0204`	`lockErr`	Illegal operation on a locked or immovable block
`$0205`	`purgeErr`	Attempt to purge an unpurgeable block
`$0206`	`handleErr`	Invalid handle
`$0207`	`idErr`	Invalid user ID
`$0208`	`attrErr`	Illegal operation for the specified attributes

Miscellaneous Tool Set

`$0301`	`badInputErr`	Bad input parameter
`$0302`	`noDevParamErr`	No device for the input parameter
`$0303`	`taskInstlErr`	Specified task is already in the heartbeat queue
`$0304`	`noSigTaskErr`	No signature detected in the task header
`$0305`	`queueDmgdErr`	Damaged heartbeat queue
`$0306`	`taskNtFdErr`	Specified task is not in the queue
`$0307`	`firmTaskErr`	Unsuccessful firmware task
`$0308`	`hbQueueBadErr`	Damaged heartbeat queue
`$0309`	`unCnctDevErr`	Dispatch attempted to an unconnected device
`$030B`	`idTagNtAvlErr`	No ID tag is available
`$0380`	`notInList`	The specified routine was not found in the queue
`$0381`	`invalidTag`	The correct signature value was not found in the header
`$0382`	`alreadyInQueue`	Specified element already in queue
`$0390`	`badTimeVerb`	Invalid `convVerb` value
`$0391`	`badTimeData`	Invalid date or time to be converted
`$034F`	`mtBufferTooSmall`	The buffer is too small

QuickDraw II

`$0401`	`alreadyInitiallized`	QuickDraw II is already initialized
`$0402`	`cannotReset`	Never used
`$0403`	`notInitialized`	QuickDraw II is not initialized
`$0410`	`screenReserved`	The screen memory is reserved
`$0411`	`badRect`	Invalid rectangle
`$0420`	`notEqualChunkiness`	Chunkiness is not equal
`$0430`	`rgnAlreadyOpen`	Region is already open
`$0431`	`rgnNotOpen`	No region is open
`$0432`	`rgnScanOverflow`	Region scan overflow
`$0433`	`rgnFull`	Region is full
`$0440`	`polyAlreadyOpen`	Polygon is already open
`$0441`	`polyNotOpen`	No polygon is open
`$0442`	`polyTooBig`	The polygon is too big
`$0450`	`badTableNum`	Invalid color table number
`$0451`	`badColorNum`	Invalid color number
`$0452`	`badScanLine`	Invalid scan line number
`$04FF`		Not implemented

Desk Manager

`$0510`	`daNotFound`	Specified desk accessory is not available

$0511	notSysWindow	The window parameter is not a pointer to a system window owned by an NDA
$0520	deskBadSelector	Selector out of range

Event Manager

$0601	emDupStrtUpErr	The Event Manager has already been started
$0602	emResetErr	Can't reset the Event Manager
$0603	emNotActErr	The Event Manager is not active
$0604	emBadEvtCodeErr	The event code is greater than 15
$0605	emBadBttnNoErr	The button number given was not 0 or 1
$0606	emQSiz2LrgErr	The size of the event queue is larger than 3639
$0607	emNoMemQueueErr	Insufficient memory for the event queue
$0681	emBadEvtQErr	The event queue is damaged
$0682	emBadQHndlErr	Queue handle damaged

Sound Tool Set

$0810	noDOCFndErr	The DOC or RAM was not found
$0811	docAddrRngErr	DOC address range error
$0812	noSAddrInitErr	The Sound Tool Set is not active
$0813	invalGenNumErr	Invalid generator number
$0814	synthModeErr	Synthesizer mode error
$0815	genBusyErr	The generator is already in use
$0817	mstrIRQNotAssgnErr	Master IRQ not assigned
$0818	sndAlreadyStrtErr	The Sound Tool Set is already started
$08FF	unclaimedSndIntErr	Unclaimed sound interrupt error

Apple Desktop Bus Tool Set

$0910	cmndIncomplete	Command not completed
$0911	cantSync	Can't synchronize with the system
$0982	adbBusy	ADB busy (command pending)
$0983	devNotAtAddr	Device not at present address
$0984	sqrListFull	SQR list is full

Integer Math Tool Set

$0B01	imBadInptParam	Bad input parameter
$0B02	imIllegalChar	Illegal character in the string
$0B03	imOverflow	Integer or longint overflow
$0B04	imStrOverflow	String overflow

Text Tool Set

$0C01	badDevType	Illegal device type
$0C02	badDevNum	Illegal device number
$0C03	badMode	Illegal operation
$0C04	unDefHW	Undefined hardware error
$0C05	lostDev	Lost device; device is no longer online
$0C06	lostFile	File is not longer available
$0C07	badTitle	Illegal file name
$0C08	noRoom	Insufficient space on the specified diskette
$0C09	noDevice	Specified volume is not online

`$0C0A`	`noFile`	Specified file is not in the directory given
`$0C0B`	`dupFile`	Duplicate file
`$0C0C`	`notClosed`	Attempt to open a file that is already open
`$0C0D`	`notOpen`	Attempt to access a closed file
`$0C0E`	`badFormat`	Error reading real or integer number
`$0C0F`	`ringBuffOFlo`	Ring buffer overflow
`$0C10`	`writeProtected`	The specified disk is write protected
`$0C40`	`devErr`	The device did not complete a read or write

Window Manager

`$0E01`	`paramLenErr`	The first word of the parameter list is the wrong size
`$0E02`	`allocateErr`	Unable to allocate the window record
`$0E03`	`taskMaskErr`	Some reserved bits were not clear in the `wmTaskMask` field of the `WmTaskRec` record
`$0E04`	`compileTooLarge`	Compiled text is larger than 64K

Menu Manager

`$0F03`	`menuNoStruct`	Returned if bit 10 of `itemFlag` is not set

Control Manager

`$1001`	`wmNotStartedUp`	The Window Manager is not initialized
`$1002`	`cmNotInitialized`	The Control Manager has not been started
`$1003`	`noCtlInList`	The control is not in the window list
`$1004`	`noCtlError`	No controls in the window
`$1005`	`noExtendedCtlError`	No extended controls in the window
`$1006`	`noCtlTargetError`	No extended control is currently the target control
`$1007`	`notExtendedCtlError`	The action is valid only for extended controls
`$1008`	`canNotBeTargetError`	The specified control cannot be made the target control
`$1009`	`noSuchIDError`	The specified control ID cannot be found
`$100A`	`tooFewParmsError`	Too few parameters were specified
`$100B`	`noCtlToBeTargetError`	No control could be made the target control
`$100C`	`noFrontWindowError`	There is no front window

Loader

`$1101`		Entry not found
`$1102`		OMF version error
`$1103`		Path name error
`$1104`		The file is not a load file
`$1107`		File version error
`$1108`		User ID error
`$1109`		Segment number out of sequence
`$110A`		Illegal load record found
`$110B`		Load segment is foreign

QuickDraw II Auxiliary

`$1211`	`badRectSize`	The height or width is negative, the destination rect is not the same size as the source rect, or the source or destination rect is not within its boundary
`$1212`	`destModeError`	The `destMode` portion of `resMode` is invalid

$1230	`badGetSysIconInput`	No icon is available for the given input

Print Manager

$1301	`missingDriver`	Specified driver is not in the drivers folder of the system folder
$1302	`portNotOn`	The specified port is not selected in the control panel
$1303	`noPrintRecord`	No print record was specified
$1304	`badLaserPrep`	The version of LaserPrep in the drivers folder is not compatible with this version of the Print Manager
$1305	`badLPFile`	The version of LaserPrep in the drivers folder is not compatible with this version of the Print Manager
$1306	`papConnNotOpen`	Connection can't be established with the LaserWriter
$1307	`papReadWriteErr`	Read-write error on the LaserWriter
$1308	`ptrConnFailed`	Connection can't be established with the ImageWriter
$1309	`badLoadParam`	The specified parameter is invalid
$130A	`callNotSupported`	Tool call is not supported by the current version of the driver
$1321	`startUpAlreadyMode`	LLDStartUp call already made
$1322	`invalidCtlVal`	Invalid control value specified

LineEdit Tool Set

$1401	`leDupStrtUpErr`	The LineEdit Tool Set has already been started
$1402	`leResetError`	Can't reset LineEdit
$1403	`leNotActiveErr`	The LineEdit Tool Set has not been started
$1404	`leScrapErr`	The desk scrap is too big to copy

Dialog Manager

$150A	`badItemType`	Inappropriate item type
$150B	`newItemFailed`	Item creation failed
$150C	`itemNotFound`	No such item
$150D	`notModalDialog`	The front-most window is not a modal dialog

Scrap Manager

$1610	`basCrapType`	No scrap of this type

Standard File

$1701	`badPromptDesc`	Invalid `promptRefDesc` value
$1702	`badOrigNameDesc`	Invalid `origNameRefDesc` value
$1704	`badReplyNameDesc`	Invalid `nameRefDesc` value in the reply record
$1705	`badReplyPathDesc`	Invalid `pathRefDesc` value in the reply record
$1706	`badCall`	`SFPGetFile`, `SFPGetFile2` and `SFPMultiGet2` are not active

Note Synthesizer

$1901	`nsAlreadyInit`	The Note Synthesizer has already been started
$1902	`nsSndNotInit`	The Sound Tool Set has not been started
$1921	`nsNotAvail`	No generators are available
$1922	`nsBadGenNum`	Invalid generator number
$1923	`nsNotInit`	The Note Synthesizer has not been started
$1924	`nsGenAlreadyOn`	The specified note is already being played

$1925	soundWrongVer	The version of the Sound Tool Set is not compatible with this version of the Note Synthesizer

Note Sequencer

$1A00	noRoomMidiErr	The Note Sequencer is already tracking 32 notes; there is no room for a MIDI `NoteOn`
$1A01	noCommandErr	The current `seqItem` is not valid in this context
$1A02	noRoomErr	The sequence is nested more than 12 levels deep
$1A03	startedErr	The Note Sequencer is already started
$1A04	noNoteErr	Can't find the note for a `NoteOff` command
$1A05	noStartErr	The Note Sequencer was not started
$1A06	instBndsErr	The specified instrument is outside of the bounds of the current instrument table
$1A07	nsWrongVer	The version of the Note Synthesizer is incompatible with the Note Sequencer

Font Manager

$1B01	fmDupStartUpErr	The Font Manager has already been started
$1B02	fmResetErr	Can't reset the Font Manager
$1B03	fmNotActiveErr	The Font Manager has not been started
$1B04	fmFamNotFndErr	Family not found
$1B05	fmFontNtFndErr	Font not found
$1B06	fmFontMemErr	Font not in memory
$1B07	fmSysFontErr	System font cannot be purged
$1B08	fmBadFamNumErr	Illegal family number
$1B09	fmBadSizeErr	Illegal font size
$1B0A	fmBadNameErr	Illegal name length
$1B0B	fmMenuErr	FixFontMenu never called
$1B0C	fmScaleSizeErr	Scaled size font exceeds limits

List Manager

$1C02	listRejectEvent	The list control did not handle the event

ACE Tool Set

$1D01	aceIsActive	The ACE Tool Set has already been started
$1D02	aceBadDP	Requested direct page location is not valid
$1D03	aceNotActive	The ACE Tool Set has not been started
$1D04	aceNoSuchParam	Requested information type not supported
$1D05	aceBadMethod	Specified compression method is not supported
$1D06	aceBadSrc	Specified source is invalid
$1D07	aceBadDest	Specified destination is invalid
$1D08	aceDataOverlap	Specified source and destination areas overlap
$1DFF	aceNotImplemented	The requested function has not been implemented

Resource Manager

$1E01	resForkUsed	The resource fork is not empty
$1E02	resBadFormat	The resource fork is not correctly formatted
$1E03	resNoConverter	No converter routine for the resource type
$1E04	resNoCurFile	No current resource file

$1E05	resDupID	The specified resource ID is already in use
$1E06	resNotFound	The specified resource was not found
$1E07	resFileNotFound	The specified ID does not match an open file
$1E08	resBadAppID	The user ID was not found; the calling program has not issued a `ResourceStartUp` call
$1E09	resNoUniqueID	No more resource IDs are available
$1E0A	resIndexRange	Index is out of range; no resource was found
$1E0B	resSysIsOpen	The system resource file is already open
$1E0C	resHasChanged	The resource has been changed and has not been updated
$1E0D	resDiffConverter	Another converter is already logged in for this resource type
$1E0E	resDiskFull	Volume full
$1E10	resNameNotFound	The named resource was not found
$1E11	resBadNameVers	Bad resource name
$1E13	resInvalidTypeOrID	The resource type or ID was not valid

MIDI Tool Set

$2000	miStartUpErr	The MIDI Tool Set has not been started
$2001	miPacketErr	Incorrect packet length received
$2002	miArrayErr	Array was an invalid size
$2003	miFullBufErr	MIDI data discarded because of buffer overflow
$2004	miToolsErr	Required tools inactive or incorrect version
$2005	miOutOffErr	MIDI output disabled
$2007	miNoBufErr	No buffer allocated
$2008	miDriverErr	Specified device driver invalid
$2009	miBadFreqErr	Unable to set MIDI clock to the specified frequency
$200A	miClockErr	MIDI clock wrapped to zero
$200B	miConflictErr	Two processes are competing for the MIDI interrupt
$200C	miNoDevErr	No device driver loaded
$2080	miDevNotAvail	MIDI interface not available
$2081	miDevSlotBusy	Specified slot not available in Control Panel
$2082	miDevBusy	MIDI interface already in use
$2083	miDevOverrun	MIDI interface overrun by input data; the interface is not being served quickly enough
$2084	miDevNoConnect	No connection to MIDI interface
$2085	miDevReadErr	Error reading MIDI data
$2086	miDevVersion	ROM version or machine type incompatible with device driver
$2087	miDevIntHndlr	Conflicting interrupt handler installed

TextEdit Tool Set

$2201	teAlreadyStarted	The TextEdit Tool Set has already been started
$2202	teNotStarted	The TextEdit Tool Set has not been started
$2203	teInvalidHandle	The `teH` does not refer to a valid `TERecord`
$2204	teInvalidDescriptor	Invalid descriptor value specified
$2205	teInvalidFlag	The specified flag word is invalid
$2206	teInvalidPCount	The specified parameter count is not valid
$2208	teBufferOverflow	The output buffer was too small to accept all data
$2209	teInvalidLine	The starting line value is greater than the number of lines in the text (can be interpreted as an end of file indication in some cases)
$220B	teInvalidParameter	A passed parameter was not valid
$220C	teInvalidTextBox2	The `LETextBox2` format codes were inconsistent
$220D	teNeedsTools	The Font Manager was not started

Media Control Tool Set

$2601	UnImp	Unimplemented for this device
$2602	BadSpeed	Invalid speed specified
$2603	BadUnitType	Invalid unit type specified
$2604	TimeOutErr	Timed out during device read
$2605	notLoaded	No driver is currently loaded
$2606	BadAudio	Invalid audio value
$2607	devRtnError	Device returned error (cannot perform the command)
$2608	unRecStatus	Unrecognized status from the device
$2609	badSelector	Invalid selector value specified
$260A	funnyData	Funny data received (try again)
$260B	invalidPort	Invalid port specified
$260C	OnlyOnce	Scans only once
$260D	NoResMgr	Resource Manager not active (must be loaded and started)
$260E	invalidPort	Invalid port specified
$260F	wasShutDown	The tool set was already shut down
$2610	wasStarted	The tool was already started
$2611	badChannel	An invalid media channel was specified
$2612	InvalidParam	An invalid parameter was specified
$2613	CallNotSupported	An invalid media control tool call was attempted

Appendix B – Resources Used in This Course

Resource Attributes

In the sections that follow this one, you will find model statements showing you how to create a resource in the resource description file that Rez compiles. In all cases, the resource model starts with a line like this one:

```
resource rMenu (id) {
```

You can also add one or more resource attributes right after the resource ID. If there is more than one attribute, separate them with commas.

Table B-1 shows the various attributes you can use, what the default is, and what the attribute will do for you.

Default	Alternative	Meaning
`unlocked`	`locked`	Locked resources cannot be moved by the Memory Manager.
`moveable`	`fixed`	Specifies whether the Memory Manager can move the block when it is unlocked.
`nonconvert`	`convert`	Convert resources require a resource converter.
`handleload`	`absoluteload`	Absolute forces the resource to be loaded at an absolute address.
`nonpurgeable`	`purgeable1` `purgeable2` `purgeable3`	Purgeable resources can be automatically purged by the Memory Manager. Purgeable3 are purged before purgeable2, which are purged before purgeable1.
`unprotected`	`protected`	Protected resources cannot be modified by the Resource Manager.
`nonpreload`	`preload`	Preloaded resources are placed in memory as soon as the Resource Manager opens the resource file.
`crossbank`	`nocrossbank`	A crossbank resource can cross memory bank boundaries.
`specialmemory`	`nospecialmemory`	A special memory resource can be loaded in banks \$00, \$01, \$E0 and \$E1.
`notpagealigned`	`pagealigned`	A page-aligned resource must be loaded with a starting address that is an even multiple of 256.

Table B-1: Resource Attribute Key Words

rAlertString $8015

`rAlertString` resources are used to form alerts using `AlertWindow`. See the description of `AlertWindow` for details about what can appear in the string itself.

```
resource rAlertString (id) {
   "string"
   };
```

`id`	This is the resource ID for the alert string.
`string`	This is the actual alert string.

rBundle $802B

`rBundle` resources are used by the Finder. You can define icons for the application and its data files, as well as tell the Finder what sort of data files your program can use.

The `rBundle` resource is used in conjunction with the `rVersion` resource. If your application has either one of these resources, it should have both.

```
resource rBundle (id, preload, nospecialmemory) {
   appIcon,
   id,
   oneDoc,
   oneDoc
   }
```

`id`	This is the resource ID for the resource. The Finder only looks for an `rBundle` resource with a resource ID of 1, so this field should always be 1.
`preload`	You must always code the `preload` attribute for `rBundle` resources. See "Resource Attributes" at the start of this appendix for a description of the various attributes.
`nospecialmemory`	You must always code the `nospecialmemory` attribute for `rBundle` resources. See "Resource Attributes" at the start of this appendix for a description of the various attributes.
`appIcon`	This is the resource ID for an `rIcon` resource. This icon will be used by the Finder when the application is displayed in a folder or on the desktop.
`id`	The resource ID for the `rBundle` resource is repeated here. The Finder makes use of this field when it builds the DeskTop file.
`oneDoc`	The last entry is any number of `oneDoc` structures. Each of these structures defines one kind of file the application can deal with in some way. You can define icons or identify your application as a program that can use the data file. You can also select the kind of file this structure applies to using some very specific methods – it's possible to identify a single file if you are specific enough.

Here's the layout for a `oneDoc` structure:

```
{
   {
      launch,
      {path},
      {largeIcon},
      {smallIcon},
      "fileDesc",
      },
   match,
   MatchFileType {{fileType}},
   MatchAuxType {{auxMask,auxType}},
   empty {},
   empty {},
   empty {},
   empty {},
   empty {},
   empty {},
   empty {},
   empty {},
   empty {},
   empty {}
   }
```

`launch` This flags word tells the Finder if your application should be executed if the user double clicks on a file that matches all of the file qualifiers for this `oneDoc` structure.

Bit 0 is a boolean flag telling the Finder whether you should be launched at all. Set this bit if your application should be launched, and clear it if the application should not be launched.

Assuming the application can be launched, bits 4 to 7 establish the application's "voting clout." The Finder uses this information to pick the application that can best handle a particular file. The Finder does this by picking the application that sets the highest voting clout bit. Only one of these bits should be set.

bit 7	This is the highest priority of all. If you set bit 7, you are telling the Finder that you are the owner of the file type. Normally you would only set this bit if you reserve a specific file type for the files for your program.
bit 6	This bit says your program knows exactly what to do with the file, and can read and write the file.
bit 5	This bit says you can read the file, and may be able to write it – but not necessarily using the same format.
bit 4	Setting this bit says you can read the file, but that's about it.

`path` The path identifier. This field is filled in by the Finder. It should be set to 0 in your application.

`largeIcon` The Finder will use this icon when it displays a document matching this `oneDoc` record on the desktop, or in a window when large icons are used.

`smallIcon` The Finder will use this icon when it displays a document matching this `oneDoc` record in a window when either small icons or display by name is selected.

`fileDesc` The Finder displays a short text description of the file when the user asks for icon information. If you can provide a better text description that the one the Finder normally uses, put that description here. If not, use a null string or omit the field.

`match` This flags field tells the Finder which of the various file selectors are actually being used. There is one bit per file selector; the bit is on if the file selector is in use, and 0 if it should be ignored.

In this course, we use bit 0 for selecting files by file type, and bit 1 for selecting files by auxiliary file type. For a description of the other bits, see *Programmer's Reference for System 6.0.*

`fileType` This is the file type for the files to match. If bit 0 of `match` is set, only files with this file type will match this `oneDoc` structure.

The `MatchFileType` that appears to the left of the file type is a Rez label; it should be typed exactly like you see it.

`auxMask, auxType`
Assuming bit 1 of `match` is set, when the Finder is checking to see if a file matches this oneDoc structure, it ands the auxiliary file type for the file with `auxMask`, then compares the result to `auxType`. If the values match, the file passes this test; if not, this `oneDoc` structure will not be applied to the file.

The `MatchAuxType` that appears to the left of the file type is a Rez label; it should be typed exactly like you see it.

`empty` There are ten other ways to check a file. These other comparisons are not used in this course, and won't be covered here. You can skip all of these tests by making sure bits 31 to 2 are 0 in match, and by coding the `empty` entry exactly as you see it in the model structure. Like `MatchFileType` and `MatchAuxType`, `empty` is a Rez label, and must appear exactly as you see it.

For complete description of these fields, see *Programmer's Reference for System 6.0*.

rComment $802A

`rComment` resources are used to place comments in a resource file. The data is a sequence of ASCII characters. The Finder uses `rComment(1)` resources (i.e. `rComment` resources with a resource ID of 1) for user comments about a file, and `rComment(2)` resources for copyright information. These comments are displayed when the user asks for information about the icon.

```
resource rComment (id) {
   "string"
   };
```

`id` This is the resource ID for the resource.

`string` The comment string.

rControlList $8003

This resource is used to define a list of controls for an `rWindParam1` resource.

```
resource rControlList (id) {
   {
      id_1,
      id_2,
      id_3
      }
   };
```

`id`	This is the resource ID for the control list.
`id_1`	This value, and the values that follow it, is a resource ID for an `rControlTemplate` resource. The various controls are attached to the window that refers to this resource. When the controls are drawn, they are drawn in reverse order of their appearance in this list.

rControlTemplate $8004

This resource is used to create controls, generally for a window control list. It is a complex resource with many options. For a detailed, control by control explanation of how to use this resource, see lessons 11, 13 and 14. For more information, see *Apple IIGS Toolbox Reference Manual: Volume 3*, page E-7.

rIcon $8001

The `rIcon` resource is used to define icons. They are used for icon controls and Finder icons in this course, but have many other applications throughout desktop programs.

For details on the declaration and use of icons, see the text of Lesson 14. For more information, see *Apple IIGS Toolbox Reference: Volume 2*, page 17-3.

```
resource rIcon (id) {
   color,
   height,
   width,
   image,
   mask,
   };
```

`id`	This is the resource ID for the icon.
`color`	Use $8000 for color icons, or $0000 for black and white icons.
`height`	Height of the icon in pixels.
`width`	Width of the icon in pixels. This value must be a multiple of 8.
`image`	This is the icon image, generally entered as hexadecimal data strings.
`mask`	This is the icon mask, generally entered as hexadecimal data strings.

rMenu $8009

The `rMenu` resource is used to define menus. This resource is used in two ways in this course: to create menus for an `rMenuBar` resource, and to create pop-up menu controls.

```
resource rMenu (id) {
   menuID,
   flags,
   title,
   {
      rMenuItem1,
      rMenuItem2,
      };
   };
```

`id`	This is the resource ID for the menu.
`menuID`	This is the menu ID, used in various calls to the Menu Manager and returned by `TaskMaster` when a menu item from this menu is selected.
`flags`	One word of flag bits, defined as follows:

bits 15-14	Defines the type of reference in `title`. In this course, the menu title reference is always a resource ID, so these bits should always be set to 10.
bits 13-12	Defines the type of reference for each of the entries in the menu item array. In this course, the menu items are always given as an `rMenuItem` resource, so these bits are always set to 10.
bits 11-9	Reserved; set to 0.
bit 8	This bit is used to support tear-off menus. The bit should be set to 0 for all programs in this course.
bit 7	If this bit is clear, the menu will be enabled. If this bit is set, the menu will be disabled.
bit 6	Reserved; set to 0.
bit 5	If this bit is set, XOR highlighting will be used. If the bit is clear, color replace highlighting is used, eliminating green apple sickness.
bit 4	This bit should be set for custom menus. All of the menus in this course are standard menus, so the bit will always be clear.
bit 3	If this bit is set, menu caching is used. Menu caching is a technique that makes drawing the menu faster. When the menu is pulled down, the Menu Manager saves a picture of the menu. As long as no changes are made to the visual

appearance of the menu, the Menu Manager will use the picture to redraw the menu in the future, which is faster than recreating the menu.

bits 2-0	Reserved; set to 0.

`title`	This is the resource ID for an `rPString` resource containing the menu title. The `rPString` resource must have an attribute of `noCrossBank`.
`rMenuItem1`	This is the first of the resource IDs for the `rMenuItem` resources that make up the menu. The menu items specified by these resources are placed in the menu in the order listed in this array of resource IDs.

There are several constants defined in the Types.rez file that can be used with this resource. All of them are used to set the `flags` word.

```
/* --------------------------------------------------*/
/* flag word for menu item
/* --------------------------------------------------*/
#Define fXOR                $0020

/* --------------------------------------------------*/
/* flag word for Menu
/* --------------------------------------------------*/
#Define fAllowCache         $0008
#Define fCustom             $0010
#Define ItemRefShift        $1000
#Define MenuTitleRefShift   $4000
```

The last two entries are used to form the position of two reference bits, and should be multiplied by one of these values:

```
#Define RefIsPtr            $0000
#Define RefIsHandle         $0001
#Define RefIsResource       $0002
```

Using these constants, a typical `flags` entry is

```
RefIsResource*ItemRefShift + RefIsResource*MenuTitleRefShift + fAllowCache
```

This sets the `flags` word to $5008.

rMenuBar $8008

The `rMenuBar` resource is generally used to set up a system menu bar by calling `NewMenuBar2`.

```
resource rMenuBar (id) {
   {
      rMenu1,
      rMenu2,
      };
   };
```

`id`	This is the resource ID for the menu bar. It is passed as a parameter to `NewMenuBar2`.

`rMenu1`, ...	These are resource IDs for `rMenu` resources that define the various menus that will appear in the menu bar. The menus will be placed in the menu bar in the order listed, proceeding from left to right across the menu bar.

rMenuItem $800A

The `rMenuItem` resource is used to define menu items. In this course, this resource is always used to define menu items for `rMenu` resources.

```
resource rMenuItem (id) {
   itemID,
   "keyEquiv","keyEquiv",
   check,
   flags,
   title
   };
```

`id`	This is the resource ID for the menu item.
`itemID`	This is the menu item ID, used in various calls to the Menu Manager and returned by `TaskMaster` when this menu item is selected.
`keyEquiv`	These two parameters are the key equivalents for the menu item. The first is the key shown when the menu is pulled down. If the menu item does not have key equivalents, omit the characters, coding `""` for each.
`check`	This is the character to use as a check mark to the left of the menu item. Use 0 if there is no check mark, or 18 for a check mark. It is also possible to use any other character, although 0 and 18 are the most common values. The check character can be specified either as the ASCII character number or as a character constant, as in 'R'.
`flags`	One word of flag bits, defined as follows:

bits 15-14	Defines the type of reference in `title`. In this course, the menu item title reference is always a resource ID, so these bits should always be set to 10.
bit 13	Reserved; set to 0.
bit 12	If this bit is set, the characters in the title will be drawn with shadowing.
bit 11	If this bit is set, the characters in the title will be drawn with outlining.
bits 10-8	Reserved; set to 0.
bit 7	If this bit is clear, the menu item will be enabled. If this bit is set, the menu item will be disabled.
bit 6	If this bit is set, there will be a divider bar under the menu item.

bit 5	If this bit is set, XOR highlighting will be used. If the bit is clear, color replace highlighting is used.
bits 4-3	Reserved; set to 0.
bit 2	If this bit is set, the characters in the title will be underlined.
bit 1	If this bit is set, the characters in the title will be italicized.
bit 0	If this bit is set, the characters in the title will be bolded.

`title` This is the resource ID for an `rPString` resource containing the menu item title. The `rPString` resource must have an attribute of `noCrossBank`.

There are several constants defined in the Types.rez file that can be used with this resource. All of them are used to set the `flags` word.

```
/* --------------------------------------------------*/
/* flag word for menu item
/* --------------------------------------------------*/
#Define fBold                $0001
#Define fItalic              $0002
#Define fUnderline           $0004
#Define fXOR                 $0020
#Define fDivider             $0040
#Define fDisabled            $0080
#Define fItemStruct          $0400
#Define fOutline             $0800
#Define fShadow              $1000
#Define ItemTitleRefShift   $4000
```

`ItemTitleRefShift` is used to form the position for a reference, and should be multiplied by one of these values:

```
#Define RefIsPtr             $0000
#Define RefIsHandle          $0001
#Define RefIsResource        $0002
```

Using these constants, a typical `flags` entry is

```
RefIsResource*ItemTitleRefShift + fDivider
```

This sets the `flags` word to $4040.

rPicture $8002

The `rPicture` resource is a recording of QuickDraw II drawing commands, generally used as pictures for picture controls. There is no resource template for pictures in types.rez, so pictures are generally created with raw data statements.

It is extremely difficult to create pictures by hand. If you want to try, refer to Apple II Technical Notes #46: DrawPicture Data Format.

```
data rPicture (id) {
   $"data"
   };
```

`id`	This is the resource ID for the picture.
`data`	Hexadecimal data.

rPString $8006

Defines a p-string. `rPString` resources are used for a variety of purposes in this course. In most cases, these resources are referenced by other resources, as when they are used to define the names of menus.

```
resource rPString (id) {
   "string"
   };
```

`id`	This is the resource ID for the string.
`string`	The p-string. The length of the string is calculated automatically by the Rez compiler.

In many cases, this resource must be allocated with the attribute `noCrossBank` so the Resource Manager doesn't put the characters in two different banks in memory. See "Resource Attributes" at the beginning of this appendix for details.

rTextForLETextBox2 $800B

Defines a string consisting of a simple sequence of ASCII characters. Resource Manager calls can be used to compute the length of the string.

```
resource rTextForLETextBox2 (id) {
   "string"
   };
```

`id`	This is the resource ID for the string.
`string`	The string.

In many cases, this resource must be allocated with the attribute `noCrossBank` so the Resource Manager doesn't put the characters in two different banks in memory. See "Resource Attributes" at the beginning of this appendix for details.

rToolStartup $8013

This resource tells `StartUpTools` what tools to start and, to some extent, how to start them.

See Table 7-1 for a list of the tools used in this course, along with their dependencies as of System Disk 6.0.

```
resource rToolStartup (id) {
   mode,
   {
      tool, version,
      tool, version
   };
```

`id`	This is the resource ID for the resource.
`mode`	This parameter is the master SCB to pass to QuickDraw II. Among other things, this value determines whether the tools will be started in 640 mode or 320 mode. To start the tools in 640 mode, use `mode320`; to start the tools in 640 mode, use `mode640`. For other possibilities, see the *Apple IIGS Toolbox Reference Manual.*
tool	This is the tool number for a tool to start. You can specify as many tools as you like, but each must be followed by a tool version number entry.
`version`	This is the minimum allowed tool version number for the tool. Version numbers as of System 6.0 are listed in Table 7-1.

Here is a sample call that shows all of the tools that are normally started in any desktop application. Other tools used in this course are commented out – to start them, remove the `/*` to the left of the tool number. You can also find this resource on the solution disk in most of the sample programs from Lesson 7 on.

```
resource rToolStartup(1) {
   mode640,
   {
       3, $0302,                       /* Misc Tool */
       4, $0307,                       /* QuickDraw II */
       5, $0304,                       /* Desk Manager */
       6, $0301,                       /* Event Manager */
/*     8, $0303,                       /* Sound Tools */
      11, $0300,                       /* Integer Math Tool Set */
      14, $0303,                       /* Window Manager */
      15, $0303,                       /* Menu Manager */
      16, $0303,                       /* Control Manager */
      18, $0304,                       /* QuickDraw II Auxiliary */
/*    19, $0301,                       /* Print Manager */
      20, $0303,                       /* LineEdit Tool Set */
      21, $0304,                       /* Dialog Manager */
      22, $0301,                       /* Scrap Manager */
      23, $0303,                       /* SFO */
/*    25, $0104,                       /* Note Synthesizer */
      27, $0303,                       /* Font Manager */
      28, $0303,                       /* List Manager */
/*    34, $0103                        /* TextEdit Tool Set */
      }
   };
```

rVersion $8029

The Finder uses an `rVersion` resource with a resource of 1 to establish version information for an application. If the application has an `rVersion` resource, it should also have an `rBundle` resource.

```
resource rVersion (id) {
   {
      major,                                       /* Major revision */
      minor,                                       /* Minor revision */
      bug,                                         /* Bug version */
      stage,                                       /* Release stage */
      release,                                     /* Non-final release # */
      },
   region,                                         /* Region code */
   "string1",                                      /* Short version number */
   "string2",                                      /* Long version number */
   };
```

`id`	This is the resource ID for the `rVersion` resource.
`major`	Major revision level.
`minor`	Minor revision level.
`bug`	Bug fix level.
`stage`	Release stage. This can be any one of `development`, `alpha`, `beta`, `final` or `release`.
`release`	This is the non-final release number, generally used during the testing cycle.
`region`	This is the region code, indicating what country or population group the program is intended for. The current regions are:

```
verUS             verFrance          verBritain
verGermany        verItaly           verNetherlands
verBelgiumLux     verFrBelgiumLux    verSweden
verSpain          verDenmark         verPortugal
verFrCanada       verNorway          verIsrael
verJapan          verAustralia       verArabia
verArabic         verFinland         verFrSwiss
verGrSwiss        verGreece          verIceland
verMalta          verCyprus          verTurkey
verYugoslavia     verYugoCroatian    verIndia
verIndiaHindi     verPakistan        verLithuania
verPoland         verHungary         verEstonia
verLatvia         verLapland         verFaeroeIsl
verIran           verRussia          verIreland
verKorea          verChina           verTaiwan
verThailand
```

`string1`	Name of the product. This string may be empty. It is displayed by the Finder when the user asks for information about the icon.
`string2`	Additional information about the product, generally copyright information. This string may be empty. It is displayed by the Finder when the user asks for information about the icon.

rWindColor $8010

This resource defines a color table for a window.

```
resource rWindColor (id) {
   frameColor,
   titleColor,
   tBarColor,
   growColor,
   infoColor,
   };
```

`id`	This is the resource ID for the color table.

See "Window Color Tables" in Appendix A for details about the color table entries.

rWindParam1 $800E

This resource is used to create a window.

```
resource rWindParam1 (id) {
   wFrameBits,
   wTitle,
   wRefCon,
   wZoom,
   wColor,
   {wXOrigin,wYOrigin},
   {wDataH,wDataW},
   {wMaxH,wMaxW},
   {wScrollHor,wScrollVer},
   {wPageHor,wPageVer},
   wInfoRefCon,
   wInfoHeight,
   wPosition,
   wPlane,
   wControlList,
   wInDesc
   };
```

`id` This is the resource ID for the resource.

`wControlList` This is a reference to the control or list of controls defined for this window, as determined by bits 7-0 of `wInDesc`.

`wInDesc` This is a flags word, defined as follows:

bits 15-12 Reserved; set to 0.
bits 11-10 Defines the kind of reference in the `wColor` field:
- 00 The reference is a pointer.
- 01 The reference is a handle.
- 10 The reference is a resource ID for an `rWindColor` resource.

bits 9-8 Defines the kind of reference in the `wTitle` field:
- 00 The reference is a pointer.
- 01 The reference is a handle.

	10 The reference is a resource ID for an `rPString` resource.
bits 7-0	Defines the kind of reference in the `wControlList` field. There are several possibilities; the ones that are used most commonly, and that apply to this course, are:
	0 The reference is a pointer to a single control template. (This is set to nil in a resource, indicating that there are no controls in the window.)
	2 The reference is a resource ID for an `rControlTemplate` resource.
	9 The reference is a resource ID for an `rControlList` resource.

See "Window Parameters" in Appendix A for details about the remaining fields in this resource.

Appendix C – Where to Go for More Information

If you think anyone sits down with a book or two and learns all there is to know about programming, or that experienced programmers never get stuck, think again. You never learn it all. You never get to the point where you can write anything without talking to someone else.

This appendix gives you a few idea about where to get more information and where to go to find kindred spirits to talk about programming or get help with tough programming problems.

All of this information is subject to change. If I'd written this appendix a year ago, a lot of the information you see here would be incorrect by now. If you are reading this appendix a year after I write it, some of the information will no doubt have changed, especially things like where to get technical notes and what online services are good. Use this appendix as a starting place and an idea center, not as the final word on where to go for information.

Apple IIGS Toolbox Reference Manuals

Apple IIGS Toolbox Reference: Volume 1
Addison-Wesley Publishing Company, Inc.
1988, $26.95
741 pages, table of contents

Apple IIGS Toolbox Reference: Volume 2
Addison-Wesley Publishing Company, Inc.
1988, $26.95
686 pages, index (includes Volume 1), table of contents

Apple IIGS Toolbox Reference: Volume 3
Addison-Wesley Publishing Company, Inc.
1990, $39.95
1032 pages, index, table of contents

Volumes 1 and 2 of the toolbox reference manuals are the original, official technical description of the Apple IIGS toolbox for programmers. Volume 3 is an update to the first two volumes, adding information about new tools, new tool calls in old tools, and clarifying and correcting the first two volumes.

These books were written by Apple's staff, drawing on technical notes written by the programmers who designed and implemented the toolbox. These books contain a tool by tool, call by call breakdown of the toolbox, along with supplementary information that gives you overviews, background, and programming tips for the toolbox. These three volumes are three of the five books you absolutely must buy if you intend to go beyond this course.

A lot of folks have complained bitterly about the toolbox reference manuals because they don't teach you to use the toolbox. Their criticism is correct, but beside the point. Complaining because these books don't teach you to write toolbox programs is like a prospective novelist complaining because the Oxford English Dictionary doesn't teach you to write good prose. So? The Oxford English Dictionary is a reference for the words in the English language, not a how-to book about writing. The Apple IIGS toolbox reference manuals are a complete reference to the toolbox. Just as a writer needs a dictionary, a toolbox programmer needs these three books. They don't teach you how to write toolbox programs (hopefully this book did!), but they are an essential reference for anyone trying to write their own toolbox programs.

Programmer's Reference for System 6.0

Programmer's Reference for System 6.0
Byte Works, Inc.
1992; $45.00 (notebook) or $35.00 (bound)
about 350 pages, index, table of contents

The 3 volumes of the Apple IIGS toolbox reference manual suite covers the Apple IIGS operating system as of System Disk 5.0. *Programmer's Reference for System 6.0*, due out about a month after this course and in final review by Apple as I write this appendix, picks up where *Apple IIGS Toolbox Reference: Volume 3* and *Apple IIGS GS/OS Reference* leave off. (The fact that the book isn't final yet explains the approximate page count.) It documents the new tools and new tool calls added to the toolbox between the release of System Disk 5.0 and System Disk 6.0, as well as the changes to GS/OS. You can also find details about Finder 6.0, especially how you write applications that work smoothly with the Finder.

Like the toolbox reference manuals, this book was developed from the technical documentation written by Apple's programmers. It was also reviewed by Apple's programming and technical support staff.

This is the fourth of the five essential books for anyone writing desktop programs for the Apple IIGS.

GS/OS Technical Reference Manual

Apple IIGS Toolbox Reference
Addison-Wesley Publishing Company, Inc.
1990; $28.95
487 pages, index, table of contents

This book is the technical description of the Apple IIGS disk operating system. It includes all of the GS/OS calls that let you manipulate disk files, as well as information about the System Loader and several major drivers and FSTs.

This is the fifth of the five essential books for Apple IIGS desktop programming.

There is a second volume to this book. The second volume was never released in a bound form, since it is intended for a very small set of programmers who want to add new drivers to the GS/OS operating system. Volume 2 is available through Resource Central. Resource Central and their relationship to the Apple II Programming Community is discussed a little later in this appendix.

Programmer's Introduction to the Apple IIGS

Programmer's Introduction to the Apple IIGS
Addison-Wesley Publishing Company, Inc.
1988; $32.95
477 pages, index, table of contents, disk

This book is a little dated, but still has some good information. It's an introduction to toolbox programming written by Apple's engineers very shortly after the Apple IIGS was released.

Firmware Reference Manual

Apple IIGS Firmware Reference
Addison-Wesley Publishing Company, Inc.
1987; $24.95
320 pages, index, table of contents

This is the technical reference for the low-level software and soft switches. It includes information about the monitor firmware, video firmware, serial port firmware, smartport firmware, interrupt handling, the keyboard microcontroller, the mouse, firmware entry points, a memory map, and softswitch locations.

You may never need this book. It's intended for people who are writing very low-level software or those creating new hardware for the Apple IIGS. You also need assembly language to make effective use of most of the information you'll find in this book.

Hardware Reference Manual

Apple IIGS Hardware Reference, Second Edition
Addison-Wesley Publishing Company, Inc.
1989; $26.95
296 pages, index, table of contents

This is the hardware reference for the Apple IIGS computer. The main audience is folks who are making cards for the computer, although there is some information about the hardware that is useful for some kinds of low-level assembly language programming. You'll find detailed information about the video displays, memory, the Ensoniq sound chip and supporting firmware, and the various input and output mechanisms.

Apple Human Interface Guidelines

Human Interface Guidelines: The Apple Desktop Interface
Addison-Wesley Publishing Company, Inc.
1987; $14.95
136 pages, index, table of contents

I almost put this book on the list of books you absolutely must have to write desktop programs, but technically you don't have to have it. Still, I highly, highly recommend it.

Knowing how to write a desktop program is a little like knowing how to use a hammer and saw – useful, but far from adequate. Besides knowing how to write desktop programs, you should also know how people use them. This book deals with that sort of issue. It talks about things like where to put buttons in dialogs, what should appear in a file menu, how to use color effectively to add to a program rather than distract the user, and so forth. You'll learn a lot about the computer, not just as a programmer, but also as a user and designer.

I recommend that you read this book cover to cover at least twice: once before you start to design your first desktop program, and once more about a year later.

Technical Notes

Apple publishes a group of information under the combined title of technical notes. Technical notes contain a lot of stuff; here's a sampling of the major kinds of information you'll find:

- Detailed information about tricky software and hardware issues, telling you how to get things done when it isn't obvious.

- Lists of the various things Apple keeps track of and assigns, like message center numbers, file types, and language numbers.

- Corrections to the Apple books listed in this appendix.

- Information about known bugs and changes in the operating system.

The technical notes are available in two forms, and are updated as needed – sometimes after just a few months. You can get the updates on disk, in either ASCII files for the Apple IIGS or as formatted files (generally using some Macintosh word processor). You can find the disks on major online services, in most user group libraries, or on Apple's developer CD ROM disks. Printed versions are also available. As I write this, you order printed versions from APDA, but be sure you say "Printed Apple II Technical Notes" at least three times, or you will end up with something for the Macintosh, or get sent somewhere else. It seems likely that the printed technical notes will be available from Resource Central in the very near future.

If you are programming professionally, or you consider yourself an advanced programmer, you should get the full set of technical notes and keep them up to date. Since the technical notes are readily available on disk, I'd suggest that everyone should pick up at least the disk based version.

Resource Central

Resource Central
P.O. Box 11250
Overland Park, KS 66207
Phone: (913) 469-6502
Fax: (913) 469-6507

The only way I can think of to describe Resource Central in a nutshell is as a worldwide Apple II advocate. Resource Central is certainly a business, but they're more than that, too. Resource Central publishes a number of newsletters concerning the Apple II, some on paper and some on disk. They manage at least one of the Apple II areas for an online service (GEnie). They have taken over as the sole source for a great deal of the technical information Apple creates for programmers and hardware engineers for the Apple II – information that isn't needed by enough people to go into a commercial book, like the toolbox reference manuals, but is still needed by some. Resource Central also holds the biggest and best conference for Apple II programmers. It's held in Kansas in July, but hey, it's held. Finally, they are an active and reputable discount mail-order distributor with the largest selection of products for the programmer I know of. (They don't limit themselves to programming products, but that's what we're discussing here.) If you're serious about the Apple II, I'd highly recommend investing in a phone call to get their latest list of products and services.

Apple Sponsored Developer Programs and Licensing

Apple Computer, Inc.
Apple Developer Programs
20525 Marriani Ave., M/S 75-2C
Cupertino, CA 95014-9968
Phone: (408) 996-1010

Apple has several programs for people who develop for Apple computers. Currently there are two categories, known as the Associates Program and the Partners Program. The Associates program is open to just about anyone, and costs $350 per year for the Macintosh and $150 for the Apple II. The Partners Program is open by application only, and you get to pay more if you're accepted. The difference is that you get a direct line to Apple's technical support staff via AppleLink, where you can ask all sorts of questions and get real responses from someone who knows what they are talking about.

You can also license a lot of software from Apple – anything from the System Disk, so people can boot your program; through the Rez compiler, which we licensed for this course.

If you are developing commercial software for the Apple II, get in touch with Apple for the latest information about their developer programs and licensing. Maybe it's something you need, and maybe not, but it's worth getting the details.

Online Services

Sometimes you just need to talk to another programmer, either to get help or just to chat with a person who shares your interest. You may also want to swap programs, source code, rumors, or wild stories about midnight entomology. On the other hand, there aren't likely to be too many Apple IIGS toolbox programmers living next door. That's where the online services come in. You can join chat sessions with other programmers, post source code for someone to help with a bug, ask questions, or search online libraries for sample code and utilities. Frankly, I think access to at least one major online service is essential for any programmer.

You will need a modem to get on any of these services, but that's about it. AppleLink and America Online will even send you the software you need, although AppleLink only runs on a Macintosh.

AppleLink

Apple Computer, Inc.
AppleLink MS 41-H
20300 Stevens Creek Blvd. Ste 245
Cupertino, CA 95014

$7 start up fee. $12 per month minimum charge. AppleLink charges by the amount of information you transfer, not by the time you spend online. The charge is $0.055 per kilo-character during prime time (6AM to 6PM Pacific Standard Time) and $0.045 per kilo-character during non-prime time.

America Online

America Online
8619 Westwood Center Dr.
Vienna, VA 22182
(703) 448-8700

$5.95 month minimum charge, which can be applied toward 1 free non-prime time hour. $10 per hour prime time (6AM to 6PM, Monday to Friday), $5 per hour during non-prime time. Call for a free startup kit, which includes all the software you need. You can contact the Byte Works on America Online by using keyword ByteWorks, or use keyword ADV to get to the Apple Developer group, which is an active group of Apple II programmers.

CompuServe

CompuServe
5000 Arlington Center Blvd.
Columbus, OH 43220

$7.95 month minimum charge, which gives unlimited access to a selection of services. For the paid areas, the charge is $6.30 at 300 baud, and $12.84 at 1200 baud or 2400 baud. Faster connections are available; call for prices. CompuServe has an active Apple II area.

GEnie

GTE Information Services
401 N. Washington Street
Rockville, MD 20850
(800) 638-9636

$4.95 month minimum charge, which gives unlimited access to a selection of services. The Apple II programming area is not in the free area. For the paid areas, the charge is $18 per hour prime time (8AM to 6PM, your local time, Monday to Friday), $6 per hour during non-prime time. The Byte Works monitors the A2Pro area, where you will find an active group of Apple II programmers.

User's Groups

If you live in a city large enough to have a stoplight, you probably live close to a user's group of some sort. If your city is large enough, you may even meet some other Apple IIGS toolbox programmers.

User's groups are a great place to meet other programmers. Many also have software libraries which include source code, Apple's system software, and the disk form of the technical notes.

Apple Computer keeps a list of all of the Apple user's groups. Call (800) 538-9696, extension 500 for information about your closest Apple user's group. Call (408) 996-1010 (Apple's main number) and ask for "Apple User's Group Connection Hotline" for more detailed information about user's groups. You might also check with local computer stores or universities for non-Apple groups of programmers.

Appendix D – Tips for ORCA/Pascal

MoreRecent

Once you start using resources, your programs will consist of at least two source files: the Pascal program and the resource description file. The easiest way to deal with programs that use more than one separately compiled file is to create a script. The course discuses this at the proper time, showing you how to write a simple script that will compile only those parts of your program that need to be compiled. To do that with shells prior to the 2.0 version of the ORCA shell, you need the `MoreRecent` utility.

`MoreRecent` is on the solution disk, stuffed inside the folder for lesson 3. To install `MoreRecent`, copy the files from the utilities folder in the Lesson.3 folder into the utilities folder for your ORCA/Pascal system. If you aren't sure where the ORCA/Pascal utilities folder is, use the `SHOW PREFIX` command from the shell or shell window – prefix 6 is the utility prefix. Next, edit your SYSCMND file, which you will find in your ORCA/Pascal system folder (that's prefix 2), and add the line

```
MORERECENT    U
```

You can actually find this line in a file in the System folder in the Lesson.3 folder.

Once you save the modified SYSCMND file, reboot. If you've done everything correctly, when you type `HELP` from the shell or shell window, `MORERECENT` will be listed as a command. Typing `HELP MORERECENT` will print a help file showing how the command is used.

Installing Rez

If you don't already have Rez installed in your ORCA/Pascal system, you will need to install it from the solution disks before you can work most of the problems in this course. If you are using floppy disks, be sure and read the next section before you try this installation.

The resource compiler is located on your solution disks in a folder called Rez. In that folder, you'll find the following files:

Languages/Rez	This is the resource compiler itself. It should be copied to your ORCA/Pascal languages folder. That's the folder where the Pascal compiler is located. Unless you've changed the name, the languages folder is called LANGUAGES.
Libraries/RInclude	This is a folder containing interface files for the resource compiler. Copy the folder and all of the files inside of the folder into the ORCA/Pascal libraries folder. That's the same folder that has the PASLIB and SYSLIB libraries, as well as the ORCAPascalDefs folder. Unless you've changed the name, the libraries folder is called LIBRARIES.
SysCMND	This is a text file containing a line that needs to be copied into the SYSCMND file in your ORCA/Pascal system folder.

Once you copy all of these files into the proper spot and reboot, you should see REZ listed as a language when you use the `SHOW LANGUAGES` command from either the shell or PRIZM's shell window.

Using Small Systems

Back when we released our first development system for the Apple IIGS, you had to have at least 512K. It didn't even take a 3.5" floppy, since we made the system available on 5.25" disks. Somehow this seem sort of quaint, now. Using the latest version of Apple's system disk, you have to have 1.25M just to run the Finder.

The smallest possible system you can use to work through this course is a 1.25M Apple IIGS (1.125M for ROM 03) with one 3.5" disk drive and one other disk drive. Frankly, you're going to have a lot of trouble if you don't have at least 1.75M of memory and a hard disk. And, unless you have both 1.75M of memory and a hard disk, you must stick with the text-based development environment – the PRIZM desktop development system can't be used with Pascal and Rez on a smaller system.

If you have a hard disk, your only real concern is how much memory you have available. With 1.25M, you can't use Rez, Pascal and PRIZM together, so you need to set up a text system. If you aren't sure how to do that, the ORCA/Pascal disks come with an installer script that will set up a text system for you.

If you are working from an 800K 3.5" floppy disk drive, you can't use PRIZM, Pascal and Rez together, no matter how much memory you have, because they won't all fit on a single floppy disk. (Technically, you can do it if you have a second 3.5" drive; see your reference manual for details.) In that case, I'd suggest setting up a minimum system. To do that, follow these steps:

1. Make a copy of your ORCA/Pascal 1.4 program disk. (If you are working with a later version of ORCA/Pascal, these steps will work, but some of the files may already be in a different place.) The rest of the steps should be made with the copy, not the original.

2. Delete these files:

   ORCA.PASCAL/SYSTEM/LOGIN
   ORCA.PASCAL/UTILITIES/HELP/= (delete all of the files in this folder)
   ORCA.PASCAL/UTILITIES/PRIZM
   ORCA.PASCAL/UTILITIES/CRUNCH
   ORCA.PASCAL/UTILITIES/MAKELIB
   ORCA.PASCAL/LIBRARIES/ORCAPASCALDEFS/ACE.INT
   ORCA.PASCAL/LIBRARIES/ORCAPASCALDEFS/DESKTOPBUS.INT
   ORCA.PASCAL/LIBRARIES/ORCAPASCALDEFS/FINDER.INT
   ORCA.PASCAL/LIBRARIES/ORCAPASCALDEFS/INTEGERMATH.INT
   ORCA.PASCAL/LIBRARIES/ORCAPASCALDEFS/MIDI.INT
   ORCA.PASCAL/LIBRARIES/ORCAPASCALDEFS/MULTIMEDIA.INT
   ORCA.PASCAL/LIBRARIES/ORCAPASCALDEFS/ORCASHELL.INT
   ORCA.PASCAL/LIBRARIES/ORCAPASCALDEFS/PRODOS.INT
   ORCA.PASCAL/LIBRARIES/ORCAPASCALDEFS/SCHEDULER.INT
   ORCA.PASCAL/LIBRARIES/ORCAPASCALDEFS/SEQUENCER.INT

   This gives you enough room to install the resource compiler (see the previous section), plus a little work space.

3. Install Rez and, if you need it, `MoreRecent`, as described in the first two sections of this appendix.