Carbon Porting Guide

(Legacy)

Carbon > Porting

2002-12-01

Contents

Introduction **Introduction to Carbon Porting Guide 9**

What Is Carbon? 9
What Are the Benefits of Carbon? 10
What Is in Carbon Today? 10
What's Not in Carbon? 11
How Does Carbon Work? 11
Carbon and the Mac OS Application Model 12
- Preemptive Scheduling and Application Threading 12
- Separate Application Address Spaces 12
- Virtual Memory 13
- Resources 13
- Code Fragments and the Code Fragment Manager 13
- Mixed Mode Manager 13
- Printing 13
- Control Panels 14
- The Trap Table 14
- Standard and Custom Definition Procedures 14
- Application-Defined Functions 14
- Data Structure Access 14

Additional Information and Feedback 15

Chapter 1 **Preparing Your Code for Carbon 17**

Using Carbon Dater 17
- Analyzing Your Application 18
- Reading the Report 18
- Additional Reports 19
- The Carbon Specification 19

Essential Steps for Porting Your Application 20
- Make Sure All of Your Code Is PowerPC-Native 20
- Update to the Current Universal Interfaces 20
- Use the Carbon SDK 20
- Target Mac OS 8 and 9 First 20
- Begin With CarbonAccessors.o 21
- Use Casting Functions to Convert DialogPtrs and WindowPtrs 21
- Replace Macro Calls to the Mixed Mode Manager With UPP Accessor Functions 22
- Move Custom Definition Procedures Out of Resources 22
- Remove Direct Access to Low-Memory Globals 23
- Use DebuggingCarbonLib 24
- Modify or Conditionalize Your Headers 25
- Update Modified or Obsolete Functions 25

Adopt Required Carbon Technologies 25
Add a 'plst' 0 Resource 26
Conditionalize Quit Menu Items 26
Additional Porting Issues 27
Determine the Appropriate CarbonLib Version 27
Draw Only Within Your Own Windows 28
Do Not Patch Traps 28
Don't Pass Pointers Across Processes 28
Do Not Write to Your Application's Resource Fork 28
Check Your OpenGL Code 29
Examine Your Plug-ins 29
Linking to Non-Carbon-Compliant Code 29
Window Manager Issues 30
Optimizing Your Code for Carbon 33
Manage Memory Efficiently 34
Avoid Polling and Busy Waiting 34
Use "Lazy" Initialization for Shared Libraries 35
Adopt HFS Plus APIs 35
Consider Mach-O Executables 35
Move Resources to Data Fork–Based Files 36
Consider Using Bundles 36
Begin Transitioning to the Aqua Interface 37
Adopt a Terse Name for the Application Menu 37
Provide Thumbnail Icons for Your Application 37

Chapter 2 **Building Carbon Applications 41**

Native Mac OS 9 Versus Mac OS X's Classic Environment 41
Development Scenarios 41
Using CodeWarrior to Build a CFM Carbon Application 41
Using CodeWarrior to Build a Mach-O Carbon Application 42
Using Project Builder to Build a Mach-O Carbon Application 42
Building a CFM Carbon Application With CodeWarrior 42
Preparing Your Development Environment 42
Building Your Application 43
Running Your Application on Mac OS 9 43
Running Your Application on Mac OS X 43
Building a Mach-O Carbon Application With CodeWarrior 44
Preparing Your Development Environment 44
Building Your Application 44
Running Your Application on Mac OS X 44
Building a Mach-O Carbon Application With Project Builder 44
Building Applications Using MPW 44
Debugging Your Application 45

Chapter 3 **A Porting Example 47**

The Sample Application 47
Obtaining the Carbon Dater Report 48
The Basic Port 50
  Make Sure All of Your Code is PowerPC–Native 50
  Update to the Current Universal Interfaces and Use the Carbon SDK 50
  Target Mac OS 8 and 9 First 51
  Begin With CarbonAccessors.o 51
  Use Casting Functions to Convert DialogPtrs and WindowPtrs 53
  Modify or Conditionalize Your Headers 53
  Replace Macro Calls to the Mixed Mode Manager With UPP Accessor Functions 53
  Move Custom Definition Procedures Out of Resources 53
  Remove Direct Access to Low-Memory Globals 53
  Use DebuggingCarbonLib 54
  Update Modified or Obsolete Functions 54
  Adopt Required Carbon Technologies 55
  Add a 'plst' 0 Resource 55
  Conditionalize Quit Menu Items 55
  Cleanup 56
Additional Changes for Aqua 57
  Adjust the Window Size 57
  Modify the About Box 57
The Carbon Version of Sample 57

Chapter 4 **New Carbon Technologies 69**

Carbon Event Manager 69
Core Foundation 70
DataBrowser 70
Multilingual Text Engine (MLTE) 71
An Example: Adding Carbon Events to Sample 71
  Standard Event Handlers 72
  The Basic Conversion 72

Appendix A **New Carbon Functions 81**

Custom Definition Procedures 81
  Changes to WDEFs 81
  Changes to MDEFs 81
Functions for Accessing Opaque Data Structures 82
  Casting Functions 82
  Accessor Functions 83
  Utility Functions 91
Functions in CarbonAccessors.o 91
Debugging Functions 95

CheckAllHeaps 95
IsHeapValid 95
IsHandleValid 95
IsPointerValid 95
Resource Chain Manipulation Functions 95
InsertResourceFile 95
DetachResourceFile 96
FSpResourceFileAlreadyOpen 96

Appendix B **The Sample Application 97**

Document Revision History 113

Index 119

Figures, Tables, and Listings

Introduction **Introduction to Carbon Porting Guide 9**

Figure I-1 Current and future composition of the Carbon API 10
Figure I-2 Calling Carbon functions on Mac OS X and Mac OS 8 and 9 12

Chapter 1 **Preparing Your Code for Carbon 17**

Figure 1-1 Outline feedback as a user resizes a window 32
Figure 1-2 Thumbnail icons in a `.icns` file, displayed in Icon Browser 38
Table 1-1 Summary of Carbon low memory accessor support 24

Chapter 3 **A Porting Example 47**

Figure 3-1 The Sample application 47
Figure 3-2 A Carbon Dater report 48
Figure 3-3 The About box for Sample 57
Figure 3-4 The Carbon version of Sample on Mac OS X 57
Table 3-1 Carbon Dater output for incompatible functions 48
Listing 3-1 Carbon version of Sample.c 57
Listing 3-2 Carbon version of SampleInit.c 66

Chapter 4 **New Carbon Technologies 69**

Listing 4-1 Application-level event handler for Sample 74
Listing 4-2 Window event handler for Sample 77

Appendix A **New Carbon Functions 81**

Table A-1 Summary of Carbon Human Interface Toolbox accessors 85
Table A-2 QuickDraw accessor functions 88
Table A-3 Functions in CarbonAccessors.o 91
Table A-4 Functions removed from CarbonAccessors.o 94
Listing A-1 Example of unsupported data structure access 83
Listing A-2 Example of using Carbon-compatible accessor functions 84

Appendix B **The Sample Application 97**

Listing B-1 Sample.c 97
Listing B-2 SampleInit.c 108

Introduction to Carbon Porting Guide

Important: The information in this document is obsolete and should not be used for new development.

The Carbon Porting Guide is intended to help experienced Macintosh developers convert existing Mac OS applications into Carbon applications that can run on Mac OS X as well as Mac OS 8 and 9. It contains detailed information about how to adapt and build your application using the Carbon API as well as step-by-step examples of the porting process.

To make the Carbon transition as smooth as possible, you should also be familiar with the following documents before beginning your port:

- *Inside Mac OS X: System Overview*. This document contains in-depth discussions of Mac OS X features and architecture. It also contains more detailed information about some topics discussed in this document.
- *Inside Mac OS X: Aqua Human Interface Guidelines*. This document provides the human interface guidelines for the Mac OS X user interface.

This chapter introduces Carbon and provides an overview of the changes you'll need to be aware of as you convert your application.

What Is Carbon?

Carbon is the set of programming interfaces derived from earlier Mac OS APIs that can run on Mac OS X. Some of these APIs have been modified or extended to take advantage of Mac OS X features such as preemptive multitasking and protected memory.

In addition to being able to run on Mac OS X, Carbon applications built for Mac OS X can also run on Mac OS 8 and 9 when the CarbonLib system extension is installed. (As always, you should test for the existence of specific features before using them.)

Carbon includes about 70 percent of the existing Mac OS APIs, covering about 95 percent of the functions used by applications. Because it includes most of the functions you rely on today, converting to Carbon is a straightforward process. Apple provides tools and documentation to help you determine the changes you will need to make in your source code, as well as the header files and libraries necessary to build a Carbon application.

Mac OS X brings important new features and enhancements that developers have asked for, and Carbon allows you to take advantage of them while preserving your investment in Mac OS source code. As Apple moves the Mac OS forward, Carbon ensures you won't be left behind.

What Are the Benefits of Carbon?

Carbon applications gain these benefits when running under Mac OS X:

- Greater stability. Protected address spaces help prevent errant applications from crashing the system or other applications.
- Improved responsiveness. Each application is guaranteed processing time through preemptive multitasking, resulting in a more responsive user experience.
- Dynamic resource allocation. More efficient use of system resources, including the elimination of fixed size heaps, means your application can allocate memory and other shared resources based on actual needs rather than predetermined values. Each application can access up to 4GB of potential addressable memory.
- Aqua look and feel. Apple's newest user interface is available only to applications that run natively on Mac OS X.

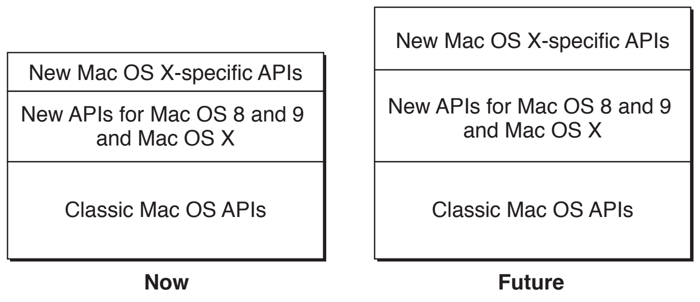
What Is in Carbon Today?

The Carbon programming interface consists of the following types of APIs:

- Classic Mac OS APIs that can run unchanged on Mac OS X. These comprise the majority of the APIs in your current application.
- Classic Mac OS APIs that have been modified to work on Mac OS X. For example, to operate properly in a preemptively-scheduled environment, a function may now require an additional parameter to specify the context (or process) to which it belongs.
- New APIs that can run on both Mac OS X and Mac OS 8 and 9. For example, Core Foundation and the Carbon Event Manager provide additional benefits for Carbon applications but are not required for porting.
- New APIs that are available only on Mac OS X.

Currently, the Classic Mac OS APIs make up the largest proportion of Carbon APIs, as shown in Figure I-1 (page 10). However, as Carbon evolves to take advantage of new features in Mac OS X, new Mac OS X-specific APIs will be added that enhance its capabilities.

Figure I-1 Current and future composition of the Carbon API


New Mac OS X-specific APIs
New APIs for Mac OS 8 and 9 and Mac OS X
Classic Mac OS APIs
Now
New Mac OS X-specific APIs
New APIs for Mac OS 8 and 9 and Mac OS X
Classic Mac OS APIs
Future

What's Not in Carbon?

If Carbon does not support a Classic Mac OS function, it is generally for one of the following reasons:

- The function performs actions that are illegal or make no sense in Mac OS X. For example, functions that are 68K-specific, or functions that allocate memory in the system heap (Mac OS X has no concept of a system heap).
- The function directly accesses hardware. The Carbon environment was designed to be fully abstracted from hardware, so such functions are not allowed.
- The function was there for legacy purposes only, and has more modern replacements (for example, File Manager functions that use working directories).

In addition, certain Classic Mac OS programming practices are no longer allowed:

- No 68K code allowed. All Carbon code must be PowerPC-based.
- No trap table access. The trap table and Patch Manager are 68K-specific.
- Limited access to data structure fields. See "Data Structure Access" (page 14).

How Does Carbon Work?

Carbon lets you create one executable file that can run on both Mac OS X and Mac OS 8 and 9. You accomplish this by linking your application with a single stub library, `CarbonLibStub`, at build time. At runtime your application links with the appropriate Carbon implementation stored as shared libraries (sometimes referred to as DLLs).

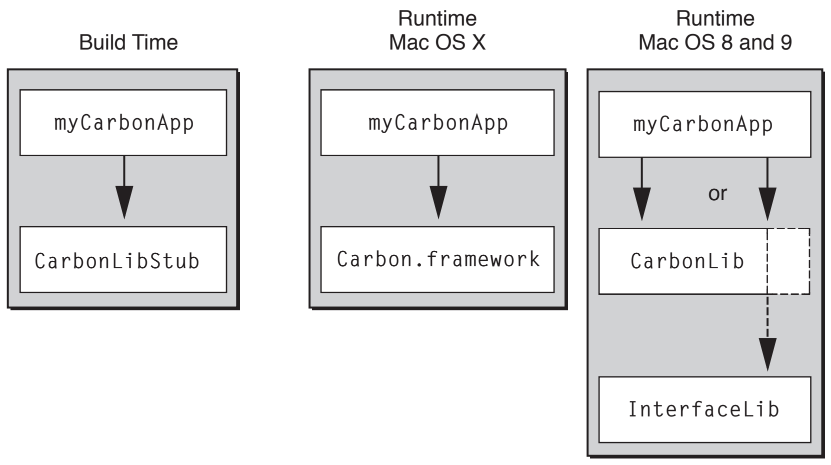
On Mac OS X, your application links dynamically to the Carbon framework, which is a hierarchy of libraries and resources that contains the implementation of Carbon.

On Mac OS 8 and 9, the Carbon implementation is stored as a system extension named `CarbonLib`. This library contains two types of elements:

- Implementations of all functions specific to Carbon.
- Exports of functions currently available in system software. For example, calls to a Menu Manager function available in both Carbon and Mac OS 8 and 9 will merely call through to the implementation in InterfaceLib.

Figure I-2 shows Carbon functions called on Mac OS X and Mac OS 8 and 9.

Figure I-2 Calling Carbon functions on Mac OS X and Mac OS 8 and 9



In general, for a pure Carbon application, the only library you should link against is `CarbonLib`. See "Linking to Non-Carbon-Compliant Code" (page 29) for special cases where you may need to link to other libraries.

Carbon and the Mac OS Application Model

The Mac OS application model remains fundamentally unchanged in Carbon. Carbon applications employ system services in essentially the same manner for both Mac OS 8 and 9 and Mac OS X. But because Mac OS 8 and 9 and Mac OS X are built on different architectures, there will be slight differences in the way your application uses some system services. This section highlights the most important changes you need to be aware of. "Preparing Your Code for Carbon" (page 17) provides more detailed information on each of these subjects.

Preemptive Scheduling and Application Threading

In Mac OS X, each Carbon application is scheduled preemptively against other Carbon applications. For calls to most low-level operating system services, Mac OS X also supports preemptive threading within an application. Because most Human Interface Toolbox functions are not reentrant, however, a multithreaded application will initially be able to call these functions only from cooperatively scheduled threads. Thread-based preemptive access to all system services—including the Human Interface Toolbox—is an important future direction for the Mac OS.

In both Mac OS 8 and 9 and Mac OS X, you can use the Multiprocessing Services API to create preemptively scheduled tasks.

Separate Application Address Spaces

In Mac OS X, each Carbon application runs in its own protected address space. An application can't reference memory locations—or corrupt another application's data—outside of its assigned address space. This separation of address spaces increases the reliability of the user's system, but it may require small programming changes to applications that use zones, system memory, or temporary memory. For example, temporary memory allocations in Mac OS X will be allocated in the application's address space, and Apple will define new functions for sharing memory between applications. "Manage Memory Efficiently" (page 34) provides more detailed information about memory management for Carbon applications.

Virtual Memory

Mac OS X uses a dynamic and highly efficient virtual memory system that is always enabled. Your Carbon application must therefore assume that virtual memory is turned on at all times. In addition, the Mac OS X virtual memory system introduces a number of changes to the addressing model that are discussed in "Manage Memory Efficiently" (page 34).

Resources

Mac OS X supports traditional Resource Manager resources, but you should consider moving resources to the data fork of your application and accessing them using Core Foundation CFBundle APIs instead. Doing so will ensure that this information will not be lost if your application is copied by a method that does not recognize resource forks. See "Move Resources to Data Fork–Based Files" (page 36) and "Consider Using Bundles" (page 36) for more information.

Note that you can no longer store executable code in resources. See "Move Custom Definition Procedures Out of Resources" (page 22) for more information.

Code Fragments and the Code Fragment Manager

Carbon fully supports the Code Fragment Manager, and the Mac OS X runtime environment supports code compiled into code fragments. For Mac OS X, however, all code fragments must contain only native PowerPC code. In addition, resource-based fragments are no longer allowed.

Mixed Mode Manager

While the Mixed Mode Manager is no longer needed to handle calls between PowerPC and 68K code, there may be instances where it must handle calls between CFM-based code and Mach-O code (the native executable format on Mac OS X). In any case, you must replace the macros for creating and disposing routine descriptors with new Carbon functions for creating, invoking, and disposing universal procedure pointers (UPPs). See "Replace Macro Calls to the Mixed Mode Manager With UPP Accessor Functions" (page 22) for more information.

Printing

Carbon introduces a new Printing Manager that allows applications to print on Mac OS 9 using current printer drivers and on Mac OS X using new printer drivers. The functions and data types defined by the Carbon Printing Manager are contained in the header files `PMApplication.h`, `PMCore.h`, and `PMDefinitions.h`. Documentation for the Carbon Printing Manager is provided with the Mac OS X Developer Tools CD and at the following website:

http://developer.apple.com/documentation/Carbon/Reference/CarbonPrintingManager_Ref/

Control Panels

Carbon does not support control panels. If possible, you should package your control panel as an application.

The Trap Table

The trap table is a 68K-specific mechanism for dispatching calls to Mac OS Toolbox functions. Because Mac OS X does not support 68K code, the Trap Manager is unavailable in Carbon, and your application should not dispatch calls through the trap table. Likewise, the Patch Manager is unsupported in Carbon, and your application should not attempt to patch the trap table or any operating system entry points. If your application relies on patches, please tell us why, so that we can help you remove this dependency.

Standard and Custom Definition Procedures

Carbon supports the standard Mac OS definition procedures (also known as defprocs) for such human interface elements as windows, menus, and controls. Custom definition procedures are also supported (as long as they are compiled as PowerPC code), but there are new procedures for creating and packaging them. These new functions are discussed in "Move Custom Definition Procedures Out of Resources" (page 22) and "Custom Definition Procedures" (page 81).

Application-Defined Functions

Carbon supports most Mac OS application-defined (callback) functions. Mac OS X fully supports callback functions within an application's address space. In Carbon, callback functions use native PowerPC conventions instead of 68K conventions, but Carbon doesn't change these function definitions. As usual you should pass universal procedure pointers when specifying your callback functions.

Data Structure Access

So that future versions of Mac OS can support access to all system services through preemptive threads, Carbon limits direct application access to some Mac OS data structures. Carbon allows three levels of data structure access, depending on which is appropriate for a given structure:

- Direct access—your application can read from and write to the data structure without restriction.
- Direct access with notification—your application can read from and write to the data structure, but after modifying the structure your application must call a function to notify the operating system that the structure has been changed.
- Indirect access—your application has no direct access to the data structure. Instead, your application can obtain and set values in the structure only by using accessor functions. Structures of this type are said to be "opaque" because their contents are not visible to applications.

Opaque data structures and the functions for using them are discussed in "Functions for Accessing Opaque Data Structures" (page 82).

Additional Information and Feedback

Apple is working hard to deliver the features and performance you expect from Carbon. You can keep abreast of current developments by visiting the Carbon website at

http://developer.apple.com/carbon/index.html

where you'll find the complete Carbon Specification, preliminary documentation, and links to other useful information.

If you have comments or suggestions about Carbon, please send them to `carbon@apple.com`.

Preparing Your Code for Carbon

This chapter describes the modifications you need to make to your source code to create a Carbon application. These changes are divided into three categories:

- Essential changes. Applications that follow these steps should run on Mac OS X, but may suffer from performance or responsiveness problems.
- Other porting issues. These are topics that could affect the porting process depending on the capabilities and needs of your application.
- Optimization steps. This section describes steps and issues to consider so your application can take best advantage of Mac OS X. Apple highly recommends that you address at least some of the topics described in this section.

Note: Carbon also provides new technologies that give additional functionality to your application. While entirely optional, these additions can improve performance, enhance the user experience, and even simplify future code development. See "New Carbon Technologies" (page 69) for more information.

For more details about porting an application, see "A Porting Example" (page 47)

Technote TN2003, "Moving Your Code to Mac OS X," contains additional porting information that you may find useful:

http://developer.apple.com/technotes/tn/tn2003.html

To make your job easier, begin by using the Carbon Dater tool to analyze the current compatibility level of your application.

Using Carbon Dater

Apple has developed a tool called Carbon Dater to analyze compiled applications and libraries for compatibility with Carbon. You can use Carbon Dater to obtain information about the compatibility of your existing code and the scope of your future conversion efforts.

Carbon Dater works by examining PEF containers in application binaries and CFM libraries. It compares the list of Mac OS symbols your code imports against Apple's database of Carbon-supported functions.

Note: Because Carbon Dater examines only PEF containers, it cannot examine 68K-based executable files. If you are porting an older 68K application, you must convert it to PowerPC before running the Carbon Dater tool.

Important: The Carbon Dater application is no longer available online and is no longer accepting reports.

Analyzing Your Application

Using Carbon Dater is a two-step process. You begin by dropping your compiled application or CFM library file onto the Carbon Dater tool. The tool examines the first PEF container in your file and outputs a text file named *filename*`.CCT` (Carbon Compatibility Test). You can drop more than one file onto the Carbon Dater tool to get a combined report, but the tool examines only the first PEF container in each file.

The CCT file contains a list of all the Mac OS functions referenced by your code. If applicable, it may also include information about your application's use of direct access to low memory addresses or about resources stored in the system heap.

The second step is to send your CCT file to Apple for analysis. The information gathered by the Carbon Dater tool is used to create a compatibility report for your application. Attach the CCT file as an email enclosure (preferably compressed) and send it to `CarbonDating@apple.com`.

Important: Carbon Dater does not expose any proprietary information about your product. The CCT file only lists calls to Mac OS functions and certain other potential compatibility issues. You can examine the CCT file to verify its contents.

Reading the Report

The CCT file you send to Apple will be processed by an automated analysis tool. The analyzer compares the list of Mac OS functions your code calls against Apple's Carbon API database, and returns a report to you by email. This report is an HTML document that provides a snapshot of your application's Carbon compatibility level.

Analysis of Imports

For each Mac OS function your code calls that is not fully supported in Carbon, the compatibility report specifies whether the function is

- supported but modified in some way from how it is used in previous versions of the Mac OS
- supported but not recommended—that is, you can use the function, but it may not be supported in the future
- unsupported
- not found in the latest version of Universal Interfaces

The report includes a chart that shows the percentages of Mac OS functions in each category. For many functions, the report also describes how to modify your application. For example, text accompanying an unsupported function might describe a replacement function or recommended workaround.

Analysis of Access to Low Memory Addresses

This section of the compatibility report lists instances where your code makes a direct access to low memory. For information on how to access low memory correctly, see "Remove Direct Access to Low-Memory Globals" (page 23). If the tested code was built with symbolic debugging information enabled, the report specifies the names of the routines that access low memory directly.

Many of the low-memory accessor functions currently defined in the Universal Interfaces are implemented as inline macros that insert load or store instructions directly in your code. Carbon Dater can't tell the difference between one of these macros and the code you wrote yourself, so you'll need to verify that you're using an approved accessor function.

Analysis of Resources Loaded Into the System Heap

This section of the compatibility report lists resources that have their system heap bit set, indicating they should be stored in the system heap. For each flagged resource, the report lists the resource type and ID, as well as the resource name if one is available. Applications do not have access to the system heap in Mac OS X, so Carbon applications cannot store resources there.

Additional Reports

You can obtain additional compatibility reports as often as you wish. This is a good way to see how much progress you've made in your porting effort. Also, as work on Mac OS X and Carbon continues, there may be changes in the level of support for some functions, which Carbon Dater may bring to your attention.

Important: The Carbon dating process cannot guarantee that your application is entirely compatible with Carbon and Mac OS X, even if your report lists no specific incompatibilities. For example, applications might access low memory in a way that is not supported but that cannot be detected by the compatibility analyzer.

The Carbon Specification

To determine compatibility, Carbon Dater uses the Carbon Specification available at

http://developer.apple.com/documentation/carbon/

You can browse this document for general compatibility information. Apple updates the Carbon Specification regularly to reflect the latest state of the Carbon APIs.

Essential Steps for Porting Your Application

This section describes the bare minimum steps you must take to port your application to Carbon. Applications ported by following these instructions will run on Mac OS 8 and 9 and Mac OS X, but may not function optimally. To further improve performance and responsiveness, see the guidelines in "Optimizing Your Code for Carbon" (page 33).

In addition to reading this section, you should also read the information provided in "Additional Porting Issues" (page 27) before beginning to port your application.

Make Sure All of Your Code Is PowerPC-Native

Because Mac OS X requires 100% native PowerPC code, you will need to remove any dependencies on 68K instructions. This applies to custom definition procedures (defprocs) and plug-ins as well as your main application. See "Move Custom Definition Procedures Out of Resources" (page 22) and "Custom Definition Procedures" (page 81) for information about new functions for creating native defprocs.

Update to the Current Universal Interfaces

Your transition to Carbon will be easier if your application already compiles using the latest version of Universal Interfaces (as of this writing, the most recent version is 3.4). Although updating is not a requirement, doing so will minimize the number of compatibility problems. Once your project compiles without errors, you should switch to the Carbon headers provided with the Carbon SDK.

You'll find the most recent Universal Interfaces on Apple's website at

http://developer.apple.com/sdk/

Use the Carbon SDK

The Carbon SDK contains the headers, stub libraries, extensions and other material that you will need to build your Carbon application. You can download it from the following website:

http://developer.apple.com/carbon/index.html

Target Mac OS 8 and 9 First

To ease the transition to Carbon, you should initially focus on getting your application running on Mac OS 8 and 9 with the CarbonLib extension. Then you can test your application on Mac OS X.

Note that just because your Carbon application runs on Mac OS 8 and 9, there is no guarantee that it will correctly run on Mac OS X. For example, Mac OS X is stricter about direct casting of types, so what is allowable on Mac OS 8 and 9 may not work on Mac OS X.

Begin With CarbonAccessors.o

`CarbonAccessors.o` is a static library that may help ease your transition to Carbon by allowing you to begin using certain Carbon features while continuing to link against `InterfaceLib` and other non-Carbon libraries.

Because many toolbox data structures are opaque in Carbon, one of the first steps you should take in porting your application is to begin using the new accessor functions. It's easier to do this if you can continue compiling as a classic `InterfaceLib`-based application, because you can keep your application running and qualify your changes incrementally. `CarbonAccessors.o` facilitates this by providing implementations of the accessor functions for opaque toolbox data structures. For a list of the functions in `CarbonAccessors.o`, see Table A-3 (page 91).

We recommend that as the first step in the porting process, you add `CarbonAccessors.o` to your link, and then begin modifying your source code to use Carbon accessor functions, one file at a time. You can do this by setting the following conditional macro at the top of each source file you plan to convert:

```
#define ACCESSOR_CALLS_ARE_FUNCTIONS 1
```

This conditional makes the prototypes for the accessor functions available to that source file.

When you have converted all of your source files to use accessor functions, you can add the following conditional macro to your build options to ensure that you are no longer directly accessing any opaque toolbox data structures:

```
#define OPAQUE_TOOLBOX_STRUCTS 1
```

At this point you have an application that uses the Carbon accessor functions but does not link against the Carbon libraries. You can continue to run and test your application on any Mac OS release, because it does not require the `CarbonLib` extension at runtime.

The next step in the conversion process is to allow only Carbon-compatible APIs in your code by adding the following conditional macro to your build options:

```
#define TARGET_API_MAC_CARBON 1
```

You can now begin modifying your code so that it no longer calls functions that are obsolete in Carbon. At this point you must stop linking against `InterfaceLib` (and `CarbonAccessors.o`) and begin linking against `CarbonLibStub` (that is, the `CarbonLib` shared library).

Note: You can also use `CarbonAccessors.o` to maintain some backwards compatibility with non-Carbon systems. For example, if you don't require functions that are available only in CarbonLib, by linking against the `CarbonAccessors.o` static library you can build an application from a Carbon-compliant code base that runs on non-Carbon systems.

Use Casting Functions to Convert DialogPtrs and WindowPtrs

You cannot directly cast values of type `DialogPtr` or `WindowPtr` to a `GrafPtr`, but instead you must use the new functions described in "Casting Functions" (page 82). Direct casting will not affect compilation, but it will cause crashes on Mac OS X.

Replace Macro Calls to the Mixed Mode Manager With UPP Accessor Functions

Carbon introduces significant changes to the Mixed Mode Manager. Static routine descriptors are not supported, and you must use the system-supplied functions for creating, invoking, and disposing of universal procedure pointers. For example, Carbon provides the following functions to replace the macros previously used to create, invoke, and dispose of universal procedure pointers:

```
ControlActionUPP NewControlActionUPP (ControlActionProcPtr userRoutine);
void InvokeControlActionUPP (ControlRef theControl,
                             ControlPartCode partCode
                             ControlActionUPP userUPP);
void DisposeControlActionUPP (ControlActionUPP userUPP);
```

Similar functions are provided for all supported UPPs. Note that Carbon does not support the generic functions `NewRoutineDescriptor`, `DisposeRoutineDescriptor`, and `CallUniversalProc`.

On Mac OS 8 and 9, the UPP creation functions allocate routine descriptors in memory just as you would expect. On Mac OS X, the implementation of UPPs depends on various factors, including the object file format you choose. Universal procedure pointers will allocate memory if your application is compiled as a CFM binary, but are likely to return a simple procedure pointer if your application is compiled as a Mach-O binary.

On Mac OS X, UPPs are opaque types that may or may not require memory allocation, depending on the particular function and the runtime they are created in. By using the system-supplied UPP functions, your application will operate correctly in either environment. You must dispose of your UPPs using the system-supplied functions to ensure that any allocated memory is released. See "Consider Mach-O Executables" (page 35) for more information about the differences between these formats.

Important: If you are using the Thread Manager, be aware that functions that did not require UPPs for designating callbacks (such as `SetThreadScheduler` and `SetThreadSwitcher`) now require them in Carbon. See the Thread Manager documentation or the header file `Threads.h` for a list of these functions and for information on the required UPP creation and disposal functions.

Your own plug-ins must be compiled as PowerPC code, so there is no need to create universal procedure pointers for them. Use normal procedure pointers instead.

Move Custom Definition Procedures Out of Resources

The resource-based format for custom definition functions (such as WDEFs, CDEFs, and so on) is defined to start with 68K instructions. Because Mac OS X does not support 68K code, you must move your custom definition functions out of resources and compile them directly in your application.

To access your custom code, you can do either of the following:

- Use new Carbon functions (`CreateCustomXXXX`) to create your objects. For example, to create a custom window, pass a universal procedure pointer (UPP) to your window definition function into the `CreateCustomWindow` function:

  ```
  OSStatus CreateCustomWindow (
      const WindowDefSpec *def,
      WindowClass windowClass,
  ```

```
        WindowAttributes attributes,
        const Rect *contentBounds,
        WindowRef *outWindow
    );
```

You can specify your function pointer in the `WindowDefSpec` structure by taking the following steps:

```
WindowDefSpec defSpec;
defSpec.defType = kWindowDefProcPtr;
defSpec.u.defProc = NewWindowDefUPP( MyWindowDefProc );
```

- Map the old `procID` s to pointers to your custom code using the `RegisterXXXDefinition` functions. For example, if you still want to use `NewCWindow` to create your window, you should call `RegisterWindowDefinition`, passing it the resource ID referenced by your `procID`s and a UPP to your window definition function:

  ```
  OSStatus RegisterWindowDefinition (
      SInt16 inResID,
      const WindowDefSpec *inDefSpec
  );
  ```

  When `NewCWindow` receives a `procID` that isn't one of the standard system `procID`s, it will look in the mapping table to find the function that's registered for the resource ID embedded in the `procID`.

  For more details about changes to custom definition procedures, see "Custom Definition Procedures" (page 81).

Remove Direct Access to Low-Memory Globals

Low-memory globals are system and application global data located below the system heap in the Mac OS 8 and 9 runtime environment. They typically fall between the hexadecimal addresses $100 and $2800. Carbon applications can continue to use many of the existing low-memory globals, although in some cases the scope and impact of the global has changed. But in all cases, Carbon applications must use the supplied accessor routines to examine or change global variables. Attempting to access them directly with an absolute address will crash your application when running on Mac OS X.

The complete list of low-memory globals supported in Carbon is not yet finalized, but your transition to Carbon will be easier if you follow these guidelines:

- Use high-level calls instead of low-memory accessors whenever possible. For example, use `GetGlobalMouse` instead of `LMGetMouseLocation`.
- If a high-level call is not available, use an accessor function.
- Rely on global data only from Mac OS managers supported in Carbon. For example, because the driver-related calls in the Device Manager are not supported in Carbon, low-memory accessors like `LMGetUTableBase` are not likely to be available. Similarly, direct access to hardware is not supported in Carbon, so calls like `LMGetVIA` will no longer be useful.

Table 2-1 lists some frequently used low-memory accessors that are unsupported in Carbon. Refer to the Carbon Specification for the most recent information.

Table 1-1 Summary of Carbon low memory accessor support

Accessor	Replacement
`LMGet/SetAuxCtlHead`	not supported
`LMGet/SetAuxWinHead`	not supported
`LMGet/SetCurActivate`	not supported
`LMGet/SetCurDeactive`	not supported
`LMGet/SetDABeeper`	not supported
`LMGet/SetDAStrings`	`GetParamText, ParamText`
`LMGet/SetDeskPort`	not supported
`LMGet/SetDlgFont`	not supported
`LMGet/SetGhostWindow`	not supported
`LMGetGrayRgn`	`GetGrayRgn`
`LMGetMBarHeight`	`GetMBarHeight`
`LMSetMBarHeight`	not supported
`LMGet/SetMBarHook`	not supported
`LMGet/SetMenuHook`	not supported
`LMGetMouseLocation`	`GetGlobalMouse`
`LMSetMouseLocation`	not supported
`LMGet/SetPaintWhite`	not supported
`LMGetWindowList`	`GetWindowList`
`LMSetWindowList`	not supported
`LMGet/SetWMgrPort`	not supported

Use DebuggingCarbonLib

The debugging version of `CarbonLib` on Mac OS 8 and 9 checks for the validity of ports and windows, so using it is a good way to quickly identify potential problem areas. However, you should be aware that it runs considerably slower than the standard version of the library.

Modify or Conditionalize Your Headers

If you plan to build your application on Mac OS X using Project Builder, be aware that the standard flat Mac OS 8 and 9 headers (such as `Dialogs.h` and `MacWindows.h`) do not correspond directly with Mac OS X frameworks. To address this issue, you can do either of the following:

- Add `-I /Developer/Headers/FlatCarbon` to the cc compiler command line when building your application. The files in the FlatCarbon folder act as a compatibility layer, mapping the standard flat header includes to the proper frameworks.
- Replace your flat headers with the single include statement `#include <Carbon/Carbon.h>`. This statement lets you access the Carbon framework directly. You should choose this method if you plan to build exclusively on Mac OS X, as it will improve compile times.

If you choose not to include the path to FlatCarbon at build time, you can also conditionalize your code to use the proper headers :

```
#if <Some flag for building on X is set>
        #include <Carbon/Carbon.h>
#else
        <The usual Mac OS 8 and 9 includes>
#endif
```

Note: `Carbon.h` is treated as a flat Mac OS 8 and 9 header, so the suggested workarounds will still apply.

Update Modified or Obsolete Functions

From the list given to you by Carbon Dater, you should replace all functions listed as “out” or “modified” with their suggested replacements. Depending on the function it may have been easier to remove them earlier in the process (such as removing A5 functions when purging all 68K-related code).

Adopt Required Carbon Technologies

Carbon requires you to replace some older system services with newer ones as follows:

- Navigation Services replaces the Standard File Package. For documentation, see the web site:

  http://developer.apple.com/documentation/Carbon/Reference/Navigation_Services_Ref/
- The Carbon Printing Manager replaces the Classic Printing Manager. For documentation, see the web site:

  http://developer.apple.com/documentation/carbon/Reference/CarbonPrintingManager_Ref/

Add a 'plst' 0 Resource

On Mac OS X, Carbon applications that do not contain a `'plst' 0` resource will open in the Classic compatibility environment and will not gain all the advantages of Mac OS X. To ensure that Mac OS X properly recognizes your application, your application must include a resource of type `'plst'` with ID 0 or a plist file in a bundle. You can also store additional information about your application in a plist resource or file. See "Consider Using Bundles" (page 36) for more information about plists and bundles.

Note: The `'plst' 0` resource supersedes the older `'carb' 0` resource. While you can continue to use the `'carb' 0` resource, the `'plst' 0` resource provides the exact same functionality while also allowing you to store additional information useful to the Mac OS X Finder.

Even if you include a `'plst' 0` resource, you can still launch the application in the Classic environment:

- If you include an empty `'plst' 0` resource, a "user choice" checkbox appears in the Finder's Get Info window, allowing the user to choose whether to launch the application in Mac OS X or the Classic compatibility environment. The default is Mac OS X. (This checkbox feature does not appear if you include only a `'carb' 0` resource).
- By specifying options in the `'plst' 0` resource, you can force the application to launch into either Mac OS X or the Classic environment.

If your application does not contain a resource fork, it launches in Mac OS X by default.

See *Inside Mac OS X: System Overview* for more information about specifying Launch Services keys in your plist file or resource.

Conditionalize Quit Menu Items

Carbon applications running on Mac OS X automatically adopt the Aqua interface. Because Aqua provides a Quit menu item under the Application menu, your application does not need to add one to the File menu. As long as your application supports the Quit Apple event, it will quit normally. However, because the Mac OS 8 and 9 user interfaces still require a Quit menu item, you must conditionalize your code to add one in the File menu when running under Mac OS 8 or 9. The easiest way to identify the user interface is to check the `gestaltMenuMgrAquaLayoutBit` bit of the `gestaltMenuMgrAttr` gestalt selector. If the bit is set, the application is using the Aqua interface, and you should not add a Quit item to the File menu.

For example, you could use code such as the following to conditionalize your menus:

```
Gestalt( gestaltMenuMgrAttr, &result);
if (result & gestaltMenuMgrAquaLayoutMask)
    menuBar = GetNewMBar(rSysXMenuBar);
else
    menuBar = GetNewMBar(rMenuBar);
```

This method uses two different `'MBAR'` resources, each with a different `'MENU'` resource for the File menu.

If you must enable and disable the Quit menu item programmatically, you can use the new functions `DisableMenuCommand` and `EnableMenuCommand` to do so. Pass `NULL` for the menu reference and `'quit'` for the command ID.

Additional Porting Issues

In addition to the steps described in the "Essential Steps for Porting Your Application" (page 20), you should be aware of these other issues that can affect the porting process.

Determine the Appropriate CarbonLib Version

Just like system software, `CarbonLib` also exists in various versions, each of which contains different levels of functionality. Because some calls to `CarbonLib` merely call through to the underlying system software, the functions available can depend on the system software version.

CarbonLib version	Reflects Universal Interfaces version	Compatible back to	Notes
1.0	3.3.1	Mac OS 8.1	*Shipped with Mac OS 9. Do not develop with this version.*
1.0.4	3.3.1	Mac OS 8.1	*Includes the following*:
			All Carbon APIs available with Mac OS 8.1
			Toolbox accessor functions
			Control, Window, and Menu properties
			Appearance Manager 1.1
			Navigation Services
			Core Foundation
			Carbon Printing Manager
1.2	3.4	Mac OS 8.6	*Adds the following:*
			DataBrowser
			Carbon Event Manager
			XML
			URL Access Manager
			Apple Type Services for Unicode Imaging (ATSUI)
			Interface Builder Services
			Font Sync

CarbonLib version	Reflects Universal Interfaces version	Compatible back to	Notes
			Apple Help Viewer
			Font Management
		Mac OS 9	*Adds the following:*
			Keychain Manager

Draw Only Within Your Own Windows

Because Mac OS X is a truly preemptive system, any number of applications may be drawing into their windows at the same time. Carbon applications, therefore, cannot draw outside their own windows. In the past you could call the `GetWMgrPort` function and use that port to draw anywhere on the screen. This port does not exist in Mac OS X, so you will need to use alternate methods to implement window dragging and resizing. For more detailed information about handling windows in Carbon, see "Window Manager Issues" (page 30).

Do Not Patch Traps

Carbon applications should not patch traps because there is no trap table in Mac OS X. The Patch Manager is unsupported, and functions like `GetTrapAddress` and `SetTrapAddress` are not available in Carbon. You can, of course, conditionalize your code and continue to patch traps when running under Mac OS 9, but your programs will be much easier to maintain if you avoid patching entirely.

Don't Pass Pointers Across Processes

In Mac OS X, every process has its own address space, so attempting to pass a pointer to another process is meaningless at best and may cause your application to misbehave. Threads or tasks created by an application (for example, Multiprocessing Services tasks or Thread Manager threads) occupy the application's address space, so you can pass pointers between them.

Do Not Write to Your Application's Resource Fork

While writing to your application's resource fork is acceptable (if not encouraged) in Mac OS 8 and 9, you should not do so in Mac OS X, as there are many common instances that will cause such write attempts to fail. Some examples:

- when file-system permissions don't allow itwhen the application resides on a network serverwhen the application resides on read-only media

If you have application-specific data that you need to save, you should store them in a preferences file.

Ideally you should remove resource forks from your application altogether and place your resources in the data fork (see "Move Resources to Data Fork–Based Files" (page 36). Note, however, that such resources are also read-only.

Note that you can still write to other resource forks (such as in document files).

Check Your OpenGL Code

If you use OpenGL in your application, you should continue to link to the OpenGLLibrary, OpenGLMemory, and OpenGLUtility stubs as you would for non-Carbon applications. On Mac OS X these functions will link with the OpenGL framework.

Note that if you are building a Mach-O–based Carbon application that uses the OpenGL header `aglMacro.h`, you must make the following call before creating any OpenGL contexts:

```
aglConfigure(AGL_TARGET_OS_MAC_OSX,GL_TRUE);
```

Do not make this call from CFM-based Carbon applications.

See "Consider Mach-O Executables" (page 35) for more information about the Mach-O format.

Examine Your Plug-ins

Carbon applications can load non-Carbon plug-ins. You must make sure, however, that your plug-ins do not link to `InterfaceLib`. On Mac OS 8 and 9 this will not cause a problem, but it can cause a crash on Mac OS X (because `InterfaceLib` is unavailable).

You can use the MPW tool DumpPEF with the `-loader i library` option to find unintentional links to non-Carbon libraries.

Linking to Non-Carbon-Compliant Code

In some cases, your CFM application may need to call code that is not Carbon-compliant to maintain cross-platform compatibility between Mac OS 8 and 9 and Mac OS X. For example, say your application makes calls to the Device Manager. The Device Manager is not part of Carbon as it cannot run on Mac OS X. However, its replacement, I/O Kit, is a Mac OS X technology that cannot run on Mac OS 8 and 9. The only way to maintain your application's functionality is to fork your code and make calls to either the Device Manager or I/O Kit, depending on the platform.

Forking your code in this manner brings up some build issues. For example, if you had set preprocessor directives to build with Carbon, the Universal Interfaces will conditionalize out any non-Carbon functions, and attempting to call non-Carbon functions will generate a compiler error indicating missing prototypes.

The easiest way to work around this problem is to compile your noncompliant code separately, using non-Carbon headers. You can package your non-Carbon code as a shared library, which you can then call from your application.

The safest method for calling non-Carbon functions in shared libraries is to prepare the fragment and locate the symbols manually. That is, call `GetSharedLibrary` to prepare the library and use `FindSymbol` to get the symbol address. You can then call the function through the returned pointer. This method gives you maximum flexibility in handling missing symbols or libraries. See the sample code included with the Mac OS X Developer Tools CD for examples.

Window Manager Issues

This section addresses common issues encountered when porting code that draws or otherwise manipulates windows.

Handling Buffered Windows

In Mac OS X, all windows are buffered. A window's contents are written first to a buffer and then the Window Manager periodically refreshes the screen with the contents of the buffers. As you don't automatically have control over when a window's contents are written to the screen, you may need to make some minor changes to your windowing code to account for buffering.

If you are writing periodically to the screen in a loop that doesn't call `WaitNextEvent`, you must call `QDFlushPortBuffer` to flush your drawing to the screen. Otherwise, you are only updating the contents of the buffer.

If you draw directly into a window's pixel map, QuickDraw cannot tell which parts of the pixel map are dirty. To work around this, you must do one of the following:

- Call `QDFlushPortBuffer` explicitly, passing a nonempty region parameter describing the modified area.
- Call, `QDGetDirtyRegion` to get the port's dirty region, add in the area you modified by calling `UnionRgn`, and then set the updated dirty region by calling `QDSetDirtyRegion`. The Window Manager will the update the region during the next call to `WaitNextEvent`.

If you draw directly into the pixel map of your windows without using QuickDraw, you'll need to wrap those blits with two new calls that signal the Window Manager not to update the window until your drawing operation completes. Here are the basic steps:

1. Use the `GetWindowPort` function to get the window's port.
2. Use the `LockPortBits` function to lock the port's pixel map. Note that this function is different from `LockPixels`; they are not interchangeable.
3. Use the `GetPortPixMap` function to get a handle to the port's pixel map. The `baseAddr` field of the `PixMap` structure contains the base address of the actual port bits in memory.

   Important: The port address is valid only after you've locked the port using the `LockPortBits` function, and is invalid after you call the `UnlockPortBits` function.

4. Perform your drawing operation as quickly as possible. Because the `LockPortBits` function blocks all other updates to the port, it's important that your drawing code be small and fast to avoid impacting system performance.

5. Call the `UnlockPortBits` function to release the port. The `PixMapHandle` is automatically disposed when you call this function. Do not attempt to reuse the handle.

Note that the `UnlockPortBits` function does not initiate a window update, it merely allows any pending or future updates to occur. An update is initiated either by the `BeginUpdate/EndUpdate` routines or when the `QDFlushPortBuffer` function is called.

Bypassing the Window Manager Port

Prior to Carbon and Mac OS X, any Mac application could access the Window Manager port, which included all available screens. Using that port, an application could write directly to the screen on top of all windows. Developers used this capability to implement a number of features, such as custom window grow outlines and custom window dragging.

Because recent releases of Mac OS 8 and 9 offer improved Window Manager functionality, as well as robust drag-and-drop support through the Drag Manager, many applications no longer need to use the Window Manager port. This is a good thing, because in Mac OS X, there is no Window Manager port, and Carbon provides no access to the Window Manager port for applications running in Mac OS 8 and 9.

The Carbon Window Manager does supply alternate mechanisms to implement features that may have relied on use of the Window Manager port. To learn more about when to use these mechanisms, see "Window Dragging and Resizing Q&A" (page 31).

If your application is drawing in the Window Manager port and you don't see an alternate mechanism described, you should consider whether you can achieve the same results by modifying your user interface. If that's not appropriate, send an email to `carbon@apple.com` explaining what you need and some APIs may be added to support additional features.

Window Dragging and Resizing Q&A

This section answers some frequently asked questions about dragging and resizing windows in Carbon and Mac OS X. For related information, see "Bypassing the Window Manager Port" (page 31).

- Q. What is the standard window dragging feedback supplied by `DragWindow`?

  A. In Mac OS X, if you call `DragWindow` for a buffered window, the Carbon Window Manager provides live dragging—that is, the contents of the window remain visible as a user moves the window around the screen.

  For a Carbon application running in Mac OS 8 or 9, `DragWindow` supplies the traditional outline feedback.

- Q. Can I still use `DragGrayRegion`?

  A. Although `DragGrayRegion` is fully supported in Carbon, it only applies to the current port. If you're currently using `DragGrayRegion` with the Window Manager port, you should instead use one of the other mechanisms described here, such as calling `DragWindow` or using Carbon event handlers.

- Q. How do I implement custom window dragging—for example, to modify the position and shape of a tool palette as the user moves it to dock with another palette?

  A. You can implement features of this type using a Carbon event handler that tracks move events. When a user starts to drag a window, your handler receives a move window event (`kEventWindowOriginChange`). If you so request, your event handler can also receive periodic move window events as the user continues to drag the window. When the user completes the move, your handler receives a window moved event that includes the final position of the window. Your handler should get the `kEventParamCurrentBounds` parameter from the event, modify the `Rect` structure as needed, update the parameter, and then return `noErr`. In the example of docking a palette to another palette, you can either make changes to the palettes during the move as the current position warrants, or you can modify them after the move is complete.

Keep in mind that using a move event handler that receives and processes events during the move may have an impact on performance.

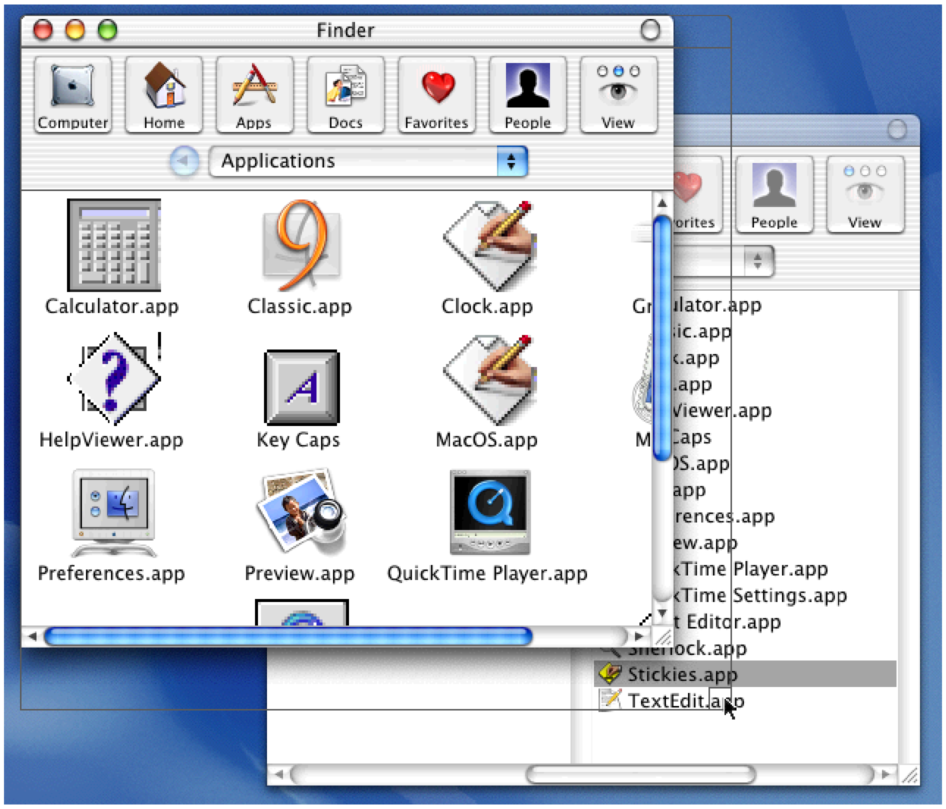
The Carbon Window Manager may also support custom dragging as part of an API to be added later. However, in Mac OS X this approach would only provide outline feedback for the drag, rather than live feedback.

- Q. What is the standard window resizing feedback supplied by `GrowWindow`?

  A. If you do not supply a resize event handler (described in another question), `GrowWindow` provides the traditional outline feedback.

  Figure 1-1 (page 32) shows the traditional outline feedback for resizing a window.

Figure 1-1 Outline feedback as a user resizes a window



- Q. How do I take advantage of live resizing in Mac OS X?

  A. If you want live resizing in Mac OS X—that is, the contents of a window remain visible and are adjusted and redrawn as needed as a user resizes the window—you must set the `kWindowLiveResizeAttribute` attribute on the window (either at creation time using `CreateNewWindow` or `CreateCustomWindow`, or by called `ChangeWindowAttributes` on an existing window) and then provide a resize event handler. The Carbon Window Manager sends an event (`kEventWindowBoundsChanged`) to your handler that indicates when it should adjust its scrollbars, redraw its content, and so on, as the user resizes the window.

Carbon applications running in Mac OS 8 and 9 will only get outline resizing.

- Q. How do I implement custom window resize feedback—for example, to make the window snap to a grid as a user resizes the window?

  A. You can implement custom resizing using the same Carbon event handler you use to support live resizing. When a user starts to resize a window, your handler receives a resize window event (`kEventWindowBoundsChanged`). Your handler also receives periodic events as the user continues the resize. When the user completes the resize, your handler receives a window resized event that includes the final size. You can modify the `kEventParamCurrentBounds` parameter to constrain resizing to the desired grid as the user resizes, or do so after the resize is complete.

  If you are already using a custom window definition (WDEF) and you do not need live resizing, the easiest way to provide custom resize feedback is to support the new WDEF message `kWindowMsgGrowImageRegion`. Your WDEF receives this message periodically as the user moves the mouse during a resize operation. You can use this message to override the region that gets displayed during resize. To get these messages, your WDEF must report the `kWindowSupportsSetGrowImageRegion` feature bit.

- Q. Do I need to make any other changes to my existing WDEF?

  A. In most cases, you should not have to change your custom window definition. Prior to Carbon and Mac OS X, custom window definitions expected to draw directly in the global port. Now the Carbon Window Manager automatically sets up an appropriate port for drawing. When your window definition gets a draw message, it can go ahead and draw—but it shouldn't assume it's drawing in a global port, because it isn't.

- Q. I use the Window Manager port to implement custom dragging with translucent drag images. How do I keep my translucent drag images without the Window Manager port?

  A. The Drag Manager has supported translucent dragging since version 1.3 and System 7.5.3. This feature is fully supported in Carbon, so you don't need to write any custom code.

- Q. How can I capture a region of the current global screen?

  A. There is currently no way to do this in Carbon, although we are considering providing an interface that will allow you to grab an arbitrary screen region.

  You should not rely on calling `CreateNewPort` and determining the location of the screen bits from the new port. This behavior is no longer supported and code that relies on it is likely to break in future versions of Mac OS X.

- Q. How can I write a screen saver or other application that needs to take over the whole screen?

  A. Use the QuickTime functions `BeginFullScreen` and `EndFullScreen`. For more information, see the QuickTime documentation at

  http://developer.apple.com/documentation/QuickTime/index.html

- Q. I don't want to modify my user interface and I don't see anything described here that will help me do what I want to do.

  A. Tell us what you need and why, so that we can help provide a solution.

Optimizing Your Code for Carbon

This section describes steps and issues you should consider for your application to take best advantage of the Mac OS X environment.

Manage Memory Efficiently

Memory management doesn't change much for Carbon applications running on Mac OS 9. You'll need all the code you use today to handle heap fragmentation, low memory situations, and stack depth.

However, there are some techniques you can adopt now that will help your application perform well when running on Mac OS X, which uses an entirely different heap structure and allocation behavior. The most significant change is in determining the amounts of free memory and stack space available. For example, you should avoid preallocating memory, as doing so will not make best use of the allocators available in Mac OS X. Similarly, using suballocators (allocating a block of memory and then allocating from within the block) is not suggested.

The functions `FreeMem`, `PurgeMem`, `MaxMem`, and `StackSpace` are all included in Carbon. You should, however, think about how and why you are using them. You'll probably want to consider additional code to better tune your performance.

The `FreeMem`, `PurgeMem`, and `MaxMem` functions behave as expected when your Carbon application is running on Mac OS 9, but they're almost meaningless when it's running on Mac OS X, where the system provides essentially unlimited virtual memory. Although you can still use these calls to ensure that your memory allocations won't fail, you shouldn't use them to allocate all available memory. Allocating too much virtual memory will cause excessive page faults and reduce system performance. Instead, determine how much memory you really need for your data, and allocate that amount.

Before Carbon, you would use the `StackSpace` function to determine how much space was left before the stack collided with the heap. This routine could not be called at interrupt time, but was useful for preventing heap corruption in code using recursion or deep call chains. But because a Carbon application may have different stack sizes under Mac OS 9 and Mac OS X, the `StackSpace` function is no longer very useful. You shouldn't rely on it for your logic to terminate a recursive function. It might still be useful as a safety check to prevent heap corruption, but for terminating runaway recursion, you should consider passing a counter or the address of a stack local variable instead of calling `StackSpace`.

The Carbon API does not include any subzone creation or manipulation routines. If you use subzones today to track system or plug-in memory allocations, you must use a different mechanism. For plug-ins, you might switch to using your own allocator routines. To prevent memory leaks, make sure all your allocations are matched with the appropriate dispose calls.

The Carbon API also removes the definition of zone headers. You can no longer modify the variables in a zone header to change the behavior of routines like `MoreMasters`. Simply call `MoreMasters` multiple times instead, which will allocate 128 master pointers each time. (You can also use the new Carbon call `MoreMasterPointers`, which allows you to specify the number of master pointers to allocate in one relocatable block.)

Avoid Polling and Busy Waiting

Polling for events or using a timer loop is allowable (but not recommended) on Mac OS 9, but it can cause severe performance problems on Mac OS X. In the Mac OS X multitasking environment, the OS gives time to all active processes. A process that is busy waiting for an event is considered active, even though it is not actually doing anything. Such waiting reduces the performance of other active processes. As an extreme example, multiple instances of a shared library, all polling for an event, can easily bog down the system. Instead of polling, your code should implement some sort of notification mechanism (such as an event queue or semaphore).

Note that triggering actions on null events (to blink the cursor, for example) does not work on Mac OS X, as the system will notify your application only when real events occur. To work around this issue you should use Carbon Event Manager timers.

Use "Lazy" Initialization for Shared Libraries

To allow Mac OS X to manage memory efficiently, you should not prepare shared libraries at application launch time, but rather only when you need them. Also, try to avoid using initialization functions if possible. See *Mac OS Runtime Architectures* for more information about initialization functions.

Adopt HFS Plus APIs

HFS Plus, the Mac OS Extended File Format, is the default file system for Mac OS X, so you should consider using HFS Plus APIs if you need to programmatically access files on hard drives. Some of the advantages of HFS Plus are as follows:

- support for long Unicode filenames (255 characters)
- support for files larger than 2 GB
- support for extended file attributes

See the File Manager documentation at

http://developer.apple.com/documentation/carbon/Reference/File_Manager

for more information about HFS Plus.

Consider Mach-O Executables

You can build Carbon applications in two object file formats: PEF, which uses the Code Fragment Manager introduced with PowerPC Macintosh computers, and Mach-O, which is the preferred format for Mac OS X. Depending on your needs, you may want to consider creating Mach-O-based Carbon applications. There are advantages and disadvantages.

Advantages:

- Applications get access to all native Mac OS X APIs such as Quartz and POSIX. CFM-based Carbon applications can access only Carbon APIs.
- Symbolic debugging is easier on Mac OS X (using GDB).
- You can take full advantage of the Interface Builder and Project Builder development tools on Mac OS X.

Disadvantages:

- Applications cannot run on Mac OS 8 and 9.
- Mach-O doesn't support the existing CFM plug-in architecture.

- Programmatic manipulation of the Code Fragment Manager (for example, calling `GetSharedLibrary`) may not work as expected.

You can also package CFM-based code and Mach-O–based executables together in bundles (as described in "Consider Using Bundles" (page 36)). Bundling creates file packages analogous to PowerPC/68K fat applications built during the transition to PowerPC. Such CFM/Mach-O packages will execute the CFM version of the application on Mac OS 8 and 9, and the Mach-O version on Mac OS X. See *Inside Mac OS X: System Overview* for more information about the Mach-O format.

Eventually, as customer focus shifts to Mac OS X, you should concentrate on building Mach-O binaries.

Move Resources to Data Fork–Based Files

While Mac OS X can handle application resources stored in the resource fork of an executable file, in general you should begin storing these resources in the data fork. The major reason for doing this is to maximize compatibilty when moving files between different file systems. Many computing environments and file copying tools recognize only single-fork files; copying uncompressed files usually results in the loss of any information stored in the resource fork.

The only exceptions at this time are the `'cfrg' 0` and `'plst' 0` (or `'carb' 0`) resources, which must remain in the resource fork for CFM-based applications so the Mac OS X Finder can launch them properly.

In general you should use the Core Foundation CFBundle APIs to package and access resources in the data fork. While you can simply move your existing resource files to the data fork, a better solution is to save each resource as an individual data fork–based file. Doing so makes it much easier to access (and perhaps modify) any individual resource.

See "Consider Using Bundles" (page 36) and *Inside Mac OS X: System Overview* for more information about bundling resources.

Consider Using Bundles

A bundle is a Mac OS X concept that lets you store all the software resources and executable files that an application requires in one package. Essentially a directory or folder hierarchy, a bundle could contain any of the following:

- images, sounds, or other files used by the application
- localized character strings
- multiple executable versions of an application.

A bundle can contain multiple sets of resources, grouped by language, locale, and platform. By combining all these resources and executables in one package, you can create one version of your application that is localized for multiple languages and can run on multiple platforms.

On Mac OS X, a bundle hierarchy normally appears as a single file, unless the bundle bit is unset, in which case it appears as a folder hierarchy.

On Mac OS 8 and 9, a bundle appears as a folder hierarchy, because the system software does not have knowledge of bundles. For this reason you should generally place an alias to your application prominently in the top level folder where the user can find it.

For maximum compatibility, you should use the Core Foundation CFBundle APIs to access bundled resources and executable files. See the Core Foundation Bundle Services documentation at

http://developer.apple.com/documentation/MacOSX/Conceptual/BPBundles/index.html

Note that if you do not wish to adopt bundles at this time, you can include some of the information stored in a bundle's `Info.plist` file in a resource of type `'plst'` with ID 0. Doing so allows you to specify attributes that provide information to the Finder (for example, what icons to use, what document types the application recognizes). You can also access data in the `'plst'` resource using Resource Manager or CFBundle APIs.

See *Inside Mac OS X: System Overview* for more specific information about packaging files in bundles.

Begin Transitioning to the Aqua Interface

By linking with CarbonLib, your Carbon application will automatically register itself with the Appearance Manager and adopt the basic Aqua look and feel. You should make sure that your interface elements are Appearance Manager–compliant; generally this means using system-defined controls, menus, and windows as much as possible.

To provide the best user experience, however, you should take the additional time to modify dialog boxes, windows, icons, controls, and so on, to conform with the Aqua specification. Doing so ensures that your application will look its best on Mac OS X. For details, see the document *Aqua Human Interface Guidelines* available at

http://developer.apple.com/documentation/macosx/

For additional information on icons, see "Provide Thumbnail Icons for Your Application" (page 37).

You should also consider adding some new interface elements introduced with Aqua, such as the following:

- Sheets: Sheets are the new window-centric modal dialogs that slide down from the title bar. Sheet functions appear in `MacWindows.h`.
- Help Tags: A help tag is a little yellow text field that appears over a control when you roll the cursor on top of it. The tag typically describes or clarifies the control's purpose. Help tags replace the Help balloons available on older Mac OS systems. Help tag functions appear in `MacHelp.h`.

Adopt a Terse Name for the Application Menu

The leftmost pull-down menu in Mac OS X is the application menu. To maximize space for other menus, you should adopt a short version of your application name (16 characters or less) for this menu. You should add this information in your application's `InfoPlist.strings` file.

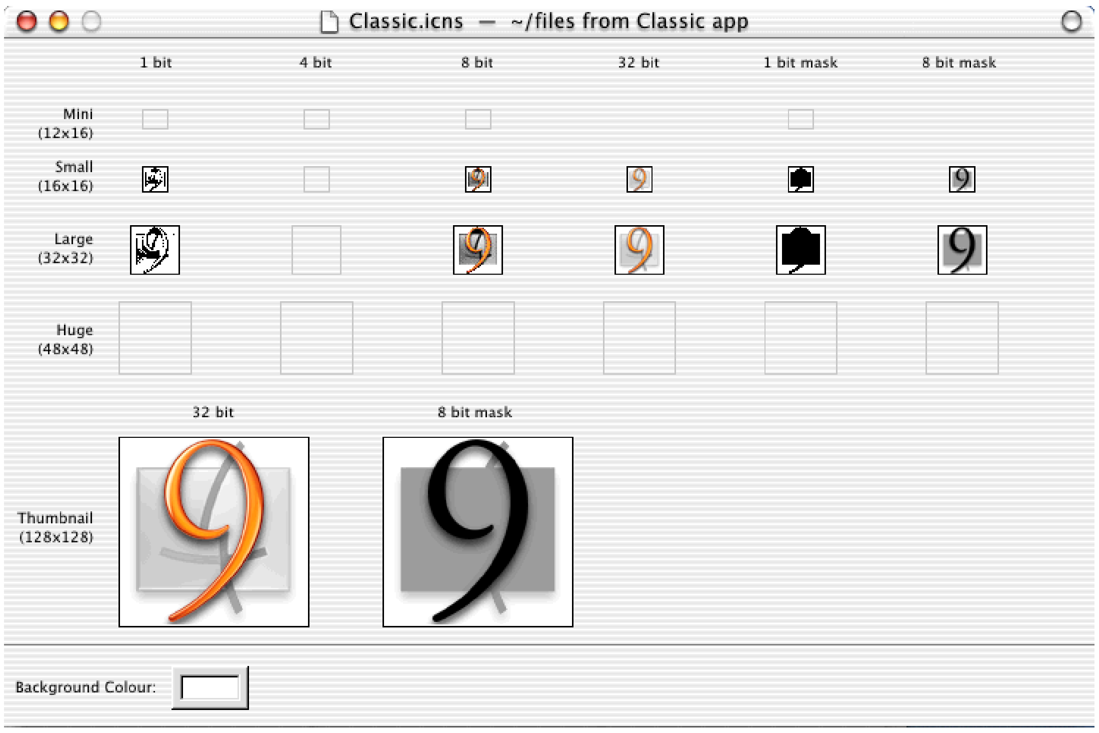
Provide Thumbnail Icons for Your Application

The information in this section supplements the document "Obtaining and Using Icons With Icon Services," available at the Carbon documentation website at

http://developer.apple.com/documentation/carbon/

In Mac OS X, a user may choose to display very large icons for the desktop, the application Dock, and so on. The Finder uses a high-quality scaling algorithm, supplied by Icon Services, to generate the variable-sized icons it needs. To help ensure a pleasing result for your application, you should provide a thumbnail icon and a thumbnail mask as part of the `'icns'` resource for your icon family. Figure 1-2 (page 38) shows the icon family, including thumbnail icons, for `Classic.app` in Mac OS X.

Figure 1-2 Thumbnail icons in a `.icns` file, displayed in Icon Browser



A thumbnail icon is 128x128 pixels with 32-bit depth. A thumbnail mask is 128x128 pixels with 8-bit depth (there is no one-bit mask for a thumbnail). Within an icon family resource, you specify thumbnail elements with the following constants:

```
enum {
    kThumbnail32BitData = 'it32',
    kThumbnail8BitMask  = 't8mk'
};
```

You can use these icon types only for an icon element within an `'icns'` icon family, not for an individual icon or icon mask resource.

Your application can continue to provide small (16x16) and large (32x32) icons as a complement to its thumbnail icons, especially if you need to preserve certain fine details at smaller resolutions. Icon Services will pick the best available icon for a particular size, so providing additional icons gives it more flexibility and gives you more control.

As of this writing, some third-party resource editor applications support editing of thumbnail icons, so you can investigate to determine which one best meets your needs.

If you want to add a thumbnail icon or mask to an icon family yourself, you can do so with the Icon Services function `SetIconFamilyData`.

```
pascal OSErr SetIconFamilyData (
                        IconFamilyHandle iconFamily,
                        OSType iconType,
                        Handle h)
```

`iconFamily`
: A handle to an `iconFamily` data structure to be used as the target.

`iconType`
: A value of type `OSType` specifying the format of the icon data you provide. For a thumbnail icon, for example, you specify `kThumbnail32BitData` in this parameter. For a thumbnail mask, you specify `kThumbnail8BitMask`.

`h`
: A handle to the icon data you provide. For a thumbnail icon, the handle contains raw image data in the form of 128x128, four bytes per pixel, RGB data. For a thumbnail mask, the data is in the same format except that it is one byte per pixel.

When you are finished constructing the icon family, you can write it to a file with the `WriteIconFile` function. For more information on these functions, see the document "Obtaining and Using Icons With Icon Services."

Building Carbon Applications

This chapter describes how to use the tools and libraries provided with the Mac OS X Developer Tools CD to build Carbon applications for both Mac OS 9 and Mac OS X. You can also install the Carbon system extension, `CarbonLib`, to run Carbon applications on Mac OS versions 8.1 and later.

Native Mac OS 9 Versus Mac OS X's Classic Environment

If you plan to build, run, and debug Carbon applications for both Mac OS 9 and Mac OS X on a single system, the Mac OS X application `Classic.app` provides a convenient environment for running your development system. You can easily switch between the two environments, and launch applications in either.

If you prefer to develop on a native Mac OS 9 system (that is, on a computer running Mac OS 9 instead of Mac OS X), you'll need to reboot to run Mac OS X and test your Carbon application in that environment.

If you have two computers, you might want to run Mac OS 9 on one computer and Mac OS X on the other. To transfer files between them, you can use one of the following methods:

- Enable file sharing on one of the machines and copy the files directly.
- Copy the files using FTP.
- Activate the Metrowerks remote debugger and select "Debug". (Doing so transfers the file to Mac OS X and begins a debugging session. After transfer, you can quit the debugging session, leaving the file ready for launch, or perhaps GDB debugging.)

Development Scenarios

There are a number of tools and processes you can use to build and debug Carbon applications. This section describes three scenarios that Apple recommends, and the advantages of each.

Using CodeWarrior to Build a CFM Carbon Application

This is the most likely scenario if you're porting an existing Mac OS 9 application to Carbon, especially if you're already using CodeWarrior. You'll continue to use the Mac OS development tools and processes you're familiar with, and you'll create CFM applications that can run on both Mac OS 9 and Mac OS X. The only difference is that you'll include the `CarbonLib` stub library in your CodeWarrior project.

Using CodeWarrior to Build a Mach-O Carbon Application

Metrowerks CodeWarrior Pro version 8.0 and later has support to build Mach-O applications on Mac OS 9, as well as build and debug applications on Mac OS X. If you have a second computer, you may also want to investigate whether Metrowerks' two-machine debugger suits your needs, as it can debug CFM applications on both platforms. Contact Metrowerks for information about these products.

Using Project Builder to Build a Mach-O Carbon Application

Project Builder is Apple's integrated development environment (IDE) for Mac OS X. It offers a comprehensive feature set that includes source-level debugging. Project Builder is a good choice if your application will run only on Mac OS X, and you want to take advantage of features available only on that platform. However, you can't use Project Builder to build a CFM application, so if you want your program to run on both platforms you'll either need to use CodeWarrior or other tools to create a CFM version for Mac OS 9.

See the Project Builder online help documentation for more information about creating Mach-O Carbon applications.

Building a CFM Carbon Application With CodeWarrior

If you plan to use Metrowerks CodeWarrior, CodeWarrior Pro version 8.0 or later is recommended. You can run CodeWarrior natively on Mac OS 9 or Mac OS X.

Preparing Your Development Environment

Before you start Carbon development with CodeWarrior, you'll need to install the tools and libraries provided with the Mac OS X Developer Tools CD or the Carbon SDK.

1. Copy the Carbon Support folder to the Metrowerks CodeWarrior folder on your hard disk. The Carbon Support folder should reside in the same folder as the CodeWarrior IDE application.
2. Copy the appropriate Carbon system extension (`CarbonLib` or `DebuggingCarbonLib`) from the Carbon Support:CarbonLib folder to your Extensions folder. You should keep only one Carbon extension in your Extensions folder at any time.
   - `CarbonLib` is the standard implementation of Carbon for Mac OS 8.1 or later.
   - `DebuggingCarbonLib` is a debugging version of CarbonLib.
3. `CarbonLib` is included in all versions of Mac OS 9 as well as the Classic environment on Mac OS X. However, to make sure you are using the latest version, you should replace the default `CarbonLib` with the latest one available (in this case version 1.6).
4. To avoid the potential for data loss in the event that you need to reinstall Mac OS X, ensure that your CodeWarrior project files and source code reside on a separate hard disk.

Building Your Application

To build a Carbon version of your application, you'll need to make the following changes to your CodeWarrior project.

1. Add the following statement to one of your source files before including any of the Carbon headers:

   ```
   #define TARGET_API_MAC_CARBON 1
   ```

   This conditional specifies that the included header files should allow only Carbon-compatible APIs and data structures. You can include the conditional in a prefix file if you wish.

   Note: Moving a project from CodeWarrior Pro 8.0 to an earlier CodeWarrior version may result in the loss of prefix file information in the C/C++ Language Preferences panel. Many of the code samples on the Mac OS X Developer Tools CD make use of a prefix file (usually `CarbonPrefix.h`) to define `TARGET_API_MAC_CARBON`, so if you try to build a sample on an older CodeWarrior system, you may need to reinstate the prefix file information.

2. Add the `CarbonLibStub` stub file to your project.

3. Ensure that your project is not linking to any libraries that are not Carbon compatible. For example, the MPW ANSI C library is not Carbon compatible. Note that you should not directly link to `InterfaceLib` when you are linking with `CarbonLib`. On Classic Mac OS, `CarbonLib` will return an error to the Code Fragment Manager if your application attempts to link to both `CarbonLib` and `InterfaceLib`, causing the application launch to fail.

4. Ensure that your CodeWarrior access paths and other target settings are correctly specified. See the sample code included with the Carbon SDK for examples of how to do this.

Running Your Application on Mac OS 9

You can launch your application from the Finder on a Mac OS 9 system by double-clicking its icon. To run Carbon applications on Mac OS 8 (version 8.1 or later), you must install the `CarbonLib` or `DebugCarbonLib` extension in the Extensions folder.

Running Your Application on Mac OS X

As long as your application resides on an HFS Plus disk, you can launch it by double-clicking its icon.You cannot launch applications from a standard HFS format disk on Mac OS X.

You can also use the command-line tool LaunchCFMApp to launch CFM applications from a terminal window in Mac OS X. If the CFM application is in the current working directory, the command is:

```
/System/Library/Frameworks/Carbon.framework/Versions/A/Support/LaunchCFMAppfilename
```

If the application is in a different directory, you must specify the path.

Note that if your application does not contain a `'plst'` resource, a bundled `Info.plist` file, or a `'carb'` resource, Mac OS X opens the application in the Classic compatibility environment. To ensure that Mac OS X properly recognizes your application, it must include a resource of type `'plst'` with ID 0, a bundled `Info.plist` file, or a resource of type `'carb'` with ID 0.

Building a Mach-O Carbon Application With CodeWarrior

Before building a Mach-O version of your application with CodeWarrior, you should follow the instructions in the previous section for building a CFM Carbon application. After you've successfully built and tested a CFM version of your application on Mac OS 9, you can use CodeWarrior to build a Mach-O version for debugging on Mac OS X.

Preparing Your Development Environment

Metrowerks CodeWarrior Pro 8.0 and later has support for Mach-O applications. See the Metrowerks documentation for more information.

Building Your Application

Refer to your Metrowerks CodeWarrior documentation for instructions on building Mach-O versions of your application.

Running Your Application on Mac OS X

CodeWarrior creates an executable Mach-O binary that includes a resource fork. As long as this file resides on an HFS Plus disk, the resource fork remains intact and you can launch the application by double-clicking its icon.

Building a Mach-O Carbon Application With Project Builder

Project Builder is included on the Mac OS X Developer Tools CD. Instructions for building Mach-O Carbon applications are available in Project Builder's online help documentation.

Building Applications Using MPW

An alternative to using CodeWarrior or ProjectBuilder for CFM applications is to use Apple's MPW development environment, which is now available as a free download from the following website:

http://developer.apple.com/tools/mpw-tools/

You should place the `CarbonLibStub` stub file in the folder `Interfaces and Libraries:Libraries:SharedLibraries`, while the Carbon version of the Universal Headers should be placed in `Interfaces and Libraries:Interfaces:CIncludes`.

You compile your application and link against `CarbonLibStub` just as you would when using CodeWarrior. Note however that Carbon applications must contain a `'SIZE'` resource, and the resource must have the `acceptSuspendResumeEvents` flag set. While CodeWarrior adds a `'SIZE'` resource automatically, you must create your own when building with MPW.

In addition, you need to override your default entry point (by using the `-m` option in PPCLink) with one of the following:

- `main` if your application does not call `exit()` and does not expect any exit procedures to run.
- `__appstart` if you want the functionality previously supplied by `__start`. `__appstart` is a stripped-down version of `__start` that initializes the environment, but does not provide support for MPW tools. To use `__appstart`, you need to link with `StdCRuntime.o` and `StdCLib` in addition to CarbonLib.

The version of `StdCRuntime.o` containing `__appstart`, as well as a Carbon-friendly version of the CreateMake tool, will be available from the following web site:

http://developer.apple.com/tools/mpw-tools/updates.html

You should also examine the PackageTool sample application in the Sample Code folder of the CarbonLib SDK and the MPW release notes for more specifics about the build process.

Debugging Your Application

You can debug Carbon applications on Mac OS 9 or Mac OS X using the Metrowerks debugger. You can use also this debugger with two networked machines (for example, one running Mac OS 9 and the other running Mac OS X). Contact Metrowerks for more information.

You can also debug Carbon applications on Mac OS X using GDB, which you can run from a terminal window. Although GDB cannot directly debug a CFM application at this time, there is a workaround that lets you perform low-level debugging on a CFM application. You'll use GDB to debug LaunchCFMApp, a Mach-O program that launches CFM applications.

This workaround has the following features and limitations:

- You can set breakpoints at Mach-O functions. Since the Carbon library is Mach-O code, you can set breakpoints at Carbon functions. However, you cannot set breakpoints at CFM functions, including those in your application.
- You can examine the memory contents at any address with the `x` command. However, you cannot view variables or expressions, since GDB cannot use the symbol names in a CFM application.
- You cannot step through your application's code.

To debug your CFM application:

1. Launch the Terminal application: `/Applications/Utilities/Terminal.app`.

2. Enter `gdb /System/Library/Frameworks/Carbon.framework/Versions/A/Support/LaunchCFMApp`.

   GDB loads the LaunchCFMApp program.

3. If you want, set breakpoints at any Carbon function with the `br` command.

   For example, you may want to set a breakpoint at the `DebugStr` function, because `DebugStr` prints its argument without stopping the program's execution. Enter `br DebugStr` at the GDB prompt.

4. At the GDB prompt, enter `r <app-pathname>`, where `<app-pathname>` is the full pathname for your CFM application.

   To enter the application's pathname, drag the application's icon to the Terminal window.

   LaunchCFMApp launches your application.

To pause your application's execution at any time, press Control-C in the Terminal application. To continue your application, enter `cont`. For more information on GDB, enter `help`.

Here are some additional hints that you might find useful:

- From the terminal window, entering `setenv CFMDebugFull 1` directs LaunchCFMApp to display debugging information at application launch time.
- Entering `setenv USERBREAK 1` enables GDB to catch C++ exceptions.
- You can set the environment variable `DYLD_IMAGE_SUFFIX` to specify an optional suffix to add to Mach-O libraries when they are loaded. For example, entering `setenv DYLD_IMAGE_SUFFIX _debug` provides an easy way to link to the debug versions of the various frameworks. You can easily toggle between the normal and debug versions of these libraries without having to rebuild your application each time. The debug versions often perform more assertions, parameter checks, and so on, which may simplify debugging.
- You can call functions in Mach-O libraries directly from the GDB command line, as long as they were explicitly or implicitly loaded. For example, you could call the `CFShow` function, which shows the contents of various Core Foundation and Cocoa objects. Because the Carbon framework is built as Mach-O binaries, you can call Carbon functions from GDB, even those not directly called by your application.
- The remote debugger nub has some command line options which you can view by entering `/usr/libexec/gdb/DebugNub -help`
- If you want to examine parameter values for CFM applications in GDB, you can do so by examining register values. For example, given a function

  ```
  void loofah (int x, int y, int z);
  ```

  then `print $r3` from the GDB command line obtains the value of x (passed in GPR3). Remember that the usual calling conventions apply in determining which parameters are passed in which registers. See *Mac OS Runtime Architectures* for more information about PowerPC calling conventions.

A Porting Example

This chapter details the process required to port a simple application to the Carbon interface. While this application is likely much simpler than your code, many of the steps are similar and you can use this example as a guideline for porting your own application.

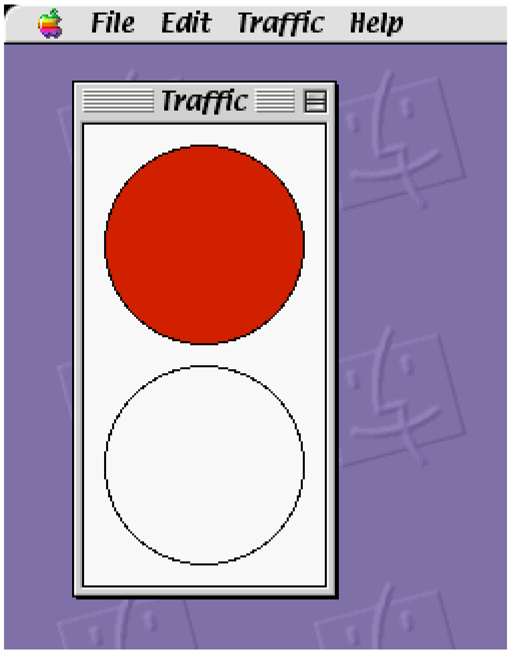
The Sample Application

The application used for this porting example is Sample (also known as TrafficLight), which is an old Mac OS demonstration program used to illustrate basic windowing and user interaction.

The original C code listing is also reproduced in "The Sample Application" (page 97).

Sample puts up a small window containing a rudimentary traffic light which toggles between red and green when you click in the window or when you select a color from the Light menu. Figure 3-1 (page 47) shows the Sample application.

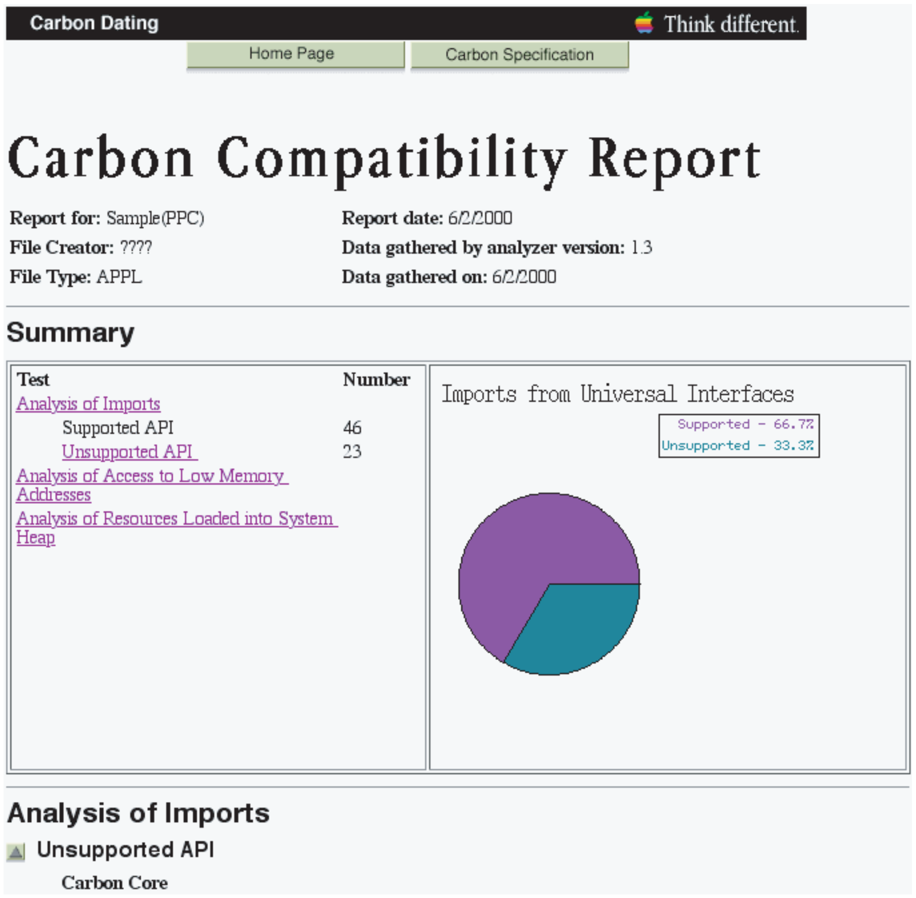
Figure 3-1 The Sample application



Obtaining the Carbon Dater Report

The first step in the porting process is to obtain a Carbon Dater report detailing what APIs will need to be changed or modified. Figure 3-2 shows the opening page of a Carbon Dater report on Sample.

Figure 3-2 A Carbon Dater report



Carbon Compatibility Report

Report for: Sample(PPC)
File Creator: ????
File Type: APPL
Report date: 6/2/2000
Data gathered by analyzer version: 1.3
Data gathered on: 6/2/2000

Summary

Test	Number
Analysis of Imports	
Supported API	46
Unsupported API	23
Analysis of Access to Low Memory Addresses	
Analysis of Resources Loaded into System Heap	

Imports from Universal Interfaces

Analysis of Imports

Unsupported API

Carbon Core

Note that while the percentage of unsupported APIs seems high (33.3 %), this fraction corresponds to only 23 functions. Larger applications typically contain many more supported functions, often resulting in compatibility ratings of 90% or higher.

Table 3-1 summarizes Carbon Dater's comments about the incompatible functions.

Table 3-1 Carbon Dater output for incompatible functions

Manager	Function Name	Comments
Memory Management Utilities	`SysEnvirons`	Uses working directories. Use `FindFolder` and `Gestalt` instead.

Manager	Function Name	Comments
Memory Manager	`ApplicationZone`	Carbon does not support zones because they do not work in a preemptively multitasked environment.
	`GetApplLimit`	Mac OS X applications have no size limit on their application partition.
	`MaxApplZone`	This routine is not needed by PowerPC-based applications because they can specify a stack size in the `'cfrg' 0` resource.
Patch Manager	`NGetTrapAddress`	Patch Manager not supported in Carbon.
Disk Initialization Manager	`DIBadMount`	Carbon does not support the Disk Initialization Manager. Disk Initialization is supported by the system. Mac OS X applications that need to initialize disks can do so using new APIs in the I/OKit.
QuickDraw Manager	`InitGraf`	In Carbon, the Mac OS automatically initializes Quickdraw for every application. When the Mac OS initializes QuickDraw, the Mac OS also automatically calls `InitGraf`.
Device Manager	`CloseDeskAcc`	Desk accessories not supported in Carbon.
	`OpenDeskAcc`	Desk accessories not supported in Carbon.
Dialog Manager	`InitDialogs`	`InitDialogs` is not supported in Carbon. There is no need to initialize the Dialog Manager as the shared library is loaded as needed.
Event Manager	`OSEventAvail`	`OSEventAvail` is not supported in Carbon. Use the `EventAvail` function instead.
	`SystemClick`	Desk accessories are not supported in Carbon.
	`SystemTask`	In Carbon, the Event Manager automatically handles all task scheduling.
Menu Manager	`CheckItem`	Replaced by `CheckMenuItem`.
	`DisableItem`	Replaced by `DisableMenuItem`.
	`EnableItem`	Replaced by `EnableMenuItem`.
	`InitMenus`	`InitMenus` is not supported in Carbon. There is no need to initialize the Menu Manager because the shared library is loaded as needed.
	`SystemEdit`	Carbon does not support desk accessories.
Window Manager	`CloseWindow`	The `CloseWindow` function is not supported because developers do not allocate their own memory for windows in Carbon. Use the `DisposeWindow` function to remove a window instead.

Manager	Function Name	Comments
	`InitWindows`	`InitWindows` is not supported in Carbon. There is no need to initialize the Window Manager because the shared library is loaded as needed.
	`InvalRect`	Calls `InvalWindowRect`, which takes a window pointer as an additional parameter. This change is necessary because invalidation works only on windows, not ports, and windows are not ports in Carbon.
Font Manager	`InitFonts`	There is no need to initialize the Font Manager because the shared library is loaded as needed.
TextEdit	`TEInit`	There is no need to initialize TextEdit because the shared library is loaded as needed.

The Carbon Dater report indicates that many of the incompatible functions are either no longer needed or obsolete (such as those related to desk accessories), while others require just a replacement. There are only a few cases which might require some thoughtful workarounds.

The Basic Port

This section describes the initial steps of the porting process. For clarity, the subsection names parallel the porting steps described in "Preparing Your Code for Carbon" (page 17)

Make Sure All of Your Code is PowerPC–Native

To qualify for this step, the Sample application should compile as a PowerPC executable, which it does. You can also make the following changes to clean up the code:

- Remove the `#pragma segment` statements. These statements indicate which segments should contain which parts of the code. PowerPC code does not use segments, so these are unnecessary.
- Remove `UnloadSeg` calls in `Main` (2 instances). Again, these calls make sense only on 68K machines.
- Remove the comment on segmentation strategy.
- Remove the `TrapAvailable` function and all references to it. Because `WaitNextEvent` is always available on current systems, `TrapAvailable` is superfluous. Also, it relies on the `NGetTrapAddress` function, which is illegal in Carbon anyway, so you might as well get rid of it now.

Update to the Current Universal Interfaces and Use the Carbon SDK

You must make sure that Sample compiles and runs using the latest version of Universal Interfaces, which should be included with the latest version of the Carbon SDK. The examples here assume you are building your code using Metrowerks CodeWarrior. The following changes are necessary to update to the latest headers:

- Remove Desk.h include statement and add Devices.h in both Sample.c and SampleInit.c.
- Change name of GetGlobalMouse function in Sample.c and Sample.h to MyGetGlobalMouse (or something similar). The local function collides with the GetGlobalMouse function in Events.h.
- Remove the reference to _DataInit in main and its external declaration. This function (part of the MPW runtime library) initializes global data on 68K machines.

Target Mac OS 8 and 9 First

This guideline requires you to focus on building a CFM-based Carbon application (at least initially), rather than a Mach-O–based one.

Begin With CarbonAccessors.o

This step is probably the most tedious as it requires you to inspect your code for data structure access that will become illegal in Carbon.

First, add the object file CarbonAccessors.o to your CodeWarrior project.

Then add #define ACCESSOR_CALLS_ARE_FUNCTIONS 1 to the beginning of Sample.c and SampleInit.c. With this setting in place, the compiler generates errors indicating places where you need to add accessor functions.

If you don't mind seeing additional compile errors initially, you can also add #define OPAQUE_TOOLBOX_STRUCTS 1. The compiler will then generate errors if it detects attempts to directly access fields of the now opaque structures. You can use this error list to identify places where you need to modify the code to use accessor functions.

For example, after setting both conditionals, CodeWarrior generates errors such as the following when attempting to compile Sample.c:

```
Error   : cannot convert
'struct OpaqueWindowPtr *' to
'struct OpaqueGrafPtr *'
Sample.c line 224   SetPort(window);    /* the window must be the current port... */

Error   : illegal use of incomplete struct/union/class 'struct OpaqueWindowPtr'
Sample.c line 225   EraseRect(&window->portRect); /* because of a bug in ZoomWindow */

Error   : illegal use of incomplete struct/union/class 'struct OpaqueWindowPtr'
Sample.c line 227   InvalRect(&window->portRect); /* to make things look better on-screen
 */
```

The code indicated by the errors (contained in the function DoEvent) is as follows:

```
...
    case inZoomIn:
    case inZoomOut:
        hit = TrackBox(window, event->where, part);
        if ( hit ) {
            SetPort(window);/* window must be the current port */
            EraseRect(&window->portRect);   /* because of a bug in ZoomWindow */
            ZoomWindow(window, part, true); /* note that we invalidate and erase... */
            InvalRect(&window->portRect);   /* to make things look better on-screen */
        }
        break;
...
```

You must use an accessor to obtain the contents of the `window.portRect` field, and then you must cast the window pointer to a graphics pointer (`GrafPtr`) before setting the port. For more information about available accessor functions and how to use them, see "Functions for Accessing Opaque Data Structures" (page 82).

After the required changes, the code might look something like this:

```
...
    case inZoomIn:
    case inZoomOut:
        hit = TrackBox(window, event->where, part);
        if ( hit ) {
            Rect portRect;              /*•• new variable to hold the value of  */
                                        /*•• the portRect field of window.      */
           GetPortBounds(GetWindowPort(window), &portRect); /*•• new accessor added */
            SetPort(GetWindowPort(window));/*•• The windowPtr is now cast to */
                                         /*•• a GrafPtr before setting */
            EraseRect(&portRect);           /* because of a bug in ZoomWindow */
            ZoomWindow(window, part, true); /* note that we invalidate and erase... */
            InvalRect(&portRect);           /* to make things look better on-screen */
        }
        break;
...
```

Note that the event record (referenced in a parameter for `TrackBox`) is not opaque. This is one of the few Carbon structures that remains accessible without accessors.

You also need to add accessor functions to the following functions:

- `DoEvent`: add an accessor to obtain the `screenbits` field of the `QDGlobals` structure.
- `AdjustCursor` : add accessors to obtain the port bounds, the visible region, and the arrow field of the `QDGlobals` structure. Instead of attempting to get the `portBits` field of the window port and setting the global origin from that, you can obtain the local port bounds, translate them to global coordinates, and set the origin to the upper left corner of those bounds. Also, you must allocate (and afterwards dispose of) a region handle to hold the visible region obtained by the accessor.
- `DoUpdate`: Add an accessor to obtain the visible region.
- `DrawWindow`: Convert the Window pointer to type `GrafPtr` before setting the port.
- `SetLight`: Add an accessor to obtain the port bounds.
- `DoCloseWindow`: You would normally want to use an accessor to obtain the `windowKind` field, but the function using it is `CloseDeskAcc`, which is not supported in Carbon anyway. So the simplest thing to do is to eliminate the code that handles the desk accessory case altogether.
- `IsAppWindow`: Add an accessor to obtain the window kind.
- `IsDAWindow`: Normally you would use an accessor to obtain the window kind, but because the entire function is useful only for desk accessories, it is simpler to remove it altogether.
- `AlertUser`: Add an accessor to obtain the arrow field of the `QDGlobals` structure.

- `Initialize` (in `SampleInit.c`): Instead of determining the size of a window record in order to allocate space for a new window, you can leave these lines out entirely, because the need to preallocate memory for windows has not been an issue for some time. Note that instead of calling `GetNewWindow`, you could call `CreateNewWindow`, which is the suggested replacement for window creation on Mac OS 8.5 and later.

Use Casting Functions to Convert DialogPtrs and WindowPtrs

Use the `GetWindowPort` function to convert window pointers to graphics pointers in the following functions, if you have not already done so: `DoEvent`, `AdjustCursor`, `DrawWindow`, and `SetLight`.

Modify or Conditionalize Your Headers

To allow compilation on both Mac OS X and Mac OS 8 and 9, replace the usual header includes with the following:

```
#define MAC_OS_X_BUILD 0
#if MAC_OS_X_BUILD
        #include <Carbon/Carbon.h>
#else
        #include <Carbon.h>
#endif
```

To build on Mac OS X, you would set the `MAC_OS_X_BUILD` flag to `1`(true).

Note that `Carbon.h` is not required; you could have included the usual Mac OS 8 and 9 headers instead. However, by including `Carbon.h`, you set the preprocessor directive

```
#define TARGET_API_MAC_CARBON 1
```

(if it wasn't already defined) which sets the previous directives (`ACCESSOR_CALLS_ARE_FUNCTIONS` and `OPAQUE_TOOLBOX_STRUCTS`) as well.

Replace Macro Calls to the Mixed Mode Manager With UPP Accessor Functions

Sample does not use universal procedure pointers, so this step is unnecessary.

Move Custom Definition Procedures Out of Resources

Sample uses no custom definition functions, so you can skip this step.

Remove Direct Access to Low-Memory Globals

Sample does not access any low-memory globals, so you can skip this step as well.

Use DebuggingCarbonLib

During development, it's usually useful to keep the debugging version of `CarbonLib` in your Extensions folder instead of the standard `CarbonLib`.

Update Modified or Obsolete Functions

Using the information obtained from Carbon Dater, you can now add the required replacements or modifications for Carbon compatibility. First, remove `CarbonAccessors.o` and `InterfaceLib` from your link path and begin linking exclusively against `CarbonLibStub`.

Any attempts to build Sample at this stage will generate linker errors for any functions that are not available in Carbon. You can use the linker errors and the Carbon Dater report as guides for making the following changes:

- In `main`: Remove `MaxApplZone` as it's not needed in PowerPC applications.
- In `DoEvent`:

  Remove `SystemClick`, which is specific to desk accessories.

  Replace `InvalRect` with `InvalWindowRect`. Note that `InvalWindowRect` takes an additional window pointer as a parameter.

  Remove `DIBadMount`. This function is hardware-specific. If you want to reproduce its functionality on Mac OS X, you must use the I/O Kit API to do so. Actually, because Carbon does not support the `diskEvt` event , you can remove this particular case altogether. The Carbon Event Manager will provide support for disk and volume events.
- In `SetLight`: Replace `InvalRect` with `InvalWindowRect`.
- In `DoMenuCommand`:

  Remove `SystemEdit` because it's specific to desk accessories.

  Remove `OpenDeskAcc` because, again, Carbon doesn't support desk accessories.
- In `DoCloseWindow`: Replace `CloseWindow` with `DisposeWindow`.
- In `AdjustMenus`:

  Replace `EnableItem` with `EnableMenuItem`.

  Replace `CheckItem` with `CheckMenuItem`.

  Normally you would replace `DisableItem` with `DisableMenuItem`. Here, `DisableItem` is used only in the desk accessory case, which will never occur. Therefore, you can remove all instances of `DisableItem` as well as the conditional to distinguish between the traffic light window and desk accessories.
- In `EventLoop`: Remove the `SystemTask` function as it is not needed in Carbon. Actually, because `WaitNextEvent` is guaranteed to be present, you can remove the conditional altogether.
- In `MyGetGlobalMouse`: Change `OSEventAvail` to `EventAvail`.
- In `Initialize`:

  Remove initialization functions `InitGraf`, `InitFonts`, `InitWindows`, `InitMenus`, `TEInit`, and `InitDialogs`, as they are not needed in Carbon.

Remove the SysEnvirons function, which is not supported in Carbon. This function is part of a check to see if WaitNextEvent is available. Because WaitNextEvent is always available in Carbon, you can remove all of the code associated with this check, which eliminates the unsupported functions.

Remove GetApplLimit and ApplicationZone as they are not supported in Carbon.

In addition, it turns out that the WaitNextEvent time LONG_MAX (referenced in the EventLoop function) is not defined in Carbon. You can replace it with 0x7FFFFFFF because there are no periodic actions that need to be taken..

After making these changes, your version of Sample should be able to run on Mac OS 8 and 9 using the CarbonLib extension.

Adopt Required Carbon Technologies

Sample does not require interfaces for printing or saving files, so this step is unnecessary.

Add a 'plst' 0 Resource

To ensure that Sample will launch as a Carbon application on the Mac OS X and not in the Classic environment, you must add a resource of type 'plst' with ID 0 to the resource file TCSample.rsrc. To add this resource, you can either modify the resource file directly using an editor such as ResEdit or Resourcerer, or you can DeRez the file, add text defining the resource, and then recompile the resource file. A minimal 'plst' resource entry would be as follows:

```
data 'plst' (0) {
    $"00"               /* . */
};
```

Conditionalize Quit Menu Items

To make sure that Sample can quit properly in Mac OS X, you must add a Quit Apple event handler to SampleInit.c such as the following:

```
/* Here is our Quit Apple event handler */
static pascal OSErr QuitAppleEventHandler (const AppleEvent *appleEvt,
                                     AppleEvent* reply, UInt32 refcon)
{
    Terminate(); /* close window and terminate gracefully */
} /* QuitAppleEventHandler */
```

To install the handler, you must call AEInstallEventHandler in the Initialize function:

```
...
OSErr err;
err = AEInstallEventHandler( kCoreEventClass, kAEQuitApplication,
                 NewAEEventHandlerUPP(QuitAppleEventHandler), 0, false );
    if (err != noErr) ExitToShell();
...
```

Note that the Apple event installer function requires one of the new UPP accessor functions, `NewAEEventHandlerUPP`. This accessor replaces the old macro `NewAEEventHandlerProc`.

In addition, you must add a case to the `DoEvent` function to process the event properly when it occurs.

```
void DoEvent (EventRecord *event)
    {
...
    /*•• Add a case to process the Quit Apple event */
        case kHighLevelEvent:
            AEProcessAppleEvent( event );
            break;
...
    }
```

After installing the handler, you must adjust the Quit menu item depending on whether Sample is running on Mac OS X or on Mac OS 8 and 9. Because the Quit item appears automatically in the application menu on Mac OS X, you should add code to the `Initialize` function to remove the Quit item from the File menu (where it normally appears for Mac OS 8 and 9) if Sample is running on Mac OS X:

```
...
    SetMenuBar(menuBar);                /* install menus */
    DisposeHandle(menuBar);

/*•• New code begins here */
long result;
MenuRef menu;

err = Gestalt(gestaltMenuMgrAttr, &result);
    if (!err && (result & gestaltMenuMgrAquaLayoutMask)) {
        menu = GetMenuHandle (mFile);
        DeleteMenuItem(menu, iQuit);
        DeleteMenuItem(menu, iQuit-1); /* the element above the Quit */
                                        /* item is a separator */
/*•• End of new code */

    DrawMenuBar();
...
```

This snippet uses `Gestalt` to get the Menu Manager attributes and checks to see if the Aqua interface is present. If it is, then Sample running on Mac OS X. It then simply deletes the Quit item and its separator from the menu bar before it gets drawn.

Cleanup

At this point the Sample application should run on both Mac OS X and Mac OS 8 and 9. However, you can remove a few extraneous bits of code to clean up Sample:

- Remove all references to the variable `gMac`, which was only used in the (now nonexistent) `SysEnvirons` call.
- Remove the `gHasWaitNextEvent` flag. Because `WaitNextEvent` is always available in Carbon, this flag is unnecessary.

If desired, you can now use the same code to build a Mach-O–based version of Sample which can run only on Mac OS X.

Additional Changes for Aqua

While the Carbon version of Sample now executes on Mac OS X, it is also important that the application adheres to the new Aqua interface. Here are a few additional changes you can make to adopt the Aqua look and feel.

Adjust the Window Size

Due to the placement of the three Aqua buttons in each window (the close, minimize, and zoom buttons) and the small default size of Sample's window, the title, Traffic, is truncated. To work around this, you can increase the dimensions of the window and avoid truncation.

Modify the About Box

About boxes in Mac OS X have a consistent appearance that is different from what you may be used to in Mac OS 8 and 9. They should be modeless dialogs that contain the application icon as well as informative text. Figure 3-3 shows an About box created for Sample.

Figure 3-3 The About box for Sample

See *Inside Mac OS X: Aqua Human Interface Guidelines* for the full specifications for the About box.

The Carbon Version of Sample

Listing 3-1 (page 57) and Listing 3-2 (page 66) show the Sample application now ported to Carbon. Changes related to the porting process are indicated by "••" in comments `/*•• just like this */`. Some discussion comments were removed for clarity.

Note: The section "An Example: Adding Carbon Events to Sample" (page 71) describes how to modify Sample to use the Carbon Event Manager.

Figure 3-4 The Carbon version of Sample on Mac OS X

Listing 3-1 Carbon version of Sample.c

```
//#define ACCESSOR_CALLS_ARE_FUNCTIONS 1 /*•• leftovers from the porting process */
//#define OPAQUE_TOOLBOX_STRUCTS 1

#define TARGET_API_MAC_CARBON 1
```

```
#define MAC_OS_X_BUILD  0
/*•• use only one include for all Carbon headers */
#if MAC_OS_X_BUILD
        #include <Carbon/Carbon.h>
    #else
        #include <Carbon.h>
        #endif

#include "Sample.h      "/* bring in all the #defines for Sample */

/* The "g" prefix is used to emphasize that a variable is global. */

/*•• removed gMac and gHasWaitNextEvent global variables, as they are never used */

/* GInBackground is maintained by our osEvent handling routines. Any part of
   the program can check it to find out if it is currently in the background. */
Boolean     gInBackground;      /* maintained by Initialize and DoEvent */


/* The following globals are the state of the window. If we supported more than
   one window, they would be attached to each document, rather than globals. */

/* GStopped tells whether the stop light is currently on stop or go. */
Boolean     gStopped;           /* maintained by Initialize and SetLight */

/* GStopRect and gGoRect are the rectangles of the two stop lights in the window. */
Rect        gStopRect;          /* set up by Initialize */
Rect        gGoRect;            /* set up by Initialize */


/* Define TopLeft and BotRight macros for convenience. Notice the implicit
   dependency on the ordering of fields within a Rect */
#define TopLeft(aRect)  (* (Point *) &(aRect).top)
#define BotRight(aRect) (* (Point *) &(aRect).bottom)


/*•• Removed _DataInit, which is 68K-specific */


void main()
{
/*•• Removed UnloadSeg call, which is 68K-specific */

    /* 1.01 - call to ForceEnvirons removed */
    /*•• Removed MaxApplZone as it's not needed in PowerPC apps */

    Initialize();                   /* initialize the program */
    /*•• Removed UnloadSeg call, which is 68K-specific */

    EventLoop();                    /* call the main event loop */
} /*main*/


void EventLoop()
{
    RgnHandle   cursorRgn;
    Boolean     gotEvent;
    EventRecord event;
    Point       mouse;

    cursorRgn = NewRgn();   /* we'll pass WNE an empty region the 1st time thru */
    do {
        /*•• WNE is always available in Carbon, so the conditional was removed. */
        MyGetGlobalMouse(&mouse);
        AdjustCursor(mouse, cursorRgn);
        /*•• Note wait period LONG_MAX not defined in Carbon. Replaced with 0x7FFFFFFF
 */
        gotEvent = WaitNextEvent(everyEvent, &event, 0x7FFFFFFF, cursorRgn);

        if ( gotEvent ) {
            /* make sure we have the right cursor before handling the event */
            AdjustCursor(event.where, cursorRgn);
            DoEvent(&event);
        }
        /*  If you are using modeless dialogs that have editText items,
            you will want to call IsDialogEvent to give the caret a chance
            to blink, even if WNE/GNE returned FALSE. However, check FrontWindow
            for a non-NIL value before calling IsDialogEvent. */
    } while ( true );   /* loop forever; we quit via ExitToShell */
} /*EventLoop*/


void DoEvent(EventRecord *event)
{
    short       part, err;
    WindowPtr   window;
    Boolean     hit;
    char        key;
    Point       aPoint;
    BitMap screenBits;      /*•• needed to hold contents of qd.screenBits */
    Rect bounds, portRect;  /*•• needed to hold contents of screenbits.bounds and */
                            /*•• the value of the portRect field of the window rec */

    switch ( event->what ) {
        case mouseDown:
            part = FindWindow(event->where, &window);
            switch ( part ) {
                case inMenuBar:         /* process a mouse menu command (if any) */
                    AdjustMenus();
                    DoMenuCommand(MenuSelect(event->where));
                    break;
                case inSysWindow:   /*•• removed SystemClick (not part of Carbon) */
                    break;
                case inContent:
                    if ( window != FrontWindow() ) {
                        SelectWindow(window);
                        /*DoEvent(event);*/ /* use this line for "do first click" */
                    } else
                        DoContentClick(window);
                    break;
                case inDrag:        /* pass screenBits.bounds to get all gDevices */
                   GetQDGlobalsScreenBits(&screenBits); /*•• use accessor to obtain */
                                                        /*•• screenBits */
                    bounds = screenBits.bounds; /*•• get bounds from screenBits.*/
                                              /*•• Note that bitmaps are not opaque */
                    DragWindow(window, event->where, &bounds);
                    break;
                case inGrow:
                    break;
                case inZoomIn:
                case inZoomOut:
                    hit = TrackBox(window, event->where, part);
                    if ( hit ) {
                        /*•• new accessor in place */
                        GetPortBounds(GetWindowPort(window), &portRect);

                        /*•• The windowPtr is now cast to a GrafPtr before setting */
                        SetPort(GetWindowPort(window));

                        EraseRect(&portRect);   /* because of a bug in ZoomWindow */
                        ZoomWindow(window, part, true); /* note that we invalidate */
                                                /* and erase...to make things look */
                                                /* better on-screen */

                        /*•• InvalRect replaced with InvalWindowRect */
                        InvalWindowRect(window, &portRect);
                    }
                    break;
            }
            break;
        case keyDown:
        case autoKey:                       /* check for menukey equivalents */
            key = event->message & charCodeMask;
            if ( event->modifiers & cmdKey )            /* Command key down */
                if ( event->what == keyDown ) {
                    AdjustMenus();      /* enable/disable/check menu items properly */
                    DoMenuCommand(MenuKey(key));
                }
            break;
        case activateEvt:
            DoActivate((WindowPtr) event->message,
                        (event->modifiers & activeFlag) != 0);
            break;
        case updateEvt:
            DoUpdate((WindowPtr) event->message);
            break;
        /*•• Add a case to process the Quit Apple event */
        case kHighLevelEvent:
            AEProcessAppleEvent( event );
            break;

        /*•• Removed diskEvt case (and DIBadMount)--not supported in Carbon

        case kOSEvent:
        /*  1.02 - must BitAND with 0x0FF to get only low byte */
            switch ((event->message >> 24) & 0x0FF) {   /* high byte of message */
                case kSuspendResumeMessage: /* suspend/resume is also an activate/ */
                                            /* deactivate */
                    gInBackground = (event->message & kResumeMask) == 0;
                    DoActivate(FrontWindow(), !gInBackground);
                    break;
            }
            break;
    }
} /*DoEvent*/


void AdjustCursor(Point mouse, RgnHandle region)
{
    WindowPtr   window;
    RgnHandle   arrowRgn;
    RgnHandle   plusRgn;
    RgnHandle   visRgn; /*•• needed to hold field value obtained by accessor function
*/
    Cursor      arrow; /*•• used to hold the contents of the qd.arrow field */
    Rect        globalPortRect, portRect;

    window = FrontWindow(); /* we only adjust the cursor when we are in front */

    if (! gInBackground) {  /*•• removed desk accessory case from conditional */
        /* calculate regions for different cursor shapes */
        arrowRgn = NewRgn();
        plusRgn = NewRgn();

        /* start with a big, big rectangular region */
        SetRectRgn(arrowRgn, kExtremeNeg, kExtremeNeg, kExtremePos, kExtremePos);

        /* calculate plusRgn */
        if ( IsAppWindow(window) ) {

            Point tempPoint;

           SetPort(GetWindowPort(window)); /* make a global version of the viewRect */
            /*•• Added accessor to cast WindowPtr to GrafPtr */
            GetPortBounds (GetWindowPort(window), &portRect);
           SetPt(&tempPoint, portRect.left, portRect.top); /*•• obtain local origin */
            LocalToGlobal(&tempPoint); /*•• translate point to global coordinates */
            SetOrigin(tempPoint.h, tempPoint.v); /*•• Set the global origin */

            /*•• Added accessor to get value for globalPortRect */
            GetPortBounds(GetWindowPort(window), &globalPortRect);
            RectRgn(plusRgn, &globalPortRect);

            visRgn = NewRgn();/*•• allocate a new region */
            /*•• Added accessor to get value for visRgn */
            GetPortVisibleRegion(GetWindowPort(window), visRgn);
            SectRgn(plusRgn, visRgn, plusRgn);
            SetOrigin(0, 0);
            DisposeRgn(visRgn);/*•• dispose of the region */
        }

        /* subtract other regions from arrowRgn */
            DiffRgn(arrowRgn, plusRgn, arrowRgn);

        /* change the cursor and the region parameter */
        if ( PtInRgn(mouse, plusRgn) ) {
                SetCursor(*GetCursor(plusCursor));
                CopyRgn(plusRgn, region);
        } else {
                SetCursor(GetQDGlobalsArrow(&arrow)); /*•• new accessor in place */
                CopyRgn(arrowRgn, region);
        }

        /* get rid of our local regions */
        DisposeRgn(arrowRgn);
        DisposeRgn(plusRgn);
    }
} /*AdjustCursor*/


/*•• "My" added to GetGlobalMouse to avoid name collision with the function in Events.h
 */

void MyGetGlobalMouse(Point *mouse)
{
    EventRecord event;

    /*•• Changed OSEventAvail to EventAvail */
    EventAvail(kNoEvents, &event);  /* we aren't interested in any events */
    *mouse = event.where;               /* just the mouse position */
} /*MyGetGlobalMouse*/


void DoUpdate(WindowPtr window)
{

    if (IsAppWindow(window)) {
        RgnHandle  visRgn; /*•• needed to hold contents of window->visRgn */

        BeginUpdate(window);                /* this sets up the visRgn */
        visRgn = NewRgn();
        /*•• Added accessor to obtain the visRgn */
        GetPortVisibleRegion (GetWindowPort(window), visRgn);
        if ( ! EmptyRgn(visRgn) )   /* draw if updating needs to be done */
            DrawWindow(window);
        EndUpdate(window);
        }
} /*DoUpdate*/


void DoActivate(WindowPtr window, Boolean becomingActive)
{
    if (IsAppWindow(window)) {

        if ( becomingActive )
            /* do whatever you need to at activation */ ;
        else
            /* do whatever you need to at deactivation */ ;
        }
} /*DoActivate*/


void DoContentClick(WindowPtr window)
{
    SetLight(window, ! gStopped);
} /*DoContentClick*/


void DrawWindow(WindowPtr window)
```

```
{
    Rect portRect; /*•• Needed to hold the contents of window->portRect */

    SetPort(GetWindowPort(window));

     /*•• Use accessor to obtain port bounds */
    GetPortBounds(GetWindowPort(window), &portRect);

    EraseRect(&portRect);   /* clear out any garbage that may linger */

    if ( gStopped )                 /* draw a red (or white) stop light */
        ForeColor(redColor);
    else
        ForeColor(whiteColor);

    PaintOval(&gStopRect);
    ForeColor(blackColor);
    FrameOval(&gStopRect);

    if ( ! gStopped )               /* draw a green (or white) go light */
        ForeColor(greenColor);
    else
        ForeColor(whiteColor);

    PaintOval(&gGoRect);
    ForeColor(blackColor);
    FrameOval(&gGoRect);
} /*DrawWindow*/


void AdjustMenus()
{
    MenuHandle  menu;

    /*•• Removed references to IsDAWindow, because desk accessories are not in Carbon*/
    /*•• removed DisableItems and all code dealing with the desk accessory case.*/

    menu = GetMenuHandle(mLight);
    EnableMenuItem(menu, iStop); /*•• replaced EnableItem with EnableMenuItem */
    EnableMenuItem(menu, iGo);

    /*•• replaced CheckItem with CheckMenuItem */
    CheckMenuItem(menu, iStop, gStopped); /* we can also determine check/uncheck */
                                        /* state, too */
    CheckMenuItem(menu, iGo, ! gStopped);
} /*AdjustMenus*/


void DoMenuCommand(long menuResult)
{
    short       menuID;             /* the resource ID of the selected menu */
    short       menuItem;           /* the item number of the selected menu */
    short       itemHit;
    Str255      daName;
    short       daRefNum;
    Boolean     handledByDA;

    menuID = HiWord(menuResult);    /* use macros for efficiency to... */
    menuItem = LoWord(menuResult);  /* get menu item number and menu number */
    switch ( menuID ) {
        case mApple:
            switch ( menuItem ) {
                case iAbout:        /* bring up alert for About */
                    itemHit = Alert(rAboutAlert, nil);
                    break;
                default:/* all non-About items in this menu are DAs */
                        /*•• removed desk accessory code (not supported in Carbon) */
                    break;
            }
            break;
        case mFile:
            switch ( menuItem ) {
                case iClose:
                    DoCloseWindow(FrontWindow());
                    break;
                case iQuit:
                    Terminate();
                    break;
            }
            break;
        case mEdit:     /* call SystemEdit for DA editing & MultiFinder */
                       /*•• removed because Carbon doesn't support desk accessories */
            break;
        case mLight:
            switch ( menuItem ) {
                case iStop:
                    SetLight(FrontWindow(), true);
                    break;
                case iGo:
                    SetLight(FrontWindow(), false);
                    break;
            }
            break;
    }
    HiliteMenu(0);      /* unhighlight what MenuSelect (or MenuKey) hilited */
} /*DoMenuCommand*/


void SetLight(WindowPtr window, Boolean newStopped)
{
    if ( newStopped != gStopped ) {

        Rect portRect;  /*•• Needed to hold port bounds */

        gStopped = newStopped;

        /*•• Use accessor to obtain port bounds */
        GetPortBounds(GetWindowPort(window), &portRect);

        /*•• Use accessor to cast WindowPtr to GrafPtr */
        SetPort(GetWindowPort(window));

        InvalWindowRect(window,&portRect);
    }
} /*SetLight*/
```

```
Boolean DoCloseWindow(WindowPtr window)
{
    /*•• Desk accessory-related code removed, because it's not supported in Carbon */

    DisposeWindow(window); /*•• replaced CloseWindow with DisposeWindow */
    return true;
} /*DoCloseWindow*/


void Terminate()
{
    WindowPtr   aWindow;
    Boolean     closed;

    closed = true;
    do {
        aWindow = FrontWindow();                /* get the current front window */
        if (aWindow != nil)
            closed = DoCloseWindow(aWindow);    /* close this window */
    }
    while (closed && (aWindow != nil));
    if (closed)
        ExitToShell();                          /* exit if no cancellation */
} /*Terminate*/


Boolean IsAppWindow(WindowPtr window)
{
    short       windowKind;

    if ( window == nil )
        return false;
    else {  /* application windows have windowKinds = userKind (8) */
        windowKind = GetWindowKind(window);
        return ( windowKind == userKind );
    }
} /*IsAppWindow*/


/* Boolean IsDAWindow(WindowPtr window)                */
/*•• Carbon does not support desk accessories, so we removed this function */
/*                                                                          */
/*IsDAWindow*/


void AlertUser()
{
    short       itemHit;
    Cursor      arrow; /*•• used to hold the contents of the qd.arrow field */

    SetCursor(GetQDGlobalsArrow(&arrow)); /*•• new accessor in place */
    itemHit = Alert(rUserAlert, nil);
    ExitToShell();
} /* AlertUser */
```

Listing 3-2 Carbon version of SampleInit.c

```
//#define ACCESSOR_CALLS_ARE_FUNCTIONS 1 /*•• leftovers from the porting process */
//#define OPAQUE_TOOLBOX_STRUCTS 1

#define TARGET_API_MAC_CARBON 1

#define MAC_OS_X_BUILD  0
/*•• use only one include for all Carbon headers */
#if MAC_OS_X_BUILD
        #include <Carbon/Carbon.h>
    #else
        #include <Carbon.h>
        #endif

#include "Sample.h      "/* bring in all the #defines for Sample */


/* The "g" prefix is used to emphasize that a variable is global. */
/* All are extern since the variables are declared in the main segment. */

/*•• removed gMac and gHasWaitNextEvent global variables, as they are never used */

/* GInBackground is maintained by our osEvent handling routines. Any part of
   the program can check it to find out if it is currently in the background. */
extern Boolean      gInBackground;      /* maintained by Initialize and DoEvent */


/* The following globals are the state of the window. If we supported more than
   one window, they would be attached to each document, rather than globals. */

/* GStopped tells whether the stop light is currently on stop or go. */
extern Boolean      gStopped;           /* maintained by Initialize and SetLight */

/* GStopRect and gGoRect are the rectangles of the two stop lights in the window. */
extern Rect     gStopRect;          /* set up by Initialize */
extern Rect     gGoRect;            /* set up by Initialize */


/*•• Here is our Quit Apple event handler */
static pascal OSErr QuitAppleEventHandler( const AppleEvent *appleEvt,
                                            AppleEvent* reply, UInt32 refcon )
{
    Terminate(); /* close window and terminate gracefully */
}   /*•• QuitAppleEventHandler */


void Initialize()
{
    Handle      menuBar;
    WindowPtr   window;
    long        total, contig;
    EventRecord event;
    short       count;
    MenuRef     menu;
    long        result;  /*•• used to hold results of Gestalt call */
    OSErr       err;
    gInBackground = false;

    /*•• Removed most of the init functions (InitGraf, etc) as they */
    /*•• are not needed in Carbon */
    InitCursor();

    /*  Call MPPOpen and ATPLoad at this point to initialize AppleTalk,
        if you are using it. */

    /*  This next bit of code is necessary to allow the default button of our
        alert be outlined.
        1.02 - Changed to call EventAvail so that we don't lose some important
        events. */

    for (count = 1; count <= 3; count++)
        EventAvail(everyEvent, &event);

    /*•• WaitNextEvent is always available in Carbon, so we eliminated the code that */
    /*•• calls SysEnvirons and checks a trap for its presence. */

    /*•• Removed calls to GetApplLimit and ApplicationZone. They are not supported */
    /*•• in Carbon, and the memory problem they protect against is no longer an issue
*/

    PurgeSpace(&total, &contig);
    if (total < kMinSpace) AlertUser();

    /*•• Removed code to preallocate space for our window, because memory */
    /*•• requirements are no longer strict enough to make it necessary */

    window = GetNewWindow(rWindow, nil, (WindowPtr) -1);

    menuBar = GetNewMBar(rMenuBar);         /* read menus into menu bar */
    if ( menuBar == nil ) AlertUser();

    SetMenuBar(menuBar);                    /* install menus */
    DisposeHandle(menuBar);
    /*•• Removed code that added desk accessories to the Apple Menu */

    /*•• Determine if we're running on Mac OS X, and if we are, remove the Quit menu */
    /*•• item and the Quit separator from the File Menu */
    err = Gestalt(gestaltMenuMgrAttr, &result);
    if (!err && (result & gestaltMenuMgrAquaLayoutMask)) {
        menu = GetMenuHandle (mFile);
        DeleteMenuItem(menu, iQuit);
        DeleteMenuItem(menu, iQuit-1); /*•• the element above the Quit item */
                                  /* •• is a separator */
        }

    DrawMenuBar();

    /*•• Install a Quit Apple Event handler to make sure that the application can */
    /*•• quit properly on Mac OS X. It's also just good programming practice on 8/9 */
    err = AEInstallEventHandler( kCoreEventClass, kAEQuitApplication,
                          NewAEEventHandlerUPP(QuitAppleEventHandler), 0, false );
    if (err != noErr) ExitToShell();

    gStopped = true;
    if ( !GoGetRect(rStopRect, &gStopRect) )
        AlertUser();                        /* the stop light rectangle */
    if ( !GoGetRect(rGoRect, &gGoRect) )
        AlertUser();                        /* the go light rectangle */
} /*Initialize*/


Boolean GoGetRect(short rectID, Rect *theRect)
{
    Handle      resource;

    resource = GetResource('RECT', rectID);
    if ( resource != nil ) {
        *theRect = **((Rect**) resource);
        return true;
    }
    else
        return false;
} /* GoGetRect */


/* TrapAvailable */
/*•• Carbon does not support traps, so this function was removed. */

/*TrapAvailable*/
```

New Carbon Technologies

This chapter describes several Mac OS technologies that are new with Carbon. While these technologies are not required in Carbon applications, adopting them can result in improved performance and user experience as well as reduced development cycles.

Carbon Event Manager

The Carbon Event Manager is an event handling API that replaces the Classic Mac OS Event Manager. It simplifies the event model and it is well-suited for the preemptive multitasking capabilities of Mac OS X. For example, when using Carbon events, an application that is performing periodic actions while idle (blinking the cursor, for example) does not have to endlessly cycle through a `WaitNextEvent` loop while doing so. The advantages of the Carbon Event Manager include the following:

- Handlers for common events (such as mouse events and keyboard events) are included. You don't need to write your own event handlers unless you want to override default behaviors.
- The Carbon Event Manager can handle any number of event types (as opposed to the16 event types available in the Classic event record).
- The Carbon Event Manager handles notifications and defproc messaging in addition to the usual events.
- The streamlining of the event handling system results in a more responsive system and a better user experience.

When adapting an application to use Carbon events, you typically replace the `WaitNextEvent` loop and event processing functions with one of more simple event handling calls. Your application must register the types of events it wishes to be notified about, and then implement handlers to address the registered events.

The Carbon Event Manager is available for Carbon applications running on Mac OS 8.6 and later, so if you do not need compatibility back to Mac OS 8.1, we highly encourage you to adopt it. The Carbon event model is flexible enough to coexist with `WaitNextEvent`, so you can make the adoption as gradual as you like. While the `WaitNextEvent` event model still works on Mac OS X, your programs and the overall system will perform better if you use Carbon events.

For more information about the Carbon Event Manager, see the documentation available at

http://developer.apple.com/documentation/carbon/Reference/Carbon_Event_Manager_Ref/index.html

as well as "An Example: Adding Carbon Events to Sample" (page 71).

Core Foundation

Core Foundation is a new set of APIs that provides a simple interface for handling many common needs for applications. For example, CFPreference APIs provide a standard interface for creating and manipulating an application's user preferences. As most Core Foundation functions are part of the Carbon API, they run on both Mac OS X and Mac OS 8 and 9. In addition, Core Foundation functions are compatible with the Foundation classes available in the Cocoa environment, which simplifies the sharing of data between Carbon and Cocoa applications.

In addition to Preferences Services, other Core Foundation services that you may find useful for your Carbon application include the following:

- Bundle Services, which provides the APIs used to access files and other resources stored in a bundle hierarchy. See "Consider Using Bundles" (page 36) for more information about bundles.
- Plug-In Services, which provides a standard plug-in architecture for Mac OS X and Mac OS 8 and 9 applications. You can package plug-in binaries for multiple platforms together, and the Plug-In Services APIs can transparently load the proper one (assuming, of course, that they share the same interface).
- String Services, which provides a simple interface for storing, converting, and manipulating Unicode strings. If your application uses (or is planning to support) Unicode, you should consider adopting String Services.
- The XML Parser, which provides an interface for writing and reading XML documents.
- Property List Services, which provides an interface to organize data into property lists ("plists"). It also allows you to convert hierarchically structured combinations of basic data types in these lists to and from standard XML.

For more information about Core Foundation Services, see *Inside Mac OS X: System Overview* and Core Foundation documentation at the Carbon documentation site:

http://developer.apple.com/documentation/Carbon/index.html

DataBrowser

DataBrowser is a new Control Manager control (defined in `ControlDefinitions.h`) that lets you display data in sortable, navigatable lists in a manner similar to the list view and column view settings of the Finder. If you need to organize data in manner that is easily accessible to the user, you should consider using the DataBrowser. Here are some advantages of the various views:

- The list view allows the user to sort by various column attributes related to the data you are displaying. For example, the Finder allows you to display files by type, size, or date modified, among other characteristics.
- The column view is useful for navigating large hierarchies or trees of data. For example both the Mac OS X Finder and the Navigation Services Save dialog box use DataBrowser to let the user quickly find a particular location in a volume's file hierarchy.

For more information about DataBrowser, see the sample code included with the Carbon SDK.

Multilingual Text Engine (MLTE)

The Multilingual Text Engine (MLTE) is the suggested replacement for TextEdit in Carbon applications. While you can still use TextEdit in Carbon, MLTE provides many additional features and simplifies the programming interface; you can accomplish more with fewer lines of code. Some of MLTE's features include the following:

- Full Unicode support, including transparent access to Apple Type Services for Unicode Imaging (ATSUI) for rendering text, and the Text Encoding Converter (TEC) for converting between encodings.
- Full support for alternate input methods using the Text Services Manager (TSM).
- Support for greater than 32 KB of text.
- Built-in scroll bar handling.
- Full justification of text.
- Built-in support for basic user actions, such as highlighting selected text, dragging selected text, and moving the caret in response to arrow key presses.
- Multiple levels of undo.
- Built-in printing support.

MLTE is available in CarbonLib 1.2 and later. You can also use it in non-Carbon applications on Mac OS 8.6 and later.

For more information about MLTE, see the documentation available at

http://developer.apple.com/documentation/Carbon/Reference/Multilingual_Text_Engine/index.html

and the MLTE SDK at

http://developer.apple.com/sdk/

An Example: Adding Carbon Events to Sample

The Carbon Event Manager is the most important of the optional technologies available with Carbon; if you plan to add only one new API to your application, it should be Carbon events. In addition to simplifying your event handling code, Carbon events will make your application a good processor-sharing citizen on Mac OS X, which will improve the performance of all running applications.

This section illustrates the adoption of the Carbon Event Manager by adding Carbon events to the Carbon version of the Sample application shown in Listing 3-1 (page 57) and Listing 3-2 (page 66).

"Determine the Appropriate CarbonLib Version" (page 27) indicates that the Carbon Event Manager is available only in CarbonLib 1.2 and later when Mac OS 8.6 or later is present. While this means that you cannot run on Mac OS 8.1, it also means that you are free to incorporate any newer APIs up to Mac OS 8.6 if that makes the work easier.

The basic model for Carbon events is that you register callback handlers for each event (or type of event) you wish to handle. You attach these handlers to specific objects, such as a window or a button. Different objects of the same type do not have to have the same handler. For example, two buttons can each have their own distinct handler. With this level of flexibility in handling events, it is important to determine the scope required for each event. For example, should an event affect a single window, all open windows, or the entire application? In most cases it is easier to think in terms of what object is affected rather than what event occurred.

After any initial setup, your application calls `RunApplicationEventLoop`, which essentially replaces the `WaitNextEvent` loop. From that point on, your application is notified only when an event you specified occurs.

Standard Event Handlers

The Carbon Event Manager provides default event handlers for many common types of events. For example, the standard handler for a window automatically handles dragging, activation, deactivation, window zooming and resizing. Of course, you will mostly likely still need to draw into or update the content region as a result of some actions, but much of the basic work is taken care of for you. A good rule of thumb is to install the standard handler first, see what actions it covers, and then write handlers for any additional actions you may want to take.

The Basic Conversion

In the Carbon version of Sample, most of the event handling occurs in the functions `EventLoop` and `DoEvent`. Here is a breakdown of the events to address:

- Adjusting the cursor shape depending on the region it occupies: Previously the cursor was adjusted each time through the event loop. Because there is no equivalent event loop that we can access in Carbon events, an alternate triggering mechanism is required. One method would be to update the cursor whenever the mouse moves. Another would be to set up a timer to adjust the cursor at regular intervals.
- Menu selections: The Carbon Event Manager can associate special command events with menu items. Many common menu items have command events defined for them in `CarbonEvents.h`, and you can assign your own using the Window Manager function `SetMenuItemCommandID`. When a menu item is selected (either through menu selection or a keyboard equivalent), the Carbon Event Manager sends the appropriate command event to the handler. Some menu selections are related to the window only (such as setting the traffic light color), while others are related to the application (such as the About command). This division suggests a menu command handler at both the window level and the application level.

  Note that the standard application handler automatically handles basic menu tracking, highlighting, and so on. All you need to do is process the menu selection.
- Keyboard events: Because the only keyboard input Sample requires are keyboard equivalents for menu items, you can handle these the same as menu selections.
- Mouse clicks in the traffic light window: The light color toggles on each click in the content region of the window. This event is entirely window-related, which suggests a window-level handler.
- Window dragging, zooming, resizing, activation, and deactivation: The standard window event handler addresses these events, so all you need to do is update the content region if necessary.
- Update events: The Carbon Event Manager posts events indicating that you should change your window contents, so you should redraw the contents at that time.
- High level (Apple) events: The only case to worry about is the Quit Apple event, for which we have already installed a handler.

- Disk Events: This case deals with bad floppy disks, and Carbon does not support the `DIBadMount` function or the `diskEvt` event. This example ignores disk events.
- Application suspend and resume events: The standard application event handler covers the basic functionality required for suspend and resume events.

This information indicates that event handlers are needed at both the window-level and the application level.

Installing the Standard Event Handlers

Before adding your application-specific event handlers, you should install the standard handlers for window and application events.

One way to assign the standard window handler to the traffic light window is to call the function `InstallStandardWindowEventHandler` after creating the window in the `Initialize` function.

```
window = GetNewWindow(rWindow, NULL, (WindowPtr)-1);
InstallStandardEventHandler(GetWindowEventTarget(window)); /* installs the default */
                                                    /* handler for window events */
ShowWindow(window);
```

The `GetWindowEventTarget` function returns an event reference (type `EventTargetRef`) to associate with the desired object (in this case, a window). Similar functions exist to create event references for controls, menus, and other objects.

However, because the application must run in Mac OS 8.6 or later, you can call the Window Manager function `CreateNewWindow` instead, which lets you specify an attribute to use the standard window handler. Doing so also provides many of the standard window controls (resize button, and so on) for free.

```
Rect windowBounds;            /* use Rect for bounds in CreateNewWindow */
WindowAttributes windowAttr;  /* to hold window attribute flags in CreateNewWindow
 */
...
windowAttr = kWindowStandardDocumentAttributes| /* standard window */
              kWindowStandardHandlerAttribute| /* standard window event handler
 */
                kWindowInWindowMenuAttribute;

SetRect (&windowBounds, 40, 60, 280, 520); /* bounds for the new window */

CreateNewWindow(kDocumentWindowClass, windowAttr, &windowBounds, &window);
SetWindowTitleWithCFString(window, CFSTR("Traffic"));

ChangeWindowAttributes(window, NULL, /* remove close box and resize tab */
                         kWindowCloseBoxAttribute|kWindowResizableAttribute);
ShowWindow(window);
```

The `SetWindowTitleWithCFString` function is a Core Foundation String Services function.

To more closely approximate the original Sample, you can call the Window Manager function `ChangeWindowAttributes` to remove the close box and the resize tab.

The standard application event handler is installed automatically when you call `RunApplicationEventLoop`, so you do not need to explicitly install it.

Registering Your Own Event Handlers

After installing the standard handlers, you must register your event handlers with the system. The Carbon Event Manager defines events by the class of event (window, mouse, and so on) as well as the type (mouse moved, content region clicked, and so on). You must specify these when registering your handlers, as in this example:

```
EventTypeSpec   appEventList[] = {{kEventClassCommand, kEventCommandProcess},
                                { kEventClassMouse, kEventMouseMoved}};

EventTypeSpec   windEventList[] = {{kEventClassWindow, kEventWindowDrawContent },
                                 { kEventClassWindow, kEventWindowClickContentRgn },
                                 { kEventClassWindow, kEventWindowBoundsChanged},
                                 { kEventClassCommand, kEventCommandProcess}};

...

    /* Installing the application event handler */
    InstallApplicationEventHandler(NewEventHandlerUPP(MyAppEventHandler),
                                2, appEventList, 0, NULL);

    /* Installing the window event handler */
    InstallWindowEventHandler(window, NewEventHandlerUPP(MyWindowEventHandler),
                                4, windEventList, 0, NULL);
```

The type `EventTypeSpec` arrays hold the pairs of event classes and types which are then passed into the appropriate handler installation calls. The calls `InstallApplicationEventHandler` and `InstallWindowEventHandler` are macros derived from the more general Carbon Event Manager function `InstallEventHandler`. Remember to pass universal procedure pointers instead of normal pointers when specifying your callback handlers. `CarbonEvents.h` defines the format for your callback handlers.

If desired, you can register individual event handlers for each event. However, for this example it is convenient to group them by object.

Note: The handlers you install complement the standard event handlers described earlier. For example, the standard window handler will resize a window in response to a resize event. However, if you indicated that you wanted to handle window resize events in your own handler, you could specify additional actions to take (such as refreshing the content window after the resize) while still allowing the standard handler to perform the window resize.

The handlers in this example essentially take the place of the `DoEvent` function, which called other functions to process the events.

The Application-Level Event Handler

Listing 4-1 shows an application-level event handler for the Sample application.

Listing 4-1 Application-level event handler for Sample

```
static pascal OSStatus MyAppEventHandler (EventHandlerCallRef myHandlerChain,
                                         EventRef event, void* userData)
{
    UInt32          whatHappened;
    HICommand       commandStruct;
    Point           wheresMyMouse;
    RgnHandle       CursorRgn;
    short           itemHit;
    OSStatus        result = eventNotHandledErr; /* report failure by default */


    whatHappened = GetEventKind(event);

    switch (whatHappened)
        {
            case kEventCommandProcess:

                    GetEventParameter (event, kEventParamDirectObject,
                                        typeHICommand, NULL, sizeof(HICommand),
                                        NULL, &commandStruct);

                    switch (commandStruct.commandID)
                        {
                            case kCommandAbout:
                                itemHit = Alert (rAboutAlert, nil);
                                result = noErr;
                                break;
                            default:
                                break;
                        }
                    break;

            case kEventMouseMoved:

                    CursorRgn = NewRgn();
                    GetEventParameter (event, kEventParamMouseLocation, typeQDPoint,
                                         NULL, sizeof(Point), NULL, &wheresMyMouse);
                    AdjustCursor(wheresMyMouse, CursorRgn);
                    DisposeRgn(CursorRgn);
                    result = noErr;
                    break;

            default:
                    break;
        }
    return result;
}
```

The event handler takes three parameters:

- The `myHandlerChain` parameter is a reference to the handler calling chain; that is, the hierarchy of event handlers that could handle this event. You would pass this reference if you wanted to call `CallNextEventHandler`, for example, which you could use to add pre- or postprocessing to the actions of a standard handler.
- The `event` parameter contains specific information related to the event (much the way the fields of an event record hold event-specific information).
- The `userData` field holds any user data you specified when you registered your handler with the call to `InstallEventHandler` (none in this case).

When an event occurs, `MyAppEventHandler` gets passed the event along with any user data you may have requested (none in this case). It then calls the `GetEventKind` function to determine the type of event that occurred and then handles the event appropriately.

The`kEventCommandProcess` function indicates a menu-related command occurred. By calling the `GetEventParameter` function, the handler determines which item was selected. At the application level only one command is possible: the About selection. Note `kCommandAbout` is not defined in `CarbonEvents.h` so, you need to define it yourself. You can do so and then call the Menu Manager function `SetMenuItemCommandID` in the `Initialize` function to register it with the system.

```
const MenuCommand kCommandAbout = FOUR_CHAR_CODE ('abou');

void Initialize()
{
    ...
    SetMenuItemCommandID (GetMenuRef(mApple), iAbout, kCommandAbout);
    ...
}
```

Note that instead of calling `SetMenuItemCommandID` to assign the command ID, you could choose the define it in an `'xmnu'` resource.

You don't need to handle the Quit event because the standard application event handler calls the default Quit Apple event handler when this occurs. If you need to take additional actions before quitting, you can install your own Quit Apple event handler. If you are not using `RunApplicationEventLoop`, (and therefore not using the standard application handler), you can process the Quit event here. Typically you call the function `QuitApplicationEventLoop` to break out of the Carbon event loop.

If you are running your application on Mac OS 8 and 9, you need to register the Quit command ID (defined as `kHICommandQuit` in `CarbonEvents.h`) using the `SetMenuItemCommandID` function, much as you had to for the About item. If you don't, the Carbon Event Manager will not properly process the event when the Quit item is selected. A convenient time to do this is when you use Gestalt to determine whether to place a Quit item in the File menu.

```
err = Gestalt(gestaltMenuMgrAttr, &result);
if (!err && (result & gestaltMenuMgrAquaLayoutMask)) {
        menu = GetMenuHandle (mFile);
        DeleteMenuItem(menu, iQuit);
        DeleteMenuItem(menu, iQuit-1); /*•• the element above the Quit item */
                                       /*•• is a separator */
    }
    else
    {   /* Assign a command ID to the Quit Item so that the Carbon Event Manager */
        /* can recognize it. */
        SetMenuItemCommandID(GetMenuRef(mFile), iQuit, kHICommandQuit);
    }
```

The other application-level event you need to handle is the mouse-moved event, which determines whether or not to adjust the cursor shape. The handler for Sample calls `GetEventParameter` to obtain the mouse position (the types of parameters you can obtain depends on the event that occurred) and then calls `AdjustCursor`.

Alternatively, you can adjust the cursor periodically by using a Carbon event timer. Doing so merely involves creating the function to call and then registering it by calling `InstallEventLoopTimer`. However, this method is similar to polling for an event, which is more processor-intensive, and therefore not suggested for Mac OS X.

The Window Event Handler

Listing 4-2 shows a window event handler for Sample.

Listing 4-2 Window event handler for Sample

```
static pascal OSStatus MyWindowEventHandler(EventHandlerCallRef myHandler,
                                            EventRef event, void* userData)
{
    WindowRef           window;
    Rect                bounds;
    UInt32              whatHappened;
    HICommand           commandStruct;
    MenuRef             theMenuRef;
    UInt16              theMenuItem;
    OSStatus            result = eventNotHandledErr; /* report failure by default */

    GetEventParameter(event, kEventParamDirectObject, typeWindowRef, NULL,
                         sizeof(window), NULL, &window);

    whatHappened = GetEventKind(event);

    switch (whatHappened)
        {
            case kEventWindowDrawContent:

                    DoUpdate(window);
                    result = noErr;
                    break;

            case kEventWindowBoundsChanged:

                    InvalWindowRect(window, GetWindowPortBounds(window, &bounds));
                    DoUpdate(window);
                    result = noErr;
                    break;

            case kEventWindowClickContentRgn:

                    DoContentClick(window);
                    DoUpdate(window);
                    AdjustMenus();
                    result = noErr;
                    break;

            case kEventCommandProcess:

                    GetEventParameter (event, kEventParamDirectObject,
                                        typeHICommand, NULL, sizeof(HICommand),
                                        NULL, &commandStruct);

                    theMenuRef = commandStruct.menu.menuRef;
                    if (theMenuRef == GetMenuHandle(mLight))
                        {
                            /* Because the event didn't occur *in* the window, the */
                            /* window reference isn't valid until we set it here */
                            window = FrontWindow();
                            theMenuItem = commandStruct.menu.menuItemIndex;
                            switch ( theMenuItem )
                                {
                                    case iStop:
                                        SetLight(window, true);
                                        break;
                                    case iGo:
                                        SetLight(window, false);
                                        break;
                                }
                            DoUpdate(window);
                            AdjustMenus();
                            result = noErr;
                        }
                    break;

            default:
                    /* If nobody handled the event, it gets propagated to the */
                    /* application-level handler. */
                    break;

        }

    return result;
}
```

As with the application-level handler, the window event handler first calls `GetEventKind` to determine the type of event and then processes them appropriately:

- `kEventWindowDrawContent` indicates that the window contents must be redrawn. This event is similar to an update event in the Classic event model. However, if you have the standard window handler installed, the Carbon Event Manager automatically calls `BeginUpdate` and `EndUpdate` for this event; all you need to do is draw in the window. To avoid nesting update calls, you should remove the duplicate `BeginUpdate` and `EndUpdate` calls in the `DoUpdate` function.
- `kEventWindowBoundsChanged` indicates that the window size has changed (by clicking the zoom button). The handler calls `DoUpdate` to redraw the content region to reflect the new size.
- `kEventWindowClickContentRgn` indicates that the user has clicked in the Traffic window. To toggle the light setting, the handler calls `DoContentClick` as before, but because Sample no longer receives update events, the handler also calls `DoUpdate` to draw the new setting. Similarly, because Sample cannot update the Traffic menu settings by calling `AdjustMenus` from the `WaitNextEvent` loop, the handler calls it here.
- `kEventCommandProcess` indicates that a menu-related command occurred. Because this is the window handler, the handler processes only cases related to the Traffic window (that is, the Red Light/Green Light items in the Traffic menu). Other commands will end up being handled by the application-level handler. To change the traffic light setting, the handler first isolates the menu item selected from the event parameter and then redraws the light (using the same calls as in the `kEventWindowClickContentRgn` case).

  If desired, instead of obtaining the menu item from the `commandStruct` structure, you can define and register constants for these items as you did for the About menu item.

Cleanup

After adapting Sample to use Carbon events, you no longer need the following functions:

- `EventLoop`: Replaced by the call to `RunApplicationEventLoop`.
- `DoEvent`: Replaced by the application and window event handlers.
- `MyGetGlobalMouse`: The application event handler now retrieves the mouse location from the event structure.
- `DoMenuCommand`: The handling of these events is split between the application and window event handlers.

New Carbon Functions

This section provides an overview of some of the new functions introduced in Carbon. Until complete documentation is available, you should refer to the header files and sample code included on the Mac OS X Developer Tools CD for additional information.

Custom Definition Procedures

Custom defprocs (that is, WDEFs, MDEFs, CDEFs, and LDEFs) must be compiled as PowerPC code and can no longer be stored in resources. Carbon introduces new variants of `CreateWindow` and similar functions (such as `NewControl` and `NewMenu`) that take a universal procedure pointer (UPP) to your custom defproc. Instead of creating a window definition as a WDEF resource, for example, you call the Carbon routine `CreateCustomWindow`:

```
OSStatus CreateCustomWindow(const WindowDefSpec *def,
        WindowClass windowClass, WindowAttributes attributes,
        const Rect *bounds, WindowPtr *outWindow);
```

The `WindowDefSpec` parameter contains a UPP that points to your custom window definition procedure.

Changes to WDEFs

You need to be aware of the following changes for custom WDEFs:

- Window defprocs no longer receive the `wCalcRgns` message. Instead they receive the `kWindowMsgGetRegion` message twice, once for the structure region and once for the content region. The structure passed in the message parameter indicates the desired region in each case. Your defproc must handle these messages.
- If you need to get the global bounds of a window's `portRect` in order to determine the structure or content regions, you should call `GetWindowBounds` and pass the `kWindowGlobalPortRgn` constant. On return, the function will supply a pointer to a `Rect` indicating the bounds in global coordinates. The pixel map (`PixMap`) bounds on Mac OS X will always start at (0,0), so you will obtain incorrect results if you attempt to manually convert the port's bounds from local to global coordinates by offsetting the bounds by the port's pixel map's bounds.

Changes to MDEFs

You need to be aware of the following changes to custom MDEFs:

- Menu defprocs no longer receive the `mChooseMsg` message. Instead they receive two new messages: `kMenuFindItemMsg` and `kMenuHiliteItemMsg`.

- Code that sets or reads the low memory global variables `TopMenuItem`, `AtMenuBottom`, `MenuDisable`, and `MBSaveLoc` should use the new `MenuTrackingData` structure instead. You can obtain the contents of the structure at any time by calling the new function `GetMenuTrackingData`.
- When a menu defproc receives a `mDrawMsg` message, it also receives a pointer to a `MenuTrackingData` structure in the `whichItem` parameter. Your defproc should read the structure to obtain the menu virtual top and bottom rather than using the low memory accessor functions `LMTopMenuItem` and `LMAtMenuBottom`.

Functions for Accessing Opaque Data Structures

A major change introduced in Carbon is that some commonly used data structures are now opaque—meaning their internal structure is hidden. Directly referencing fields within these structures is no longer allowed, and will cause a compiler error. QuickDraw global variables, graphics ports, regions, window and dialog records, controls, menus, and TSMTE dialogs are all opaque to Carbon applications. Anywhere you reference fields in these structures directly, you must use new casting and accessor functions described in the following sections.

Casting Functions

Many applications assume that window pointer (`WindowPtr`) and dialog pointer (`DialogPtr`) types have a graphics port (`GrafPort`) embedded at the top of their structures. In fact, the standard Universal Interfaces defines dialog pointers and window pointers as graphics pointers so that you don't have to cast them to a type `GrafPtr` before using them. For example:

```
void DrawIntoWindow(WindowPtr window)
{
    SetPort(window);
    MoveTo(x, y);
    LineTo(x + 50, y + 50);
}
```

If you compile the above code using the Carbon interfaces, you'll get a number of compilation errors due to the fact that window pointers are no longer defined as graphics pointers. But you can't simply cast these variables to type `GrafPtr` because doing so will cause your application to crash under Mac OS X.

Instead, Carbon provides a set of casting functions that allow you to obtain a pointer to a window's `GrafPort` structure or vice versa. Using these new functions, code like the previous example must be updated as follows to be Carbon-compliant and compile without errors:

```
void DrawIntoWindow(WindowPtr window)
{
    SetPort(GetWindowPort(window));
    MoveTo(x, y);
    LineTo(x + 50, y + 50);
}
```

Casting functions are provided for obtaining graphics ports from windows, windows from dialogs, and various other combinations. By convention, functions that cast up (that is, going from a lower-level data structure like a graphics port to a window or going from a window to a dialog pointer) are named `GetHigherLevelTypeFromLowerLevelType`. Functions that cast down are named `GetHigherLevelTypeLowerLevelType`.

Examples of functions that cast up include:

```
pascal DialogPtr GetDialogFromWindow(WindowPtr window);
pascal WindowPtr GetWindowFromPort(CGrafPtr port);
```

Functions that cast down include:

```
pascal WindowPtr GetDialogWindow(DialogPtr dialog);
pascal CGrafPtr GetWindowPort(WindowPtr window);
```

Accessor Functions

Carbon includes a number of functions to allow applications to access fields within system data structures that are now opaque. Listing A-1 (page 83) shows an example of some typical coding practices that must be modified for Carbon.

Listing A-1 Example of unsupported data structure access

```
void WalkWindowsAndDoSomething(WindowPtr firstWindow)
{
    WindowPtr currentWindow = firstWindow;

    while (currentWindow != NULL)
    {
        if ((WindowPeek) currentWindow->visible)
            && RectIsFourByFour(&currentWindow->portRect))
        {
            DoSomethingSpecial(currentWindow);
        }
        currentWindow = (WindowPtr) ((WindowPeek) currentWindow->nextWindow);
    }
}
```

There are four problems in Listing A-1 (page 83) that will cause compiler errors when building a Carbon application.

1. Checking the `visible` field directly is not allowed because the `WindowPeek` type is no longer defined (it's only useful when you can assume that type `WindowPtr` can be cast to a `WindowRecord` pointer, which is not the case in Carbon).

2. The `currentWindow` variable is treated as a graphics port. You need to use the casting functions discussed above to access a window's `GrafPort` structure.

3. Graphics ports are now opaque data structures, so you must use an accessor to get the port's bounding rectangle.

4. Accessing the `nextWindow` field directly from the `WindowRecord` structure is not allowed.

To compile and run under Carbon, the code above would have to be changed as shown in Listing A-2.

Listing A-2 Example of using Carbon-compatible accessor functions

```
void WalkWindowsAndDoSomething(WindowPtr firstWindow)
{
    WindowPtr currentWindow = firstWindow;

    while (currentWindow != NULL)
    {
        Rect windowBounds;

        if (IsWindowVisible(currentWindow) &&
            RectIsFourByFour(GetPortBounds(GetWindowPort(currentWindow),
                &windowBounds))
        {
            DoSomethingSpecial(currentWindow);
        }
        currentWindow = GetNextWindow(currentWindow);
    }
}
```

One thing to note is that the `GetPortBounds` function returns a pointer to the input rectangle as a syntactic convenience, to allow you to pass the result of `GetPortBounds` directly to another function. Many of the accessor functions return a pointer to the input in the same way, as a convenience to the caller.

With a few exceptions as noted below, all accessor functions return copies to data, not the data itself. You must make sure to allocate storage before you access non-scalar types such as regions and pixel patterns. For example, if you use code like this to test the visible region of a graphics port:

```
if (EmptyRgn(somePort->visRgn))
    DoSomething();
```

you'll have to change it as shown below in order to allow the accessor to copy the port's visible region into your reference:

```
RgnHandle visibleRegion;

visibleRegion = NewRgn();
if (EmptyRgn(GetPortVisibleRegion(somePort, visibleRegion)))
    DoSomething();
DisposeRgn(visibleRegion);
```

A few accessor functions continue to return actual data rather than copied data. `GetPortPixMap`, for example, is provided specifically to allow calls to `CopyBits`, `CopyMask`, and similar functions, and should only be used for these calls. The interface for the `CopyBits`-type calls will be changing to work around this exception, but for now be aware that this exception exists. The QuickDraw bottleneck routines, which are stored in a `GrafProc` record, continue to operate just like their classic Mac OS equivalents. That is, the actual pointer to the structure is returned rather than creating a copy. Other instances where the actual handle is passed back include cases where user-specified data is carried in a data structure, such as the `UserHandle` field in `ListHandle` records.

Table A-1 lists common accessor functions for Human Interface Toolbox structures.

Table A-1 Summary of Carbon Human Interface Toolbox accessors

Data structure	Element	Accessor
Controls		
`ControlRecord`	`nextControl`	Use Control Manager embedding hierarchy functions. (See Mac OS 8 Control Manager Reference.)
	`contrlOwner`	`Get/SetControlOwner`. May be replaced in favor of `Embed/DetachControl`.
	`contrlRect`	`Get/SetControlBounds`
	`contrlVis`	`IsControlVisible, SetControlVisibility`
	`contrlHilite`	`GetControlHilite, HiliteControl`
	`contrlValue`	`Get/SetControlValue, Get/SetControl32BitValue`
	`contrlMin`	`Get/SetControlMinimum, Get/SetControl32BitMinimum`
	`contrlMax`	`Get/SetControlMaximum, Get/SetControl32BitMaximum`
	`contrlDefProc`	not supported
	`contrlData`	`Get/SetControlDataHandle`
	`contrlAction`	`Get/SetControlAction`
	`contrlRfCon`	`Get/SetControlReference`
	`contrlTitle`	`Get/SetControlTitle`
`AuxCtlRec`	`acNext`	not supported
	`acOwner`	not supported
	`acCTable`	not supported
	`acFlags`	not supported
	`acReserved`	not supported
	`acRefCon`	Use `Get/SetControlProperty` if you need more `refCon`s.
`PopupPrivateData`	`mHandle`	Use `Get/SetControlData` with proper tags.
	`mID`	Use `Get/SetControlData` with proper tags.
Dialog Boxes		
`DialogRecord`	`window`	Use `GetDialogWindow` to obtain the value. There is no equivalent function for setting the value.

Data structure	Element	Accessor
	items	AppendDITL, ShortenDITL, AppendDialogItemList, InsertDialogItem, RemoveDialogItems
	textH	GetDialogTextEditHandle
	editField	GetDialogKeyboardFocusItem
	editOpen	Get/SetDialogCancelItem
	aDefItem	Get/SetDialogDefaultItem
Menus		
MenuInfo	menuID	Get/SetMenuID
	menuWidth	Get/SetMenuWidth
	menuHeight	Get/SetMenuHeight
	menuProc	SetMenuDefinition
	enableFlags	Enable/DisableMenuItem, IsMenuItemEnabled
	menuData	Get/SetMenuTitle
Windows		
WindowRecord CWindowRecord	port	Use GetWindowPort to obtain the value. There is no equivalent function for setting the value.
	windowKind	Get/SetWindowKind
	visible	Hide/ShowWindow, ShowHide, IsWindowVisible
	hilited	HiliteWindow, IsWindowHilited
	goAwayFlag	ChangeWindowAttributes
	spareFlag	ChangeWindowAttributes
	strucRgn	GetWindowRegion
	contRgn	GetWindowRegion
	updateRgn	GetWindowRegion
	windowDefProc	not supported
	dataHandle	not supported
	titleHandle	Get/SetWTitle
	titleWidth	GetWindowRegion

Data structure	Element	Accessor
	controlList	GetRootControl
	nextWindow	GetNextWindow
	windowPic	Get/SetWindowPic
	refCon	Get/SetWRefCon
AuxWinRec	awNext	not supported
	awOwner	not supported
	awCTable	Get/SetWindowContentColor
	reserved	not supported
	awFlags	not supported
	awReserved	not supported
	awRefCon	Use Get/SetWindowProperty if you need more reference constants.
Lists		
ListRec	rView	Get/SetListViewBounds
	port	Get/SetListPort
	indent	Get/SetListCellIndent
	cellSize	Get/SetListCellSize
	visible	Use GetListVisibileCells to obtain the value. No equivalent function for setting the value.
	vScroll	GetListVerticalScrollBar, use new API (TBD) to turn off automatic scroll bar drawing.
	hScroll	GetListHorizontalScrollBar, use new API (TBD) to turn off automatic scroll bar drawing.
	selFlags	Get/SetListSelectionFlags
	lActive	LActivate, GetListActive
	lReserved	not supported
	listFlags	Get/SetListFlags
	clikTime	Get/SetListClickTime
	clikLoc	GetListClickLocation

Data structure	Element	Accessor
	mouseLoc	GetListMouseLocation
	lClickLoop	Get/SetListClickLoop
	lastClick	SetListLastClick
	refCon	Get/SetListRefCon
	listDefProc	not supported
	userHandle	Get/SetListUserHandle
	dataBounds	GetListDataBounds
	cells	LGet/SetCell
	maxIndex	LGet/SetCell
	cellArray	LGet/SetCell

Table A-2 provides a summary of accessor functions you can use to access common QuickDraw data structures.

Table A-2 QuickDraw accessor functions

Data structure	Element	Accessor
GrafPort	device	not supported
	portBits	Use GetPortBitMapsForCopyBits or IsPortColor.
	portRect	Get/SetPortBounds
	visRgn	Get/SetPortVisibleRegion
	clipRgn	Get/SetPortClipRgn
	bkPat	not supported
	fillPat	not supported
	pnLoc	Get/SetPortPenLocation
	pnSize	Get/SetPortPenSize
	pnMode	Get/SetPortPenMode
	pnPat	not supported
	pnVis	Use GetPortPenVisibility or Show/HidePen.
	txFont	Use GetPortTextFont or TextFont.

Data structure	Element	Accessor
	txFace	Use GetPortTextFace or TextFace.
	txMode	Use GetPortTextMode or TextMode.
	txSize	Use GetPortTextSize or TextSize.
	spExtra	Use GetPortSpExtra or SpaceExtra.
	fgColor	not supported
	bkColor	not supported
	colrBit	not supported
	patStretch	not supported
	picSave	IsPortPictureBeingDefined
	rgnSave	not supported
	polySave	not supported
	grafProcs	not supported
CGrafPort	device	not supported
	portPixMap	GetPortPixMap
	portVersion	IsPortColor
	grafVars	not supported
	chExtra	GetPortChExtra
	pnLocHFrac	Get/SetPortFracHPenLocation
	portRect	Get/SetPortBounds
	visRgn	Get/SetPortVisibleRegion
	clipRgn	Get/SetPortClipRegion
	bkPixPat	Use GetPortBackPixPat or BackPixPat.
	rgbFgColor	Use GetPortForeColor or RGBForeColor.
	rgbBkColor	Use GetPortBackColor or RGBBackColor.
	pnLoc	Get/SetPortPenLocation
	pnSize	Get/SetPortPenSize
	pnMode	Get/SetPortPenMode

Data structure	Element	Accessor
pnPixPat		Get/SetPortPenPixPat
	fillPixPat	Get/SetPortFillPixPat
	pnVis	Use GetPortPenVisibility or Show/HidePen.
	txFont	Use GetPortTextFont or TextFont.
	txFace	Use GetPortTextFace or TextFace.
	txMode	Use GetPortTextMode or TextMode.
	txSize	Use GetPortTextSize or TextSize.
	spExtra	Use GetPortSpExtra or SpaceExtra.
	fgColor	not supported
	bkColor	not supported
	colrBit	not supported
	patStretch	not supported
	picSave	IsPortPictureBeingDefined
	rgnSave	not supported
	polySave	not supported
	grafProcs	Get/SetPortGrafProcs
QDGlobals	randSeed	GetQDGlobalsRandomSeed
	screenBits	GetQDGlobalsScreenBits
	arrow	GetQDGlobalsArrow
	dkGray	GetQDGlobalsDarkGray
	ltGray	GetQDGlobalsLightGray
	gray	GetQDGlobalsGray
	black	GetQDGlobalsBlack
	white	GetQDGlobalsWhite
GrafPtr	thePort	GetQDGlobalsThePort

Utility Functions

Carbon includes a number of utility functions to make it easier to port your application. Under the classic Mac OS API, new graphics ports were created by allocating non-relocatable memory the size of a `CGrafPort` record and calling `OpenCPort`. Because `GrafPort` records are now opaque, and their size is system-defined, Carbon includes new routines to create and dispose of graphics ports:

```
pascal CGrafPtr CreateNewPort()
pascal void DisposePort(CGrafPtr port)
```

These functions provide access to commonly used bounding rectangles:

```
pascal OSStatus GetWindowBounds(WindowRef window,
                WindowRegionCode regionCode, Rect *bounds);
pascal OSStatus GetWindowRegion(WindowRef window,
                WindowRegionCode regionCode, RgnHandle windowRegion);
```

Often you'll find the need to set the current port to the one that belongs to a window or dialog box. `SetPortWindowPort` and `SetPortDialogPort` allow you to do this:

```
pascal void SetPortWindowPort(WindowPtr window)
pascal void SetPortDialogPort(DialogPtr dialog)
```

The new function `GetParamText` replaces `LMGetDAStrings` as the method to retrieve the current `ParamText` setting. Pass `NULL` for a parameter if you don't want a particular string.

```
pascal void GetParamText(StringPtr param0, StringPtr param1,
                         StringPtr param2, StringPtr param3)
```

Functions in CarbonAccessors.o

`CarbonAccessors.o` is a static library that contains implementations of the Carbon functions for accessing opaque toolbox data structures. See "Begin With CarbonAccessors.o" (page 21) for information on how you can use this library to assist in porting your code to Carbon.

Note: You can also use `CarbonAccessors.o` to maintain some backwards compatibility with non-Carbon systems. For example, if you don't require functions that are available only in CarbonLib, by linking against the `CarbonAccessors.o` static library you can build an application from a Carbon-compliant code base that runs on non-Carbon systems.

Table A-3 lists the Carbon functions implemented in `CarbonAccessors.o`. The "•" symbol indicates a function added since the Developer Preview 3 version of this document. "••" indicates a function added since Developer Preview 4.

Table A-3 Functions in CarbonAccessors.o

`AEFlattenDesc`••	`AEGetDescData`
`AEGetDescDataSize`	`AEReplaceDescData`•
`AESizeOfFlattenedDesc`••	`AEUnflattenDesc`••

c2pstrcpy•	CopyCStringToPascal•
CopyPascalStringToC•	CreateNewPort
DisposePort	GetControlBounds
GetControlDataHandle	GetControlHilite
GetControlOwner	GetControlPopupMenuHandle
GetControlPopupMenuID	GetDialogCancelItem
GetDialogDefaultItem	GetDialogFromWindow
GetDialogKeyboardFocusItem	GetDialogPort
GetDialogTextEditHandle	GetDialogWindow
GetGlobalMouse	GetListActive
GetListCellIndent	GetListCellSize
GetListClickLocation	GetListClickLoop
GetListClickTime	GetListDataBounds
GetListDataHandle	GetListDefinition
GetListFlags	GetListHorizontalScrollBar
GetListMouseLocation	GetListPort
GetListRefCon	GetListSelectionFlags
GetListUserHandle	GetListVerticalScrollBar
GetListViewBounds	GetListVisibleCells
GetMenuHeight	GetMenuID
GetMenuTitle	GetMenuWidth
GetNextWindow•	GetParamText
GetPixBounds	GetPixDepth
GetPortBackColor	GetPortBackPixPat
GetPortBackPixPatDirect	GetPortBitMapForCopyBits•
GetPortBounds	GetPortChExtra
GetPortClipRegion	GetPortFillPixPat
GetPortForeColor	GetPortFracHPenLocation

GetPortGrafProcs	GetPortHiliteColor
GetPortOpColor	GetPortPenLocation
GetPortPenMode	GetPortPenPixPat
GetPortPenPixPatDirect	GetPortPenSize
GetPortPenVisibility	GetPortPixMap
GetPortPrintingReference	GetPortSpExtra
GetPortTextFace	GetPortTextFont
GetPortTextMode	GetPortTextSize
GetPortVisibleRegion	GetQDGlobals
GetQDGlobalsArrow	GetQDGlobalsBlack
GetQDGlobalsDarkGray	GetQDGlobalsGray
GetQDGlobalsLightGray	GetQDGlobalsRandomSeed
GetQDGlobalsScreenBits	GetQDGlobalsThePort
GetQDGlobalsWhite	GetRegionBounds
GetTSMDialogDocumentID	GetTSMTEDialogTSMTERecHandle•
GetWindowFromPort	GetWindowKind
GetWindowList•	GetWindowPort
GetWindowPortBounds	GetWindowSpareFlag
GetWindowStandardState	GetWindowUserState
InvalWindowRect	InvalWindowRgn
IsControlHilited	IsPortColor•
IsPortOffscreen	IsPortPictureBeingDefined
IsPortRegionBeingDefined	IsRegionRectangular
IsTSMTEDialog•	IsWindowHilited
IsWindowUpdatePending	IsWindowVisible
p2cstrcpy•	SetControlBounds
SetControlDataHandle	SetControlOwner
SetControlPopupMenuHandle	SetControlPopupMenuID

SetListCellIndent	SetListClickLoop
SetListClickTime	SetListFlags
SetListLastClick	SetListPort
SetListRefCon	SetListSelectionFlags
SetListUserHandle	SetListViewBounds
SetMenuHeight	SetMenuID
SetMenuTitle	SetMenuWidth
SetPortBackPixPat	SetPortBackPixPatDirect
SetPortBounds	SetPortClipRegion
SetPortDialogPort	SetPortFracHPenLocation
SetPortGrafProcs	SetPortOpColor
SetPortPenMode	SetPortPenPixPat
SetPortPenPixPatDirect	SetPortPenSize
SetPortPrintingReference	SetPortVisibleRegion
SetPortWindowPort	SetQDError•
SetQDGlobalsArrow	SetQDGlobalsRandomSeed
SetTSMDialogDocumentID	SetTSMTEDialogTSMTERecHandle•
SetWindowKind	SetWindowStandardState
SetWindowUserState	ValidWindowRect
ValidWindowRgn	

The following functions were removed from `CarbonAccessors.o`.

Table A-4 Functions removed from CarbonAccessors.o

DisableMenuItem	EnableMenuItem
GetControlColorTable	GetControlDefinition
GetTSMDialogPtr	GetTSMDialogTextEditHandle
GetWindowGoAwayFlag	SetControlColorTable

Debugging Functions

The following functions have been added to `MacMemory.h` to aid in debugging.

CheckAllHeaps

```
pascal Boolean CheckAllHeaps(void);
```

Checks all applicable heaps for validity. Returns `false` if there is any corruption.

IsHeapValid

```
pascal Boolean IsHeapValid(void);
```

Similar to `CheckAllHeaps`, but checks only the application heap for validity.

IsHandleValid

```
pascal Boolean IsHandleValid(Handle h);
```

Returns `true` if the specified handle is valid. You cannot pass `NULL` or an empty handle to `IsHandleValid`.

IsPointerValid

```
pascal Boolean IsPointerValid(Ptr p);
```

Returns `true` if the specified pointer is valid. You cannot pass `NULL` or an empty pointer to `IsPointerValid`.

Resource Chain Manipulation Functions

Three functions have been added to `Resources.h` to facilitate resource chain manipulation in Carbon applications.

InsertResourceFile

```
OSErr InsertResourceFile(SInt16 refNum, RsrcChainLocation where);
```

If the file is already in the resource chain, it is removed and re-inserted at the location specified by the `where` parameter. If the file has been detached, it is added to the resource chain at the specified location. Returns `resFNotFound` if the file is not currently open. Valid constants for the `where` parameter are:

```
// RsrcChainLocation constants for InsertResourceFile
enum short
```

```
{
    kRsrcChainBelowAll = 0,          /* Below all other app files in
                                         the resource chain */
    kRsrcChainBelowApplicationMap = 1, /* Below the application's
                                         resource map */
    kRsrcChainAboveApplicationMap = 2  /* Above the application's
                                         resource map */
};
```

DetachResourceFile

```
OSErr DetachResourceFile(SInt16 refNum);
```

If the file is not currently in the resource chain, this function returns `resNotFound`. Otherwise, the resource file is removed from the resource chain.

FSpResourceFileAlreadyOpen

```
Boolean FSpResourceFileAlreadyOpen (
                        const FSSpec *resourceFile,
                        Boolean *inChain,
                        SInt16 *refNum);
```

This function returns `true` if the resource file is already open and known by the Resource Manager (that is, if the file is either in the current resource chain or if it's a detached resource file). If the file is in the resource chain, the `inChain` parameter is set to `true` on exit and the function returns `true`. If the file is open but currently detached, `inChain` is set to `false` and the function returns `true`. If the file is open, the `refNum` to the file is returned.

The Sample Application

The main code for Sample is included in `Sample.c` and `SampleInit.c` as shown in Listing B-1 and Listing B-2 (page 108). Sample also includes a definition file, `Sample.h`, and a compiled resource file, `TCSample.rsrc`.

The chapter "A Porting Example" (page 47) describes how to port the Sample application to Carbon. "An Example: Adding Carbon Events to Sample" (page 71) describes how to add Carbon events to the Carbon version of Sample.

Listing B-1 Sample.c

```
/*
    File:       Sample.c

    Contains:   Sample is an example application that demonstrates how to
                initialize the commonly used toolbox managers, operate
                successfully under MultiFinder, handle desk accessories,
                and create, grow, and zoom windows.

                It does not by any means demonstrate all the techniques
                you need for a large application. In particular, Sample
                does not cover exception handling, multiple windows/documents,
                sophisticated memory management, printing, or undo. All of
                these are vital parts of a normal full-sized application.

                This application is an example of the form of a Macintosh
                application; it is NOT a template. It is NOT intended to be
                used as a foundation for the next world-class, best-selling,
                600K application. A stick figure drawing of the human body may
                be a good example of the form for a painting, but that does not
                mean it should be used as the basis for the next Mona Lisa.

                We recommend that you review this program or TESample before
                beginning a new application.

    Written by:

    Copyright:  Copyright © 1988-1999 by Apple Computer, Inc., All Rights Reserved.

    You may incorporate this Apple sample source code into your program(s) without
    restriction. This Apple sample source code has been provided "AS IS" and the
    responsibility for its operation is yours. You are not permitted to redistribute
    this Apple sample source code as "Apple sample source code" after having made
    changes. If you're going to re-distribute the source, we require that you make
    it clear in the source that the code was descended from Apple sample source
    code, but that you've made changes.

    Change History (most recent first):
    8/13/1999   Karl Groethe    Updated for Metrowerks Codewarror Pro 2.1
*/
```

```
/* Segmentation strategy:

   This program consists of three segments.
   1. "Main" contains most of the code, including the MPW libraries, and the
      main program.  This segment is in the file Sample.c
   2. "Initialize" contains code that is only used once, during startup, and
      can be unloaded after the program starts.  This segment is in the file
      SampleInit.c.
   3. "%A5Init" is automatically created by the Linker to initialize globals
      for the MPW libraries and is unloaded right away. */


/* SetPort strategy:

   Toolbox routines do not change the current port. In spite of this, in this
   program we use a strategy of calling SetPort whenever we want to draw or
   make calls which depend on the current port. This makes us less vulnerable
   to bugs in other software which might alter the current port (such as the
   bug (feature?) in many desk accessories which change the port on OpenDeskAcc).
   Hopefully, this also makes the routines from this program more self-contained,
   since they don't depend on the current port setting. */

#pragma segment Main

#include <Limits.h>
#include <Types.h>
#include <Resources.h>
#include <QuickDraw.h>
#include <Fonts.h>
#include <Events.h>
#include <Windows.h>
#include <Menus.h>
#include <TextEdit.h>
#include <Dialogs.h>
#include <Desk.h>
#include <ToolUtils.h>
#include <Memory.h>
#include <SegLoad.h>
#include <Files.h>
#include <OSUtils.h>
#include <DiskInit.h>
#include <Packages.h>
#include <Traps.h>
#include "Sample.h      "/* bring in all the #defines for Sample */

/* The "g" prefix is used to emphasize that a variable is global. */

/* GMac is used to hold the result of a SysEnvirons call. This makes
   it convenient for any routine to check the environment. */
SysEnvRec   gMac;               /* set up by Initialize */

/* GHasWaitNextEvent is set at startup, and tells whether the WaitNextEvent
   trap is available. If it is false, we know that we must call GetNextEvent. */
Boolean     gHasWaitNextEvent;  /* set up by Initialize */

/* GInBackground is maintained by our osEvent handling routines. Any part of
   the program can check it to find out if it is currently in the background. */
```

```
Boolean     gInBackground;      /* maintained by Initialize and DoEvent */


/* The following globals are the state of the window. If we supported more than
   one window, they would be attached to each document, rather than globals. */

/* GStopped tells whether the stop light is currently on stop or go. */
Boolean     gStopped;           /* maintained by Initialize and SetLight */

/* GStopRect and gGoRect are the rectangles of the two stop lights in the window. */
Rect        gStopRect;          /* set up by Initialize */
Rect        gGoRect;            /* set up by Initialize */


/* Define TopLeft and BotRight macros for convenience. Notice the implicit
   dependency on the ordering of fields within a Rect */
#define TopLeft(aRect)  (* (Point *) &(aRect).top)
#define BotRight(aRect) (* (Point *) &(aRect).bottom)


/* This routine is part of the MPW runtime library. This external
   reference to it is done so that we can unload its segment, %A5Init. */

#ifndef THINK_C
  extern void _DataInit();
#endif


void main()
{
#ifndef THINK_C
    UnloadSeg((Ptr) _DataInit);     /* note that _DataInit must not be in Main! */
#endif

    /* 1.01 - call to ForceEnvirons removed */

    /*  If you have stack requirements that differ from the default,
        then you could use SetApplLimit to increase StackSpace at
        this point, before calling MaxApplZone. */
    MaxApplZone();          /* expand the heap so code segments load at the top */

    Initialize();                   /* initialize the program */
    UnloadSeg((Ptr) Initialize);    /* note that Initialize must not be in Main! */

    EventLoop();                    /* call the main event loop */
} /*main*/


/*  Get events forever, and handle them by calling DoEvent.
    Get the events by calling WaitNextEvent, if it's available, otherwise
    by calling GetNextEvent. Also call AdjustCursor each time through the loop. */

void EventLoop()
{
    RgnHandle   cursorRgn;
    Boolean     gotEvent;
    EventRecord event;
    Point       mouse;
    cursorRgn = NewRgn();   /* we'll pass WNE an empty region the 1st time thru */
    do {
        /* use WNE if it is available */
        if ( gHasWaitNextEvent ) {
            GetGlobalMouse(&mouse);
            AdjustCursor(mouse, cursorRgn);
            gotEvent = WaitNextEvent(everyEvent, &event, LONG_MAX, cursorRgn);
        }
        else {
            SystemTask();
            gotEvent = GetNextEvent(everyEvent, &event);
        }
        if ( gotEvent ) {
            /* make sure we have the right cursor before handling the event */
            AdjustCursor(event.where, cursorRgn);
            DoEvent(&event);
        }
        /*  If you are using modeless dialogs that have editText items,
            you will want to call IsDialogEvent to give the caret a chance
            to blink, even if WNE/GNE returned FALSE. However, check FrontWindow
            for a non-NIL value before calling IsDialogEvent. */
    } while ( true );   /* loop forever; we quit via ExitToShell */
} /*EventLoop*/

/* Do the right thing for an event. Determine what kind of event it is, and call
 the appropriate routines. */

void DoEvent(EventRecord *event)
{
    short       part, err;
    WindowPtr   window;
    Boolean     hit;
    char        key;
    Point       aPoint;

    switch ( event->what ) {
        case mouseDown:
            part = FindWindow(event->where, &window);
            switch ( part ) {
                case inMenuBar:         /* process a mouse menu command (if any) */
                    AdjustMenus();
                    DoMenuCommand(MenuSelect(event->where));
                    break;
                case inSysWindow:       /* let the system handle the mouseDown */
                    SystemClick(event, window);
                    break;
                case inContent:
                    if ( window != FrontWindow() ) {
                        SelectWindow(window);
                        /*DoEvent(event);*//* use this line for "do first click" */
                    } else
                        DoContentClick(window);
                    break;
                case inDrag:    /* pass screenBits.bounds to get all gDevices */
                    DragWindow(window, event->where, &qd.screenBits.bounds);
                    break;
                case inGrow:
```

```
                break;
            case inZoomIn:
            case inZoomOut:
                hit = TrackBox(window, event->where, part);
                if ( hit ) {
                    SetPort(window); /* the window must be the current port... */
                    EraseRect(&window->portRect);   /* because of a bug in */
                                                    /* ZoomWindow */
                    ZoomWindow(window, part, true); /* note that we invalidate */
                                                    /* and erase... */
                    InvalRect(&window->portRect);   /* to make things look */
                                                    /* better on-screen */
                }
                break;
        }
        break;
    case keyDown:
    case autoKey:                       /* check for menukey equivalents */
        key = event->message & charCodeMask;
        if ( event->modifiers & cmdKey )            /* Command key down */
            if ( event->what == keyDown ) {
                AdjustMenus();      /* enable/disable/check menu items properly */
                DoMenuCommand(MenuKey(key));
            }
        break;
    case activateEvt:
        DoActivate((WindowPtr) event->message,
                    (event->modifiers & activeFlag) != 0);
        break;
    case updateEvt:
        DoUpdate((WindowPtr) event->message);
        break;
    /*  1.01 - It is not a bad idea to at least call DIBadMount in response
        to a diskEvt, so that the user can format a floppy. */
    case diskEvt:
        if ( HiWord(event->message) != noErr ) {
            SetPt(&aPoint, kDILeft, kDITop);
            err = DIBadMount(aPoint, event->message);
        }
        break;
    case kOSEvent:
    /*  1.02 - must BitAND with 0x0FF to get only low byte */
        switch ((event->message >> 24) & 0x0FF) {       /* high byte of message */
            case kSuspendResumeMessage: /* suspend/resume is also an */
                                        /* activate/deactivate */
                gInBackground = (event->message & kResumeMask) == 0;
                DoActivate(FrontWindow(), !gInBackground);
                break;
        }
        break;
    }
} /*DoEvent*/


/*  Change the cursor's shape, depending on its position. This also calculates the region
    where the current cursor resides (for WaitNextEvent). If the mouse is ever outside
 of
    that region, an event would be generated, causing this routine to be called,
    allowing us to change the region to the region the mouse is currently in. If
    there is more to the event than just "the mouse moved", we get called before the
    event is processed to make sure the cursor is the right one. In any (ahem) event,
    this is called again before we fall back into WNE. */

void AdjustCursor(Point mouse, RgnHandle region)
{
    WindowPtr   window;
    RgnHandle   arrowRgn;
    RgnHandle   plusRgn;
    Rect        globalPortRect;

    window = FrontWindow(); /* we only adjust the cursor when we are in front */
    if ( (! gInBackground) && (! IsDAWindow(window)) ) {
        /* calculate regions for different cursor shapes */
        arrowRgn = NewRgn();
        plusRgn = NewRgn();

        /* start with a big, big rectangular region */
        SetRectRgn(arrowRgn, kExtremeNeg, kExtremeNeg, kExtremePos, kExtremePos);

        /* calculate plusRgn */
        if ( IsAppWindow(window) ) {
            SetPort(window);    /* make a global version of the viewRect */
            SetOrigin(-window->portBits.bounds.left, -window->portBits.bounds.top);
            globalPortRect = window->portRect;
            RectRgn(plusRgn, &globalPortRect);
            SectRgn(plusRgn, window->visRgn, plusRgn);
            SetOrigin(0, 0);
        }

        /* subtract other regions from arrowRgn */
        DiffRgn(arrowRgn, plusRgn, arrowRgn);

        /* change the cursor and the region parameter */
        if ( PtInRgn(mouse, plusRgn) ) {
            SetCursor(*GetCursor(plusCursor));
            CopyRgn(plusRgn, region);
        } else {
            SetCursor(&qd.arrow);
            CopyRgn(arrowRgn, region);
        }

        /* get rid of our local regions */
        DisposeRgn(arrowRgn);
        DisposeRgn(plusRgn);
    }
} /*AdjustCursor*/


/*  Get the global coordinates of the mouse. When you call OSEventAvail
    it will return either a pending event or a null event. In either case,
    the where field of the event record will contain the current position
    of the mouse in global coordinates and the modifiers field will reflect
    the current state of the modifiers. Another way to get the global
    coordinates is to call GetMouse and LocalToGlobal, but that requires
    being sure that thePort is set to a valid port. */
```

```
void GetGlobalMouse(Point *mouse)
{
    EventRecord event;

    OSEventAvail(kNoEvents, &event);    /* we aren't interested in any events */
    *mouse = event.where;               /* just the mouse position */
} /*GetGlobalMouse*/


/*  This is called when an update event is received for a window.
    It calls DrawWindow to draw the contents of an application window.
    As an efficiency measure that does not have to be followed, it
    calls the drawing routine only if the visRgn is non-empty. This
    will handle situations where calculations for drawing or drawing
    itself is very time-consuming. */

void DoUpdate(WindowPtr window)
{
    if ( IsAppWindow(window) ) {
        BeginUpdate(window);                /* this sets up the visRgn */
        if ( ! EmptyRgn(window->visRgn) )   /* draw if updating needs to be done */
            DrawWindow(window);
        EndUpdate(window);
    }
} /*DoUpdate*/


/*  This is called when a window is activated or deactivated.
    In Sample, the Window Manager's handling of activate and
    deactivate events is sufficient. Other applications may have
    TextEdit records, controls, lists, etc., to activate/deactivate. */

void DoActivate(WindowPtr window, Boolean becomingActive)
{
    if ( IsAppWindow(window) ) {
        if ( becomingActive )
            /* do whatever you need to at activation */ ;
        else
            /* do whatever you need to at deactivation */ ;
    }
} /*DoActivate*/


/*  This is called when a mouse-down event occurs in the content of a window.
    Other applications might want to call FindControl, TEClick, etc., to
    further process the click. */

void DoContentClick(WindowPtr window)
{
    SetLight(window, ! gStopped);
} /*DoContentClick*/


/* Draw the contents of the application window. We do some drawing in color, using
   Classic QuickDraw's color capabilities. This will be black and white on old
   machines, but color on color machines. At this point, the window’s visRgn
   is set to allow drawing only where it needs to be done. */
```

```
void DrawWindow(WindowPtr window)
{
    SetPort(window);

    EraseRect(&window->portRect);   /* clear out any garbage that may linger */
    if ( gStopped )                 /* draw a red (or white) stop light */
        ForeColor(redColor);
    else
        ForeColor(whiteColor);

    PaintOval(&gStopRect);
    ForeColor(blackColor);
    FrameOval(&gStopRect);

    if ( ! gStopped )               /* draw a green (or white) go light */
        ForeColor(greenColor);
    else
        ForeColor(whiteColor);

    PaintOval(&gGoRect);
    ForeColor(blackColor);
    FrameOval(&gGoRect);
} /*DrawWindow*/


/*  Enable and disable menus based on the current state.
    The user can only select enabled menu items. We set up all the menu items
    before calling MenuSelect or MenuKey, since these are the only times that
    a menu item can be selected. Note that MenuSelect is also the only time
    the user will see menu items. This approach to deciding what enable/
    disable state a menu item has the advantage of concentrating all
    the decision-making in one routine, as opposed to being spread throughout
    the application. Other application designs may take a different approach
    that is just as valid. */

void AdjustMenus()
{
    WindowPtr   window;
    MenuHandle  menu;

    window = FrontWindow();

    menu = GetMenuHandle(mFile);
    if ( IsDAWindow(window) )       /* we can allow desk accessories to be */
                                    /* closed from the menu */
        EnableItem(menu, iClose);
    else
        DisableItem(menu, iClose);  /* but not our traffic light window */

    menu = GetMenuHandle(mEdit);
    if ( IsDAWindow(window) ) {     /* a desk accessory might need the edit menu… */
        EnableItem(menu, iUndo);
        EnableItem(menu, iCut);
        EnableItem(menu, iCopy);
        EnableItem(menu, iClear);
        EnableItem(menu, iPaste);
    } else {                        /* …but we don’t use it */
        DisableItem(menu, iUndo);
        DisableItem(menu, iCut);
        DisableItem(menu, iCopy);
        DisableItem(menu, iClear);
        DisableItem(menu, iPaste);
    }

    menu = GetMenuHandle(mLight);
    if ( IsAppWindow(window) ) {    /* we know that it must be the traffic light */
        EnableItem(menu, iStop);
        EnableItem(menu, iGo);
    } else {
        DisableItem(menu, iStop);
        DisableItem(menu, iGo);
    }
    CheckItem(menu, iStop, gStopped); /* we can also determine the check/uncheck */
                                     /* state,too */
    CheckItem(menu, iGo, ! gStopped);
} /*AdjustMenus*/


/*  This is called when an item is chosen from the menu bar (after calling
    MenuSelect or MenuKey). It performs the right operation for each command.
    It is good to have both the result of MenuSelect and MenuKey go to
    one routine like this to keep everything organized. */

void DoMenuCommand(long menuResult)
{
    short       menuID;             /* the resource ID of the selected menu */
    short       menuItem;           /* the item number of the selected menu */
    short       itemHit;
    Str255      daName;
    short       daRefNum;
    Boolean     handledByDA;

    menuID = HiWord(menuResult);    /* use macros for efficiency to... */
    menuItem = LoWord(menuResult);  /* get menu item number and menu number */
    switch ( menuID ) {
        case mApple:
            switch ( menuItem ) {
                case iAbout:        /* bring up alert for About */
                    itemHit = Alert(rAboutAlert, nil);
                    break;
                default:            /* all non-About items in this menu are DAs */
                    /* type Str255 is an array in MPW 3 */
                    GetMenuItemText(GetMenuHandle(mApple), menuItem, daName);
                    daRefNum = OpenDeskAcc(daName);
                    break;
            }
            break;
        case mFile:
            switch ( menuItem ) {
                case iClose:
                    DoCloseWindow(FrontWindow());
                    break;
                case iQuit:
                    Terminate();
                    break;
            }
            break;
        case mEdit:                 /* call SystemEdit for DA editing & MultiFinder */
            handledByDA = SystemEdit(menuItem-1);/* since we don't do any Editing */
            break;
        case mLight:
            switch ( menuItem ) {
                case iStop:
                    SetLight(FrontWindow(), true);
                    break;
                case iGo:
                    SetLight(FrontWindow(), false);
                    break;
            }
            break;
    }
    HiliteMenu(0);              /* unhighlight what MenuSelect (or MenuKey) hilited */
} /*DoMenuCommand*/


/* Change the setting of the light. */

void SetLight(WindowPtr window, Boolean newStopped)
{
    if ( newStopped != gStopped ) {
        gStopped = newStopped;
        SetPort(window);
        InvalRect(&window->portRect);
    }
} /*SetLight*/


/* Close a window. This handles desk accessory and application windows. */

/*  1.01 - At this point, if there was a document associated with a
    window, you could do any document saving processing if it is 'dirty'.
    DoCloseWindow would return true if the window actually closed, i.e.,
    the user didn't cancel from a save dialog. This result is handy when
    the user quits an application, but then cancels the save of a document
    associated with a window. */

Boolean DoCloseWindow(WindowPtr window)
{
    if ( IsDAWindow(window) )
        CloseDeskAcc(((WindowPeek) window)->windowKind);
    else if ( IsAppWindow(window) )
        CloseWindow(window);
    return true;
} /*DoCloseWindow*/


/**************************************************************************************
*** 1.01 DoCloseBehind(window) was removed ***

    1.01 - DoCloseBehind was a good idea for closing windows when quitting
    and not having to worry about updating the windows, but it suffered
    from a fatal flaw. If a desk accessory owned two windows, it would
    close both those windows when CloseDeskAcc was called. When DoCloseBehind
    got around to calling DoCloseWindow for that other window that was already
    closed, things would go very poorly. Another option would be to have a
    procedure, GetRearWindow, that would go through the window list and return
    the last window. Instead, we decided to present the standard approach
    of getting and closing FrontWindow until FrontWindow returns NIL. This
    has a potential benefit in that the window whose document needs to be saved
    may be visible since it is the front window, therefore decreasing the
    chance of user confusion. For aesthetic reasons, the windows in the
    application should be checked for updates periodically and have the
    updates serviced.
**************************************************************************************/


/* Clean up the application and exit. We close all of the windows so that
 they can update their documents, if any. */

/*  1.01 - If we find out that a cancel has occurred, we won't exit to the */
/*          shell, but will return instead. */

void Terminate()
{
    WindowPtr   aWindow;
    Boolean     closed;

    closed = true;
    do {
        aWindow = FrontWindow();                /* get the current front window */
        if (aWindow != nil)
            closed = DoCloseWindow(aWindow);    /* close this window */
    }
    while (closed && (aWindow != nil));
    if (closed)
        ExitToShell();                          /* exit if no cancellation */
} /*Terminate*/


/*  Check to see if a window belongs to the application. If the window pointer
    passed was NIL, then it could not be an application window. WindowKinds
    that are negative belong to the system and windowKinds less than userKind
    are reserved by Apple except for windowKinds equal to dialogKind, which
    mean it is a dialog.
    1.02 - In order to reduce the chance of accidentally treating some window
    as an AppWindow that shouldn't be, we'll only return true if the windowkind
    is userKind. If you add different kinds of windows to Sample you'll need
    to change how this all works. */

Boolean IsAppWindow(WindowPtr window)
{
    short       windowKind;

    if ( window == nil )
        return false;
    else {  /* application windows have windowKinds = userKind (8) */
        windowKind = ((WindowPeek) window)->windowKind;
        return ( windowKind == userKind );
    }
} /*IsAppWindow*/
```

```
/* Check to see if a window belongs to a desk accessory. */

Boolean IsDAWindow(WindowPtr window)
{
    if ( window == nil )
        return false;
    else    /* DA windows have negative windowKinds */
        return ( ((WindowPeek) window)->windowKind < 0 );
} /*IsDAWindow*/


/*  Display an alert that tells the user an error occurred, then exit the program.
    This routine is used as an ultimate bail-out for serious errors that prohibit
    the continuation of the application. Errors that do not require the termination
    of the application should be handled in a different manner. Error checking and
    reporting has a place even in the simplest application. The error number is used
    to index an 'STR#' resource so that a relevant message can be displayed. */

void AlertUser()
{
    short       itemHit;

    SetCursor(&qd.arrow);
    itemHit = Alert(rUserAlert, nil);
    ExitToShell();
} /* AlertUser */
```

Listing B-2 SampleInit.c

```
/* File:            SampleInit.c */

/* Repeated comments from Sample.c removed */

#pragma segment Initialize

#include <Limits.h>
#include <Types.h>
#include <Resources.h>
#include <QuickDraw.h>
#include <Fonts.h>
#include <Events.h>
#include <Windows.h>
#include <Menus.h>
#include <TextEdit.h>
#include <Dialogs.h>
#include <Desk.h>
#include <ToolUtils.h>
#include <Memory.h>
#include <SegLoad.h>
#include <Files.h>
#include <OSUtils.h>
#include <DiskInit.h>
#include <Packages.h>
#include <Traps.h>
#include <OSUtils.h>
#include "Sample.h      "/* bring in all the #defines for Sample */
```

```
/* The "g" prefix is used to emphasize that a variable is global. */
/* All are extern since the variables are declared in the main segment. */

/* GMac is used to hold the result of a SysEnvirons call. This makes
   it convenient for any routine to check the environment. */
extern SysEnvRec    gMac;               /* set up by Initialize */

/* GHasWaitNextEvent is set at startup, and tells whether the WaitNextEvent
   trap is available. If it is false, we know that we must call GetNextEvent. */
extern Boolean      gHasWaitNextEvent;  /* set up by Initialize */

/* GInBackground is maintained by our osEvent handling routines. Any part of
   the program can check it to find out if it is currently in the background. */
extern Boolean      gInBackground;      /* maintained by Initialize and DoEvent */


/* The following globals are the state of the window. If we supported more than
   one window, they would be attached to each document, rather than globals. */

/* GStopped tells whether the stop light is currently on stop or go. */
extern Boolean      gStopped;           /* maintained by Initialize and SetLight */

/* GStopRect and gGoRect are the rectangles of the two stop lights in the window. */
extern Rect     gStopRect;          /* set up by Initialize */
extern Rect     gGoRect;            /* set up by Initialize */


/*  Set up the whole world, including global variables, Toolbox managers,
    and menus. We also create our one application window at this time.
    Since window storage is non-relocateable, how and when to allocate space
    for windows is very important so that heap fragmentation does not occur.
    Because Sample has only one window and it is only disposed when the application
    quits, we will allocate its space here, before anything that might be a locked
    relocatable object gets into the heap. This way, we can force the storage to be
    in the lowest memory available in the heap. Window storage can differ widely
    amongst applications depending on how many windows are created and disposed. */

/*  1.01 - The code that used to be part of ForceEnvirons has been moved into
    this module. If an error is detected, instead of merely doing an ExitToShell,
    which leaves the user without much to go on, we call AlertUser, which puts
    up a simple alert that just says an error occurred and then calls ExitToShell.
    Since there is no other cleanup needed at this point if an error is detected,
    this form of error- handling is acceptable. If more sophisticated error recovery
    is needed, an exception mechanism, such as is provided by Signals, can be used. */

void Initialize()
{
    Handle      menuBar;
    WindowPtr   window;
    long        total, contig;
    EventRecord event;
    short       count;

    gInBackground = false;

    InitGraf((Ptr) &qd.thePort);
    InitFonts();
```

```
InitWindows();
InitMenus();
TEInit();
InitDialogs(nil);
InitCursor();

/*  Call MPPOpen and ATPLoad at this point to initialize AppleTalk,
    if you are using it. */
/*  NOTE -- It is no longer necessary, and actually unhealthy, to check
    PortBUse and SPConfig before opening AppleTalk. The drivers are capable
    of checking for port availability themselves. */

/*  This next bit of code is necessary to allow the default button of our
    alert be outlined.
    1.02 - Changed to call EventAvail so that we don't lose some important
    events. */

for (count = 1; count <= 3; count++)
    EventAvail(everyEvent, &event);

/*  Ignore the error returned from SysEnvirons; even if an error occurred,
    the SysEnvirons glue will fill in the SysEnvRec. You can save a redundant
    call to SysEnvirons by calling it after initializing AppleTalk. */

SysEnvirons(kSysEnvironsVersion, &gMac);

/* Make sure that the machine has at least 128K ROMs. If it doesn't, exit. */

if (gMac.machineType < 0) AlertUser();

/*  1.02 - Move TrapAvailable call to after SysEnvirons so that we can tell
    in TrapAvailable if a tool trap value is out of range. */

gHasWaitNextEvent = TrapAvailable(_WaitNextEvent, ToolTrap);

/*  1.01 - We used to make a check for memory at this point by examining ApplLimit,
    ApplicationZone, and StackSpace and comparing that to the minimum size we told
    MultiFinder we needed. This did not work well because it assumed too much about
   the relationship between what we asked MultiFinder for and what we would actually
    get back, as well as how to measure it. Instead, we will use an alternate
    method comprised of two steps. */

/*  It is better to first check the size of the application heap against a value
    that you have determined is the smallest heap the application can reasonably
    work in. This number should be derived by examining the size of the heap that
    is actually provided by MultiFinder when the minimum size requested is used.
    The derivation of the minimum size requested from MultiFinder is described
    in Sample.h. The check should be made because the preferred size can end up
    being set smaller than the minimum size by the user. This extra check acts to
    insure that your application is starting from a solid memory foundation. */

if ((long) GetApplLimit() - (long) ApplicationZone() < kMinHeap) AlertUser();

/*  Next, make sure that enough memory is free for your application to run. It
    is possible for a situation to arise where the heap may have been of required
    size, but a large scrap was loaded which left too little memory. To check for
    this, call PurgeSpace and compare the result with a value that you have
    determined is the minimum amount of free memory your application needs at
```

```
        initialization. This number can be derived several different ways. One way that
        is fairly straightforward is to run the application in the minimum size
        configuration as described previously. Call PurgeSpace at initialization and
        examine the value returned. However, you should make sure that this result is
not
        being modified by the scrap's presence. You can do that by calling ZeroScrap
        before calling PurgeSpace. Remove this call before shipping, though. */

    /* ZeroScrap(); */

    PurgeSpace(&total, &contig);
    if (total < kMinSpace) AlertUser();

    /*  The extra benefit to waiting until after the Toolbox Managers have been
        initialized to check memory is that we can now give the user an alert to tell
        him/her what happened. Although it is possible that the memory situation could
        be worsened by displaying an alert, MultiFinder would gracefully exit the
        application with an informative alert if memory became critical. Here we are
        acting more in a preventative manner to avoid future disaster from low-memory
        problems. */

    /*  We will allocate our own window storage instead of letting the Window
        Manager do it because GetNewWindow may load in temp. resources before
        making the NewPtr call, and this can lead to heap fragmentation. */

    window = (WindowPtr) NewPtr(sizeof(WindowRecord));
    if ( window == nil ) AlertUser();
    window = GetNewWindow(rWindow, (Ptr) window, (WindowPtr) -1);

    menuBar = GetNewMBar(rMenuBar);         /* read menus into menu bar */
    if ( menuBar == nil ) AlertUser();
    SetMenuBar(menuBar);                    /* install menus */
    DisposeHandle(menuBar);
    AppendResMenu(GetMenuHandle(mApple), 'DRVR');   /* add DA names to Apple menu */
    DrawMenuBar();

    gStopped = true;
    if ( !GoGetRect(rStopRect, &gStopRect) )
        AlertUser();                        /* the stop light rectangle */
    if ( !GoGetRect(rGoRect, &gGoRect) )
        AlertUser();                        /* the go light rectangle */
} /*Initialize*/


/*  This utility loads the global rectangles that are used by the window
    drawing routines. It shows how the resource manager can be used to hold
    values in a convenient manner. These values are then easily altered without
    having to re-compile the source code. In this particular case, we know
    that this routine is being called at initialization time. Therefore,
    if a failure occurs here, we will assume that the application is in such
    bad shape that we should just exit. Your error handling may differ, but
    the check should still be made. */

Boolean GoGetRect(short rectID, Rect *theRect)
{
    Handle      resource;

    resource = GetResource('RECT', rectID);
    if ( resource != nil ) {
        *theRect = **((Rect**) resource);
        return true;
    }
    else
        return false;
} /* GoGetRect */


/*  Check to see if a given trap is implemented. This is only used by the
    Initialize routine in this program, so we put it in the Initialize segment.
    The recommended approach to see if a trap is implemented is to see if
    the address of the trap routine is the same as the address of the
    Unimplemented trap. */
/*  1.02 - Needs to be called after call to SysEnvirons so that it can check
    if a ToolTrap is out of range of a pre-MacII ROM. */

Boolean TrapAvailable(short tNumber, TrapType tType)
{
    if ( ( tType == ToolTrap ) &&
        ( gMac.machineType > envMachUnknown ) &&
        ( gMac.machineType < envMacII ) ) {     /* it's a 512KE, Plus, or SE */
        tNumber = tNumber & 0x03FF;
        if ( tNumber > 0x01FF )                 /* which means the tool traps */
            tNumber = _Unimplemented;           /* only go to 0x01FF */
    }
    return NGetTrapAddress(tNumber, tType) !=
                                    NGetTrapAddress(_Unimplemented, ToolTrap);
} /*TrapAvailable*/
```

Document Revision History

This table describes the changes to *Carbon Porting Guide*.

Date	Notes
	Updated path to LaunchCFMApp in "Debugging Your Application" (page 45) to match that in "Running Your Application on Mac OS X" (page 43).
	Updated CodeWarrior Mach-O information in "Building Carbon Applications" (page 41) to remove mention of the older cross-compiler.
2001-06-23	Spelling correction: "supercedes" should be "supersedes."
	Added Important note to "Replace Macro Calls to the Mixed Mode Manager With UPP Accessor Functions" (page 22) indicating that Thread Manager functions that did not previously require UPPs for function pointers now require them in Carbon.
	Added new section: "Move Custom Definition Procedures Out of Resources" (page 22).
	Added additional information to "Add a 'plst' 0 Resource" (page 26) about how the Mac OS X Finder interprets the presence of resource forks and `'plst' 0` resources in determining what environment to launch an application.
	IB Carbon Runtime in "Determine the Appropriate CarbonLib Version" (page 27) is now called Interface Builder Services.
	Added new section "Do Not Write to Your Application's Resource Fork" (page 28).
	Changed text in "Running Your Application on Mac OS X" (page 43) to reflect new path to `LaunchCFMApp`: `/System/Library/Frameworks/Carbon.framework/Versions/A/Support/LaunchCFMApp`.
	Correction in "Carbon Event Manager" (page 69): The Carbon Event Manager does not replace the functionality of the Notification Manager.
2000-12-04	The Aqua guidelines document, *Adopting the Aqua Interface*, is now called *Inside Mac OS X: Aqua Human Interface Guidelines*.
	Changed "Add a 'carb' 0 Resource" to "Add a 'plst' 0 Resource" (page 26). The `'plst' 0` resource supersedes the `'carb' 0` resource.
	Correction in "Consider Using Bundles" (page 36): A bundle appears as a folder hierarchy on Mac OS X if the bundle bit is unset (not set as previously stated).

Date	Notes
	Added information about __appstart to "Building Applications Using MPW" (page 44).
	Indicated in "Update Modified or Obsolete Functions" (page 54) and "The Basic Conversion" (page 72) that Carbon does not support the diskEvt event. Removed diskEvt case from Carbon version of Sample.
	Further subdivided "The Basic Conversion" (page 72) by adding sections "Installing the Standard Event Handlers" (page 73) and "Registering Your Own Event Handlers" (page 74).
	Added information about the event handler parameters in "The Application-Level Event Handler" (page 74).
2000-11-15	General correction: specific references to CarbonLib 1.1 updated to CarbonLib 1.2.
	General correction: CarbonStub now renamed CarbonLibStub.
	In "Preparing Your Code for Carbon" (page 17), added a link to Technote TN2003, "Moving Your Code to Mac OS X."
	Changed URL link in "Use the Carbon SDK" (page 20) to http://developer.apple.com/carbon/index.html, as this location is more Carbon-specific and allows ADC members to download prerelease versions of the SDK.
	Declaration that CarbonAccessors.o is only a porting tool softened in "Begin With CarbonAccessors.o" (page 21) and "Functions in CarbonAccessors.o" (page 91). You can link against CarbonAccessors.o to simplify building non-Carbon applications from a Carbon code base.
	In "Conditionalize Quit Menu Items" (page 26), reworded the text to emphasize that the position of the Quit item is a feature of the user interface (Aqua versus Mac OS 8 and 9) rather than of the underlying system.
	Added more information to "Begin Transitioning to the Aqua Interface" (page 37), including details about Appearance Manager compliance, sheets, and help tags.
	In "Adopt a Terse Name for the Application Menu" (page 37) changed the recommended storage location for the name from Info.plist to InfoPlist.strings, because the latter allows the name to be localized.
	Simplified methods of transferring files between Mac OS X and Mac OS 8 and 9 in "Native Mac OS 9 Versus Mac OS X's Classic Environment" (page 41), as file sharing is now fully supported.
	Added About box screen shot to "Modify the About Box" (page 57).
	Mentioned in "The Application-Level Event Handler" (page 74) that you could assign a command ID for a menu item in an 'xmnu' resource instead of calling SetMenuItemCommandID.

Date	Notes
	Bug fix in Listing 4-1 (page 74): Added `DisposeRgn` call to the `kEventMouseMoved` case of the application event handler to deallocate memory from the `NewRgn` call.
2000-10-24	Revised text in "Resources" (page 13) and "Move Resources to Data Fork–Based Files" (page 36) to emphasize that the preferred method for accessing application resources is by using CFBundle APIs. Also added that the `'cfrg' 0` and `'carb' 0` resources need to remain in the resource fork for CFM-based Carbon applications so the Mac OS X Finder can launch them properly.
	The Carbon Event Manager constant `kEventWindowSizeChanged` now replaced by `kEventWindowBoundsChanged`.
	In section "Check Your OpenGL Code" (page 29), the OpenGLMemoryLibrary library is now compatible with Carbon. Also, if you are building a Mach-O Carbon applicatiojn that uses OpenGL, you must call the `aglConfigure` function before creating any OpenGL contexts.
	Added screen shot of the ported Sample application to "The Carbon Version of Sample" (page 57).
	Added information about new Carbon technology "Multilingual Text Engine (MLTE)" (page 71).
	Added Index.
2000-10-12	Correction in "How Does Carbon Work?" (page 11): `CarbonLibStub` changed to `CarbonStub`.
	Changed section "Adopt the Carbon.h Header" to "Modify or Conditionalize Your Headers" (page 25). You no longer need to adopt `Carbon.h`. The path required for the cc compiler is now `-I /Developer/Headers/FlatCarbon`. You can also use this path with the conventional Mac OS 8 and 9 headers when building on Mac OS X.
	Changed text in "Begin Transitioning to the Aqua Interface" (page 37) to indicate that your application automatically registers with the Appearance Manager when you link with `CarbonLib`.
	Added new chapters, "A Porting Example" (page 47) and "New Carbon Technologies" (page 69).
	Added new appendix, "The Sample Application" (page 97) which contains the source code to be ported in "A Porting Example" (page 47).
2000-09-07	Updated software and header versions to reflect the latest available.
	Added new porting guideline "Modify or Conditionalize Your Headers" (page 25).

Date	Notes
	Added info about using plists to signify Mac OS X Carbon applications in "Add a 'plst' 0 Resource" (page 26) and "Running Your Application on Mac OS X" (page 43).
	Correction in "Determine the Appropriate CarbonLib Version" (page 27):: Appearance Manager 1.1 is available in all versions of CarbonLib, not just 1.1 and later. Also, DataBrowser is now available back to System 8.6.
	"Handling Buffered Windows" (page 30) section added, which incorporates information from the older section "Drawing into Windows Without QuickDraw".
	Added more specific event information (for example, which event to wait on) in answers to questions in "Window Dragging and Resizing Q&A" (page 31).
	The Aqua guidelines document referenced in "Begin Transitioning to the Aqua Interface" (page 37) is now *Adopting the Aqua Interface*. Added URL pointer to the document.
	Added new porting guidelines: "Move Resources to Data Fork–Based Files" (page 36) and "Consider Using Bundles" (page 36).
	Added new build section "Building Applications Using MPW" (page 44).
	Added new sections "Changes to WDEFs" (page 81) and "Changes to MDEFs" (page 81) under "Custom Definition Procedures" (page 81).
	Removed private functions from `CarbonAccessors.o` list in Table A-3 (page 91).
	In Table A-3 (page 91), `QError` should be `QDError`.
	Removed functions accidentally identified as removed from `CarbonAccessors.o` in Table A-3 (page 91): `GetWindowKind`, `SetWindowKind`, `GetKeys`, `GetWindowSpareFlag`, `InvalWindowRect`, and `InvalWindowRgn`,.
2000-07-11	Major reorganization of material.
	"Introduction to Carbon Porting Guide" (page 9) rewritten to reflect the current state of Carbon.
	Porting guidelines reorganized into sections: "Essential Steps for Porting Your Application" (page 20), "Additional Porting Issues" (page 27), and "Optimizing Your Code for Carbon" (page 33).
	Some existing porting sections were renamed to better integrate wth the new sections.

Date	Notes
	New porting guideline sections added: "Use the Carbon SDK" (page 20), "Target Mac OS 8 and 9 First" (page 20), "Use DebuggingCarbonLib" (page 24), "Adopt Required Carbon Technologies" (page 25), "Update Modified or Obsolete Functions" (page 25), "Determine the Appropriate CarbonLib Version" (page 27), "Examine Your Plug-ins" (page 29), "Adopt HFS Plus APIs" (page 35), "Consider Mach-O Executables" (page 35), "Adopt a Terse Name for the Application Menu" (page 37).
	Softened requirements for the contents of a `carb'0'` resource in "Add a 'plst' 0 Resource" (page 26). The resource can contain arbitrary data.
	Comparision of CFM versus Mach-O object file formats moved to the porting guidelines chapter under "Consider Mach-O Executables" (page 35).
	"Linking to Non-Carbon-Compliant Code" (page 29) moved to porting guidelines chapter.
	Directory paths in Mac OS X have changed:
	Path `/System/Developer/Tools/LaunchCFMApp` is now `/Developer/Tools/LaunchCFMApp`.
	Path `System/Administration/Terminal.app` is now `/Applications/Utilities/Terminal.app`.
	Function descriptions and other reference-like material moved to the Appendix: "Custom Definition Procedures" (page 81), "Functions for Accessing Opaque Data Structures" (page 82), "Functions in CarbonAccessors.o" (page 91), "Debugging Functions" (page 95), and "Resource Chain Manipulation Functions" (page 95).
	Revised contents of `CarbonAccessors.o` in Table A-3 (page 91) and Table A-4 (page 94).
2000-05-01	Updated software and header versions to reflect the latest available (for example, CarbonLib 1.1 and Universal Interfaces 3.4d2).
	Added new section, "The Carbon Specification" (page 19).
	Added new sections describing preparations for Carbon conversion:"Don't Pass Pointers Across Processes" (page 28)"Avoid Polling and Busy Waiting" (page 34)"Use Casting Functions to Convert DialogPtrs and WindowPtrs" (page 21)"Use "Lazy" Initialization for Shared Libraries" (page 35)"Check Your OpenGL Code" (page 29)"Begin Transitioning to the Aqua Interface" (page 37)"Provide Thumbnail Icons for Your Application" (page 37)
	Added information about avoiding preallocation and suballocators in "Manage Memory Efficiently" (page 34).
	Created new section, "Window Manager Issues" (page 30), to cover Window Manager porting issues in detail.

Date	Notes
	In Table A-1 (page 85), added `SetMenuDefinition` as the accessor function for the `MenuProc` element in a `MenuInfo` structure.
	Added Table A-2 (page 88) listing QuickDraw accessor functions.
	Added information about transferring files between Mac OS 9 and Mac OS X computers in "Native Mac OS 9 Versus Mac OS X's Classic Environment" (page 41).
	Revised contents of `CarbonAccessors.o` in Table A-3 (page 91)..
	Added list of functions removed from `CarbonAccessors.o` in Table A-4 (page 94).
	Emphasized that you cannot link with `InterfaceLib` if you link to `CarbonLib` in "Using CodeWarrior to Build a CFM Carbon Application" (page 41)
	Created new section, "Linking to Non-Carbon-Compliant Code" (page 29).
	Revised "Debugging Your Application" (page 45) to include specific information about debugging Carbon applications using GDB.
	Added this document revision history.

Index

Symbols

76
__appstart entry point in MPW 45
__startentry point in MPW 45

A

About box 57, 76
accessor functions 83
  for handling universal procedure pointers 22
  for Human Interface Toolbox structures 84
  for QuickDraw data structures 88
ACCESSOR_CALLS_ARE_FUNCTIONS 21, 51
aglConfigure, calling from Mach-O Carbon applications 29
Apple events 26, 55
application address spaces 12
application menu, naming convention for 37
application threading 12
application-defined functions 14
Aqua
  adopting in Carbon applications 37
  changes to Sample application for 57
  Quit menu item in 26, 55

B

benefits of Carbon 10
buffered windows, handling 30
building Carbon applications using MPW 44
building Carbon applications with CodeWarrior. *See* CodeWarrior
bundles 35
bundles 36

C

Carbon Dater tool 17, 48
Carbon Event Manager
  adding to Sample application 71
  handling menu selections using 72
  installing event handlers 73
  keyboard events 72
  overview 69
  quitting the application 76
  resizing windows with 33
  standard event handlers 72
  timers, in place of null events 76
  tracking window movement using 31
Carbon Event Manager
  timers, in place of null events 34
Carbon events *See* Carbon Event Manager
Carbon Printing Manager 25
Carbon Specification 19
Carbon Support folder 42
Carbon.h header 25, 53
Carbon
  events. *See* Carbon Event Manager
  API composition of 10
  benefits of 10
  defined 9
  feedback URL for 15
  frameworks on Mac OS X 11
  functions unavailable in 11
  implementation on Mac OS X and Mac OS 8 and 9 11
  memory management in 34
  plug-ins 29
  printing in 13
  SDK, location of 20
CarbonAccessors.o 21
  contents of 91
  using with non-Carbon applications 21
CarbonAccessors.o
  used to port Sample application 51
CarbonLib
  versions of 27
  versus Carbon frameworks 11

CarbonLibStub stub library 11
'carb'0 resource 26, 43
casting functions 21, 82
CFBundle APIs 36
CheckAllHeaps function 95
Classic environment versus Mac OS 9 41
Code Fragment Manager 13, 35
CodeWarrior
  building Sample application with 50
  using to build Carbon applications 41, 42
command IDs 76
composition of Carbon 10
control panels 14
converting types WindowPtr and DialogPtr 21
Core Foundation, overview 70
custom definition procedures 14, 81
custom menu definitions 81
custom window definitions 33, 81

D

data fork, resources in 13, 36
data structures, opaque 14, 82
DataBrowser, overview 70
debugging Carbon applications 45
debugging functions, new 95
DebuggingCarbonLib 24
defprocs 14
DetachResourceFile function 96
dialog pointers 82
Drag Manager, translucent dragging with 33
dragging and resizing windows in Carbon 31
DumpPEF tool 29

E

event handlers, Carbon Event Manager 72
events. *See* Carbon Event Manager
exit() in MPW 45
Extended File Format 35

F

feedback URL 15
FlatCarbon headers 25
frameworks, Carbon 11
FSpResourceFileAlreadyOpen function 96
functions unavailable in Carbon 11

G

GDB debugger 45
gestaltMenuMgrAquaLayoutMask selector 26, 56
GetSharedLibrary for loading non-Carbon–compliant code 29
graphics pointers 21
  versus window pointers 82
graphics ports, creating and disposing in Carbon 91

H

headers
  FlatCarbon 25
headers
  Universal 20
HFS Plus 35

I

icons, creating and adding to applications 37
idle events 34
Info.plist file 37
infoPlist.strings file 37
initialization functions 35
InsertResourceFile function 95
InterfaceLib, linking with 21, 43
IsHandleValid function 95
IsHeapValid function 95
IsPointerValid function 95

K

keyboard events 72

L

LaunchCFMApp tool 43
launching applications on Mac OS X using LaunchCFMApp 43
lazy initialization 35
linking to non-Carbon–compliant code 29
low-memory globals 19, 23

M

Mac OS 8 and 9
  API compatibility with CarbonLib 27
Mac OS 9 versus Classic environment 41
Mach-O file format, advantages and disadvantages of 35
main entry point in MPW 45
MDEFs 81
memory management in Mac OS X 34
menu item selection using Carbon events 72
Metrowerks CodeWarrior. *See* CodeWarrior
Mixed Mode Manager 13, 22
MLTE, overview 71
MPW, building Carbon applications using 44
Multilingual Text Engine, overview 71

N

Navigation Services 25
null events 34

O

opaque data structures 14, 82
OPAQUE_TOOLBOX_STRUCTS 21, 51
OpenGL in Carbon 29

P

passing pointers across processes 28
Patch Manager 28
PEF containers 35
PEF containers 17
pixel maps, drawing into 30
plist file 37
plist resource. *See* 'plst'0 resource
'plst'0 resource 26, 43, 55
  versus Info.plist file 37
'plst'0 resource 26, 43, 55
  versus Info.plist file 37
plug-ins 29
polling 34
preemptive scheduling of applications 12
prefix file information in CodeWarrior 43
printing in Carbon 13
Printing Manager 25
protected memory 12

Q

quit events 76
Quit menu item 26, 55
quitting Carbon applications 26, 76

R

resources
  'SIZE' 45
  in data fork 13, 36
  'carb'0 26, 43
  'plst'0 26, 37, 43

S

Sample application
  adding accessors to 51
  adding Carbon events to 71
  Carbon Dater report on 48
  described 47
  listings of Carbon version 57
  original code listings 97
  porting to Carbon 50
screen savers in Carbon 33
SDK, Carbon 20
SetMenuItemCommandID function 76
shared libraries, lazy initialization of 35
'SIZE' resource 45
standard definition procedures 14
standard event handlers for Carbon events 72
Standard File Package 25

T

TARGET_API_MAC_CARBON 21, 43, 53
TrafficLight. *See* Sample application
transferring files between Mac OS 9 and Mac OS X 41
trap tables 14, 28

U

Universal Interfaces 20, 50
universal procedure pointers 13
universal procedure pointers 14, 22
update events 72, 78
utility functions for porting 91

V

versions of CarbonLib 27
virtual memory on Mac OS X 13

W

WDEFs 33, 81
Window Manager Port 31
window pointers 82