Carbon Porting Guide

For Mac OS X Public Beta

Preliminary

7/17/00

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada

Contents

Figures, Listings, and Tables

Introduction

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. 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 based on the Classic Mac OS APIs that will run on Mac OS X. Here we define Classic Mac OS as being those technologies and functions that were designed for Mac OS 9 and earlier.

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 have 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?

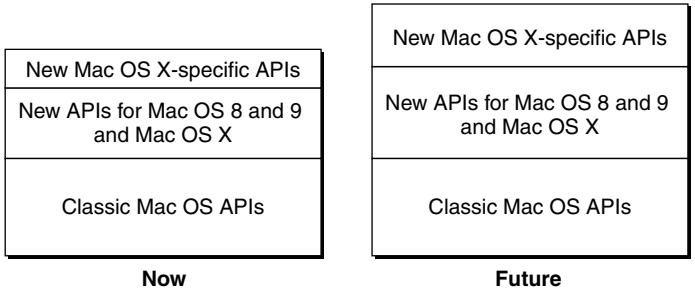
The Carbon programming interface consists of the following types of APIs:

- 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, some functions may now require a context (or process) ID parameter to distinguish itself in a preemptively-scheduled environment.

- 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 can run only on Mac OS X.

Currently, the Classic Mac OS APIs take up the largest proportion of Carbon APIs, as shown in Figure 1-1. However, because Carbon is a first class application framework on Mac OS X, new X-specific APIs will be added in the future to enhance its capabilities.

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



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 17).

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. To make this happen, your application links with a single stub library, CarbonLibStub at build time. At runtime your application links with the appropriate Carbon implementation stored as shared libraries (or DLLs).

Note: At this point in the development cycle, the implementation of Carbon functions on Mac OS X does not match those in CarbonLib. To work around this discrepancy, there are two different stub libraries, `CarbonStub9` and `CarbonStubX`, and you should link against the appropriate one depending on the platform you are targeting. This implementation discrepancy will be removed before Mac OS X GM.

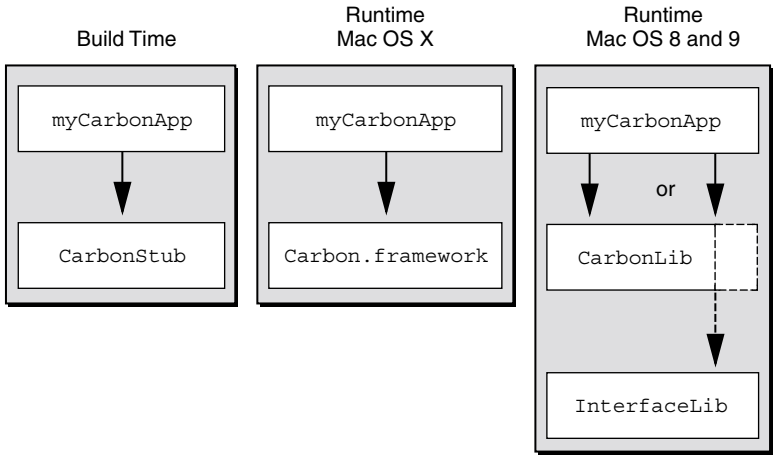
On Mac OS X, your application links dynamically to the Carbon framework, which is a hierarchy of libraries and resources that contains the implementation of Carbon.

On Mac OS 8 and 9, the Carbon implementation is stored as a system extension, CarbonLib. This library contains

- 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 1-2 shows Carbon functions called on Mac OS X and Mac OS 8 and 9.

Figure 1-2 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 33) 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. Chapter 2, "Preparing Your Code For Carbon," 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 39) 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 39).

Resources

Mac OS X supports resources, but you should consider moving all your resources to the data fork of your application. 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.

Note that you can no longer store executable code in resources.

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 25) 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 file `PMApplication.h`. Preliminary documentation for the Carbon Printing Manager is provided with the OS X Beta Developer Tools CD and at the following website:

```
http://developer.apple.com/techpubs/carbon/multimedia/
CarbonPrintingManager/carbonprintingmgr.html
```

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 "Custom Definition Procedures" (page 57).

Application-Defined Functions

Carbon supports most Mac OS application-defined (callback) functions. Mac OS X will fully support 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 58).

Additional Information and Feedback

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

`http://developer.apple.com/macosx/carbon/`

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.

To make your job easier, we recommend you 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.

You'll find the Carbon Dater tool and complete instructions online at

```
http://developer.apple.com/macosx/carbon/dater.html
```

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 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 e-mail 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 via e-mail. 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 26). 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 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

Carbon Dater uses the Carbon Specification available at

`http://developer.apple.com/techpubs/carbon/Carbon_Specification/CarbonSpecTOC.html`

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

Essential Steps for Carbonizing Your Application

This section describes the bare minimum steps you need to Carbonize your application. 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 39).

In addition to reading this section, you should also read the information provided in "Additional Porting Issues" (page 30) before beginning to port your application.

Begin With 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 development version is 3.4d2). 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 this SDK.

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

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

Make Sure All 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 "Custom Definition Procedures" (page 57) for information about new functions for creating native defprocs.

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/sdk/#Carbon`

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 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 70).

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 `CarbonLib`.

Important
Apple does not support the use of `CarbonAccessors.o` as anything other than a porting tool. To build a Carbon application you must link against `CarbonLib`..

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 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 ProcPtr 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 it is 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 42) for more information about the differences between these formats.

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

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 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 2-1 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 Casting Functions to Convert DialogPtrs and WindowPtrs

You cannot directly cast a `DialogPtr` or `WindowPtr` to a `GrafPtr`, but instead you must use the new functions described in "Casting Functions" (page 58). Direct casting will not affect compilation, but it will cause inexplicable crashes on Mac OS X.

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 nondebugging version of the library.

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.

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/techpubs/carbon/Files/NavigationServices/navigationservices.html`
- The Carbon Printing Manager replaces the old Mac OS Printing Manager. For documentation, see the web site:
  `http://developer.apple.com/techpubs/carbon/multimedia/CarbonPrintingManager/carbonprintingmgr.html`

Add a 'carb' 0 Resource

On Mac OS X, Carbon applications that do not contain a `'carb' 0` resource will open in the Classic Compatibility Environment and will not gain all the advantages of Mac OS X. To ensure that your application opens in the Mac OS X environment, your application must include a resource of type `'carb'` with ID 0 . The contents may be arbitrary, typically four bytes of zero data.

Conditionalize “Quit” Menu Items

Mac OS X applications are automatically assigned a “Quit” menu item under the Application menu, so your application does not need to add one to the File menu as in the past. As long as your application supports the `'quit'` Apple event, it will quit normally. However, because Mac OS 8 and 9 applications still require a “Quit” 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 current operating system is to check the `gestaltMenuMgrAquaLayoutBit` bit of the `gestaltMenuMgrAttr` gestalt selector. If the bit is set, the application is running on Mac OS X.

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” 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 Carbonizing Your Application" (page 22), 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 Navigation Services Core Foundation Carbon Printing Manager

CarbonLib Version	Reflects Universal Interfaces Version	Compatible Back to	Notes
1.1	3.4	Mac OS 8.6	*Adds the following:*
			Appearance Manager 1.1
			Carbon Event Manager
			XML
			URL Access Manager
			Apple Type Services for Unicode Imaging (ATSUI)
			IB Carbon Runtime
			Font Sync
			Apple Help Viewer
			Font Management
		Mac OS 9	*Adds the following:*
			DataBrowser
			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 34).

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.

Check Your OpenGL Code

If you use OpenGL in your application, be aware that the APIs in the OpenGLMemory library are not Carbon-compliant, as they address issues that do not exist on Mac OS X. Otherwise, you should continue to link to the OpenGLLibrary and OpenGLUtility stubs as you would for non-Carbon applications. On Mac OS X these functions will link with the OpenGL framework.

Examine Your Plug-Ins

Carbonized applications can load nonCarbon plug-ins. However, you must be careful 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 nonCarbon 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; attempting to call nonCarbon 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 OS X Beta Developer Tools CD for examples.

Window Manager Issues

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

Drawing into Windows without QuickDraw

If you draw directly into the bitmap 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.
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. That's 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 35).

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, let us know what you need. We may add some APIs 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 34).

- 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 window that isn't buffered, the Window Manager provides the traditional outline feedback.

  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. 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. 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 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 2-1 shows the traditional outline feedback for resizing a window.

Figure 2-1 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 provide a resize event handler. The Carbon Window Manager sends events to your handler that indicate 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. 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 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 drag and drop 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/techpubs/quicktime/quicktime.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. That's not really a question, but let us know what you need.

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 (that is, 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 no longer can 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 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" Initializations 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+ APIs

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

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

See the File Manager documentation at

```
http://developer.apple.com/techpubs/carbon/Files/FileManager/
filemanager.html
```

for more information about HFS+.

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 Buildler development tools on Mac OS X.

Disadvantages:

- Applications cannot run on Mac OS 8 and 9.
- 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 Mach-O–based executables together in bundles in a manner analogous to the 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 and packaging files in bundles.

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

Begin Transitioning to the Aqua Interface

If you are using Appearance Manager 1.1 or later, your Carbon application will automatically adopt the basic Aqua look and feel. However, to provide the best user experience, you should begin modifying dialog boxes, windows, icons, and so on to

meet the Aqua specification. See the document *Aqua Layout Guidelines* for details. For additional information on icons, see "Provide Thumbnail Icons for Your Application" (page 43).

Adopt a Terse Name for the Application Menu

The leftmost pulldown 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 for this menu.

Provide Thumbnail Icons for Your Application

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

```
http://developer.apple.com/techpubs/macos8/HumanInterfaceToolbox/
IconServUtili/IconServices/index.html
```

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 2-2 shows the icon family, including thumbnail icons, for the `Classic.app` application in Mac OS X.

Figure 2-2 Thumbnail icons in a `.icns` file, displayed by 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 may want to 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 Beta 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 vs. 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` (formerly known as the "Blue Box") provides a convenient environment for running your development system. You can easily switch between the two environments, and launch Carbon applications in either.

For performance reasons, however, you may prefer to develop on a native Mac OS 9 system (that is, a computer running Mac OS 9 instead of Mac OS X), as your development tools are likely to run somewhat slower in the Classic environment. In this case 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.

You can connect the two computers using Ethernet, and transfer files between them using FTP. However, doing so requires you to split the resource and data forks for transmission. Instead, you can use the following simpler methods:

- Launch the Classic.app application on your Mac OS X machine and enable file sharing. You can then mount the hard drive on your Mac OS 9 machine and simply copy files using the Finder. Note that the Classic application does not support TCP/IP, so you must use AppleTalk instead (AppleTalk must be enabled on the Mac OS X machine as well, using the Setup Assistant or the Network control panel). In addition, on systems prior to Developer Preview 4, make sure your Classic environment is not using the same IP address as the underlying Mac OS X system.
- 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 has developed a cross-compiler that you can use to build Mach-O applications with CodeWarrior on Mac OS 9. You may want to create a Mach-O version of your application in order to debug it on Mac OS X using Project Builder. However, if you have a second computer you may want to investigate whether Metrowerks' two-machine debugger better 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 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 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, we recommend CodeWarrior Pro version 4.0 or later.

You can run CodeWarrior on either a native Mac OS 9 system or in the Classic environment on Mac OS X. You must install CodeWarrior on a disk or partition that uses the HFS Plus volume format ("Mac OS Extended") if you plan to run CodeWarrior in the Classic environment.

Preparing Your Development Environment

Before you start Carbon development with CodeWarrior, you'll need to install the tools and libraries provided with the OS X Beta 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 1.1).
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 4.0 to earlier CodeWarrior versions will result in the loss of prefix file information in the C/C++ Language Preferences panel. Many of the code samples on the OS X Beta 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 `CarbonLib` stub library 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. 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 + 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:

`/Developer/Tools/LaunchCFMApp` *filename*

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

Note that if your application does not contain a `'carb'` resource, Mac OS X opens the application in the Classic Compatibility Environment. To ensure that your application opens in the Mac OS X environment, your application must include 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

To build a Mach-O application with CodeWarrior, you'll need to install the Mach-O cross-compiler tools available from Metrowerks.

Building Your Application

Refer to your Metrowerks CodeWarrior documentation for instructions on using the Mach-O cross-compiler.

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+ 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 OS X Beta Developer Tools CD. Instructions for building Mach-O Carbon applications are available in Project Builder's online help documentation.

Debugging Your Application

You can debug Carbon applications on Mac OS 9 using the Metrowerks debugger. You can also use this debugger with two networked machines, 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 these 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 /Developer/Tools/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.

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 Beta 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 calls (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.

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 globals, 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 WindowPtr and DialogPtr types have a GrafPort embedded at the top of their structures. In fact, the current Universal Interfaces define DialogPtrs and WindowPtrs as GrafPtrs so that you don't have to cast them to a 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 WindowPtrs are no longer defined as GrafPtrs. But you can't simply cast these variables to GrafPtrs because it 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 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 GrafPorts 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 GrafPort 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 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 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 a WindowPtr can be cast to a WindowRecord pointer, which is not the case in Carbon).
2. The `currentWindow` variable is treated as a GrafPort. You need to use the casting functions discussed above to access a window's GrafPort.
3. GrafPorts 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 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 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 UserHandles in ListHandles.

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 refCons.

Table A-1 Summary of Carbon Human Interface Toolbox Accessors (continued)

Data Structure	Element	Accessor
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.
	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

Table A-1 Summary of Carbon Human Interface Toolbox Accessors (continued)

Data Structure	Element	Accessor
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`
	`controlList`	`GetRootControl`
	`nextWindow`	`GetNextWindow`
	`windowPic`	`Get/SetWindowPic`
	`refCon`	`Get/SetWRefCon`

Table A-1 Summary of Carbon Human Interface Toolbox Accessors (continued)

Data Structure	Element	Accessor
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 refCons.
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
	mouseLoc	GetListMouseLocation
	lClickLoop	Get/SetListClickLoop

Table A-1 Summary of Carbon Human Interface Toolbox Accessors (continued)

Data Structure	Element	Accessor
	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.
	txFace	Use GetPortTextFace or TextFace.

Table A-2 QuickDraw Accessor functions (continued)

Data Structure	Element	Accessor
	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

Table A-2 QuickDraw Accessor functions (continued)

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 GrafPorts were created by allocating non-relocatable memory the size of a CGrafPort and calling `OpenCPort`. Because GrafPorts 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 24) for information on how you can use this library to assist in porting your code to Carbon.

Important
Apple does not support the use of `CarbonAccessors.o` as anything other than a porting tool. To build a Carbon application you must link against `CarbonLib`

Table A-3 lists the Carbon functions implemented in `CarbonAccessors.o`. An "•" 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•`	`CallMenuBar••`
`CopyCStringToPascal•`	`CopyPascalStringToC•`
`CreateNewPort`	`DisposePort`
`FindTSMTEDDialog•`	`GetControlBounds`
`GetControlDataHandle`	`GetControlHilite`
`GetControlOwner`	`GetControlPopupMenuHandle`
`GetControlPopupMenuID`	`GetCorrectPort•`
`GetDialogCancelItem`	`GetDialogDefaultItem`
`GetDialogFromWindow`	`GetDialogKeyboardFocusItem`
`GetDialogPort`	`GetDialogTextEditHandle`
`GetDialogWindow`	`GetGlobalMouse`
`GetKCHRDeadKeyState•`	`GetKCHRDeadKeyStatePtr••`
`GetUCHRDeadKeyState•`	`GetListActive`
`GetListCellIndent`	`GetListCellSize`
`GetListClickLocation`	`GetListClickLoop`
`GetListClickTime`	`GetListDataBounds`
`GetListDataHandle`	`GetListDefinition`
`GetListFlags`	`GetListHorizontalScrollBar`
`GetListMouseLocation`	`GetListPort`
`GetListRefCon`	`GetListSelectionFlags`
`GetListUserHandle`	`GetListVerticalScrollBar`

Table A-3 Functions in CarbonAccessors.o (continued)

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	GetTSMTEDialogListStorage•
GetTSMTEDialogTSMTERecHandle•	GetWindowFromPort
GetWindowList•	GetWindowPort
GetWindowPortBounds	GetWindowSpareFlag
GetWindowStandardState	GetWindowUserState
GrabSpecifiedCFMSymbol•	GrowPortGrafVars•
InvalWindowRect	InvalWindowRgn

Table A-3 Functions in CarbonAccessors.o (continued)

IsControlHilited
IsKCHRAvailable•
IsPortOffscreen
IsPortRegionBeingDefined
IsTSMTEDialog•
IsWindowHilited
IsWindowVisible
p2cstrcpy•
SetControlDataHandle
SetControlPopupMenuHandle
SetKCHRDeadKeyState•
SetListCellIndent
SetListClickTime
SetListLastClick
SetListRefCon
SetListUserHandle
SetMenuHeight
SetMenuTitle
SetPortBackPixPat
SetPortBounds
SetPortDialogPort
SetPortGrafProcs
SetPortPenMode
SetPortPenPixPatDirect
SetPortPrintingReference
SetPortWindowPort
SetQDGlobalsArrow
SetTSMDialogDocumentID
SetWindowStandardState
TSMDialogIsValid•
TSMSetDeadKeyState•
IsCurrentTSMDocumentUnicode••
IsPortColor•
IsPortPictureBeingDefined
IsRegionRectangular
IsUCHRAvailable••
IsWindowUpdatePending
NeedKCHRState••
SetControlBounds
SetControlOwner
SetControlPopupMenuID
SetUCHRDeadKeyState•
SetListClickLoop
SetListFlags
SetListPort
SetListSelectionFlags
SetListViewBounds
SetMenuID
SetMenuWidth
SetPortBackPixPatDirect
SetPortClipRegion
SetPortFracHPenLocation
SetPortOpColor
SetPortPenPixPat
SetPortPenSize
SetPortVisibleRegion
SetQError•
SetQDGlobalsRandomSeed
SetTSMTEDialogTSMTERecHandle•
SetWindowUserState
TSMGetDeadKeyState•

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

Table A-4 Functions removed from CarbonAccessors.o

`DisableMenuItem`	`EnableMenuItem`
`GetControlColorTable`	`GetControlDefinition`
`GetKeys`	`GetTSMDialogPtr`
`GetTSMDialogTextEditHandle`	`GetWindowGoAwayFlag`
`GetWindowKind`	`GetWindowSpareFlag`
`InvalWindowRect`	`InvalWindowRgn`
`SetControlColorTable`	`SetControlOwner`
`SetWindowKind`	`SetKCHRDeadKeyStatePtr`
`ValidWindowRect`	`ValidWindowRgn`

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. It is invalid to pass `NULL` or an empty handle to `IsHandleValid`.

IsPointerValid

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

Returns `true` if the specified pointer is valid. It is invalid to 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.

Document Version History

This appendix lists changes to the Carbon Porting Guide:

Table B-1 Carbon Porting Guide revision history

Version	Notes
Public Beta	Major reorganization of material. "Introduction" (page 9) rewritten to reflect the current state of Carbon. Porting guidelines reorganized into sections: "Essential Steps for Carbonizing Your Application" (page 22), "Additional Porting Issues" (page 30), and "Optimizing Your Code for Carbon" (page 39). Some existing porting sections were renamed to better integrate wth the new sections. New porting guideline sections added: "Use the Carbon SDK" (page 23), "Target Mac OS 8 and 9 First" (page 23), "Use DebuggingCarbonLib" (page 28), "Adopt Required Carbon Technologies" (page 28), "Update Modified or Obsolete Functions" (page 28), "Determine the Appropriate CarbonLib Version" (page 30), "Examine Your Plug-Ins" (page 32), "Adopt HFS+ APIs" (page 41), "Consider Mach-O Executables" (page 42), "Adopt a Terse Name for the Application Menu" (page 43). Softened requirements fot the contents of a `carb'0'` resource in "Add a 'carb' 0 Resource" (page 29). 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 42).

Table B-1 Carbon Porting Guide revision history (continued)

Version	Notes
	"Linking to Non-Carbon-Compliant Code" (page 33) 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 57), "Functions For Accessing Opaque Data Structures" (page 58), "Functions in CarbonAccessors.o" (page 69), "Debugging Functions" (page 73), and "Resource Chain Manipulation Functions" (page 74).
	Revised contents of `CarbonAccessors.o` in Table A-3 (page 70) and Table A-4 (page 73).
Developer Preview 4	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 22).
	Added new sections describing preparations for Carbon conversion:"Don't Pass Pointers Across Processes" (page 32)"Avoid Polling and Busy Waiting" (page 40)"Use Casting Functions to Convert DialogPtrs and WindowPtrs" (page 28)"Use "Lazy" Initializations for Shared Libraries" (page 41)"Check Your OpenGL Code" (page 32)"Begin Transitioning to the Aqua Interface" (page 42)"Provide Thumbnail Icons for Your Application" (page 43)
	Added information about avoiding preallocation and suballocators in "Manage Memory Efficiently" (page 39).
	Created new section, "Window Manager Issues" (page 34), to cover Window Manager porting issues in detail.
	In Table A-1 (page 62), added `SetMenuDefinition` as the accessor function for the `MenuProc` element in a `MenuInfo` structure.
	Added Table A-2 (page 66) listing QuickDraw accessor functions.

Table B-1 Carbon Porting Guide revision history (continued)

Version	Notes
	Added information about transferring files between Mac OS 9 and Mac OS X computers in "Native Mac OS 9 vs. Mac OS X's Classic Environment" (page 47).
	Revised contents of `CarbonAccessors.o` in Table A-3 (page 70)..
	Added list of functions removed from `CarbonAccessors.o` in Table A-4 (page 73).
	Emphasized that you cannot link with `InterfaceLib` if you link to `CarbonLib` in "Using CodeWarrior to Build a CFM Carbon Application" (page 48)
	Created new section, "Linking to Non-Carbon-Compliant Code" (page 33).
	Revised "Debugging Your Application" (page 53) to include specific information about debugging Carbon applications using GDB.
	Added this document revision history.