Carbon Porting Guide

Converting Mac OS 8 Applications to the **Carbon** Programming Interface

Developer Preview Edition

4/30/99

Contents

C H A P T E R 1

Introduction

Contents

Introduction

This *Carbon Porting Guide* is intended to help experienced Macintosh developers convert existing Mac OS 8 applications into Carbon applications that can run on Mac OS X as well as Mac OS 8. This chapter introduces Carbon and provides an overview of the changes you'll need to be aware of as you convert your application.

Understanding Carbon

Carbon is the set of programming interfaces you can use to build Mac OS X applications that can also run on Mac OS 8 (version 8.1 or later). 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 is providing tools and documentation to help you determine the changes you'll need to make in your source code, as well as the header files and libraries necessary to build a Carbon application.

Carbon allows you to take advantage of all the great new features in Mac OS X with a minimum of effort. And you don't need to maintain separate source code versions because Carbon supports both the Mac OS 8 and Mac OS X runtime environments. (As always, you should test for the existence of specific features before using them.)

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

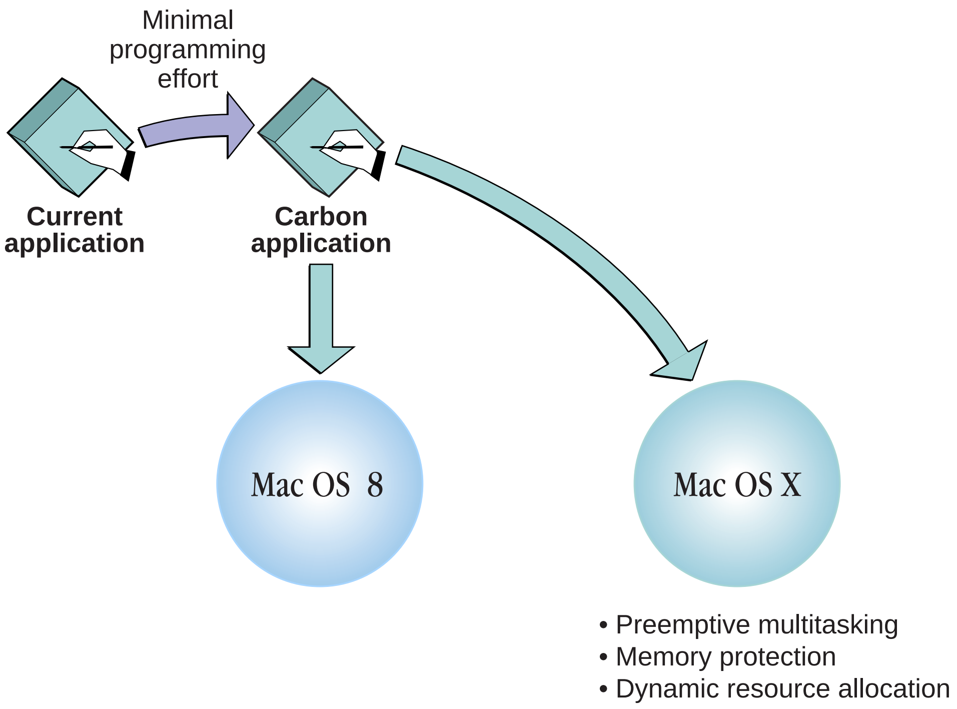
- **Greater stability**
  Protected address spaces help prevent errant applications from crashing the system or other applications.
- **Improved responsiveness**
  Each application is guaranteed processing time through preemptive multitasking, resulting in a more responsive user experience.
- **Dynamic 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.

The Carbon Advantage

The Mac OS has proven to be the strongest development platform for building innovative applications. Thousands of world-class programs have been created on the Macintosh, and it continues to be the platform of choice for creative professionals in the design, publishing, education, and new media markets. Flexible, extensible, and complete, the Mac OS has matured and evolved while retaining its leading-edge characteristics.

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 8 source code. As Apple moves the Mac OS forward, Carbon ensures you won't be left behind.

Figure 1-1 The Carbon advantage


Minimal programming effort
Current application
Carbon application
Mac OS 8
Mac OS X
• Preemptive multitasking
• Memory protection
• Dynamic resource allocation

An Easy Transition

Apple's goal is to ease your transition to Mac OS X by making as few changes as possible to the Mac OS API. Carbon accomplishes this goal by providing a compatible set of interfaces on which to base both existing and future applications.

Based on feedback from developers, and our own experience porting large applications, the level of programming effort required for Carbon compatibility is about the same as was needed for converting 68K applications to PowerPC, and can usually be accomplished in less than two weeks. Smaller applications have been ported in just a few days.

Carbon Today

Apple continues to make progress implementing Carbon on both Mac OS 8 and Mac OS X. You can begin developing Carbon applications today using the tools and libraries included on the Mac OS X Developer Preview CD.

By adopting Carbon now, you'll be ready to deliver Mac OS X compatibility to your customers when the new operating system is released. Better still, your applications will take advantage of all the latest performance, stability, and interface improvements of both Mac OS 8 and Mac OS X.

As you begin your porting effort, it's important that you understand where we are with Carbon today, and where we plan to take Carbon in the future:

- All new functions added to Mac OS 8 (versions 8.5 and later) will be part of Carbon, where applicable. Exceptions are functions for hardware drivers and other APIs that are not required by applications.
- Core Foundation is a new set of services available to Carbon applications. Some of the benefits provided by Core Foundation are data and code sharing, plug-in support, and internationalization support. For complete details, see "Overview of Core Foundation" (CFOverview.pdf) on the Developer Preview CD.
- Carbon does not currently support applications that bypass or control operating system services. For example, disk utilities that bypass the file system are not supported. We are investigating how to best address these needs for Mac OS X.

- Control panels are not supported by Carbon at this time. If possible, you should repackage your control panel as an application.
- System extensions that install device drivers will not be supported in Carbon. For other types of system extensions, Apple is considering how best to support programmatic extensibility in Carbon.

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.

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 Mac OS X. But because Mac OS 8 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 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 25) 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 25).

Code Fragments and the Code Fragment Manager

Carbon fully supports the Code Fragment Manager, and the Mac OS X runtime environment fully supports code compiled into code fragments. For Mac OS X, however, all code fragments must contain only native PowerPC code.

Mixed Mode Manager

For source code compatibility, Carbon supports universal procedure pointers (UPPs) transparently. Because Mac OS X does not run 68K code, the Mixed Mode Manager will not provide any useful functionality on that operating system. However, you may keep Mixed Mode Manager calls in your application to maintain source code compatibility with Mac OS 8.

Printing

Carbon introduces a new Printing Manager that allows applications to print on Mac OS 8 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 on Developer Preview CD.

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 applications 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 8 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 27).

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.

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 27).

Preparing Your Code For Carbon

Contents

Preparing Your Code For Carbon

This chapter describes the kinds of modifications you may need to make to your source code to create a Carbon application. To make your job easier, we recommend you begin by using the Carbon Dater tool to analyze the current compatibility level of your application.

The process of converting a typical large application can usually be accomplished in less than two weeks, depending on how closely you've followed Apple's recommended programming practices. The Carbon Dater report, and the information provided in this chapter, will help you gauge the extent of your porting effort.

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 Universal Interfaces 3.2

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 "Avoid Using Low-Memory Globals" (page 23). If the tested code was built with symbolic debugging information enabled, the report specifies the names of the routines that access low memory directly.

Note
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. ▲

Carbon Coding Guidelines

This section lists requirements and recommendations for creating Carbon-compatible code.

Begin With the Current Universal Interfaces

Your transition to Carbon will be easier if your application already compiles using the latest version of the Universal Interfaces (currently at 3.2). 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/

Compile Native PowerPC Code

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 27) for information about new functions for creating native defprocs.

Review Your Mixed Mode Calls

Carbon introduces significant changes to the Mixed Mode Manager. Static routine descriptors are not supported, and you must use the system-supplied UPP creation functions (such as `NewModalFilterProc`) for system callback UPPs. On Mac OS 8, these functions will allocate routine descriptors in memory just as you would expect. On Mac OS X they are likely to simply return the target function's address. By using the system-supplied UPP creation functions, your application will operate correctly in both environments. You still need to dispose of your UPPs using `DisposeRoutineDescriptor`, so that any allocated memory can be released when your application is running on Mac OS 8.

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

Avoid Using Low-Memory Globals

Low-memory globals are system and application global data located below the system heap in the classic Mac OS 8.x 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

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 8, but your programs will be much easier to maintain if you avoid patching entirely.

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. If you were using this technique for custom dragging or zooming feedback, use `DragWindow` or other Window Manager or Drag Manager functions instead.

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 will signal the Window Manager not to update the screen until your drawing operation completes. These functions are not yet available.

Manage Memory Efficiently

Memory management doesn't change much for Carbon applications running on Mac OS 8. 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 you'll need to make is in determining amounts of free memory and stack space available.

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 8, 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 8 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'll need to 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.

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 on the Developer Preview 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'll have to 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 functions 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 `Get`*HigherLevelType*`From`*LowerLevelType*. Functions that cast down are named `Get`*HigherLevelTypeLowerLevelType*.

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. Table 2-2 (page 33) provides a summary of accessor functions you can use for common Mac OS data types.

Listing 2-1 shows an example of some typical coding practices that must be modified for Carbon.

Listing 2-1 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 2-1 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 2-2.

Listing 2-2 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.

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)
```

Table 2-2 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`.
`PopupPrivateData`	`mHandle`	Use `Get/SetControlData` with proper tags.
	`mID`	Use `Get/SetControlData` with proper tags.

Table 2-2 Summary of Carbon Human Interface Toolbox Accessors (continued)

Data Structure	**Element**	**Accessor**
Dialog Boxes		
`DialogRecord`	`window`	Use `GetDialogWindow` to obtain the value. There is no equivalent function for setting the value.
	`items`	`AppendDITL, ShortenDITL, AppendDialogItemList, InsertCheckBoxDialogItem, InsertControlDialogItem, InsertEditTextDialogItem, InsertIconDialogItem, InsertPictureDialogItem, InsertPushButtonDialogItem, InsertRadioButtonDialogItem, InsertStaticTextDialogItem, InsertUserDialogItem, RemoveDialogItem, SetDialogItem`
	`textH`	`GetDialogTextEditHandle`
	`editField`	`GetDialogKeyboardFocusItem`
	`editOpen`	`Get/SetDialogCancelItem`
	`aDefItem`	`Get/SetDialogDefaultItem`
Menus		
`MenuInfo`	`menuID`	`Get/SetMenuID`
	`menuWidth`	`Get/SetMenuWidth`
	`menuHeight`	`Get/SetMenuHeight`
	`menuProc`	not supported
	`enableFlags`	`Enable/DisableMenuItem, IsMenuItemEnabled`
	`menuData`	`Get/SetMenuTitle`
Windows		
`WindowRecord` `CWindowRecord`	`port`	Use `GetWindowPort` to obtain the value. There is no equivalent function for setting the value.
	`windowKind`	`Get/SetWindowKind`
	`visible`	`Hide/ShowWindow, ShowHide, IsWindowVisible`

Table 2-2 Summary of Carbon Human Interface Toolbox Accessors (continued)

Data Structure	Element	Accessor
	`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`
`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 `refCon`s.

Table 2-2 Summary of Carbon Human Interface Toolbox Accessors (continued)

Data Structure	**Element**	**Accessor**
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`
	`lastClick`	`SetListLastClick`
	`refCon`	`Get/SetListRefCon`
	`listDefProc`	not supported
	`userHandle`	`Get/SetListUserHandle`
	`dataBounds`	`GetListDataBounds`
	`cells`	`LGet/SetCell`
	`maxIndex`	`LGet/SetCell`
	`cellArray`	`LGet/SetCell`

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.

C H A P T E R 3

Building Carbon Applications

Contents

This chapter describes how to use the tools and libraries provided with the Mac OS X Developer Preview CD to build Carbon applications for both Mac OS 8 (version 8.1 or later) and Mac OS X.

Platform-Specific Considerations

This section discusses some key differences between Mac OS 8 and Mac OS X, and how they may affect your choice of development environments.

Object File Formats: CFM and Mach-O

On Mac OS 8, Carbon applications use the CFM runtime architecture, in which code fragments are stored in PEF containers and managed by the Code Fragment Manager.

While Mac OS X supports the Code Fragment Manager and the CFM runtime architecture for Carbon applications, it also supports the Mach object file format, known as Mach-O.

For Carbon applications that run on Mac OS 8, you must use the CFM object file format. On Mac OS X, however, you may choose to create a Mach-O executable.

Besides being more familiar to Mac OS 8 developers, an advantage of CFM is that a single executable file will run on both Mac OS 8 and Mac OS X. CFM also provides support for existing plug-in architectures.

The primary advantage of Mach-O for Carbon developers is that it is currently the best format for debugging your application on Mac OS X. Project Builder, Apple's integrated development environment for Mac OS X, supports symbolic debugging of Mach-O executables using the GDB debugger.

Metrowerks is preparing a two-machine debugger for Mac OS X that may provide another option for Carbon developers. Contact Metrowerks for more information.

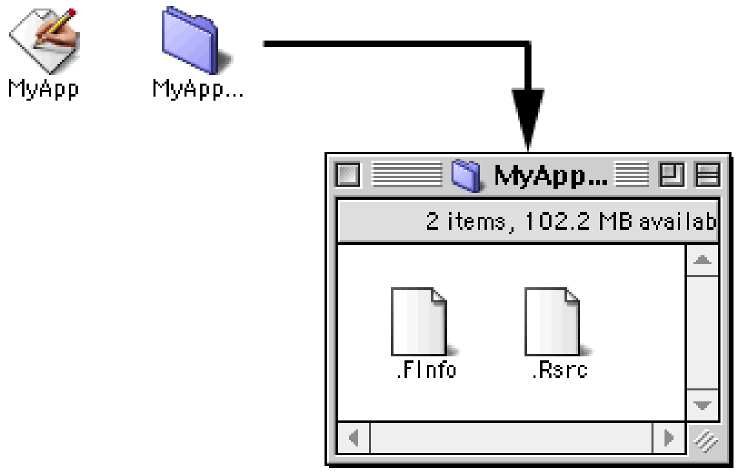
File System Formats: UFS and HFS Plus

Mac OS X currently uses the UFS volume format for the disk or partition containing its system files. Because Mac OS 8 systems (including the Blue Box) cannot directly access UFS disks, you'll probably find it more convenient to do your Carbon development and testing on an HFS Plus disk.

You can use the Workspace Manager on Mac OS X to transfer files between UFS and HFS Plus disks, or you can transfer files over a network. However, if you move an application from an HFS Plus disk to a UFS disk, the file's resource fork and Finder information will not be copied, because UFS does not provide built-in support for these features.

To support development and testing of Carbon applications on UFS disks, Apple has defined a directory structure and naming conventions for the files that contain an application's resources and Finder information. This standard, which is depicted in Figure 3-1, requires that you create a directory with the same name as your application but ending with three periods (...). Inside this directory you must have a file named `.Finfo` that contains the Finder information, and optionally a file named `.Rsrc` containing your application's resources.

Figure 3-1 Folder hierarchy for resources and Finder info on UFS disks



The "UFS Converter" tool, provided on the Developer Preview CD, is a drag-and-drop application that you can use to create a resource folder and files for your application. You can then transfer the application and its resource folder to a UFS disk and launch your program.

Native Mac OS 8 vs. Blue Box

If you plan to build, run, and debug Carbon applications for both Mac OS 8 and Mac OS X on a single system, 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 8 system (that is, a computer running Mac OS 8 instead of Mac OS X), as your development tools are likely to run somewhat slower in the Blue Box. 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 8 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.

Implementation Issues

This section contains important information about the Mac OS X Developer Preview release. These issues will be resolved in upcoming releases.

Spaces in Filenames on Mac OS X

On Mac OS X you cannot currently launch an application located on an HFS Plus disk if the filename or pathname contains any spaces. If you plan to build or run your application on Mac OS X, you should remove any spaces from the name of the HFS Plus disk and the folders containing your project.

Carbon on Mac OS X

There is currently a disparity between the number of Carbon functions implemented on Mac OS 8 and Mac OS X. Because Mac OS X does not yet include support for all the Carbon functions provided in `CarbonLib` on Mac OS 8, Apple is providing a stub library, `LiteCarbonLib`, that exports only the entry points currently supported on Mac OS X. If you develop on Mac OS 8, you can link against `LiteCarbonLib` to ensure that your application doesn't call any functions that aren't yet supported on Mac OS X.

PreCarbon.o

The object file `PreCarbon.o` is provided for developers who have ported their code to Carbon but also want to create a version of their application that does not require the `CarbonLib` shared library at runtime. You will need to conditionalize some of your sources (for example, you'll need to initialize Human Interface Toolbox managers when building a pre-Carbon version, and you'll have to package your defprocs as resources instead of using the new custom definition creation functions), but `PreCarbon.o` will make the job of back-porting easier.

If you are building a Carbon application you should not use `PreCarbon.o`. Instead, you should link to the `CarbonLib` shared library. `PreCarbon.o` does not include all the functions in `CarbonLib`, and Apple will not be actively supporting or encouraging developers to use `PreCarbon.o`. Table 3-1 lists the Carbon functions implemented in `PreCarbon.o`.

Table 3-1 Functions in PreCarbon.o

AEGetDescData	AEGetDescDataSize
CreateNewPort	DisableMenuItem
DisposePort	EnableMenuItem
GetControlBounds	GetControlColorTable
GetControlDataHandle	GetControlDefinition
GetControlHilite	GetControlOwner
GetControlPopupMenuHandle	GetControlPopupMenuID
GetDialogCancelItem	GetDialogDefaultItem
GetDialogFromWindow	GetDialogKeyboardFocusItem
GetDialogPort	GetDialogTextEditHandle
GetDialogWindow	GetGlobalMouse
GetListActive	GetListCellIndent
GetListCellSize	GetListClickLocation
GetListClickLoop	GetListClickTime
GetListDataBounds	GetListDataHandle
GetListDefinition	GetListFlags
GetListHorizontalScrollBar	GetListMouseLocation
GetListPort	GetListRefCon
GetListSelectionFlags	GetListUserHandle
GetListVerticalScrollBar	GetListViewBounds
GetListVisibleCells	GetMenuHeight
GetMenuID	GetMenuTitle

Table 3-1 Functions in PreCarbon.o (continued)

GetMenuWidth	GetParamText
GetPixBounds	GetPixDepth
GetPortBackColor	GetPortBackPixPat
GetPortBackPixPatDirect	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
GetTSMDialogPtr	GetTSMDialogTextEditHandle
GetWindowFromPort	GetWindowGoAwayFlag
GetWindowKind	GetWindowPort
GetWindowPortBounds	GetWindowSpareFlag
GetWindowStandardState	GetWindowUserState
InvalWindowRect	InvalWindowRgn
IsControlHilited	IsPortOffscreen
IsPortPictureBeingDefined	IsPortRegionBeingDefined
IsRegionRectangular	IsWindowHilited
IsWindowUpdatePending	IsWindowVisible
SetControlBounds	SetControlColorTable
SetControlDataHandle	SetControlOwner
SetControlPopupMenuHandle	SetControlPopupMenuID
SetListCellIndent	SetListClickLoop
SetListClickTime	SetListFlags

Table 3-1 Functions in PreCarbon.o (continued)

`SetListLastClick`	`SetListPort`
`SetListRefCon`	`SetListSelectionFlags`
`SetListUserHandle`	`SetListViewBounds`
`SetMenuHeight`	`SetMenuID`
`SetMenuTitle`	`SetMenuWidth`
`SetPortBackPixPat`	`SetPortBackPixPatDirect`
`SetPortBounds`	`SetPortClipRegion`
`SetPortDialogPort`	`SetPortFracHPenLocation`
`SetPortGrafProcs`	`SetPortOpColor`
`SetPortPenMode`	`SetPortPenPixPat`
`SetPortPenPixPatDirect`	`SetPortPenSize`
`SetPortPrintingReference`	`SetPortVisibleRegion`
`SetPortWindowPort`	`SetQDGlobalsArrow`
`SetQDGlobalsRandomSeed`	`SetTSMDialogDocumentID`
`SetTSMDialogTextEditHandle`	`SetWindowKind`
`SetWindowStandardState`	`SetWindowUserState`
`ValidWindowRect`	`ValidWindowRgn`

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 8 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 8 and Mac OS X. The only difference is that you'll include the `CarbonLib` shared 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 8. 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 8.

Instructions for getting started with Project Builder are provided separately in "Using Project Builder to Build and Debug a Carbon Application" (MacOSXDevTools.pdf) on the Developer Preview CD.

Building a CFM Carbon Application with CodeWarrior

If you plan to use Metrowerks CodeWarrior, we recommend CodeWarrior Pro version 4.0 or later. The Carbon project stationary provided on the Developer Preview CD requires CodeWarrior Pro 4.0, but if you want to use an earlier version of CodeWarrior you can use the BasicApp sample project, which builds with CodeWarrior Pro 2.0 or later, as a starting point for your project.

You can run CodeWarrior on either a native Mac OS 8 system or in the Blue Box 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 Blue Box.

Preparing Your Development Environment

Before you start Carbon development with CodeWarrior, you'll need to install the tools and libraries provided with the Developer Preview CD.

1. Copy the `Carbon Support` folder from the Developer Preview CD to the `CodeWarrior Pro 4:Metrowerks CodeWarrior` folder on your hard disk. The `Carbon Support` folder must reside in the same folder as the CodeWarrior IDE application.
2. Copy the Carbon project stationary folder from the Developer Preview CD to your `Metrowerks CodeWarrior:(Project Stationery)` folder.
3. Copy the appropriate shared libraries from your `CodeWarrior Pro 4: Metrowerks CodeWarrior:Carbon Support:CarbonLib` folder to your Extensions folder. Note that you may want to keep only one type of Carbon library in your Extensions folder at any time to ensure that the Code Fragment Manager selects the correct library at runtime.
   - `CarbonLib` is the standard implementation of Carbon for Mac OS 8.1 or later.
   - `DebuggingCarbonLib` is a debugging version of `CarbonLib`.

Building Your Application

To build a Carbon version of your application:

1. Open your project in CodeWarrior and add the following statement to one of your source files before including any of the Carbon headers:

   ```
   #define TARGET_CARBON 1
   ```

   The `TARGET_CARBON` conditional specifies that the included header files should allow only Carbon-compatible APIs and data structures. You can include this 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 Developer Preview CD make use of a prefix file (usually `CarbonPrefix.h`) to define `TARGET_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` library to your project.
3. Press the Make button.

Running Your Application on Mac OS 8

To launch your application from the Finder on a Mac OS 8 system (version 8.1 or later), you must install the `CarbonLib` or `DebugCarbonLib` shared library in the System Folder or in the same folder as your application.

Running Your Application on Mac OS X

As long as your application resides on an HFS Plus disk, you can launch it from the Mac OS X Workspace Manager by double-clicking its icon. If you move your application to a UFS disk, you'll need to create a resource directory as described in "File System Formats: UFS and HFS Plus" (page 43). You cannot launch applications from a standard HFS format disk on Mac OS X.

IMPORTANT
In the Developer Preview release of Mac OS X you cannot execute a file on an HFS Plus disk if the filename or pathname contains any spaces. Make sure that your HFS Plus volume name and your project folder names do not contain spaces. ▲

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:

`/usr/Carbon/bin/LaunchCFMApp` *filename*

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

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 8, 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

If the application is located on an HFS Plus disk

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

IMPORTANT

In the Developer Preview release of Mac OS X you cannot execute a file on an HFS Plus disk if the filename or pathname contains any spaces. Make sure that your HFS Plus volume name and your project folder names do not contain spaces. ▲

If the application is located on a UFS disk

If you move your application to a UFS disk, you'll need to create a resource directory as described in "File System Formats: UFS and HFS Plus" (page 43). You can then transfer the application file and resource directory to a UFS disk using FTP, or by copying the file in the Workspace Manager if both disks are on the same computer.

You can launch your application from the Workspace Manager by double-clicking its icon, or from a terminal window by typing the application's name (including pathname if the file is not in the current directory).

If the file does not launch using either of these methods, make sure that the executable permissions are set properly. With the file selected in the Workspace Manager, type Command-4 or choose Tools>Inspector>Access from the menu. Place checkmarks in each of the Execute boxes if they are not already checked. Click OK and close the Inspector window.

Building a Mach-O Carbon Application with Project Builder

Project Builder and its documentation are included on the Mac OS X Developer Preview CD. Instructions for building Carbon applications with Project Builder are provided in "Using Project Builder to Build and Debug a Carbon Application" (MacOSXDevTools.pdf).

Debugging Your Application

You can debug Carbon applications on Mac OS 8 using the Metrowerks debugger. You can also use this debugger with two networked machines, one running Mac OS 8 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 or the Project Builder development environment. However, GDB provides only limited support for CFM applications at this time. The techniques for debugging Carbon applications with GDB are described in "Using Project Builder to Build and Debug a Carbon Application" (MacOSXDevTools.pdf) on the Developer Preview CD.