Porting CodeWarrior Projects to Xcode

Tools > Xcode

2006-10-26

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

Simultaneously published in the United States and Canada.

Contents

Introduction Introduction to Porting CodeWarrior Projects to Xcode 9

Prerequisites 9
Further Reading 10
Organization of This Document 10
Feedback and Mail List 11

Chapter 1 Xcode From a CodeWarrior Perspective 13

The Basic Development Environment 13
- Some Special Features of Xcode 14
- The Project Window 14
- Customizing the Environment 17

Limitations 19
Companion Applications 19
Header Files 20
Framework-Style Headers 20
Cross-Development 21
Precompiled Headers and Prefix Files 21
Pragma Statements 22
C and C++ Libraries 25
- Runtime and Library Issues 25

Support for wchar_t and wstring 26
The GCC Compiler 26
- More on GCC Compiler Versions 28
- Additional Compiler Information 29

C++ Code in C Files 29
Inline ASM 29
The Linker 31
- Dead Code Stripping 31

The Information Property List and .plc Files 32
Working With Resources 32
Building Code 33
- Native Build System 34
- Build Configurations 34
- Creating Debug and Non-Debug Products 35
- Automating the Build Process 36
- Makefiles 37

Exporting Symbols 37
Debugging 38
Prebinding 39
Source Trees 40
Source Control 41
PowerPlant 41
Maintaining Parallel Projects in Xcode and CodeWarrior 41
Executable Format 41
Framework and System Headers 42
Using Precompiled Headers and Prefix Files 42
Property Lists 43
Code Differences 43
Linking 44

Chapter 2 Preparing a CodeWarrior Project for Importing 45

Convert Classic Applications to Use Carbon 45
Use the Application Package Type 45
Build the Application in the Mach-O Format 46
Convert to Mach-O Format 46
Use Framework Headers 47
Use C99 Standard in Language Settings 47
Migrate from MSL to System C and C++ Libraries 48
Replace Your Prefix File 48
Test Your Mach-O Target 48
Remove Unnecessary Targets From the Project 48

Chapter 3 Importing a CodeWarrior Project Into Xcode 49

About the Importer 49
For Best Results When Importing 49
Known Issues With the Importer 50
Importing a Project 50

Chapter 4 After Importing a Project 53

Build Your Project 53
Check for Common Conversion Issues 53
Resolve Undefined Symbols 54
Make Code Changes for GCC Compatibility 54
Conform to the C99 Standard 54
Inlining Thresholds 55
Dealing With Pragmas 55
The Mach-O bool Type is Four Bytes, Not One 57
Guarding Header Includes for C++ 58
Friend of Template Specialization Can't Access Private Members 58

Float to Integer Conversions Give Incorrect Output 58
Some Casts Don't Work in Function Lists 59
Standard C Library Functions Are in Global Namespace 59
Vector Load Indexed Intrinsic Is Not Const Correct 59
CodeWarrior Allows Anonymous Unused C Function Arguments 59
GCC Interprets Some #define Values Differently Than CodeWarrior 59
GCC asm Intrinsics are Preprocessor Macros 60
Xcode's Rez tool Doesn't Support Certain Operators 60
Move Settings From the .plc File to an Info.plist File 61
Make Changes to PowerPlant 62
Add PowerPlant Headers to the Project and Target 63
Create a Prefix File for PowerPlant 63
Make Minor Changes to the PowerPlant Code 65

Chapter 5 Using PowerPlant in Universal Binaries 69

Changing LStream Code 69
LStream.h 70
LStream.cp 72
LControl.cp 72
LListBox.cp 73
LPane.cp 73
LPrintout.cp 74
LScroller.cp 74
LTable.cp 74
LView.cp 75
LWindow.cp 75
LPopupGroupBox.cp 75
LControlView.cp 75
LScrollerView.cp 76
LPageController.cp 76
Flipping the 'DBC#' Resource Type 76

Chapter 6 Where to Go From Here 79

After Moving Your Project to Xcode 79

Document Revision History 81

Figures and Listings

Chapter 1 Xcode From a CodeWarrior Perspective 13

Figure 1-1 Xcode project window for a simple Carbon application 15
Figure 1-2 Inspector window showing Language settings in Build pane 17
Figure 1-3 Some GCC build settings in an inspector window 23
Figure 1-4 Adding compiler flags for one or more files 24
Figure 1-5 Warnings settings in an inspector window 28
Figure 1-6 The Configurations pane 35
Figure 1-7 A debug window in Xcode 39
Listing 1-1 A typedef with the same name as an op code 30
Listing 1-2 Preprocessor directives for isolating compiler specific code 43

Chapter 3 Importing a CodeWarrior Project Into Xcode 49

Figure 3-1 The Import CodeWarrior Project pane 51

Chapter 4 After Importing a Project 53

Figure 4-1 The Properties pane in a target editor 62
Listing 4-1 A structure definition that is interpreted differently 56
Listing 4-2 Code that generates a size incompatibility 56
Listing 4-3 Code that generates a register difference 57
Listing 4-4 Code that generates error 59
Listing 4-5 `#define` test code 60
Listing 4-6 An Xcode prefix header file for PowerPlant 63
Listing 4-7 Modified switch statement in `LGATabsControlImp.cp` 66
Listing 4-8 `LStream.h` modifications at line 152 66
Listing 4-9 `LStream.h` modifications at line 172 67
Listing 4-10 `LException.h` modifications at line 33 67
Listing 4-11 `LException.cp` modifications at line 94 68
Listing 4-12 `LDebugStream.cp` modification at line 1150 68
Listing 4-13 Modification to `LCommanderTree.cp` at line 87 (and to `LPaneTree.cp` at line 81) 68

Chapter 5 Using PowerPlant in Universal Binaries 69

Listing 5-1 Calls that may require swapping bytes 69
Listing 5-2 Code that flips the `'DBC#'` resource type 77

Introduction to Porting CodeWarrior Projects to Xcode

This document describes how to move Mac OS software projects from CodeWarrior to Xcode, the Apple integrated development environment. It lists similarities and differences between the two environments, describes how to import a CodeWarrior project into Xcode, and provides detailed information on many conversion issues.

The Xcode application is part of the developer tools distributed with Mac OS X version 10.3 and later. It provides a powerful user interface to many industry-standard and open-source tools, including GCC, `javac`, `jikes`, and GDB. Xcode contains templates for creating applications, frameworks, libraries, plug-ins, Java applications and applets, and command-line tools. Xcode supports both Cocoa and Carbon development, using C, C++, Objective-C, and Java.

Although this document generally describes how to convert CodeWarrior projects that build applications, much of the information can be applied to projects that build plug-ins, bundles, frameworks, and other kinds of software.

Note: This document assumes that your Xcode project will use the GCC 4.0 compiler, available with Xcode 2.0 and later. Xcode 2.0 requires Mac OS X version 10.4 or later.

CodeWarrior descriptions and examples in this document are based on CodeWarrior Pro version 8.3 for Macintosh.

Prerequisites

This document is intended for CodeWarrior users, and assumes that you have some familiarity with the Mac OS, including Mac OS X.

- For detailed information on the development tools available with Xcode, see Mac OS X Developer Tools in *Mac OS X Technology Overview*.

  Among other new and revised documents, the Tools Documentation includes updated GCC documentation: *GNU C/C++/Objective-C 4.0.1 Compiler User Guide* and *GNU C 4.0 Preprocessor User Guide*.

- For introductory information on Mac OS X, see *Mac OS X Technology Overview* .

Note: These documents are part of the Apple Developer Documentation installed on your system with the developer tools. They're accessible through Xcode, and are also available at http://developer.apple.com.

The primary documentation for performing operations with Xcode is *Xcode User Guide*.

Further Reading

The following documents provide information on moving other kinds of software to Mac OS X.

- **UNIX or Linux software**
  - *Porting UNIX/Linux Applications to Mac OS X*.
  - Technical Note 2071: Porting Command Line UNIX Tools to Mac OS X
- **Windows software**
  - *Porting to Mac OS X from Windows Win32 API*

You can find additional information about porting code to Mac OS X in the Porting Documentation area.

Organization of This Document

This document contains the following:

- This Introduction describes the audience for the document and summarizes the contents.
- "Xcode From a CodeWarrior Perspective " (page 13) describes similarities and differences in key features of Xcode and CodeWarrior. It also describes how to use certain Xcode features.
- "Preparing a CodeWarrior Project for Importing" (page 45) describes steps you can take to modify your CodeWarrior project before importing it into Xcode.
- "Importing a CodeWarrior Project Into Xcode" (page 49) provides a brief walk-through of importing a CodeWarrior project.
- "After Importing a Project" (page 53) describes steps you may need to take to successfully build an imported CodeWarrior project in Xcode.
- "Using PowerPlant in Universal Binaries" (page 69) describes modification you may need to make to build universal binaries that use PowerPlant.
- "Where to Go From Here" (page 79) points to some tools and performance documents you'll want to consider as you work on your Mac OS X software.

Feedback and Mail List

You can get quick answers to your day-to-day Xcode questions by sending email to xcode-users@lists.apple.com. You can become a member of this list at Apple Mailing Lists.

Your feedback and suggestions for Xcode are welcome. For feedback on this document, use the link at the bottom of each page. To request a feature or report a bug in Xcode, use the Apple Bug Reporter.

To report bugs or to receive the bi-weekly Apple Developer Connection News email newsletter, you must be a member of Apple Developer Connection (ADC). You can sign up for a free ADC Online membership.

Xcode From a CodeWarrior Perspective

This chapter introduces key features of Xcode from the perspective of CodeWarrior users. Understanding the similarities and differences in these features should help you put your CodeWarrior experience to work in Xcode. It will also be useful in converting your CodeWarrior projects.

Important: This chapter concentrates on differences between Xcode and CodeWarrior, and does not provide a comprehensive overview of Xcode. For a more complete feature list, see *Xcode User Guide*. For a brief tutorial introduction to Xcode, see *Xcode Quick Tour Guide*.

Though there are many minor differences between Xcode and CodeWarrior, you'll find that Xcode supports most of the features CodeWarrior users are familiar with. Xcode also provides a great deal of flexibility in organizing the environment for the way you like to work, as described in "Customizing the Environment" (page 17).

The Basic Development Environment

Like CodeWarrior, Xcode is focused on the use of projects, targets, and files. But it also takes advantage of some of the popular user interface features found in iTunes and other Apple consumer applications, such as useful grouping of files and other items, fast searching for items, and a clean, relatively simple interface.

The development environments for Xcode and CodeWarrior contain the same major components, including project window, text editor, build system, debugger, symbol navigation, find facilities, and help system. One significant difference is that Xcode includes a number of open source components, including the GCC compiler and GDB debugger, as well as other tools written by the UNIX open-source community.

Behind the user interface, Xcode performs many tasks by launching command-line tools as Mach processes. That allows Xcode to implement features such as distributed and multiprocessor builds, where each GCC compile can be handled by a separate CPU, even on separate machines.

Some Special Features of Xcode

Xcode supports several features with no exact equivalent in CodeWarrior, and others that put a new twist on everyday features. For details on how to use these features, as well as possible limitations, see *Xcode User Guide*.

Making life easier for many common tasks:

- **Fast searching** allows you to quickly find a file, symbol, warning, error, or other item in a list by filtering out any items that don't match the characters you type. You can search using string matching, regular expressions, or wildcard patterns.
- **Smart groups** let you intelligently organize your project's content and data, using groups that match a certain rule or pattern. For example, built-in smart groups include the Errors and Warnings group and the Find Results group.
- **Inspectors** allow you to examine and edit the objects in your project.
- **Integrated Mac OS X API documentation** provides full access to all installed Mac OS X documentation, with rapid API searching by language. It includes many searching and viewing options to help find the programming information you need. Automatic detection of documentation updates lets you download the latest technical information from Apple.

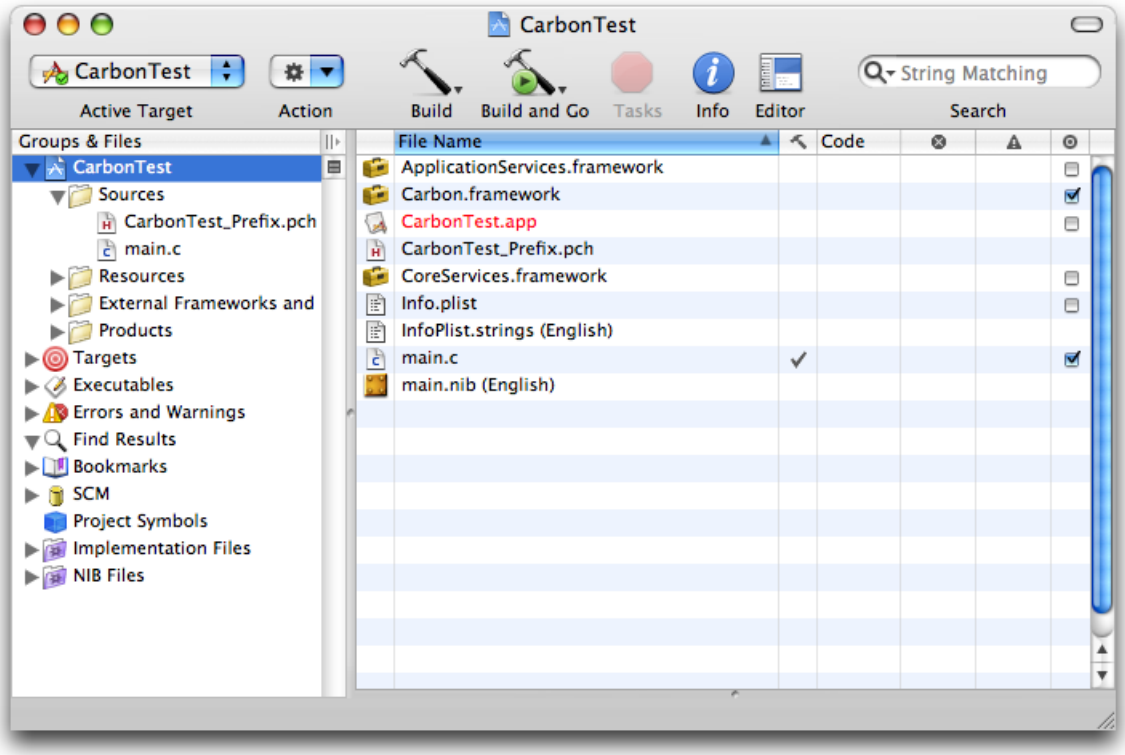
For improving your building and debugging experience:

- **Distributed builds** can dramatically reduce build time for large projects by distributing compiles to available computers on the network.
- **ZeroLink** shortens link time for development builds and allows you to very quickly relaunch your application after making changes.
- **Fix and Continue** improves your debugging efficiency by allowing you to change the source in a file, recompile just that file, and run the changed code without stopping the current debugging session. (The Fix icon looks like a tape dispenser.)

The Project Window

Figure 1-1 shows the Xcode project window for a simple Carbon application.

Figure 1-1 Xcode project window for a simple Carbon application



You can get a complete listing of Xcode features in *Xcode User Guide*, but the following list highlights some of the most frequently used interface features.

- Instead of tabs (as in CodeWarrior and Project Builder), the default project window in Xcode uses a hierarchical list of groups in the Groups & Files list. Xcode includes several different project window layouts, however, including one that provides a tabbed interface to your project's contents.
- The contents of the selected item or items in the Groups & Files list are listed in a detail view.
- Text you type in the Search field is used to filter items in the detail view. The Search field supports simple string matching, regular expressions, and wildcard patterns.
- For source files, the detail view shows build status, SCM status, size, errors, warnings, and so on.
- Some of the items that can appear in the Groups & Files list include:
  - The project group, at the top of the Groups & Files list, shows all of the files, frameworks, and libraries included in the project. The previous figure shows the project group "CarbonTest."

    Source groups, identified by yellow folder icons, can contain other source groups. They help you organize the files in your project into more manageable chunks.
  - The Targets group contains all the targets in the project.
  - The Executables group contains all of the executable environments defined in the project. Executable environments specify the executable file that is launched when you run or debug; typically, this corresponds to a product that the project builds.

The following items in the Groups & Files list are smart groups:

- The Errors and Warnings group lists the errors and warnings generated when you build your project.
- The SCM group lets you view the project files currently under version control, as well as their current state.
- The Bookmarks group contains all of the locations you have saved as bookmarks for this project.
- The Find Results group contains the results of any searches you perform in your project. Each search creates a new entry in this group.
- The Project Symbols group lists all of the symbols defined in your project.
- The NIB Files group contains any `.nib` files used to create your product's user interface.
- The Implementation Files group contains all of the implementation files in your project, such as those ending in `.cpp`, `.c`, and `.m`.

You can define your own smart groups and the rules for what they contain. User-defined smart groups are indicated with a purple folder icon.

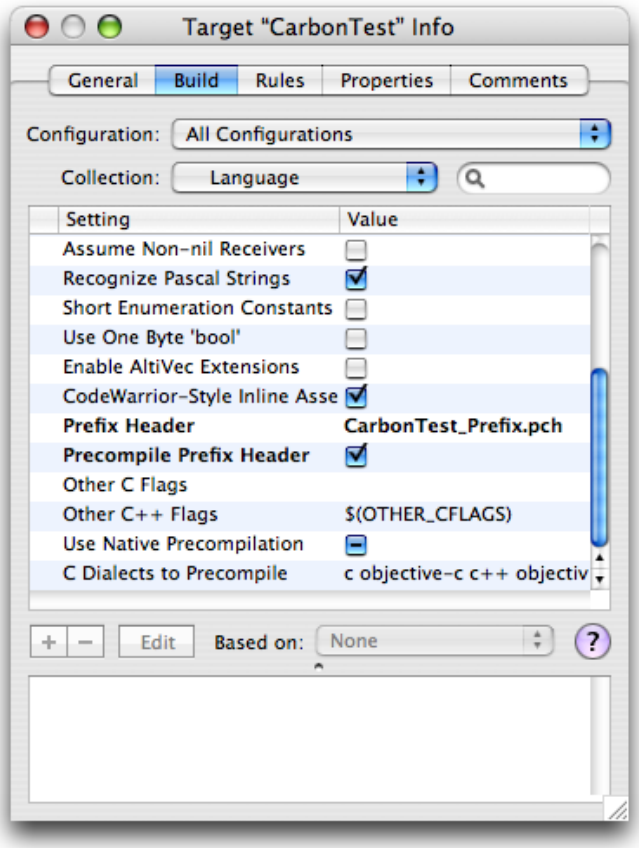
- Inspector windows and Info windows handle much of the interesting work, allowing you to easily display and edit project data. (These windows differ in that inspector windows are floating utility windows that track the current selection, while Info windows are standard windows that continue to display information about the same item.)

  Among other things, you can use these windows to:

  - display per-project settings
  - display per-target settings

    Figure 1-2 shows the target inspector. A useful feature of the build settings interface is a section below the build settings table that provides a description of the currently selected build setting. If this section isn't visible, you can display it by dragging the separator bar.

Figure 1-2 Inspector window showing Language settings in Build pane



- display per-file settings and information, such as location, encoding, tab settings, line endings, and so on
- select multiple project items—such as files or targets—and change settings on all of them (for settings for which this makes sense)

- The Find string is persistent across dialogs (and even many Mac OS applications).
- Xcode uses shell paths (slash-delimited) exclusively, not colon-delimited paths.
- Xcode supports per-file compiler settings.
- The Xcode editor supports code completion; as you type identifiers and keywords, Xcode suggests likely matches based on the current context.

Customizing the Environment

If your fingers are hard-wired for CodeWarrior keystroke equivalents (or BBEdit, or even MPW), don't worry. Xcode provides many features you can customize to fit the way you like to work. These include all the standard features you can set up in the Xcode Preferences window, such as indentation style, syntax coloring, Source Code Management (SCM) options, and so on.

Xcode provides additional control over your environment with the following features:

- You can customize Command-key equivalents for menu items, as well as keystroke equivalents for editing tasks, to use the keystrokes you are most familiar with. Xcode provides predefined sets that are compatible with BBEdit, CodeWarrior, and even MPW (Macintosh Programmer's Workshop, the Apple development environment for Mac OS 9). To learn more about customizing keystroke equivalents, see Customizing Key Equivalents in *Xcode User Guide*.
- You can select from a number of different project window layouts, to choose the configuration that works best for you. These layouts are:
  - Condensed. This layout provides a project window that contains tabs for Files, Targets, and all other smart groups. You use separate windows for tasks other than project navigation. This layout should be the most familiar to CodeWarrior users.
  - All-In-One. This layout provides a single window for all of the tasks typical of software development, such as debugging, viewing build results, and searching.
  - Default. This layout provides the default Xcode project window, described in "The Project Window" (page 14).

  For more on project window layouts, see The Project Window in *Xcode User Guide*.
- You can choose what information Xcode displays in the project window, by customizing the set of smart groups that appear there. You can rearrange, show, and hide any of the groups that appear in the project window.
- You can choose the editors to use for files in your project. You can use the Xcode editor or use an external editor such as BBEdit or Emacs. Or you can choose to open a file with the application chosen for it by the Finder.
- You can easily customize the toolbar for any project window, adding or removing items, or choosing the default set.
- You can conveniently use shell scripts in Xcode in several ways:
  - You can select text in any file and execute it as a shell command.
  - You can have Xcode run a startup shell script each time you launch it.

    A default version of the startup script creates the User Scripts menu from any available menu definition files (which are just shell scripts). Xcode provides default menu definition files that add useful items to the User Scripts menu and demonstrate how to work with menu scripts.

    You can customize the built-in startup script and menu definition files, or you can create your own.
  - Xcode provides a number of built-in script variables and utility scripts you can use in menu scripts or other user scripts.
  - You can add a shell-file build phase that lets you add the execution of shell script files to the build process for a target. (Select a target and choose Project > New Build Phase > New Run Script Build Phase.)

For details on how to customize, see Customizing Xcode in *Xcode User Guide*.

Limitations

Throughout this document, you will find comparisons between CodeWarrior and Xcode, including differences, issues, and in many cases, alternate approaches. This section lists some Xcode limitations to be aware of, with links to further information where available.

- Compile and build speed are greatly improved, and will get faster, but do not yet equal CodeWarrior.
- GCC provides limited support for CodeWarrior pragmas.

  See "Pragma Statements" (page 22).
- There is no support for `wchar_t` and `wstring` prior to Mac OS X version 10.3 (Panther).

  See "Support for wchar_t and wstring" (page 26).
- There is no projectwide graphical tree browser; however, Xcode 2.0 and later include a class modeling tool that you can use to make individual graphical models of your project's classes.

While not strictly a limitation, there are many differences between GCC and the Metrowerks compilers. Information on these differences is provided in many places in this document. For a jumping-off point, see "The GCC Compiler" (page 26).

Companion Applications

CodeWarrior provides the Constructor application for designing user interfaces based on the PowerPlant framework. Xcode provides the Interface Builder application for creating graphical user interfaces for Mac OS X applications. Interface Builder works with both Carbon and Cocoa applications and lets you lay out interface objects (including windows, controls, menus, and so on), resize them, set their attributes, and make connections to other objects and to source code. The layout tools in Interface Builder include built-in support for the Aqua interface guidelines. You can, of course, continue to use Constructor even when moving your code to Xcode.

To learn more about using Interface Builder with Carbon applications, see *Unarchiving Interface Objects With Interface Builder Services* and *Interface Builder Services Reference*.

CodeWarrior provides the Profiler application for examining the behavior of a running application and fine-tuning performance. Xcode Tools includes the Shark and Sampler profiling tools. For tracking memory usage and bugs, Xcode provides the MallocDebug application, which helps debug memory problems that you would debug on Mac OS 9 with CodeWarrior's ZoneRanger. To learn more about performance optimization and available tools, see *Performance Overview*.

Note: For descriptions of the wide range of development tools available with Xcode, see Mac OS X Developer Tools in *Mac OS X Technology Overview*, as well as the documents available in the Tools Documentation area.

Header Files

For a CodeWarrior project, you don't have to explicitly add header files. Instead, you can add recursive access paths, and as long as the header files are somewhere on a specified path, CodeWarrior will find them. (You may notice that this causes a delay on opening projects and can cause errors if you have different headers with the same name.)

Xcode also supports recursive search paths; however, it is generally recommended that your project include explicit references to its header files. For example, when you add a `.c` file to your Xcode project, you don't typically add a search path for the associated header file. Instead, you drag the actual header file into a group in the Groups & Files list in the project to add a reference to the header.

The Build pane in the target and project inspector windows includes build settings for Header, Framework, and Library search paths. By default, these paths are empty; access paths defined in your CodeWarrior project are not brought over by the importer. If you specify a search path in one of these build settings, Xcode uses that path to locate header files during compiling and linking.

To have Xcode search the contents of the entire directory tree located at a given path, select the Recursive option next to that path. This option is not enabled by default when you add a new search path. You should be aware, however, that using recursive search paths can result in longer build times and cause problems if you have multiple files with the same name. For more on using search paths in Xcode, see Build Settings in *Xcode User Guide*.

Framework-Style Headers

One important transition in moving to Mac OS X development is the switch from including individual headers to including framework-style headers in your source code files. A framework is a type of bundle that packages shared resources, such as a dynamic shared library and its associated resource files, header files, and reference documentation. An umbrella framework is a framework that includes a number of related frameworks. The Carbon framework (`Carbon.framework`) is an umbrella framework that contains just one header, `Carbon.h`. That header in turn includes headers from many additional frameworks, such as Core Services, HIToolbox, and others.

A framework-style include statement specifies a framework (typically an umbrella framework) and its main header file. For example, a Carbon source file should use the statement `#include <Carbon/Carbon.h>`, rather than separate include statements for each individual header file it may require. Using framework-style includes will help ensure that your project builds correctly and remains reliable over time.

Note: You can add system frameworks by dragging them into the Groups & Files list of your Xcode project (from `/System/Library/Frameworks`), or with the Project > Add to Project menu item.

For related information, see "Header Files" (page 20), "Precompiled Headers and Prefix Files" (page 21), "Cross-Development" (page 21), "Use Framework Headers" (page 47), and "Use C99 Standard in Language Settings" (page 47). For more information on frameworks in Mac OS X, see *Framework Programming Guide*.

Cross-Development

Cross-development refers to the ability to develop software that can be deployed on, and take advantage of features from, specified versions of Mac OS X, including versions different from the one you are developing on. CodeWarrior provides some support for cross-development for versions of the Mac OS through Universal Interfaces, but those headers have not been updated for recent versions of Mac OS X.

Xcode supports cross-development for various versions of Mac OS X. You can specify which version of Mac OS X headers and libraries to build with, as well as the earliest Mac OS X system version on which the software will run.

To use cross-development for a target in an Xcode project, you make two selections:

- In the General pane of the inspector window for the project group, you use the Cross-Develop Using Target SDK pop-up to select an OS version to develop for, such as Mac OS X 10.4.0. All targets in the project are built as though you were building in that version of the operating system.
- In the Build pane of the inspector window for the target or project, you choose a Mac OS X deployment version, such as 10.2. For this target or project, this specifies the earliest Mac OS X system version on which your software will run.

You can also use the SDK headers and libraries to support cross-development in CodeWarrior projects. For details, see "Maintaining Parallel Projects in Xcode and CodeWarrior" (page 41).

For detailed documentation on cross-development, see *Cross-Development Programming Guide*.

Precompiled Headers and Prefix Files

Precompiled headers are binary files that represent the compiler's intermediate form of the headers required to compile a source file. Generally, all source files in a given target use a large common subset of system and project headers, so by precompiling these headers into intermediate form once and reusing it for many source files, the compiler can build source files more quickly.

CodeWarrior users must manage precompiled headers manually. In CodeWarrior, you can use a precompiled header interchangeably with a normal header file. It can be included in an `#include` directive or used as a prefix header in a target's Target Settings. You can create a precompiled header file using the Precompile menu item, or by including a `.pch` file in your project (you use a `#pragma`

directive in that file to define the name of the precompiled header itself, which usually has a `.mch` suffix). In many larger projects, developers have separate targets (or even whole projects) that just build common precompiled headers that are shared among multiple projects.

In Xcode, precompiled headers are generated automatically and invisibly by the development environment. By setting the Prefix Header build setting of a target to your `.pch` file (or any other header file that contains `#include` statements for your framework headers and common target headers), Xcode will generate the precompiled header file and use it when compiling every source file in that target. (The Precompile Prefix Header build setting must also be enabled.)

The precompiled header is regenerated whenever any files it depends on are changed, so you don't need to manually build or maintain the precompiled header.

You can see the Prefix Header and Precompile Prefix Header build settings in the Build pane in Figure 1-3 (page 23). For more information on using precompiled headers in Xcode, see Optimizing the Edit-Build-Debug Cycle in *Xcode User Guide*.

Pragma Statements

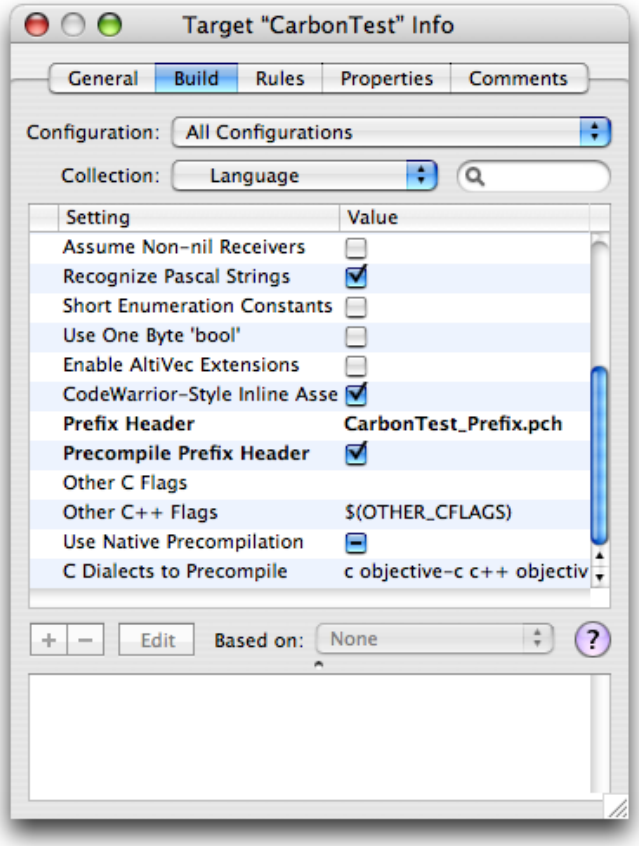
A pragma statement is a directive for providing additional information to the preprocessor or compiler beyond what is conveyed in the language itself. There are a small number of pragma types defined by the C standard; individual compilers may support any number of additional types and values.

The CodeWarrior compiler supports a number of pragmas, including pragma equivalents for most of the compiler settings you can set in the user interface. It also supports some pragmas you can't set in the user interface. Some examples of pragmas CodeWarrior supports are `#pragma unused` and `#pragma once`. In CodeWarrior, pragmas can turn a feature on and off within an individual compilation unit. For example, you can use the pragma statements `#pragma export on` and `#pragma export off` in a source file to bracket any symbols you want to export from a shared library.

Xcode supports some CodeWarrior-defined pragmas. However, Xcode supplies build settings for many of the features the CodeWarrior compiler supports through pragmas. These settings should be set automatically when you import a CodeWarrior project, but you can also add, delete, or modify build settings. Figure 1-3 shows some of the Language settings you can set for a target in the Build pane in an inspector window in Xcode.

Note: When a file is compiled, build settings are passed to the compiler as command line flags. If a GCC compiler option is not available in the Xcode user interface, you can set individual flags with the Other C Flags setting.

Figure 1-3 Some GCC build settings in an inspector window



Although language settings can be set on a file, target, or project basis, you cannot use them for inline control of compiler options. As a result, the smallest unit for which you can set a compiler option is a single source file. To modify build settings on a target-wide basis, select the target in the Groups & Files list and open an Info window (by clicking the Info button or by choosing File > Get Info). Select the Build pane, then make your changes.

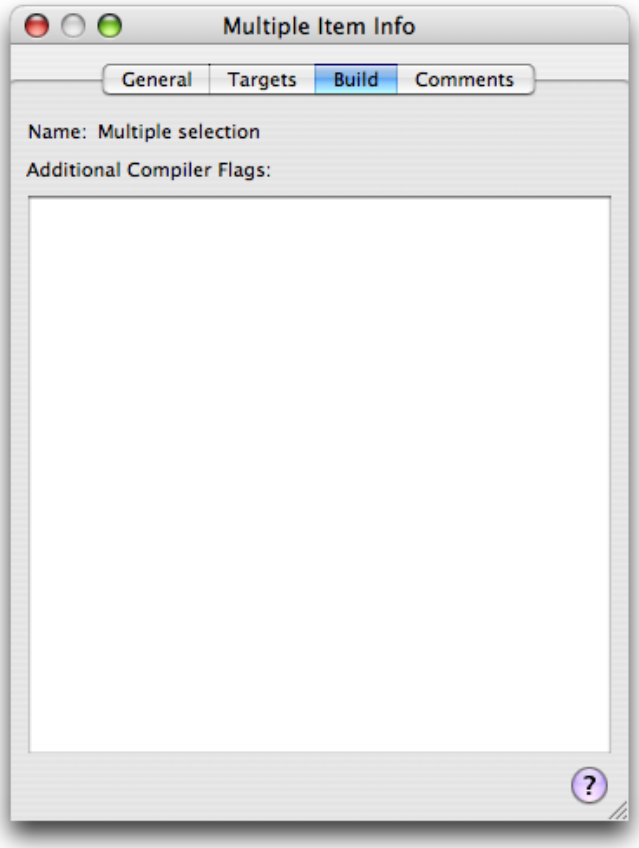
Note: You also modify the settings in an inspector window, which you open by holding the Option key and choosing File > Show Inspector.

Xcode provides a different interface for modifying options for a single file (or several files). You can add additional compiler flags for one or more files with the following steps:

1. Open the target in the Groups & Files list.
2. Open the Compile Sources build phase.
3. Select the file (or files) for which you want to modify the settings.

4. Open an Info window by choosing File > Get Info.
5. Select the Build pane (shown in Figure 1-4 for a multiple selection).

Figure 1-4 Adding compiler flags for one or more files



6. Add any desired compiler flags. Multiple flags should be separated by spaces.

If your CodeWarrior project follows the common pattern of placing pragmas in a header file that is included by other source files, you won't have to make changes to individual files—you can just set the corresponding values for an entire target or a subset of files. However, if you have code that uses pragmas to turn settings on and off within a single file, and if you need to maintain that level of control, you may need to break up some files so that code that shares common build settings is in one file.

For additional information on modifying your source files, see "Maintaining Parallel Projects in Xcode and CodeWarrior" (page 41). For information on using pragmas with respect to identifying export symbols, see "Exporting Symbols" (page 37).

For further information on setting per-target and per-file compiler options in Xcode, see Build Settings in *Xcode User Guide*. For a list of pragmas supported by GCC, see the sections "Pragmas" in *GNU C 4.0 Preprocessor User Guide* and "Pragmas Accepted by GCC" in *GNU C/C++/Objective-C 4.0.1 Compiler User Guide*.

For more information on code changes you may need to make as a result of pragma statements in your CodeWarrior code, see "Create a Prefix File for PowerPlant" (page 63).

C and C++ Libraries

When you create Carbon applications in CodeWarrior, you can link against MSL C and C++ libraries (in debug and nodebug variants) to create CFM-style executables that can run in Mac OS 9 or Mac OS X. There is no system-supplied C library in Mac OS 9, so your application relies on the development environment, which for CodeWarrior, means the MSL libraries.

In Mac OS X, there are system-supplied C and C++ libraries. The C library comes in static and dynamic variants. The dynamic version is strongly recommended for most software, though you may need to link the static version in certain situations (such as for KEXTs, which must load when the dynamic loader isn't available).

Beginning with Mac OS X 10.3.9, the standard C++ library is packaged as a dynamic shared library. In prior versions of Mac OS X, the C++ library is packaged as a static library. Packaging the standard C++ library as a dynamic shared library provides a number of benefits, such as smaller binaries and improved performance. To link against the shared library version, `libstdc++.dylib`, you must use GCC 4.0. If your application must run on versions of Mac OS X prior to 10.3.9, however, you must link against the static library, `libstdc++.a` . To learn more about the C++ runtime and using the shared library version of `libstdc++` , see *C++ Runtime Environment Programming Guide*.

CodeWarrior also supplies MSL libraries to create Mach-O style executables that run on Mac OS X only. In some cases, the MSL library calls through to the Mac OS X System framework library, and in some cases it provides missing features (such as `wchar_t` support, not available in Mac OS X prior to Panther).

If your CodeWarrior project is already building a Mach-O style executable, you should have an easier time in moving it to Xcode. If not, you should consider converting it, as described in "Preparing a CodeWarrior Project for Importing" (page 45). You'll primarily be changing linker settings, access paths, and precompiled headers. In addition, if you load plug-ins via CFM, you'll need to rewrite that code to use the CFBundle or CFPlugin APIs.

Important: Static libraries currently will not be linked in Xcode unless they are named according to the format `lib*.a`. That is, they must start with `lib` and have the extension `.a`.

For related information, see "The GCC Compiler" (page 26).

Runtime and Library Issues

The following are some runtime and library issues you may encounter in switching from MSL libraries to the Mac OS X C and C++ libraries:

- There is no runtime support for `wchar_t` and `wstring` prior to Mac OS X version 10.3 (Panther). For details, see "Support for wchar_t and wstring" (page 26).
- While CodeWarrior supplies nonstandard flags to perform diagnostics on use of the standard template library (STL), Xcode does not include a debug version of the STL.

- Some code will be larger when built with the Xcode standard C and C++ libraries because, unlike the MSL library, they do not currently support container optimization for templates.
- The standard C library function `clock` (or `ctime` for C++) returns a value of type `clock_t` in units of `CLOCKS_PER_SEC`, defined in `time.h`. This constant currently has a value of 100, which results in a low resolution that is inadequate for some tasks. CodeWarrior defines `CLOCKS_PER_SEC` as 1000000, for microsecond resolution.
- The function `itoa` is not a standard C library function and it is not included in the system libraries provided by Mac OS X. CodeWarrior does supply this function, in the header `extras.h`. If your code uses this function, you should replace it with `printf` or some other equivalent function.

Support for wchar_t and wstring

The standard C and C++ wide character type `wchar_t` and the `wstring` template class that makes use of it are supported in the system libraries for Mac OS X version 10.3 and later. As a result, you can freely use these data types in software created with Xcode that will run only in Panther and later. However, if you need to use these data types in software that will run in versions of Mac OS X prior to Panther, you should obtain a third-party standard C library, such as the one available from Dinkumware, Ltd. .

Important: Although `wchar_t` and `wstring` are standards, they are somewhat vaguely defined and ambiguous ones. Mac OS X provides far better support for Wide and Unicode characters through the CFString APIs. Apple strongly recommends using these APIs on Mac OS X, as almost all other Mac OS X frameworks expect wide character strings to be in this format, and none support `wchar_t` and `wstring`.

CodeWarrior supports `wchar_t` and `wstring` for CFM applications because they are supported in the MSL C and C++ libraries for CFM. However, because MSL libraries for Mach-O in CodeWarrior do not anticipate `wchar_t` support in the Apple standard C libraries, `wchar_t` support is turned off in the MSL Mach-O standard library projects. This means that if you use CodeWarrior in Mac OS X v.10.3 or later, you'll get build errors when you rebuild the CodeWarrior MSL libraries because `wchar_t` support is turned off but the Mac OS X headers now use `wchar_t`.

To resolve this issue, when rebuilding your MSL libraries on Panther, you should select the “Enable wchar_t Support” checkbox in the C/C++ Language settings for the Mach-O library projects.

The GCC Compiler

CodeWarrior provides a proprietary compiler that supports C and C++ and includes support for Altivec. CodeWarrior and Xcode C and C++ libraries are discussed in “C and C++ Libraries” (page 25).

Xcode uses the open-source GNU C Compiler, or GCC. When you compile C, C++, Objective-C, or Objective-C++ code in Xcode, you’re using GCC. The default version of GCC for Xcode 2.2 is 4.0. For versions of Xcode earlier than Xcode 2.0, the default version of GCC is 3.3. If necessary, you can choose from the the available GCC versions by selecting a target in the project window, opening an inspector

window, and selecting the Rules pane. There you can use the pop-up menus to view the current system rules and to add rules for which compiler to use for C, C++, and other types of files in that target. For related information, see "More on GCC Compiler Versions" (page 28).

Note: Versions of the Xcode Tools package prior to Xcode 2.0 also included GCC versions 2.95 and 3.1; these versions are not included with subsequent versions of the tools.

When you import a CodeWarrior project, the rule for processing C files is left unchanged. This means that C files are compiled with the current system version of GCC, which depending on the system, is either 3.3 or 4.0. This document assumes that you will use version 4.0 to build your imported project.

Compile speed has improved dramatically in GCC 3.3 and 4.0, but does not yet quite equal CodeWarrior. However, Xcode sports features such as distributed and multiprocessor builds, where each compile is handled by a separate CPU, even on separate machines. These features can dramatically reduce build time for large projects. Xcode also provides additional features to improve the overall development process, including ZeroLink and Fix and Continue, described in "Some Special Features of Xcode" (page 14).

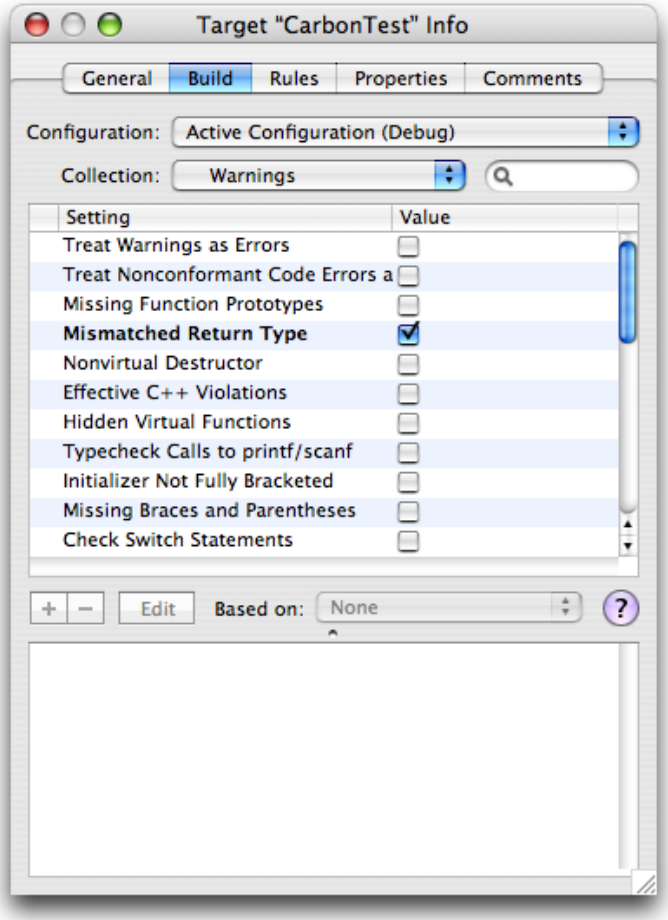
Note: You can view the full compile commands, as well as other detailed build information, in a display area in the Build Results window. You can open the Build Results window with Build > Build Results. You can open the display area by clicking the build log button at the bottom of the window's topmost pane.

Figure 1-3 (page 23) shows some GCC build settings in an inspector window for an Xcode target. Build settings in the Xcode user interface are converted to command-line options to GCC when you compile your source code. If you want to use a feature of GCC that isn't available in the Xcode user interface, you can add flags to the Other C Flags setting. To view this setting, see the detailed steps provided in "Inline ASM" (page 29). Multiple flags should be separated by spaces.

There are a number of significant differences between GCC and the CodeWarrior compiler. One that you're likely to notice right away is that GCC has a reputation for strictness, and is likely to generate many warnings for code that may produce few warnings in CodeWarrior. Because C and C++ are very permissive languages, most of these warnings have been found to be useful over the years. Getting a compiler error or warning about suspect code can save days or even weeks of difficult debugging.

Still, you have options to reduce the number of warnings. To change the warnings for a selected target, open the inspector window, select the Build pane, and choose Warnings under GNU C/C++ Compiler in the Collections menu. You can then select any of the warning settings and see a description for it displayed beneath the list of settings, as shown in Figure 1-5.

Figure 1-5 Warnings settings in an inspector window



You can also set additional GCC warnings by modifying the Other Warning Flags build setting in the Build pane.

More on GCC Compiler Versions

The C++ ABI changed between GCC 4.0 and GCC 3.3 (the default compiler for Xcode 1.5 and earlier). The GCC 3.3 compiler in turn has a different ABI than either the GCC 3.1 compiler (the default compiler for the December 2002 release of the Developer Tools) or the GCC 2.95 compiler (the default compiler for earlier releases of the Developer Tools). All your C++ code, including libraries and frameworks, must be built with the same compiler. Note that you should not use the GCC 4.0 compiler to build C++ programs for versions of Mac OS X prior to 10.3.9.

IOKit-based device drivers and other kernel extensions built with the GCC 4.0 compiler will run in Mac OS X versions 10.2 through 10.4. You must use GCC 2.95.2 for kernel extensions that need to run on Mac OS X version 10.1.

Additional Compiler Information

Additional compiler-related differences are described throughout this document, especially in the following sections:

- "C++ Code in C Files" (page 29)
- "Inline ASM" (page 29)
- "C and C++ Libraries" (page 25)
- "Precompiled Headers and Prefix Files" (page 21)
- "Pragma Statements" (page 22)
- "Cross-Development" (page 21)
- "Support for wchar_t and wstring" (page 26)
- "Code Differences" (page 43)
- "Migrate from MSL to System C and C++ Libraries" (page 48)
- "Use C99 Standard in Language Settings" (page 47)
- "Make Code Changes for GCC Compatibility" (page 54)
- "Make Changes to PowerPlant" (page 62)

Current documentation for GCC (through version 4.0) is available in Tools Documentation.

C++ Code in C Files

GCC is literal in its interpretation of file suffixes. While CodeWarrior applies the C++ preprocessor and parser to C++ and C files alike, GCC will emit errors if C++ code is used in `.c` files. You can override this by changing the file's type to a C++ filetype (`sourcecode.cpp`) in the General pane of the file inspector. As long as the Compile Sources As (GCC_INPUT_FILETYPE) build setting is set to According to File Suffix, changing the filetype will cause Xcode to use the `-x c++` option when compiling files whose type is a C++ filetype. Any explicitly set value for the Compile Sources As build setting overrides the filetype you specify in the General pane of the file inspector.

By default, when you import a CodeWarrior project, the Compile Sources As build setting is set to According to File Suffix. However, if you have Force C++ compilation set in your CodeWarrior target settings, the importer in Xcode 2.2 sets the Compile Sources As build setting for that target to `sourcecode.cpp.cpp`.

Inline ASM

GCC supports CodeWarrior-style inline asm code. To enable this support, you add the following flag to your build settings:

```
-fasm-blocks
```

When you import a CodeWarrior project that uses inline asm into Xcode, this flag is not set automatically, so you will have to add it yourself. You can do so with the following steps:

1. Select a target in the project window.
2. Open an inspector window like the one shown in Figure 1-3 (page 23) by clicking the Info button or choosing File > Get Info.
3. Select the Build pane.
4. If the GNU C/C++ Compiler settings aren't visible, select GNU C/C++ Compiler from the Collection menu.
5. Click the checkbox in the Value column next to the CodeWarrior-Style Inline Assembly setting.

Important: There is a setting Allow 'asm', 'inline', 'typeof' in the Language Settings in the Build pane. This setting, which is enabled by default, is not related to CodeWarrior-style inline asm support. It governs the use of the old-style GNU asm statements, which are not actually part of the ISO standard.

In general, the GCC implementation works the same as the CodeWarrior implementation, and any discrepancies you note should be reported as bugs (see "Feedback and Mail List" (page 11)). For example, if your project contains hand-tweaked asm code, it should work correctly in the new project, unless your code specifically targets CFM-related features. (CFM refers to the executable architecture supported by the Code Fragment Manager in Mac OS 9.)

Function calls, such as `bl foo`, don't currently work, and are treated as though `foo` is a label whose definition is missing.

Calling conventions are generally the same as for CFM, but global variables are not available through the TOC register `r2`. For instance, in CFM, if `aglob` is a global, then `lwz aglob(r2)` works to get the value of `aglob` into `r3`. The Mach-O equivalent is complicated, involving multiple internal labels, and at present can't be handled with inline asm.

GCC passes the inline asm through to the assembler and doesn't interpret it, so any errors reported from the asm code come from the assembler. In unusual cases, such as when a typedef has the same name as an op code, GCC may parse the code differently from CodeWarrior. For example, the code in listing Listing 1-1 will result in warnings and a body with one `nop` if using CodeWarrior, and a body with two instructions (`mr r1,r2` and `nop 0` if using GCC. (This should probably be considered a bug in GCC's implementation.)

Listing 1-1 A typedef with the same name as an op code

```
typedef int mr;
asm int foo()
{
 mr r1,r2;
 nop
}
```

For a related issue, see "Inlining Thresholds" (page 55).

The Linker

In CodeWarrior, you can choose from among several linkers in the Target Settings pane. The "Macintosh PowerPC linker" creates executables based on the format specified by the Code Fragment Manager (CFM) architecture. The "Apple Mach-O PowerPC linker" makes use of Apple's `ld` linker and `libtool` command-line tool. The "Mac OS X PowerPC Mach-O linker" also creates Mach-O executables, but uses the Metrowerks Mac OS linker.

Xcode uses the `ld` linker, which is designed to work with Mach-O object files, and supports dynamic shared libraries, two-level and flat name spaces, and other features.

The `ld` linker supports the automatic stripping of unused (dead) code beginning with the June 2004 release of the Xcode tools. For more information, see "Dead Code Stripping" (page 31). Using the `ld` linker may also result in larger binary files than in your CodeWarrior project. You can display man page documentation for the linker in Xcode with Help > Open man page, or in a Terminal window by typing `man ld`.

Dead Code Stripping

The CodeWarrior linker supports dead code stripping (the removal of unused code). Beginning with version 1.5, Xcode also supports dead code stripping. To enable dead code stripping in your project, do the following:

1. Select a target in the project window.
2. Open an inspector window like the one shown in Figure 1-3 (page 23) by clicking the Info button or choosing File > Get Info.
3. Select the Build pane.
4. If the Linking settings aren't visible, select Linking from the Collection menu.
5. Click the checkbox in the Value column next to the Dead Code Stripping setting.

In versions of Xcode prior to Xcode 1.5, a project may generate errors at link time because it contains unused code that refers to undefined symbols. For more information on this issue, see "Resolve Undefined Symbols" (page 54).

You can of course take your own steps to eliminate code you know will never be called. Or, if you have reasons for not stripping certain unused code, you can take steps to avoid exporting symbols for that code. For example, if you create an order file, you can increase the likelihood that dead code in your application will never actually be loaded.

From a performance standpoint, it is always worthwhile to reduce both your code size and the number of exported symbols. For more information in this document, see "Exporting Symbols" (page 37). The document *Code Size Performance Guidelines* contains the following sections:

- "Improving Locality of Reference" describes how to create an order file.
- "Minimizing Your Exported Symbols" describes how to reduce the symbols exported by your application.

To learn more about support for dead code stripping in Xcode see "Dead Code Stripping" in the chapter "Linking" in *Xcode User Guide*.

The Information Property List and .plc Files

Any packaged Mac OS X software, including applications, bundles, plug-ins, and frameworks, requires an information property list file named `Info.plist`. That file contains key-value pairs that specify various information that is used at runtime, such as the version number for the software. This information is used by the application and by Launch Services (an API for launching applications in Mac OS X) as well.

In CodeWarrior, projects that create packaged Mac OS X software use a `.plc` file to specify information for the `Info.plist` file. For example, if you use CodeWarrior project stationery to create a project for a Mach-O based Mac OS X application or a bundled Carbon application, the project includes a default `.plc` file. CodeWarrior reads the `.plc` file at build time, follows any included header files or prefix file chains, and creates a property list from it, according to settings you specify in the Property List pane of the target settings window.

An advantage of this approach is that you can define symbols in header files and include them into both your code and your `.plc` file. As a result, you can be confident you are using the same data in both your code and your `Info.plist` file.

Xcode provides a user interface for directly specifying property list settings, but it does not support conversion of a `.plc` file into an information property list, so you cannot share symbols between your code and your property list file. However, Xcode does obtain information for the property list from your CodeWarrior project during the import process.

You supply or modify property list information by opening an Inspector window for a target, and making changes in the Properties pane. Xcode also supports preprocessing of the `Info.plist` file using the GNU C preprocessor; you can include headers, use conditional statements, and define preprocessor macros for use when preprocessing the file. You can also reference any build settings in effect for the target; Xcode evaluates those build settings and replaces them with the appropriate value at build time. For more information on editing property lists in Xcode, see *Xcode User Guide*. For details, see "Move Settings From the .plc File to an Info.plist File" (page 61).

Working With Resources

CodeWarrior and Xcode both work with Resource Manager `.r` and `.rsrc` resource files. In CodeWarrior, you can set Rez Options, Derez Options, and Common Options in the Rez target settings pane. In Xcode, you can set Resource Manager settings in the Build pane of a target inspector window. If you drag resources into your Xcode project, or import them as part of a CodeWarrior project, Xcode should place those resources in the Build ResourceManager Resources build phase. Xcode compiles `.r` files in this build phase and copies any `.rsrc` files into the product bundle. For more information on build phases in Xcode, see Build Phases in *Xcode User Guide*.

Xcode does not provide a user interface to rez and derez individual files, but the Rez and DeRez tools are available in `/Developer/Tools` and you can run them from the command line and in scripts. There are man pages for Rez and DeRez, but documentation for the rez language is a bit hard to find. It's documented in Appendix C of Building and Managing Programs with MPW.

Important: Due to a bug in Rez, if output is written to a data-fork file (as it almost always is when building with Xcode), then all `.rsrc` files included with an include statement must also be data-fork files.

In Mac OS X and for Carbon applications generally, resources should be put in the data fork of a separate resource file, not in the resource fork of the executable, as described in the section "*Move Resources to Data Fork–Based Files*" in *Carbon Porting Guide*. The primary reason for moving application resources out of resource forks is to enable applications to be seamlessly moved around different file systems without loss of their resources.

In addition to using `.r` and `.rsrc` files for resources, Carbon applications can use Interface Builder resource files (called nib files because they have an extension of `.nib`). When you create a new project in Xcode, the Carbon Application project template creates a nib-based Carbon application. Interface Builder provides an easy-to-use graphical method for designing and implementing a GUI, so there is potentially a lot to gain by using nib files for your resources. Interface Builder is described in "Companion Applications" (page 19). To learn more about using Interface Builder with Carbon applications, see *Interface Builder*, *Unarchiving Interface Objects With Interface Builder Services* and *Interface Builder Services Reference*.

Mac OS X also provides a very useful mechanism for storing language-dependent resources in localized directories within an application or other bundle. By using APIs such as CFBundle, your application can work with localized resources in a seamless manner. For more information on resources, application packaging, and bundles, see the document *Mac OS X Technology Overview*, as well as the document *Bundle Programming Guide*.

Building Code

In CodeWarrior, you can have one or more named build targets, each of which specifies a set of files and project settings used to build an output file. The project manager keeps track of project dependencies and calls on the build system to build the current target.

With Xcode, you also define named targets, containing all the files and instructions required to create a final product. In addition, you can define different build configurations for each target, which allow you to build the target with different build settings without creating a whole new target.

The following sections describe the Xcode build system and compare various build-related features in CodeWarrior and Xcode. For additional details on building code, such as how to modify build settings, start a build, display the Build Results window, and so on, see "The Build System" in *Xcode User Guide*.

Native Build System

Xcode includes a fast, accurate dependency system built into the IDE (hence "native"). While Xcode fully supports legacy Project Builder projects that use a JAM-based build system, all new projects and targets—including targets imported from CodeWarrior projects—use the native build system.

Note: In Project Builder version 2.0 and earlier, the dependency analysis among project files was done by an external tool called JAM ("Just Another Make") developed by Perforce Software. JAM-based targets will continue to be supported in Xcode until native targets support all of the same functionality.

Native targets have certain advantages over JAM targets. Because the dependency checking is done in the IDE itself, it is faster, which means that your builds start faster. In addition, many Xcode features, including Zero Link, Fix and Continue, Distributed Builds, and SDK Support, are only implemented for native targets.

Native targets use inspector windows for viewing and editing target settings, while JAM targets use an older-style Target Settings window. JAM targets are represented in the Xcode user interface by the traditional bull's-eye icon, while the icon for a native target varies according to the product type—for example, applications are represented by a stylized "A" icon.

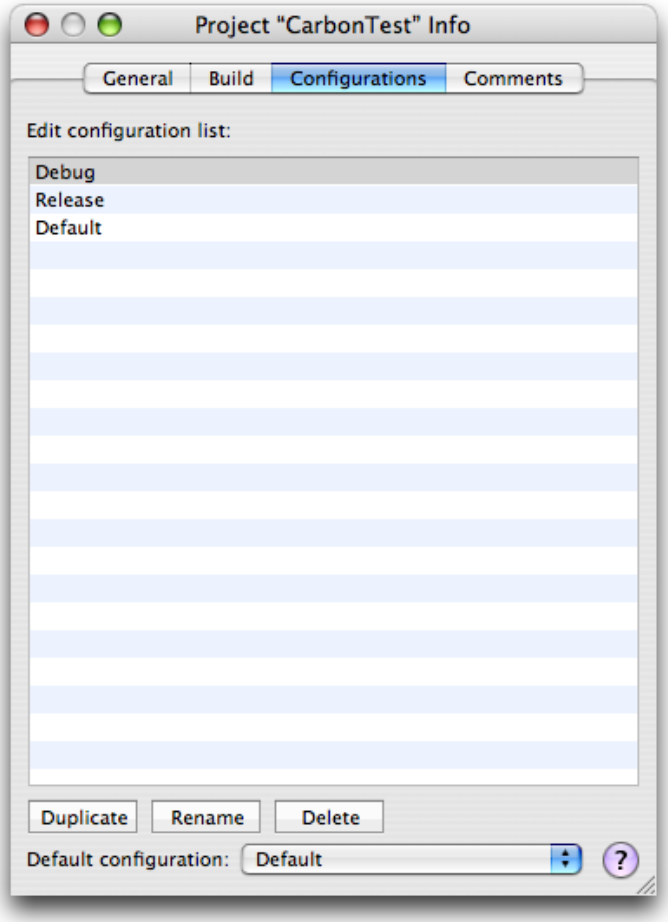
Note: The Targets group *always* uses the bull's-eye icon. Individual targets within the group use the icons as described.

Build Configurations

A build configuration is a variation on a target that lets you build a target using different build settings without creating a whole new target. If you need to create products that differ only in their build settings, you might create one target with several build configuration. If you need to create products that differ in other types of settings (such as build phases or information property list entries), you should create separate targets for each.

The list of build configuration names is defined at the project level. To add or remove build configurations, select the project source group in the project window, click the Info button to open an inspector window, and select the Configurations pane, which is shown in Figure 1-6 (page 35). By default, new projects created in Xcode contain both Debug and Release build configurations, described below. Projects you import from CodeWarrior have a single build configuration, named Imported CodeWarrior Settings.

Figure 1-6 The Configurations pane



To add a build configuration to a project, select an existing build configuration and copy it.

Although build configuration names are defined at the project level, each target in the project, and the project itself, can define a different set of build settings for a build configuration. You edit the build settings in a build configuration in the Build pane of the target or project inspector. In this pane, select the build configuration you want to edit in the Configuration menu, then add or delete settings and insert or modify values in the list below the menu. You use the minus and plus buttons at the bottom of the list to add or delete settings.

To select the current build configuration, use the Active Build Configuration pop-up menu in the toolbar of the project or Build Results window or choose Set Active Build Configuration.

Creating Debug and Non-Debug Products

In Xcode, Debug and Release build configurations typically take the place of CodeWarrior's Debug and Final targets, respectively. When you create a new project, by default the project contains these two build configurations. You are free to modify their definitions at the target or project levels, or to add other build configurations. By default, the Debug build configuration produces debug symbols and turns off code optimization, while the Release build configuration does the opposite.

As mentioned in the previous section, projects imported from CodeWarrior contain a single build configuration, called Imported CodeWarrior Settings, which contains the target settings brought over from your CodeWarrior targets. You can easily create Debug and Release build configurations by duplicating the Imported CodeWarrior Settings configuration and customizing each of the new Debug and Release configurations with the appropriate settings.

Note: If you import a CodeWarrior project that has both Debug and Final targets, the new Xcode project will also have both Debug and Final targets.

In both CodeWarrior and Xcode, you can define a common prefix file that includes all the headers needed by a project to create debug or non-debug products. You can then define separate headers that set required preprocessor definitions (for example, `#define debug 1`), then include the appropriate header into the prefix file, depending on which type of product you want to build.

In CodeWarrior, this requires separate prefix files because the only way to set a macro is with a `#define` statement in a file (you can't set values on a command line).

In Xcode, you can take another approach. You can have one common prefix file for all targets. You can then define preprocessor options in the target settings. This allows you, for example, to define a master `debug` flag you can test for anywhere in your code. Since it's part of the target settings, the prefix file will be precompiled differently, depending on the target settings for the product.

In fact, it's recommended that you delete any redundant targets and move the differentiating flags from the prefix headers to the appropriate build configuration of the primary target. The targets with the fewest settings are the best candidates to replace with a build configuration.

Automating the Build Process

CodeWarrior provides a substantial AppleScript dictionary, with terminology that allows some automation of the build process.

Xcode also has a substantial scripting dictionary, and provides AppleScript terminology for controlling many aspects of the build process. To examine the Xcode terminology, you can:

- Drag the Xcode application icon onto Script Editor in the Finder.
- Choose File > Open Dictionary in Xcode
- Choose File > Open Dictionary in Script Editor

You'll find classes such as `bookmark`, `breakpoint`, `source directory`, and `target dependency`, along with events such as `build` and `clean`, to name just a few.

You can also call the `xcodebuild` tool from a shell window or a script to build a project. The `xcodebuild` tool reads your project file and builds it just as if you had used the Build command from within Xcode, although there are some differences you should read about in *Xcode User Guide*.

In addition, you can use the `osascript` and `osaexecute` commands to run AppleScript scripts from a shell window or shell script. These commands can target Xcode or any other scriptable application. You can use the Terminal application, available in `/Applications/Utilities`, to open shell windows, execute shell scripts, and so on.

Finally, you can use AppleScript's `do shell script` command to launch the `xcodebuild` command-line tool from an AppleScript script. However, if you're writing an AppleScript script anyway, it probably makes more sense to use the Xcode scripting terminology.

Makefiles

CodeWarrior has a Makefile Importer wizard to import makefiles.

The Xcode project importer does not import makefiles, and the IDE has no automated support for setting up a project to follow the rules in a makefile. However, it is often the case that developers don't necessarily want to convert a makefile into a project, they would just like to use the makefile with a project. Xcode supports that goal with the external target.

An external target is a target that Xcode doesn't maintain build rules for. For all other targets, Xcode stores information on how its product should be built and installed. As you add and remove files, Xcode keeps track of how to compile them for you. For a legacy target, you have to maintain that information yourself, usually by creating your own build file.

Important: External targets cannot take advantage of some Xcode features, such as ZeroLink and Fix and Continue.

You add an external target by choosing Project > New Target and selecting External Target.

Exporting Symbols

As part of building a shared library, plug-in, or other software that will export symbols, you want to specify which symbols are available to clients of the software. The fewer symbols your software exports, the more quickly it will load at runtime. In addition, you typically do not want to expose internal entry points, due to both competitive and support issues.

CodeWarrior provides several options for specifying which symbols a shared library will export.

- Use an export (or `.exp`) file. When you run the Make command on a library target, CodeWarrior creates a `.exp` file containing all the global variables and routines in the project. After creating this file, you can comment out any symbols you don't want to export, including CodeWarrior runtime symbols. You then add the `.exp` file to the appropriate target and rebuild the project.

  Note: CodeWarrior also supports the use of a `.axp` file, which lists symbols that should not be exported. CodeWarrior exports all symbols other than those listed in this file.

- You can use the pragma statements `#pragma export on` and `#pragma export off` to bracket any symbols you want to export. CodeWarrior exports only symbols bracketed by these statements.
- Combine the two previous mechanisms. Only the bracketed symbols are placed in the `.exp` file, which you can still edit as necessary.

You can include your export file in an Xcode project as well, by setting the Exported Symbols File build setting to the name of your export file. Xcode passes the appropriate options to the static linker.

Note: The static linker also supports the `-unexported_symbols_list` option to specify a file containing symbols that should not be exported. However, there is currently no built-in build setting in the Xcode interface for setting this option.

You can't use the pragma statements `#pragma export on` and `#pragma export off` to specify export symbols in Xcode because GCC does not support these pragma statements. However, GCC does have its own pragmas for controlling symbol visibility, described in *C++ Runtime Environment Programming Guide*.

Beginning with Xcode 2.2, the importer interprets the Export Symbols setting in your CodeWarrior target. Based on the following values for that setting, the importer configures the corresponding Xcode target differently:

- None. The importer enables the Symbols Hidden by Default (GCC_SYMBOLS_PRIVATE_EXTERN) and Inline Functions Hidden (GCC_INLINES_ARE_PRIVATE_EXTERN) build settings for the new Xcode target. When these build settings are enabled, all symbols are declared 'private extern' unless explicitly marked otherwise.
- All Globals. The importer disables the Symbols Hidden by Default (GCC_SYMBOLS_PRIVATE_EXTERN) build setting in the new Xcode target.
- 'Use ".exp" file' or 'Use #pragma and ".exp" file.' The importer sets the Exported Symbols File (EXPORTED_SYMBOLS_FILE) build setting to the path to the first export file that it finds in your CW target. It also enables the Symbols Hidden by Default and Inline Functions Hidden build settings.

  Note that, while CodeWarrior targets may have multiple `.exp` files, an Xcode target can have only one. If your CW target has more than one export file, the importer uses the first one it encounters and ignores the rest.

The importer does not interpret the Referenced Globals or Use #pragma settings.

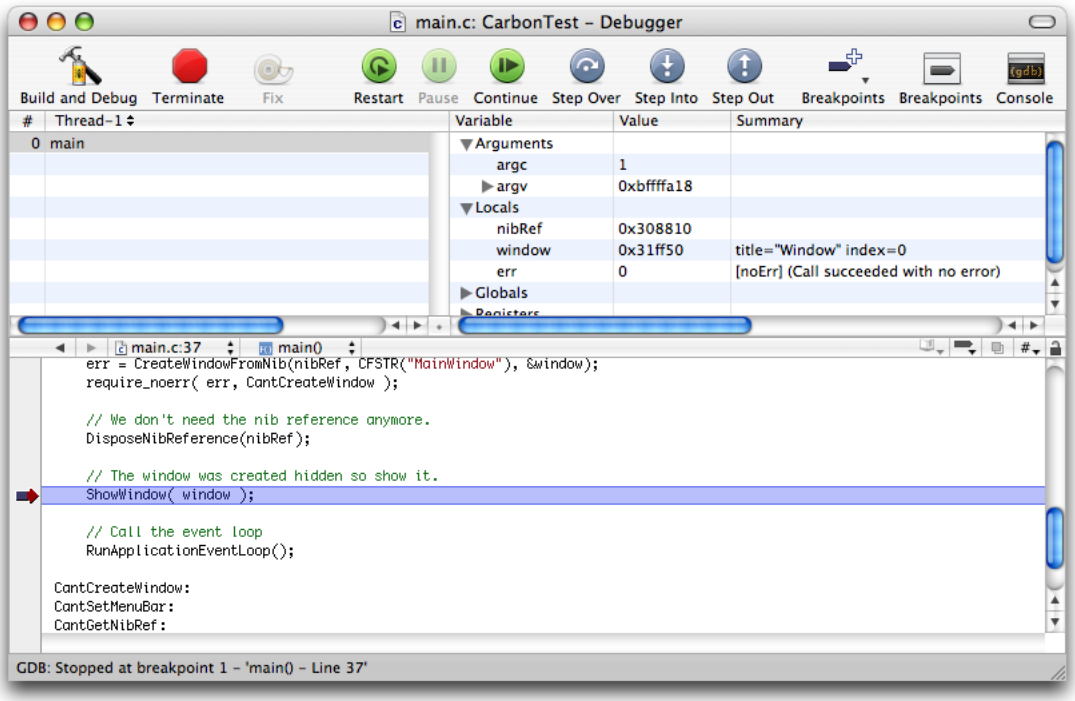
For related information in this document, see "Runtime and Library Issues" (page 25), "Dead Code Stripping" (page 31), and "Pragma Statements" (page 22).

For additional information on examining and managing your software's exported symbols, and on other techniques for producing more efficient code, see *Code Size Performance Guidelines*. In particular, the section "Minimizing Your Exported Symbols" describes how to reduce the symbols exported by your application.

Debugging

Basic debugging is very similar in CodeWarrior and Xcode, though Xcode uses the open source GDB debugger and provides some additional features, such as Fix and Continue (live patching), to make debugging more efficient. Xcode supplies a graphical interface to GDB, but you can also use the command line interface (in place of or in parallel to the UI) to get full access to any GDB feature not supported in the user interface. Figure 1-7 shows the debugger window stopped in a simple Carbon application.

Figure 1-7 A debug window in Xcode



For GDB documentation, see *Debugging with GDB*.Note that some debugging features are currently unavailable in Xcode: for example, you cannot set event points in the UI.

In CodeWarrior, “event points” are the general name for a set of actions that the CodeWarrior IDE debugger can perform at a breakpoint (such as perform a test, log a message, play a sound, speak a phrase, or execute a script). A “watch point” is a range of memory that you are watching while debugging to note changes; the debugger stops execution when it notices that a certain range of memory has been changed.

Beginning with Xcode 2.1, Xcode includes support for both breakpoint actions and watchpoints. Similar to event points, breakpoint actions let you perform an action at a breakpoint. Xcode includes breakpoint actions that let you execute a script, print a message to the console, send a command to the debugger, and more.

While CodeWarrior supplies nonstandard flags to perform diagnostics on use of the standard template library (STL), Xcode does not include a debug version of the STL. However, Xcode does support breaking on C++ `try`, `catch` and `throw` constructs, and allows stepping through template instantiation.

Prebinding

Prebinding is the process of computing at build time the addresses for the symbols imported by libraries and applications, so that less work needs to be performed by the dynamic linker at runtime.

Note: Prebinding is essential for applications running on Mac OS X version 10.3.3 or earlier. If your application is running on Mac OS X version 10.3.4 or later, you no longer need to prebind your application.

If a user installs your software on a system with a set of libraries that is different than those present when the application was built, the prebinding information cannot be used and application launch is slower. If you use the PackageMaker application to package your software and let users install it with the Installer application, Installer automatically attempts to update prebinding information for the software it installs by running the `update_prebinding` tool. If your software uses drag-and-drop installation, you can provide instructions for how to run this tool through a Read Me file, printed documentation, or other mechanism.

Prebinding is only applicable for Mach-O executables. (Mach-O is the native executable format in Mac OS X and is the only format supported by Xcode.) In addition, there are some circumstances where prebinding information cannot be updated or used. For details, see the following:

- "Prebinding Your Application" in *Launch Time Performance Guidelines*.
- The man pages for `update_prebinding` and `redo_prebinding`. (You can search for and view man pages in Xcode with Help > Open man page.)

For more information on distributing your software, see *Software Delivery Guide*.

Source Trees

CodeWarrior supports the use of source trees, a kind of root path commonly used in projects that will be worked on by developers in different locations and on different machines. Source trees can be used to define common access paths and target outputs. Global source trees (defined in the CodeWarrior preferences) apply to all projects, while project source trees (defined in the Target Settings for a project) apply only to files in that project.

Xcode supports only a global source tree, defined in the Source Trees pane of the Preferences window (though it does also support search paths for headers, libraries, and frameworks, on a project basis).

When you import a CodeWarrior project (see "Importing a Project" (page 50)), Xcode determines the location of the CodeWarrior root folder (commonly referred to as `{Compiler}` in CodeWarrior's access paths). It then adds an entry to the Source Trees list in the Preferences window, with the Setting Name "CodeWarrior," Display Name "CodeWarrior Folder," and the specified path. If a Source Tree entry for CodeWarrior already exists, it is overwritten.

Depending on where CodeWarrior is located, a typical path will look something like `/Applications/Metrowerks_CodeWarrior_8.0/Metrowerks CodeWarrior`. This CodeWarrior-relative source tree is very useful for building projects that use PowerPlant.

Note: If your project uses source trees, make sure that everyone working on the project has those source trees defined. (The source trees can point at different places, of course, but the names should be the same.)

Source Control

CodeWarrior provides a Version Control System (VCS) menu that supports Perforce and other systems through plug-ins. Plug-ins can provide features that are very specific to the particular system, but are not tightly integrated into CodeWarrior itself.

Xcode takes a different approach. It currently supports three systems: the open source standard Concurrent Versions System (CVS), Subversion, and Perforce. Whichever system you choose, the Source Code Management (SCM) menu has the same commands. The number of supported commands is deliberately kept limited, but the commands are more tightly integrated into the environment. You'll find the most commonly used source control commands, but no access is provided to more specific (and possibly obscure) commands that are supported only by a particular system. This allows for a cleaner integration; for example, status output is more usefully integrated into the user interface.

Important: Diff and Compare commands in Xcode do not currently deal correctly with Mac-style line endings.

PowerPlant

If you move a project that uses CodeWarrior's PowerPlant framework to Xcode, you'll have to build PowerPlant as part of the project. When you import such a project, Xcode should add all the required PowerPlant files to the new Xcode project. You'll then have to supply a prefix file and make a few minor modifications to the PowerPlant source code to build successfully with GCC 4.0. These steps are described in "Make Changes to PowerPlant" (page 62).

Maintaining Parallel Projects in Xcode and CodeWarrior

This section provides tips on maintaining parallel projects in Xcode and CodeWarrior that operate on the same code base.

Executable Format

The more similar the CodeWarrior and Xcode projects are, the easier the task will be, so unless you absolutely need a Mac OS 9 product, convert your CodeWarrior project to use the Mach-O executable format (as all Xcode projects must do).

When you use Mach-O with CodeWarrior, you can choose either the Metrowerks linker or the Apple linker, and choose to use either MSL or the standard Mac OS X libraries. Again, it's simplest to have consistency between your products, so unless you're using specific features of the Metrowerks linker or MSL, it's recommended that you use the Apple linker and Mac OS X standard libraries.

Framework and System Headers

Most CodeWarrior projects are still built against Universal Interfaces. These headers have not been updated for current versions of Mac OS X, so projects that use them don't have access to all the functions and constants that are available in the latest headers.

If you convert your project to Mach-O, you'll most likely also move it to using framework-style headers. This means that your source files should include just the Carbon framework header (or any other framework headers you need), rather than the raft of individual header files traditionally used in CarbonLib or Classic development. To include the Carbon framework header, you use a statement like this:

```
#include <Carbon/Carbon.h>
```

Note: When you include the Carbon framework header, it in turn includes a number of other framework headers.

If you haven't converted to framework-style headers (or are building for CFM in CodeWarrior, which doesn't allow you to use framework includes), you can use the Xcode SDK Support feature to get access to the headers of the current system (or a past or future version, if you wish). Just change your system access path from `{Compiler}MacOS support` to `/Developer/SDKs/MacOSX10.2.8.sdk/Developer/Headers/CFMSupport` (to get the Jaguar headers) and add an access path to `/Developer/SDKs/MacOSX10.2.8.sdk/System/Library/CFMSupport` (to get the Carbon CFM link libraries). Of course, you can use different SDKs depending on which OS you wish to target, and you can use weak linking as usual to run on older systems.

You can find additional information on weak linking in Technote 2064, "Ensuring Backwards Binary Compatibility—Weak Linking and Availability Macros on Mac OS X".

See "Cross-Development" (page 21) for information on using SDK support in Xcode.

Using Precompiled Headers and Prefix Files

Because CodeWarrior compiler and Xcode use different precompiled header mechanisms, you'll have to have separate precomps for each environment. You can manage this by using the automatic-precompilation features of both environments; Metrowerks will automatically precompile a header file that ends in `.pch`, and Xcode automatically precompiles any prefix file. So the recommended way to set this up is as follows. For Xcode:

1. Create a `.pch` file that #includes all the headers you want to precompile (framework, project, and utility).

2. Use that `.pch` file as the prefix file in the appropriate targets of your project by entering the name in the "Prefix Header" Language setting in the Build pane for each target.

3. Enable the “Precompile Prefix Header” Language setting in the Build pane for each target.

For CodeWarrior:

1. In the `.pch` file, add the following directive:

```
#ifdef __MWERKS_
        #pragma precompile_target "Precomps.mch"
  #endif
```

2. Use the filename defined in the previous step (`Precomps.mch`) as your prefix file in appropriate targets of your project.
3. Create a CodeWarrior target that contains just the `.pch` file; when built, this will generate the `.mch` file.
4. Make other targets depend on the target that generates the `.mch` file.

For related information, see “Precompiled Headers and Prefix Files” (page 21).

Property Lists

CodeWarrior uses a property list compiler and special property list source files (`.plc` files) to generate `Info.plist` files for packages. Xcode uses GUI settings in the target inspector window to set those values, and generates the `Info.plist` file automatically. If you change property list settings, you'll have to change them in both places; it's worth inserting a comment into your `.plc` file to remind other people that if they make changes there, they should change the Xcode project file as well.

For more information on working with property lists, see “The Information Property List and .plc Files” (page 32).

Code Differences

While both CodeWarrior and GCC compilers do a good job of implementing standard C and C++ features, they differ on which compiler-specific extensions are supported. You should isolate compiler-specific code using the following preprocessor directives:

Listing 1-2 Preprocessor directives for isolating compiler specific code

```
#ifdef __MWERKS__
    // CodeWarrior-only code should go here
#endif

#ifdef __GNUC__
    // GCC-only code should go here
#endif
```

The following are some examples of CodeWarrior-only code you should isolate. You’ll find more details on these items in “Make Code Changes for GCC Compatibility” (page 54):

- Most CodeWarrior-defined `#pragma` directives. Though GCC ignores them, it's good discipline to remind yourself that they're compiler-specific.
- Metrowerks-specific extensions to BSD functionality, such as `FSp_fopen()`, `SIOUX`, `console.h`, and so on.
- C-style cases and functional casts in an argument list (standards-compliant but not supported in GCC).
- Instantiating a `struct` with a member template within a template definition (standards-compliant but not supported in GCC).
- `#if true` (CodeWarrior-specific extension).
- Anonymous unused arguments in C function definitions (legal in C99 but unsupported in GCC).
- Structs in `vararg` lists (Metrowerks extension).
- Using a const global variable in the definition of another const global (Metrowerks extension).
- A class declared as a friend of template specialization that accesses private members.

Some examples of GCC-only code:

- GCC inline assembler. GCC 3.3 and 4.0 can assemble CodeWarrior-style inline assembler, but CodeWarrior cannot handle the GCC syntax for inline assembly.

Linking

CodeWarrior offers a choice of using in-code `#pragma` statements or an `.exp` file to control which symbols are exported from your end-product libraries; GCC uses only the `.exp` file.

For more information on exporting symbols, see "Exporting Symbols" (page 37).

Preparing a CodeWarrior Project for Importing

This section describes steps you can take to modify your CodeWarrior project before importing it into Xcode. Taking an incremental approach should make it easier to get your project building successfully in Xcode.

Important: It is recommended that you use CodeWarrior Pro version 8.3 or later for projects that you will import into Xcode. You are more likely to experience problems with older versions of CodeWarrior.

Note: This chapter describes preparation for converting a project that builds an application, but converting a project that builds shared libraries or other types of software requires many of the same steps.

Convert Classic Applications to Use Carbon

If you have a classic application—that is, an application designed for versions of the Mac OS earlier than Mac OS X—you should convert your code to use Carbon. Carbon is a set of programming interfaces that allows applications (including those originally designed for Mac OS 8 and 9) to run natively in Mac OS X. If an application uses older Mac OS APIs that aren't part of Carbon, it cannot run reliably in Mac OS X, except in Classic (or emulation) mode.

The amount of effort required to convert a project to Carbon depends on its complexity and on the programming tactics and Mac OS APIs it uses. For full information, see *Carbon Porting Guide*.

Use the Application Package Type

You should set the Project Type setting for your CodeWarrior project to Application Package. Packaging an application consists of putting the application's code and resources in prescribed directory locations inside the application bundle.

Note: A bundle is a directory in the file system that stores executable code and the software resources related to that code. Application packages are presented by the Finder as a single file.

To build an application as a package, you choose Application Package in the Project Type pop-up in the PPC Target pane. When you create a project from CodeWarrior stationery that uses one of the bundled types, such as "C Toolbox Carbon Bundle," the Project Type is automatically set to Application Package.

Important: You may want to change application package type at the same time you switch to the Mach-O executable format, described in the next section. If so, you'll set the project type on the PPC Mac OS X Target pane.

Packaged applications require an information property list, named `Info.plist`. "The Information Property List and .plc Files" (page 32) describes this list, as well as differences in how to create it in CodeWarrior and Xcode.

Build the Application in the Mach-O Format

Mach-O is the native executable format in Mac OS X and is the only format supported by Xcode. Applications that use Mach-O format have access to all native Mac OS X APIs, such as Quartz and POSIX, and can more easily support symbolic debugging with GDB. However, Mach-O applications cannot run in earlier versions of the Mac OS. For a summary of the advantages and issues of using Mach-O format, see "Use the Mach-O Binary Format" in *Performance Overview*.

If your project currently builds a CFM application or library, you may want to add a target to the project to build a Mach-O version. This will allow you to isolate changes you'll need for Mach-O before switching to Xcode. And you'll still benefit from the automated setup performed when you import the project into Xcode. However, for a simple project, you may end up doing extra work.

Conversely, if your CodeWarrior project is not particularly complex, you may want to import it directly into Xcode (and switch to the Mach-O executable format at the same time). If you do, many changes will be handled automatically, including changing your linker settings and moving from MSL libraries to the standard C and C++ libraries. A disadvantage is that you'll be making more changes in a new environment, where problems may be harder to isolate, so this approach isn't recommended for more complex projects.

Whichever approach you take, you will still have some work to do after importing, as described in "After Importing a Project" (page 53).

Convert to Mach-O Format

"C and C++ Libraries" (page 25) describes differences between the CodeWarrior MSL libraries and the standard C and C++ libraries for Mach-O. To build your application in Mach-O format, you'll need to make these changes:

- On the Target pane of the Target Settings window, change your Linker setting to "Apple Mach-O PowerPC" (preferred) or "Mac OS X PowerPC Mach-O".

When you save your settings, CodeWarrior adds a Frameworks tab to your project window, if it didn't already have one.

- If you load plug-ins via CFM, you'll need to rewrite that code to use the CFBundle or CFPlugin APIs.

  You can read more about CFBundle in the document *Bundle Programming Guide*.

- If your code makes assumptions about the size of the C++ `bool` type, you may have to make changes: in Mach-O, a `bool` is four bytes wide, not one. See "Make Code Changes for GCC Compatibility" (page 54) for details, and for other possible code issues.

The sections that follow list additional steps you'll need to take in converting to use Mach-O format.

Use Framework Headers

"Framework-Style Headers" (page 20) provides background on working with framework-style headers. To continue making the switch to a Mach-O target, you'll need to take these steps to start using framework headers:

- Remove `{Compiler}MacOS Support` from the Access Paths pane of the Target Settings window.
- Add necessary frameworks to your project. For example, you should include `System.framework`. You'll probably also want to include `Carbon.framework`, and may need to include other frameworks, such as `CoreAudio.framework` or `OpenGL.framework`, if your project uses the APIs they define.

  Remember that the Carbon framework is an umbrella framework (it includes many other frameworks), so it may include most of the framework headers you need. Look in the header file `Carbon.h` within the framework for a complete list of the frameworks the Carbon framework includes.

  To add a framework, you can drag it from the Finder to the Frameworks tab in your project window. Or you can choose Project > Add Files and navigate to `/System/Library/Frameworks`.

  When you first add a framework such as the Carbon framework, CodeWarrior adds a path like the following to your access paths:

  ```
  {OS X Volume}System/Library/Frameworks
  ```

These steps are recommended, but not required.

- Remove all `#include <MacHeader.h>` statements from your source and header files.
- Use just the statement `#include <Carbon/Carbon.h>` in your prefix file. More information on the prefix file is provided in "Replace Your Prefix File" (page 48).

Use C99 Standard in Language Settings

Applications that change from using Universal Interfaces to using framework-style include statements (in either CodeWarrior or Xcode), must conform to the C99 standard, as described in "Conform to the C99 Standard" (page 54). In CodeWarrior, you can select the "Enable C99 Extensions" setting on the C/C++ Language pane in the Target Settings window.

Migrate from MSL to System C and C++ Libraries

"C and C++ Libraries" (page 25) describes differences between the libraries you use in CodeWarrior and Xcode. To continue the switch to a Mach-O target, perform the following steps to change libraries:

- Remove any MSL libraries (such as `MSL_All_CarbonD.Lib`) from the Libraries folder in the Files tab of your project.
- Remove any access paths to MSL libraries from the Access Paths pane of the Target Settings window for the target.
- Add paths to the Access Paths pane for `{OS X Volume}usr/include` and `{OS X Volume}usr/lib`.
- Add the file `/usr/lib/libSystem.dylib` to your project. This is a symlink that will resolve to the appropriate version of the system library.
- You may need to add the file `crt1.o` (located in `/usr/lib`) to the project and make it the first file in the Link Order settings tab.

Replace Your Prefix File

"Precompiled Headers and Prefix Files" (page 21) describes differences in how you use prefix files in CodeWarrior and Xcode. To continue the switch to a Mach-O target, you'll have to move away from the CodeWarrior precompiled headers. You can do that in one of two ways:

- Make your own prefix file and precompile it. As mentioned in a previous section, some projects can use just the statement `#include <Carbon/Carbon.h>` in your prefix file. Of course you may need to construct a more complex prefix file.

  In the C/C++ Language pane of the Target Settings window, enter the name of your file in the Prefix File text field.
- Or you can modify the source for the CodeWarrior precompiled header so that it precompiles against the headers in `/usr/lib`, rather than the MSL headers, then re-precompile it.

Test Your Mach-O Target

At this point you should be ready to test your Mach-O target.

Remove Unnecessary Targets From the Project

To simplify the conversion process even further, you can make a copy of the project and delete any targets other than the targets you will import into Xcode. That will make importing faster and the resulting project will have just the desired target. Of course, you can also delete unneeded targets after importing into Xcode.

Importing a CodeWarrior Project Into Xcode

This chapter provides a brief walk-through of importing a CodeWarrior project into Xcode. It assumes you've read "Xcode From a CodeWarrior Perspective " (page 13) and taken any appropriate preparatory steps listed in "Preparing a CodeWarrior Project for Importing" (page 45).

About the Importer

The project importer in Xcode communicates with CodeWarrior through Apple events. Once you've specified a CodeWarrior project to import, the importer tells CodeWarrior to make a copy of the project, then sets various settings and performs other operations to prepare the copy. The importer tells CodeWarrior to export the copied project, then reads the exported XML and uses that information to set up the new Xcode project. The importer displays a status dialog while importing. On completion, the status dialog is dismissed and the new project opens.

Some other features include:

- By default, the CodeWarrior importer adds a Source Tree entry to your Xcode preferences named "CodeWarrior". This entry maps to CodeWarrior's `{Compiler}` relative path access path. The importer will not overwrite an existing source tree of the same name.
- The importer can optionally import global source tree data from CodeWarrior.
- The new project uses the Xcode native build system for all imported CW targets. This system is described in "Native Build System" (page 34).
- The importer supports importing cross-project references.
- The importer supports importing dependent projects automatically.
- You can write an AppleScript script to automate importing CodeWarrior projects, using the `import` AppleScript command (in the Xcode Application Suite).

For Best Results When Importing

Important: CodeWarrior must be running when you import your project.

For best results when importing, you should follow these steps:

- Some import steps can be CPU-intensive and time-consuming—for example, importing multiple projects (due to dependent projects) can launch many threads, while searching for included files in a large project can take some time.

  Avoid performing other compute-intensive tasks during large imports. If possible, allow the machine to work only on the import.
- Use CodeWarrior Pro version 8.3 or later for projects that you will import into Xcode. The earlier your version of CodeWarrior, the more likely it is that problems will occur.

Known Issues With the Importer

The following are some problems that have been observed. For the latest information, see the *Xcode 3.0 Release Notes*.

- The importer does not currently support bringing in CW Targets that use the Java Linker or the Mac OS Merge Linker.
- A successful import does not imply that your project will build successfully, since (by design) the importer does not modify source files. See "After Importing a Project" (page 53) for additional steps you may need to take.
- It may happen that an AppleScript or CodeWarrior error occurs during the import. If so, check the Console for details and try importing again.

  One possible cause is that you previously quit CodeWarrior with a project open, then moved or deleted that project. When the importer sends an Apple event to launch CodeWarrior, CodeWarrior puts up an error dialog reporting it can't find the previously open project. That freezes the process, and eventually the Apple event times out and the import stops. The importer status dialog may remain on screen until you next quit and relaunch Xcode.

  If an import fails repeatedly, please file a bug with as much detail as possible. See "Feedback and Mail List" (page 11) for information on filing bugs.
- If CodeWarrior prompts you to install the Interface Builder connection and your project fails to import, make sure that you have CodeWarrior running before trying the import again.
- Even though your CodeWarrior application uses Carbon, if it doesn't use framework-style headers, the importer may not add the Carbon framework to the new Xcode project.

Importing a Project

Importing a CodeWarrior project is very straightforward. First, make sure you've read the information in "For Best Results When Importing" (page 49) and "Known Issues With the Importer" (page 50). Then follow these steps:

1. Choose File > Import Project, which opens the Import Project assistant.

2. Select Import CodeWarrior Project and click the Next button to bring up the pane shown in Figure 3-1.

Figure 3-1 The Import CodeWarrior Project pane

3. Click the Choose button and navigate to the CodeWarrior project file to import (or type in the path and filename).

   When you choose or type a project file to import, Xcode automatically uses the same name for the new project. However, you are free to change the project name in the New Project Name field. The new project is created in the same folder as the CodeWarrior project file, and has the extension `.xcodeproj`. (If you are using Xcode 2.0 or earlier, the project has the extension `.xcode`.)

4. When you import a project, Xcode determines the location of the CodeWarrior root folder (commonly referred to as `{Compiler}` in CodeWarrior's access paths) and adds it to the Source Trees list in the Preferences window, with the Setting Name "CodeWarrior" and Display Name "CodeWarrior Folder".

   Select the checkbox Import "Global Source Trees" from CodeWarrior if you want the importer to add any global source trees from CodeWarrior's preferences to the Source Trees list in Xcode. For more information about Source Trees, see "Source Trees" (page 40).

   If you select the checkbox "Import referenced projects," Xcode also imports any projects that the selected CodeWarrior project references. For any referenced projects it imports, the importer uses the name of the referenced CodeWarrior project, with an extension of `.xcodeproj`.

5. Click the Finish button to dismiss the assistant and start the import.

6. You can follow the process of the import in the Import status window. Once the import is complete, the new project window opens, and in some cases asks you to choose an encoding.

In most cases, you should be able to choose the suggested encoding (in this case, "Western (Mac OS Roman)"). If you are unsure, click the Give Me Advice button for more information.

For most projects, it is unlikely that the software will build and run immediately after importing. See "After Importing a Project" (page 53) for steps you may need to take to get your imported CodeWarrior project to build in Xcode.

After Importing a Project

Importing into Xcode is a good starting point for converting your project, but you'll typically have some work to do before it successfully builds and runs. For example, the Xcode importer does a good job of duplicating the structure of a CodeWarrior project, but you'll need to make sure the build settings it chooses are correct for your project. It's likely that you are the best judge of settings such as compiler optimization, and you should check other build settings as well.

Of course if you prepared your code carefully before importing your project, and if your code is already fantastically clean and standard, you will have fewer changes to make after importing. But you are still likely to face some of the simple problems or some of the more subtle issues described in this chapter.

Build Your Project

Once you've imported your project into Xcode, use one of the commands from the Build menu, such as Command-B, to build it. To view the full build report, choose Build > Build Results (or type Command-Shift-B). In the resulting Build Results window, you can open the build log to show the most detailed results; to do so, click the button to the left of the arrow, below the top pane of the window. This group of buttons allow you to control the amount of information displayed.

You can also view any errors or warnings generated during the build in the Errors and Warnings group in the project window. See the section "The GCC Compiler" (page 26) for information on how to control the actual level of warnings generated by GCC.

Check for Common Conversion Issues

The following are some common steps you may need to take to get your project to build:

- Specify a prefix file. For an overview, see "Precompiled Headers and Prefix Files" (page 21). For the actual steps, see "Using Precompiled Headers and Prefix Files" (page 42).
- Your application may require compiler settings for GCC that the importer does not pick up. See "The GCC Compiler" (page 26) for information about working with compiler settings.
- You may need to add a Mac OS library to the project. You typically only need to explicitly include a library if you are statically linking your software.

You will also need to rename any static libraries so that they match the format `lib*.a`. Otherwise, Xcode will not link them.

- For additional issues, see these sections:
  - "C and C++ Libraries" (page 25)
  - "Runtime and Library Issues" (page 25)
  - "Migrate from MSL to System C and C++ Libraries" (page 48)

Resolve Undefined Symbols

You may get undefined symbol errors when you build an Xcode project imported from CodeWarrior. This is most likely because the import process did not pick up a needed source file or framework. To add these items:

- You can drag any missing source files into the project group in the Groups & Files list of the project window or add them with Project > Add to Project.

  In either case, Xcode adds a reference to the file to your project. You also get a chance to specify whether to copy the file into the project directory, and to specify which targets to add the file to.
- Add any needed frameworks to the project, by dragging them to the External Frameworks and Libraries group or using Project > Add to Project, which displays an Open dialog. Again, you will also specify which target to add the framework to.

Make Code Changes for GCC Compatibility

This section describes a number of code changes you may need to make to ensure compatibility with GCC version 4.0. It also includes some changes related to switching from MSL libraries to the standard Mac OS X C and C++ libraries. See also "Code Differences" (page 43) for a brief list of code differences.

Note: Some of the issues listed here are due to current GCC compiler bugs, which may be fixed in future releases.

Conform to the C99 Standard

Applications that change from using Universal Interfaces to using framework-style include statements (in either CodeWarrior or Xcode), must conform to the C99 standard. This is required because the header `MacTypes.h` defines the `bool` type as defined in the C99 standard. This header is included into frameworks such as the Carbon and Cocoa frameworks, and thus a large percentage of Mac OS software projects. Projects that pick up the header `MacTypes.h` through framework headers (for example, with `#include <Carbon/Carbon.h>`) don't have a choice to not include individual header files.

Because C99 is largely a superset of the C89 standard, most code that conforms to the C89 standard should compile with the C99 setting. To avoid compilation errors, however, observe the following restrictions:

- The `main()` function must be type `int` and have an argument list of either `(void)` or `(int argc, char * argv)`.
- Some implicit conversions that are valid in C89/C90 are illegal in C99.
- Keywords (such as `bool`, `true`, and `false`) that are free for use by developer code in C89/C90 are defined by the compiler or standard headers in C99 and may cause redefined-symbol errors.

Inlining Thresholds

Inlining is an optimization the compiler performs to trade off code size for speed. On the theory that function call overhead can be high for small functions called within loops, inlining places the body of the function "in line" with the caller, so no function call overhead is needed.

The trade-off of code size for speed can backfire, however, if it generates too much inlined code; the bloat in code size can itself become a performance problem. Both CodeWarrior and GCC allow control of how much inlining is enough, but the two mechanisms differ substantially, and there's no direct correlation between them. When moving a project from CodeWarrior to GCC you may need to experiment with the inlining settings to get the results you want.

CodeWarrior allows control of inlining depth: if an inlined function calls another inlined function, both are expanded in line. CodeWarrior lets you set the depth at which inlining stops.

The GCC compiler has no specific control over inlining depth, but does let you control the total expanded size of inlined functions. By default, inlined functions over 600 "pseudo-instructions" will not be inlined, regardless of whether the size is a result of deep inlining or not. The custom compiler flag `-finline-limit=n` lets you control the maximum size of inlined functions. You can set this compiler flag in the Other C Flags build setting in the Build pane in the target inspector window.

Note: Between GCC 3.3 and GCC 4.0, significant improvements have been made to the algorithms for inlining functions. As a result, using the `-finline-limit=n` compiler flag produces different results when compiling with GCC 3.3 than when compiling with GCC 4.0, even for the same value of the `-flnline-limit` compiler flag. You should profile your code and test your inlining settings with the appropriate compiler version. For more information on differences in optimization between GCC 3.3 and GCC 4.0, see *GCC Porting Guide*.

Dealing With Pragmas

Before reading this section, you should be familiar with the information in "Pragma Statements" (page 22), which describes some of the differences in how pragmas are used by the CodeWarrior and GCC compilers.

In general, GCC supports only a small number of pragmas, including those for structure alignment, and instead uses compiler settings to accomplish similar tasks. Unsupported pragma statements are generally ignored, depending on the current warning settings. Compiler settings can be set on a per-project or a per-file basis.

GCC does not support `#pragma export on` and `#pragma export off`. See "Exporting Symbols" (page 37) for information on how to work with exported symbols.

If you have header files that use the statement `#pragma options(!pack_enums), enum=int, enumsalwaysint on`, you can instead add the flag `-fshort-enums` for those files by enabling the Short Enumeration Constants build setting.

If you need to isolate GCC-specific or CodeWarrior-specific code, use the conditional statements described in "Code Differences" (page 43).

Packing and Alignment Differences

This section lists some packing and alignment differences between the CodeWarrior compiler and GCC.

Different size and alignment of enums

GCC gives all enums `int` (4-byte) size and alignment. By default, CodeWarrior bases the size and alignment of enums on the smallest integer type able to contain the declared range of values. However, you can use the CodeWarrior C/C++ Language setting "Enums Always Ints" to specify the use of int-sized enums.

Different alignment of doubles

GCC gives doubles 8-byte alignment only when they are the first member of a struct. CodeWarrior gives all doubles appearing at the "top level" in a struct 8-byte alignment if the first member of the struct is a double. In some unusual cases, such as the example shown in Listing 4-1, this can lead to different alignment.

Listing 4-1 A structure definition that is interpreted differently

```
struct s { double f1; char f2; double f3; };
```

With this code, GCC will give the variable `f3` 4-byte alignment, while CodeWarrior will give it 8-byte alignment.

Code size incompatibility with power alignment

When compiling the code in Listing 4-2, GCC and CodeWarrior disagree on the proper size of the following struct when compiled with `power` alignment. For CodeWarrior, `sizeof(ValueRangeCriteriaData) = 68`, while for GCC, `sizeof(ValueRangeCriteriaData) = 72`.

Listing 4-2 Code that generates a size incompatibility

```
#pragma options align=power
struct ValueRangeCriteriaValue
{
    SInt64 value;
    SInt64 valueOffset;
    SInt64 valueOffsetMultiplier;
};
typedef struct ValueRangeCriteriaValue ValueRangeCriteriaValue;
```

```
struct ValueRangeCriteriaData
{
    ValueRangeCriteriaValue startValue;
    ValueRangeCriteriaValue endValue;
    UInt32              reserved[5];
};
typedef struct ValueRangeCriteriaData ValueRangeCriteriaData;
```

Data packed differently in a register

When compiling the code in Listing 4-3, Codewarrior packs the data for the variable `i` into the high two bytes of the register, but GCC packs the data into the low two bytes of the register. This causes an incompatibility wherever a type like this is used.

Listing 4-3 Code that generates a register difference

```
union Opaque16 {
    char   b[2];
    short  notanInt
};

void bar (Opaque16 i)
{
printf("i = %d\n");
}

void main (void)
{
  Opaque16 i;

  i = 5;
  bar(i);
}
```

The Mach-O bool Type is Four Bytes, Not One

In GCC 4.0, the default size of a `bool` variable is four bytes. In CodeWarrior, the size is one byte. You should avoid using code that counts on the size of a `bool`. You should also use care in working with `bool` pointers.

Note: GCC does provide the flag `-mone-byte-bool` to force GCC to use one byte for `bool` types. However, code generated with this flag set may not be binary compatible with code generated without it or with Mac OS X frameworks. The Xcode build setting for setting this flag is Use One Byte 'bool' (GCC_ONE_BYTE_BOOL).

Note: The size and alignment of the bool data type is different for 32-bit and 64-bit architectures. This document assumes that you are building for a 32-bit architecture. For more information on building for a 64-bit architecture, see *64-Bit Transition Guide*.

Guarding Header Includes for C++

There are cases where if you explicitly include individual Mac OS X system headers from C++ code, you'll get link failures with the system libraries. To prevent this, you should isolate your header includes by adding this at the beginning of your file, after the include guard:

```
#ifdef __cplusplus
extern "C" {
#endif
```

and this at the end, before the include guard:

```
#ifdef __cplusplus
}
#endif
```

Friend of Template Specialization Can't Access Private Members

With GCC, a class declared as a friend of a template specialization does not have access to private members.

Float to Integer Conversions Give Incorrect Output

The ANSI C standard states that the result of converting an unrepresentable floating point value to integer is undefined, but most implementations produce a result that can be usefully detected. However, for some such conversions, GCC 3.3 produces a meaningless result. For example, consider the results of the following casts:

```
d = -1.79769e+308;
(long)d                 // result: -2147483648 (should be -2147483648)
(int64_t)d              // result: 1 (should be -9223372036854775808)
(long long)d            // result: 1 (should be -9223372036854775808)
(unsigned char)d        // result: 0 (should be 0)
```

Some Casts Don't Work in Function Lists

In some cases in GCC, C style casts or functional style casts do not work correctly in an argument list because argument conversions are applied before the cast conversion is applied.

Standard C Library Functions Are in Global Namespace

With GCC, standard C library functions are in the global namespace, not in `std::`.

Vector Load Indexed Intrinsic Is Not Const Correct

The `vec_ld` (vector load indexed) intrinsic is not const correct, and generates an unclear diagnostic message. For example, the code in Listing 4-4 generates the error message

```
invalid conversion from `vector <anonymous>*' to `vector <anonymous>*'
```

As shown in the listing, you can use a cast to eliminate the warning.

Listing 4-4 Code that generates error

```
const vector unsigned char* srcPtr;
    ...
    // This line gives an INCORRECT warning
    vector unsigned charv1 = vec_ld(0, srcPtr);

    // casting away the constness gets rid of the warning,
    //  but shouldn't be necessary and complicates the code
   vector unsigned charv2 = vec_ld(0, const_cast<vector unsigned char*>(srcPtr));
```

CodeWarrior Allows Anonymous Unused C Function Arguments

CodeWarrior allows unused arguments to C functions to have their names omitted.

```
int foo(float)
{
  return 0;
}
```

This is allowed by the C99 standard, but generates a warning in GCC.

GCC Interprets Some #define Values Differently Than CodeWarrior

The CodeWarrior compiler treats "true" the same as "1" in `#define` statements, but GCC treats them differently. For example, in Listing 4-5 (page 60), both code snippets generate the same result in CodeWarrior. With GCC, however, <some code> doesn't get executed in the first snippet.

Listing 4-5 `#define` test code

```
// Snippet 1: <some code> executed only in CodeWarrior
#define MY_FEATURE true

#if MY_FEATURE
<some code>
#endif

// Snippet 2: <some code> executed in CodeWarrior or GCC

#define MY_FEATURE 1

#if MY_FEATURE
<some code>
#endif
```

GCC also does not currently provide a warning when an undefined identifier is evaluated in a `#if` directive (Snippet 1).

GCC asm Intrinsics are Preprocessor Macros

Compilers support intrinsics as an alternative to using asm statements within functions. However, in some cases GCC uses macros, not true functions, so they can't substitute arguments. As a result, code that compiles on CodeWarrior generates an error with GCC.

For example, compiling the following line with GCC generates an error that the number of parameters does not match (expected 5, got 3):

```
sFlags = __rlwimi(sFlags, inValue, INSERT_HALT);
```

However, `INSERT_HALT` is a `#define`:

```
#define INSERT_HALT 8,23,23
```

To work around this, you can replace the define with the actual value:

```
sFlags = __rlwimi(sFlags, inValue, 8,23,23);
```

Note: If you use intrinsics in CodeWarrior, you can continue to do so in Xcode by including the file `ppcintrinsics.h`.

Para

Xcode's Rez tool Doesn't Support Certain Operators

In CodeWarrior, you can use << and >> as shift operators in a resource file, but Xcode's version of Rez doesn't currently support using these symbols as operators.

Move Settings From the .plc File to an Info.plist File

Any packaged Mac OS X software requires an information property list file named `Info.plist`. CodeWarrior projects that create similar types of software use a `.plc` file to supply this information. You can read about differences between these approaches in "The Information Property List and .plc Files" (page 32).

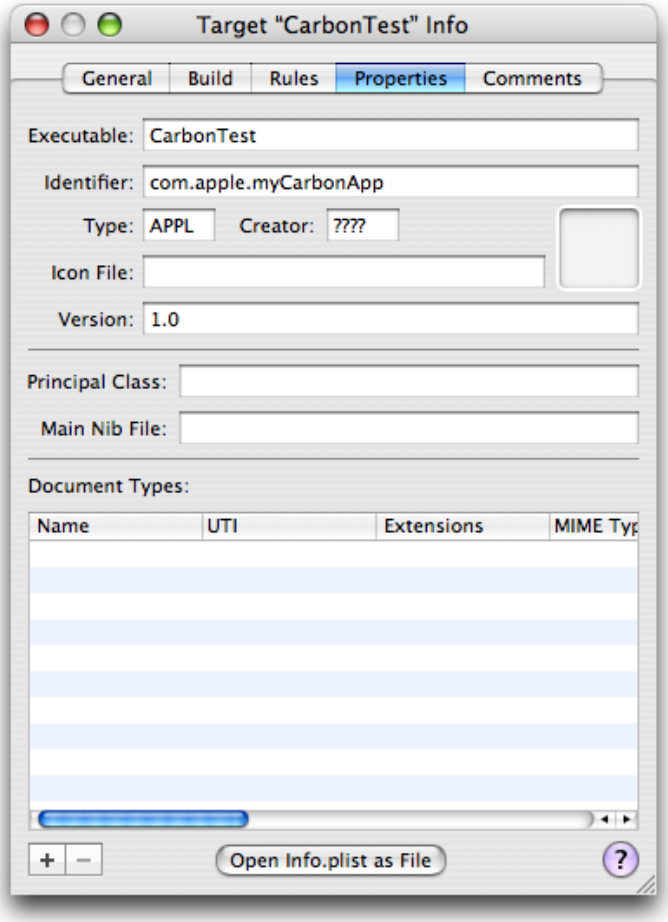
When you import a project into Xcode, the importer does extract property list setting information from the CodeWarrior project, but you should check the information and make any necessary additions or modifications. If your CodeWarrior project doesn't have a `.plc` file, you'll have to supply all the necessary information.

In Xcode, you supply or modify this information by opening a target editor and making changes in the Properties pane. Figure 4-1 shows the Properties pane after importing a project based on CodeWarrior project stationery.

You can also move property list entries from your CodeWarrior project to your Xcode project by the following steps:

1. In CodeWarrior, verify that the Property List Compiler Output pop-up on the Property List pane of the target settings window is set to "Output as file."
2. Build your CodeWarrior project, so that it creates an `Info.plist` file from the `.plc` file. The property list file is part of the application bundle, so it is not typically visible in the Finder.
3. Open the `Info.plist` file as text. You can do this by Control-clicking the application file and choosing Show Package Contents, navigating to the file within the `Contents` folder, and double-clicking it (which should open the file in a CodeWarrior editor window).
4. In the project window of your Xcode project, open the Resources group within the project group in the Groups & Files list. You should see an `Info.plist` file.
5. Double-click the `Info.plist` file to open it in an Xcode editor window.
6. You can cut and paste key-value entries from the CodeWarrior property list to the Xcode property list. Of course, you'll have to observe the XML format.

Figure 4-1 The Properties pane in a target editor



Make Changes to PowerPlant

If your project uses PowerPlant, importing the project should include all the PowerPlant source files. To build successfully with GCC 4.0, however, you'll have to make a few changes to the code.

To build PowerPlant in your Xcode project, you perform these steps, described in the following sections:

1. Add PowerPlant's headers to the project and target.
2. Create a prefix file in Xcode that includes the PowerPlant headers and the necessary definitions.
3. Make minor changes to the PowerPlant code to build with the GCC 4.0 compiler.

For related information in other sections, see "The GCC Compiler" (page 26), "Framework-Style Headers" (page 20), and "Precompiled Headers and Prefix Files" (page 21).

Add PowerPlant Headers to the Project and Target

This requirement should be taken care of automatically when you use the Xcode project importer to import a project that uses PowerPlant. If you find that any files are missing when you attempt to build the project, you can drag them into the project from the Finder, or use Project > Add to Project in Xcode.

Create a Prefix File for PowerPlant

After adding the PowerPlant header files, you need to disentangle the header files from the CodeWarrior precompiled header mechanism. To do this, you need to create a prefix header for PowerPlant in Xcode that includes information from the following PowerPlant files: `CommonMach-OPrefix.h`, `DebugMach-OPrefix.pch++`, `PP_ClassHeaders.cp`, `PP_DebugHeaders.cp`, and `PP_MacHeadersMach-O.c`.

Listing 4-6 shows a header file that contains the needed definitions and include statements from these CodeWarrior header files. You can use this header file for a debug or a final target. See "Creating Debug and Non-Debug Products" (page 35) for more information.

Listing 4-6 An Xcode prefix header file for PowerPlant

```
/*
 *  PP_Xcode.h
 *
 *  Created on Wed Jan 29 2003.
 *  Copyright (c) 2003 __MyCompanyName__. All rights reserved.
 *
 */

#define _STD std
#define _CSTD std
#define __dest_os __mac_os_x

#define PP_Target_Carbon 1

#define PP_Target_Classic (!PP_Target_Carbon)


// ---------------------------------------------------------------------
//  Options

#define PP_Uses_PowerPlant_Namespace 0
#define PP_Supports_Pascal_Strings 1
#define PP_StdDialogs_Option PP_StdDialogs_NavServicesOnly

#define PP_Uses_Old_Integer_Types 0
#define PP_Obsolete_AllowTargetSwitch 0
#define PP_Obsolete_ThrowExceptionCode 0
#define PP_Warn_Obsolete_Classes 1

#define PP_Suppress_Notes_2 211

//
// Carbon headers
```

```
#include <Carbon/Carbon.h>
//
// PowerPlantheaders
    // Action Classes
#include <LAction.h>
#include <LUndoer.h>
#include <UTETextAction.h>
#include <UTEViewTextAction.h>
    // AppleEvent Classes
#include <LModelDirector.h>
#include <LModelObject.h>
#include <LModelProperty.h>
#include <UAppleEventsMgr.h>
#include <UExtractFromAEDesc.h>
    // Array Classes
#include <LArray.h>
#include <LArrayIterator.h>
#include <LComparator.h>
#include <LRunArray.h>
#include <LVariableArray.h>
#include <TArray.h>
#include <TArrayIterator.h>
    // Commander Classes
#include <LApplication.h>
#include <LCommander.h>
#include <LDocApplication.h>
#include <LDocument.h>
#include <LSingleDoc.h>
    // Feature Classes
#include <LAttachable.h>
#include <LAttachment.h>
#include <LBroadcaster.h>
#include <LDragAndDrop.h>
#include <LDragTask.h>
#include <LEventDispatcher.h>
#include <LListener.h>
#include <LPeriodical.h>
#include <LSharable.h>
    // File & Stream Classes
#include <LDataStream.h>
#include <LFile.h>
#include <LFileStream.h>
#include <LHandleStream.h>
#include <LStream.h>
    // Pane Classes
#include <LButton.h>
#include <LCaption.h>
#include <LCicnButton.h>
#include <LControl.h>
#include <LDialogBox.h>
#include <LEditField.h>
#include <LFocusBox.h>
#include <LGrafPortView.h>
#include <LListBox.h>
#include <LOffscreenView.h>
#include <LPane.h>
#include <LPicture.h>
#include <LPlaceHolder.h>
```

```
#include <LPrintout.h>
#include <LRadioGroupView.h>
#include <LScroller.h>
#include <LStdControl.h>
#include <LTabGroupView.h>
#include <LTableView.h>
#include <LTextEditView.h>
#include <LView.h>
#include <LWindow.h>
#include <UGWorld.h>
#include <UQuickTime.h>
    // PowerPlant Headers
#include <PP_Constants.h>
#include <PP_KeyCodes.h>
#include <PP_Macros.h>
#include <PP_Messages.h>
#include <PP_Prefix.h>
#include <PP_Resources.h>
#include <PP_Types.h>
                                    // Support Classes
#include <LClipboard.h>
#include <LFileTypeList.h>
#include <LGrowZone.h>
#include <LMenu.h>
#include <LMenuBar.h>
#include <LRadioGroup.h>
#include <LString.h>
#include <LTabGroup.h>
#include <UDesktop.h>
                                    // Utility Classes
#include <UAttachments.h>
#include <UCursor.h>
#include <UDebugging.h>
#include <UDrawingState.h>
#include <UDrawingUtils.h>
#include <UEnvironment.h>
#include <UException.h>
#include <UKeyFilters.h>
#include <UMemoryMgr.h>
#include <UModalDialogs.h>
#include <UPrinting.h>
#include <UReanimator.h>
#include <URegions.h>
#include <URegistrar.h>
#include <UScrap.h>
#include <UScreenPort.h>
#include <UTextEdit.h>
#include <UTextTraits.h>
#include <UWindows.h>
```

Make Minor Changes to the PowerPlant Code

To build PowerPlant in Xcode with GCC 4.0, you need to make a relatively small number of changes to the source code. These changes are described here, with accompanying code listings. The line numbers are based on source files from CodeWarrior Pro version 8.3.

To avoid a GCC error, in the file LGATabsControlImp.cp, starting at line 964, move the declaration of tabButton outside the kControlTabEnabledFlagTab case, as shown in Listing 4-7.

Listing 4-7 Modified switch statement in LGATabsControlImp.cp

```
    switch (inTag) {

//      case kControlTabContentRectTag: {
//          Rect contentRect = *(Rect *)inDataPtr;
                // ••• Now what do we do with this? Do we resize the
                // control to fit the content rect?
//          break;
//      }

        LGATabsButton*tabButton;
        case kControlTabEnabledFlagTag: {
            Boolean enableIt = *(Boolean *)inDataPtr;
            tabButton = GetTabButtonByIndex(inPartCode);
            if (tabButton != nil) {
                if (enableIt) {
                    tabButton->Enable();
                } else {
                    tabButton->Disable();
                }
            }
            break;
        }

        case kControlTabInfoTag: {
            tabButton = GetTabButtonByIndex(inPartCode);
            if (tabButton != nil) {
                ControlTabInfoRec*info = (ControlTabInfoRec*) inDataPtr;
                tabButton->SetDescriptor(info->name);
                tabButton->SetIconResourceID(info->iconSuiteID);
            }
            break;

        default:
            LGAControlImp::SetDataTag(inPartCode,
                                inTag, inDataSize, inDataPtr);
            break;
        }
    }
```

In LStream.h, you'll need to enclose several method definitions in conditional directives to avoid redefinition with GCC. Listing 4-8 shows the first change, at line 152.

Note: Commenting out these methods makes them unavailable to client code. Usage of these methods is rare, but if they are used in your application, you may need to make additional source changes.

Listing 4-8 LStream.h modifications at line 152

```
#ifndef __GNUC__
    LStream&operator << (long double inNum)
                        {
                            (*this) << (double) inNum;
```

```
                        return (*this);
                    }

    LStream&operator << (short double inNum)
                    {
                        (*this) << (double) inNum;
                        return (*this);
                    }
#endif
```

Listing 4-9 shows the second change to `LStream.h`, at line 172.

Listing 4-9 `LStream.h` modifications at line 172

```
#ifndef __GNUC__
    LStream&operator >> (long double &outNum)
                    {
                        doublenum;
                        (*this) >> num;
                        outNum = num;
                        return (*this);
                    }

    LStream&operator >> (short double &outNum)
                    {
                        double num;
                        (*this) >> num;
                        outNum = (short double) num;
                        return (*this);
                    }
#endif
```

You'll also have to make a change to `LException.h` for GCC. Change the definition of the class destructor to agree with the standard C++ library exception definition, as shown in Listing 4-10. The class listing begins at line 23, while the changes begin at line 33.

Listing 4-10 `LException.h` modifications at line 33

```
class LException : public PP_STD::exception {
public:
                    LException(
                            SInt32      inErrorCode,
                            ConstStringPtrinErrorString = nil);

                    LException( const LException& inException );

    LException& operator = ( const LException& inException );

#ifndef __GNUC__
    virtual         ~LException();
#else
    virtual         ~LException() throw();
#endif
    virtual const char*what() const throw();
```

You'll also have to change the implementation of the class destructor to agree with standard C++ library exception definition, as shown in Listing 4-11. The class listing begins at line 90, while the changes begin at line 94.

Listing 4-11 LException.cp modifications at line 94

```
// ---------------------------------------------------------------------
//  • ~LException               Destructor        [public]
// ---------------------------------------------------------------------

#ifdef __GNUC__
LException::~LException() throw()
#else
LException::~LException()
#endif
{
}
```

Changes for a Debug Build

The following changes are required after importing a CodeWarrior PowerPlant project with a Debug target.

In LDebugStream.cp, you'll have to modify the following (occurring at line 1150):

```
UInt8* theFirstByte = static_cast<UInt8*>(const_cast<UInt8*>(inPtr));
```

You can instead use the line shown in Listing 4-12:

Listing 4-12 LDebugStream.cp modification at line 1150

```
const UInt8* theFirstByte = ((const UInt8*) inPtr);
```

In LCommanderTree.cp (at line 87) and in LPaneTree.cp (at line 81), you'll have to modify the following lines (which have a slightly different error message in LPaneTree.cp):

```
#if __option(RTTI)
    #include <typeinfo>
#else
    #error "RTTI option disabled -- Must be enabled for LCommanderTree to function"
#endif
```

You can instead use the lines shown in Listing 4-13:

Listing 4-13 Modification to LCommanderTree.cp at line 87 (and to LPaneTree.cp at line 81)

```
#ifdef __GNUC__
    #include <typeinfo>
#else
    #if __option(RTTI)
        #include <typeinfo>
    #else
        #error "RTTI option disabled -- Must be enabled for LCommanderTree
to function"
    #endif
#endif
```

Using PowerPlant in Universal Binaries

You can use PowerPlant on an Intel-based Macintosh computer only after you make the changes detailed in this section. Most of the substantial changes that you need to make are to the PowerPlant `LStream` class. PowerPlant uses `LStream` for a number of different I/O tasks, of which the most important is reading to and writing from `'PPob'` resources. "Changing LStream Code" (page 69) provides a detailed description of all the `LStream`-related changes that are required.

If you use the `LDataBrowser` PowerPlant class and the `'DBC#'` resource, you might also need to write a resource flipper. See "Flipping the 'DBC#' Resource Type" (page 76).

Changing LStream Code

The changes discussed in this section cause `LStream` to always read and write big-endian data. After you make these changes

- `PPob` resources will work correctly on both PowerPC-based and Intel-based Macintosh computers, with the possible exception of custom types used by your application. Custom types will also work correctly if you ensure that the `LStream` constructors used by your custom types employ the streaming methods of `LStream` rather than the byte-manipulation methods.
- Data read from and written to disk or to the network and that is written with `LStream` will be portable between PowerPC-based and Intel-based Macintosh computers, with the exception of calls to the methods shown in Listing 5-1. For calls to those methods, your code should either swap bytes to big-endian format or switch to using the streaming methods of `LStream`.

Make sure that you evaluate all calls to the methods in Listing 5-1 (including calls to any subclassed versions of these methods) to see if they require byte swapping.

Listing 5-1 Calls that may require swapping bytes

```
LStream::PutBytes
LStream::WriteData
LStream::WriteBlock
LStream::operator << (Handle inHandle)
LStream::GetBytes
LStream::ReadData
LStream::ReadBlock
LStream::PeekData
LStream::operator >> (Handle &outHandle)
```

```
LStream::WritePtr
LStream::ReadPtr
LStream::WriteHandle
LStream::ReadHandle
```

If you have custom classes in a `PPob` resource, you should change their `LStream` constructors to avoid calling `LStream::ReadData`. Specifically, you need to change the code in the same way as the `LStream` constructors for `LControl`, `LListBox`, and other PowerPlant classes are changed in the following sections. The sections, organized by the files you need to change, describe the required code changes:

- "LStream.h" (page 70)
- "LStream.cp" (page 72)
- "LControl.cp" (page 72)
- "LListBox.cp" (page 73)
- "LPane.cp" (page 73)
- "LPrintout.cp" (page 74)
- "LScroller.cp" (page 74)
- "LTable.cp" (page 74)
- "LView.cp" (page 75)
- "LWindow.cp" (page 75)
- "LPopupGroupBox.cp" (page 75)
- "LControlView.cp" (page 75)
- "LScrollerView.cp" (page 76)
- "LPageController.cp" (page 76)

LStream.h

Change the `operator << (const Rect&inRect)` method from a single `WriteBlock` call to the following:

```
Rect rect;
rect.top = CFSwapInt16HostToBig(inRect.top);
rect.left = CFSwapInt16HostToBig(inRect.left);
rect.right= CFSwapInt16HostToBig(inRect.right);
rect.bottom = CFSwapInt16HostToBig(inRect.bottom);
WriteBlock(&rect, sizeof(rect));
```

Change the `operator << (const Point &inPoint)` method from a single `WriteBlock` call to the following:

```
Point pt;
pt.v = CFSwapInt16HostToBig(inPoint.v);
pt.h = CFSwapInt16HostToBig(inPoint.h);
WriteBlock(&pt, sizeof(pt));
```

Change the `operator << (SInt16 inNum)` method from a single `WriteBlock` call to the following:

```
SInt16 n;
n = CFSwapInt16HostToBig(inNum);
WriteBlock(&n, sizeof(n));
```

Change the `operator << (UInt16 inNum)` method from a single `WriteBlock` call to the following:

```
UInt16 n;
n = CFSwapInt16HostToBig(inNum);
WriteBlock(&n, sizeof(n));
```

Change the `operator << (SInt32 inNum)` method from a single `WriteBlock` call to the following:

```
SInt32 n;
n = CFSwapInt32HostToBig(inNum);
WriteBlock(&n, sizeof(n));
```

Change the `operator << (UInt32 inNum)` method from a single `WriteBlock` call to the following:

```
UInt32 n;
n = CFSwapInt32HostToBig(inNum);
WriteBlock(&n, sizeof(n));
```

Change the `operator << (float inNum)` method from a single `WriteBlock` call to the following:

```
CFSwappedFloat32 swappedFloat;
swappedFloat = CFConvertFloat32HostToSwapped(inNum);
WriteBlock(&swappedFloat, sizeof(swappedFloat));
```

Change the `operator << (bool inBool)` method from a single `WriteBlock` call to the following:

```
UInt32 boolValue;
boolValue = CFSwapInt32HostToBig(inBool);
WriteBlock(&boolValue, sizeof(boolValue));
```

In the `operator >> (Rect &outRect)` method, add this after the `ReadBlock` call:

```
outRect.top = CFSwapInt16BigToHost(outRect.top);
outRect.left = CFSwapInt16BigToHost(outRect.left);
outRect.right= CFSwapInt16BigToHost(outRect.right);
outRect.bottom = CFSwapInt16BigToHost(outRect.bottom);
```

In the `operator >> (Point &outPoint)` method, add the following after the `ReadBlock` call:

```
outPoint.v = CFSwapInt16BigToHost(outPoint.v);
outPoint.h = CFSwapInt16BigToHost(outPoint.h);
```

In the `operator >> (SInt16 &outNum)` method, add the following after the `ReadBlock` call:

```
outNum = CFSwapInt16BigToHost(outNum);
```

In the `operator >> (UInt16 &outNum)` method, add the following after the `ReadBlock` call:

```
outNum = CFSwapInt16BigToHost(outNum);
```

In the `operator >> (SInt32 &outNum)` method, add the following after the `ReadBlock` call:

```
outNum = CFSwapInt32BigToHost(outNum);
```

In the `operator >> (UInt32 &outNum)` method, add the following after the `ReadBlock` call:

```
outNum = CFSwapInt32BigToHost(outNum);
```

In the `operator >> (float &outNum)` method, replace the `ReadBlock` call with the following:

```
CFSwappedFloat32 swappedFloat;
ReadBlock(&swappedFloat, sizeof(swappedFloat));
outNum = CFConvertFloat32SwappedToHost(swappedFloat);
```

In the `operator >> (bool &outBool)` method, replace the `ReadBlock` call with the following:

```
UInt32 boolValue;
ReadBlock(&boolValue, sizeof(boolValue));
outBool = CFSwapInt32BigToHost(boolValue);
```

LStream.cp

In the `operator << (double inNum)` method, change the `#if TARGET_CPU_PPC` block to the following:

```
#if TARGET_CPU_PPC || TARGET_CPU_X86
// PowerPC and Intel doubles -- they're 8 bytes already, so just swap
// if necessary and write.

Assert_(sizeof(inNum) == 8);
CFSwappedFloat64 swappedDouble = CFConvertDoubleHostToSwapped(inNum);
WriteBlock(&swappedDouble, sizeof(swappedDouble));
```

In the `operator >> (double& outNum)` method, change the `#if TARGET_CPU_PPC` block to the following:

```
#if TARGET_CPU_PPC || TARGET_CPU_X86
// PowerPC and Intel doubles -- they're 8 bytes already, so just read
// and swap if necessary.

Assert_(sizeof(outNum) == 8);
CFSwappedFloat64 swappedDouble;
ReadBlock(&swappedDouble, sizeof(swappedDouble));
outNum = CFConvertDoubleSwappedToHost(swappedDouble);
```

LControl.cp

In the `LStream` constructor, replace this line:

```
inStream->ReadData(&controlInfo, sizeof(SControlInfo));
```

with the following lines:

```
*inStream >> controlInfo.valueMessage;
*inStream >> controlInfo.value;
*inStream >> controlInfo.minValue;
*inStream >> controlInfo.maxValue;
```

LListBox.cp

In the `LStream` constructor, replace this line:

```
inStream->ReadData(&listInfo, sizeof(SListBoxInfo));
```

with the following lines:

```
*inStream >> listInfo.hasHorizScroll;
*inStream >> listInfo.hasVertScroll;
*inStream >> listInfo.hasGrow;
*inStream >> listInfo.hasFocusBox;
*inStream >> listInfo.doubleClickMessage;
*inStream >> listInfo.textTraitsID;
*inStream >> listInfo.LDEFid;
*inStream >> listInfo.numberOfItems;
```

In the `RestorePlace(LStream*inPlace)` method, replace the following line:

```
 inPlace->ReadData(&theRect, sizeof(Rect));
```

with this line:

```
*inPlace >> theRect;
```

Further down in the method, just under the `if (vScroll != nil)` line, replace this line:

```
inPlace->ReadData(&theRect, sizeof(Rect));
```

with this line:

```
*inPlace >> theRect;
```

And once more in the same method, just under the `if (hScroll != nil)` line, replace this line:

```
inPlace->ReadData(&theRect, sizeof(Rect));
```

with this line:

```
*inPlace >> theRect;
```

LPane.cp

In the `LStream` constructor, replace this line:

```
inStream->ReadData(&thePaneInfo, sizeof(SPaneInfo));
```

with the following lines:

```
SInt32 viewPtr;
*inStream >> thePaneInfo.paneID;
*inStream >> thePaneInfo.width;
*inStream >> thePaneInfo.height;
*inStream >> thePaneInfo.visible;
*inStream >> thePaneInfo.enabled;
```

```
*inStream >> thePaneInfo.bindings.left;
*inStream >> thePaneInfo.bindings.top;
*inStream >> thePaneInfo.bindings.right;
*inStream >> thePaneInfo.bindings.bottom;
*inStream >> thePaneInfo.left;
*inStream >> thePaneInfo.top;
*inStream >> thePaneInfo.userCon;
*inStream >> viewPtr;
thePaneInfo.superView = reinterpret_cast<LView *>(viewPtr);
```

LPrintout.cp

In the `LStream` constructor, replace this line:

```
inStream->ReadData(&thePrintoutInfo, sizeof(SPrintoutInfo));
```

with the following lines:

```
*inStream >> thePrintoutInfo.width;
*inStream >> thePrintoutInfo.height;
*inStream >> thePrintoutInfo.active;
*inStream >> thePrintoutInfo.enabled;
*inStream >> thePrintoutInfo.userCon;
*inStream >> thePrintoutInfo.attributes;
```

LScroller.cp

In the `LStream` constructor, replace this line:

```
inStream->ReadData(&scrollerInfo, sizeof(SScrollerInfo));
```

with the following lines:

```
*inStream >> scrollerInfo.horizBarLeftIndent;
*inStream >> scrollerInfo.horizBarRightIndent;
*inStream >> scrollerInfo.vertBarTopIndent;
*inStream >> scrollerInfo.vertBarBottomIndent;
*inStream >> scrollerInfo.scrollingViewID;
```

LTable.cp

In the `LStream` constructor, replace this line:

```
inStream->ReadData(&tableInfo, sizeof(STableInfo));
```

with the following lines:

```
*inStream >> tableInfo.numberOfRows;
*inStream >> tableInfo.numberOfCols;
*inStream >> tableInfo.rowHeight;
*inStream >> tableInfo.colWidth;
*inStream >> tableInfo.cellDataSize;
```

LView.cp

In the `LStream` constructor, replace this line:

```
inStream->ReadData(&viewInfo, sizeof(SViewInfo));
```

with the following lines:

```
*inStream >> viewInfo.imageSize.width;
*inStream >> viewInfo.imageSize.height;
*inStream >> viewInfo.scrollPos.h;
*inStream >> viewInfo.scrollPos.v;
*inStream >> viewInfo.scrollUnit.h;
*inStream >> viewInfo.scrollUnit.v;
*inStream >> viewInfo.reconcileOverhang;
```

LWindow.cp

In the `LStream` constructor, replace this line:

```
inStream->ReadData(&windowInfo, sizeof(SWindowInfo));
```

with the following lines:

```
*inStream >> windowInfo.WINDid;
*inStream >> windowInfo.layer;
*inStream >> windowInfo.attributes;
*inStream >> windowInfo.minimumWidth;
*inStream >> windowInfo.minimumHeight;
*inStream >> windowInfo.maximumWidth;
*inStream >> windowInfo.maximumHeight;
*inStream >> windowInfo.standardSize.width;
*inStream >> windowInfo.standardSize.height;
*inStream >> windowInfo.userCon;
```

LPopupGroupBox.cp

In the `LStream` constructor, replace this line:

```
inStream->ReadData(&cInfo, sizeof(SControlInfo));
```

with the following lines:

```
*inStream >> cInfo.valueMessage;
*inStream >> cInfo.value;
*inStream >> cInfo.minValue;
*inStream >> cInfo.maxValue;
```

LControlView.cp

In the `LStream` constructor, replace this line:

```
inStream->ReadData(&cInfo, sizeof(SControlInfo));
```

with the following lines:

```
*inStream >> cInfo.valueMessage;
*inStream >> cInfo.value;
*inStream >> cInfo.minValue;
*inStream >> cInfo.maxValue;
```

LScrollerView.cp

In the `LStream` constructor, replace this line:

```
inStream->ReadData(&scrollerInfo, sizeof(SScrollerViewInfo));
```

with the following lines:

```
*inStream >> scrollerInfo.horizBarLeftIndent;
*inStream >> scrollerInfo.horizBarRightIndent;
*inStream >> scrollerInfo.vertBarTopIndent;
*inStream >> scrollerInfo.vertBarBottomIndent;
*inStream >> scrollerInfo.scrollingViewID;
```

LPageController.cp

In the `LStream` constructor, replace these lines:

```
inStream->ReadData( &mBackColor, sizeof(RGBColor));
inStream->ReadData( &mFaceColor, sizeof(RGBColor));
inStream->ReadData( &mPushedTextColor, sizeof(RGBColor));
```

with the following lines:

```
*inStream >> mBackColor.red;
*inStream >> mBackColor.green;
*inStream >> mBackColor.blue;
*inStream >> mFaceColor.red;
*inStream >> mFaceColor.green;
*inStream >> mFaceColor.blue;
*inStream >> mPushedTextColor.red;
*inStream >> mPushedTextColor.green;
*inStream >> mPushedTextColor.blue;
```

Flipping the 'DBC#' Resource Type

If you use the `LDataBrowser` PowerPlant class, you might be using a `'DBC#'` resource. If so, you will need to write a resource flipper similar to that shown in Listing 5-2.

Listing 5-2 Code that flips the 'DBC#' resource type

```
struct DataBrowserColumnSetup {
    DataBrowserTableViewColumnID    propertyID;         // UInt32
    DataBrowserPropertyType         propertyType;       // unsigned long
    SInt32                          nameStrIndex;
    DataBrowserPropertyFlags        propertyFlags;      // UInt32
    UInt16                          minimumWidth;
    UInt16                          maximumWidth;
    UInt16                          initialWidth;
    ControlContentType              btnContentType;     // SInt16
    ControlButtonGraphicAlignment   btnContentAlign;    // SInt16
    SInt16                          btnContentDataID;
    ControlButtonGraphicAlignment   titleAlignment;     // SInt16
    ControlButtonTextPlacement      titlePlacement;     // SInt16
    SInt16                          titleFontTypeID;
    SInt16                          titleFontStyle;
    SInt16                          titleFontSize;
    UInt16                          titleOffset;
};
typedef struct DataBrowserColumnSetup   DataBrowserColumnSetup;

OSStatus FlipDBC_(OSType dataDomain, OSType dataType,
                  short id, void* dataPtr, UInt32 dataSize,
                  Boolean currentlyNative, void* refcon)
{
    DataBrowserColumnSetup *dbc;
    UInt16 count;
    int i;

    if (currentlyNative) {
        count = *(UInt16 *) dataPtr;
        *(UInt16 *) dataPtr = EndianU16_NtoB(count);
    } else {
        *(UInt16 *) dataPtr = EndianU16_BtoN(count);
        count = *(UInt16 *) dataPtr;
    }
    dataPtr += sizeof(UInt16);
    dbc = (DataBrowserColumnSetup *) dataPtr;
    for(i = 0; i < count; i++, dbc++) {
        dbc->propertyID = Endian32_Swap(dbc->propertyID);
        dbc->propertyType = Endian32_Swap(dbc->propertyType);
        dbc->nameStrIndex = Endian32_Swap(dbc->nameStrIndex);
        dbc->propertyFlags = Endian32_Swap(dbc->propertyFlags);
        dbc->minimumWidth = Endian16_Swap(dbc->minimumWidth);
        dbc->maximumWidth = Endian16_Swap(dbc->maximumWidth);
        dbc->initialWidth = Endian16_Swap(dbc->initialWidth);
        dbc->btnContentType = Endian16_Swap(dbc->btnContentType);
        dbc->titleAlignment = Endian16_Swap(dbc->titleAlignment);
        dbc->titlePlacement = Endian16_Swap(dbc->titlePlacement);
        dbc->titleFontTypeID = Endian16_Swap(dbc->titleFontTypeID);
        dbc->titleFontStyle = Endian16_Swap(dbc->titleFontStyle);
        dbc->titleFontSize = Endian16_Swap(dbc->titleFontSize);
        dbc->titleOffset = Endian16_Swap(dbc->titleOffset);
    }
    return noErr;
}
```

Where to Go From Here

This chapter points to some tools and performance documents you'll want to consider as you work on your software in Xcode.

After Moving Your Project to Xcode

Now that you've imported your project into Xcode and hopefully gotten it building, what's next? Well, this would be the right time to learn more about how to make your application a first-class citizen on Mac OS X. And there are several good places to get started:

- Xcode is a rich programming environment, and this document has only touched on its features. For a more detailed list, with specific instructions for taking advantage of new and enhanced features, see *Xcode User Guide*.
- Great applications take performance seriously, and Xcode comes with a lot of documentation to help you make sure your application is finely-tuned. You can start with *Performance Overview*, and follow up with the other documents available in the Performance Documentation area.
- If you're not familiar with the tools available with Xcode, take a look at Mac OS X Developer Tools in *Mac OS X Technology Overview*, as well as the documents available in the Tools Documentation area.

Most of the documents mentioned here are available in Xcode—choose Help > Documentation. Between releases of the developer tools, you can obtain the latest documentation through Xcode's documentation update mechanism or from the ADC website. And please use the information in "Feedback and Mail List" (page 11) to supply your feedback.

Document Revision History

This table describes the changes to *Porting CodeWarrior Projects to Xcode*.

Date	Notes
2006-10-26	Corrected errors. Changed title from "Moving CodeWarrior Projects to Xcode."
	Updated "For Best Results When Importing" (page 49) and "Known Issues With the Importer" (page 50) to note that CodeWarrior must be running when importing.
2006-11-07	Corrected errors.
	Revised statement of pragma support in GCC.
2006-09-05	Corrected errors.
	Fixed missing semicolon in Listing 4-3 (page 57).
	Updated "Known Issues With the Importer" (page 50) to reflect importer support for CW object targets.
2006-05-23	Made document-flow change.
	Changed the order of the steps to update the PowerPlant source files in "Make Changes to PowerPlant" (page 62) to make them easier to follow.
2005-11-09	Updated for Xcode 2.2 and corrected errors.
	Added "Using PowerPlant in Universal Binaries" (page 69).
	Updated "Exporting Symbols" (page 37) to reflect support for importing `.exp` files in Xcode 2.2.
	Corrected description in "C and C++ Libraries" (page 25) of options passed by Xcode to the compiler when changing filetype.
	In "Working With Resources" (page 32), added links to additional information on resource files and build phases.

Date	Notes
	Added note, in "Known Issues With the Importer" (page 50), about the importer not supporting targets whose Project Type is Object.
	Added note about including `ppcintrinsics.h` to use intrinsics in Xcode.
	Fixed typos.
2005-08-11	Fixed typos and made minor editorial corrections.
	Updated "Building Code" (page 33) to reflect default Imported CodeWarrior Settings build configuration for imported projects.
	Fixed typos.
2005-06-06	Updated for Xcode 2.1 and GCC 4.0.
	Added section on "Build Configurations" (page 34); removed mention of build styles.
	Updated description of Search field in "Some Special Features of Xcode" (page 14) to include wildcard and regular expression searching.
	Added descriptions of project window layouts and showing / hiding smart groups to "Customizing the Environment" (page 17).
	Revised discussion of header files and search paths in "Header Files" (page 20) to reflect Xcode 2.0 support for recursive search paths.
	Updated build phase names throughout.
	Added mention of breakpoint actions and watchpoints to "Debugging" (page 38).
	Updated "The GCC Compiler" (page 26) for GCC 4.0; also added note about removal of GCC 3.1 and 2.95.
	Noted that prebinding is no longer necessary if you are running on 10.3.4 or later in "Prebinding" (page 39).
	Updated "C and C++ Libraries" (page 25) to reflect the switch to packaging the standard C++ library as a dynamic shared library.
2005-04-29	Fixed broken links.
2005-01-11	Corrected typos.
	Fixed typos in Listing 4-6 (page 63) and in "Migrate from MSL to System C and C++ Libraries" (page 48).
	Updated for July 2004 Xcode Tools.
	Updated screenshots.

Date	Notes
	Added information on using dead code stripping in Xcode to section "Dead Code Stripping" (page 31).
	Added a note about a Rez bug to section "Working With Resources" (page 32).
	First public release.

Document Revision History