Cross-Development Programming Guide

Tools > Xcode

2006-11-07

Apple Inc.
© 2006 Apple Computer, Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled computers.

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

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

.Mac is a registered service mark of Apple Inc.

Apple, the Apple logo, Carbon, Cocoa, Mac, Mac OS, Macintosh, Objective-C, Panther, QuickTime, Tiger, and Xcode are trademarks of Apple Inc., registered in the United States and other countries.

Finder is a trademark of Apple Inc.

Intel and Intel Core are registered trademarks of Intel Corportation or its subsidiaries in the United States and other countries.

OpenGL is a registered trademark of Silicon Graphics, Inc.

PowerPC and and the PowerPC logo are trademarks of International Business Machines Corporation, used under license therefrom.

UNIX is a registered trademark of The Open Group

Simultaneously published in the United States and Canada.

Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Introduction **Introduction to Cross-Development Programming Guide 7**

Organization of This Document 7

Chapter 1 **How Cross-Development Works 9**

SDK Header Files and Stub Libraries 9
SDK Settings 10
Building 11
Linking 11
Loading 12
How SDK Settings Affect The Build 12
How SDK Settings Affect Other Features of Xcode 13
Limitations 14

Chapter 2 **Using Cross-Development 17**

Configuring Your Project for Cross Development 17
  Configuring an Xcode Project 17
  Configuring a Makefile-Based Project 20
Set the Prefix File 21
Check for Undefined Function Calls 21
Conditionally Compile for Different SDKs 22
Finding Deprecated API 23

Chapter 3 **Cross-Development and Universal Binaries 25**

Chapter 4 **Determining the Version of a Framework 29**

Document Revision History 31

2006-11-07 | © 2006 Apple Computer, Inc. All Rights Reserved.

2006-11-07 | © 2006 Apple Computer, Inc. All Rights Reserved.

Figures, Tables, and Listings

Chapter 1 **How Cross-Development Works 9**

Figure 1-1 Cross-development timeline 11

Chapter 2 **Using Cross-Development 17**

Figure 2-1 Selecting the SDK in Xcode 18
Figure 2-2 Selecting the deployment OS in Xcode 19
Listing 2-1 A function that checks for a null function pointer 21

Chapter 3 **Cross-Development and Universal Binaries 25**

Table 3-1 Xcode cross-development settings with per-architecture variants 25

2006-11-07 | © 2006 Apple Computer, Inc. All Rights Reserved.

2006-11-07 | © 2006 Apple Computer, Inc. All Rights Reserved.

Introduction to Cross-Development Programming Guide

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.

Cross-development support is available starting with the Xcode Tools distributed with Mac OS X version 10.3 and later. It includes **Software Development Kits (SDKs)**, which are complete sets of header files and stub libraries as shipped in previous versions of Mac OS X. To use cross-development, you specify which version (or SDK) of Mac OS X to build with. When you choose an SDK, your software is built just as though you were building in that version of the operating system. You can also specify the earliest Mac OS X system version on which your software will run.

Important: You cannot use cross-development in Xcode unless, when you install the Xcode development tools, you install all the cross-development SDKs you plan to work with. During installation, after selecting an installation disk, click the Customize button and select the Cross-Development checkbox.

The Mac OS X 10.1 SDK is no longer shipped with the Xcode Tools. If you want to develop for Mac OS X version 10.1.5, you can obtain this SDK package as part of the Xcode Legacy Tools package, available in the Downloads section of the ADC Member Site. Note that you must install both GCC 2.95.2 and the Mac OS X 10.1.5 SDK.

You can take advantage of cross-development in a number of ways:

- You can build a target that is optimized for one version of the operating system and is forward-compatible with later versions but doesn't take specific advantage of their features.
- You can build a target for a range of operating system versions, so that it can still launch in older versions but can take advantage of features in newer ones. This allows you to deliver software that provides new value to customers who have upgraded to a new system version, but still runs for those who haven't.
- You can have one project with separate subprojects (for example, for bundles or frameworks) for different system versions, so your software can load and execute platform-specific code without having to test for operating system version or function availability throughout the code.
- In Mac OS X v10.4 and later, you can use GCC 4.0 to add cross-development support to makefile-based projects.

For possible development issues, see "Limitations" (page 14).

Organization of This Document

This document contains the following:

- "How Cross-Development Works" (page 9) describes how this feature is supported in Xcode and lists some limitations you may encounter.

2006-11-07 | © 2006 Apple Computer, Inc. All Rights Reserved.

- "Using Cross-Development" (page 17) shows how to set up cross-development in your project and how to deal with undefined functions.
- "Cross-Development and Universal Binaries" (page 25) describes how to use cross-development to build for Mac OS X v10.4 for Intel-based Macintosh computers while supporting earlier versions of Mac OS X for PowerPC-based Macintosh computers.
- "Determining the Version of a Framework" (page 29) describes how to check at runtime for new features in the Application Kit or Foundation frameworks using the framework version.

2006-11-07 | © 2006 Apple Computer, Inc. All Rights Reserved.

How Cross-Development Works

This chapter describes how cross-development works. For information on how to configure your Xcode projects or makefiles to use cross-development, see "Configuring Your Project for Cross Development" (page 17).

To develop software that can be deployed on, and take advantage of features from, different versions of Mac OS X, you specify which version (or SDK) of Mac OS X headers and libraries to build with. You can also specify the earliest Mac OS X system version on which the software will run.

Cross-development depends on the following features of the development environment:

- Weak linking, which was introduced in Mac OS X version 10.2 (Jaguar) and supported by the December 2002 Development Tools. Weak linking allows your software to run on versions of Mac OS X whose libraries may not export all the entry points your software requires.

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

  Important: See "Limitations" (page 14) for possible issues with weak linking.

- Mac OS SDKs, introduced in Mac OS X version 10.3, which are complete sets of header files and stub libraries as shipped in previous versions of Mac OS X, so you can compile and link against specific headers and libraries.
- Cross-Development support, first available in the Xcode Tools distributed with Mac OS X version 10.3. This support allows you to specify the earliest version of Mac OS X your software is intended to run on and the latest version your software uses features from.

SDK Header Files and Stub Libraries

When you first install the Developer Tools, you can choose which cross-development SDKs are installed by using the Customize option in the installer. The cross-development SDKs are not installed by default; therefore you must do a custom installation to add them to your system. If you already installed the developer tools on your system, you can reinstall the SDKs from the Xcode Tools CD. Install the `CrossDevelopment.mpkg` package or run the general installer and install the Cross-Development components.

When you install the cross-development SDKs, the installer creates a `/Developer/SDKs` directory. This directory contains several subdirectories, each of which provides the complete set of header files and stub libraries that shipped for a particular version of Mac OS X. Typically, the latest minor revision of every major revision is represented—for example, 10.2.8 is the latest version of Mac OS X version 10.2. This gives you access to features introduced in System Updates without too much redundancy.

2006-11-07 | © 2006 Apple Computer, Inc. All Rights Reserved.

Note: Using SDKs allows you to use new APIs introduced in a System Update. When new functionality is added as part of a System Update, the System Update itself does not typically contain updated header files reflecting the change. The SDKs, however, do contain updated header files.

Each SDK resembles the directory hierarchy of the operating system release it represents: it has `usr`, `System`, `Library`, and `Developer` directories at its top level. Each of these directories is in turn populated with directories containing the headers and libraries that would be present in that version of the operating system with Apple Developer Tools installed.

The libraries in an SDK are **stub libraries**; that is, they do not contain executable code but just the exported symbols for the purpose of linking. This allows SDKs to take up far less disk space and reduces the possibility that an SDK library will be loaded at runtime.

Note: You can create stub libraries of your own with the `strip (1)` tool and its new `-c` option.

SDK Settings

To use cross-development for a target in an Xcode project, you make two selections (described in more detail in "Configuring Your Project for Cross Development" (page 17)):

- In the General pane of the project inspector, you select a target SDK. This is the OS version you want to develop for, such as Mac OS X 10.3.9. This is shown in Figure 2-1 (page 18).

  Your software is 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. This is shown in Figure 2-2 (page 19).

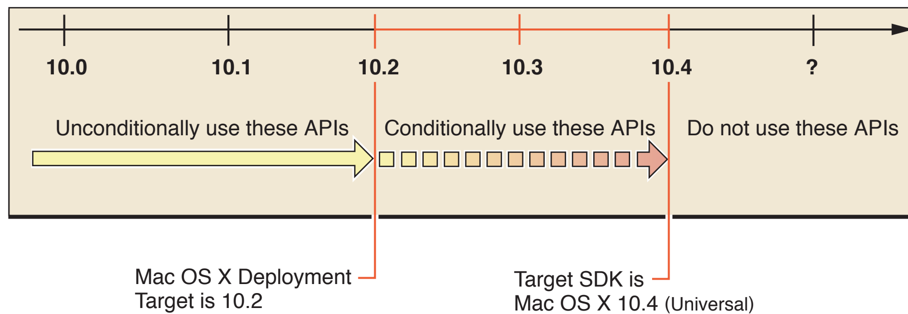
  This specifies the earliest Mac OS X system version on which your software will run.

Xcode uses information from these settings during building and linking of your software. Together, these settings define the range of OS versions from which you can use features. You can unconditionally use features from OS versions up to and including the system version that you have specified as your deployment OS. You can use features from system versions later than the deployment OS—up to and including the system version you've selected as your target SDK—but you must check for the availability of the feature, as described in "Check for Undefined Function Calls" (page 21).

Figure 1-1 (page 11) shows the "timeline" for a project targeting the Mac OS X 10.4 (Universal) SDK, with a deployment target set to Mac OS X 10.2.

2006-11-07 | © 2006 Apple Computer, Inc. All Rights Reserved.

Figure 1-1 Cross-development timeline



In this example, the software can freely use any features from Mac OS X versions 10.0 through 10.2. It can also take advantage of features from Mac OS X versions 10.3 and 10.4, but it must first check that the feature is available.

Of course, you should also check to see if you are using deprecated API; while still available, these APIs are not guaranteed to be supported in the future. Using cross-development, the compiler can warn you about the presence of deprecated API.

To use cross-development in a makefile-based project, you add special options to your compiler and linker options to build against the target SDK. For more information, see "Using Cross-Development" (page 17).

Building

In Xcode, the SDK settings in the project inspector control the value of a build setting called `SDKROOT`. This build setting points to the root directory in which to find headers and libraries to build against. If `SDKROOT` is not set, a project is compiled against the headers and linked against the runtime libraries in the current system (`/usr/include`, `/usr/lib`, and `/System/Library/Frameworks`).

In makefile-based projects, the SDK settings are controlled by the options you pass to the compiler and linker.

When you build a target, GCC compiles your source files using the headers and frameworks from the SDK you selected. This means that your source can refer to functions, classes, methods, and constants that are defined for use with that version of the operating system. Functions and methods that exist in that version but not earlier ones are defined as "weak" imports; that is, they will not cause an error at runtime if they are not found in the system libraries.

Linking

Your target is linked using "stub libraries" that represent the frameworks and other shared libraries in the system specified in the SDK you selected. Your software will be optimized for that version of Mac OS X (that is, it will be "prebound" to the standard locations of the frameworks in that version).

2006-11-07 | © 2006 Apple Computer, Inc. All Rights Reserved.

> **Note:** 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. In Mac OS X v10.3.3 and earlier, prebinding can improve the user experience by providing a faster launch. In later versions of Mac OS X, prebinding is not necessary.
>
> You can read more about prebinding in “Prebinding Your Application” in “Launch Time Performance” in the Performance Documentation area.

Only system dynamic libraries and frameworks are provided in stub-library form. Static libraries (such as those required by GCC) are provided in the SDK and included in your code as if they have been drawn from the `/usr/lib` directory of the operating system being targeted.

Loading

When the object code for the target is loaded on a Mac OS X system, the dynamic loader (`dyld`) attempts to resolve its imports with the libraries that exist on the running system. If it is loaded on a system earlier than the one defined by the deployment OS version, then it may fail to load (if, for example, it uses function calls not supported in that version of the operating system, or it is loaded in Mac OS 10.1, which does not support weak linking).

However, if the system supports weak linking and the deployment OS was set for that system version or later, then your object code loads, but certain function calls may be undefined. Your code should avoid calling those functions in system versions that do not support them (either by checking the system version globally, or checking each function pointer for a null value before calling it). For an example see "Check for Undefined Function Calls" (page 21).

How SDK Settings Affect The Build

In Xcode, when you select an SDK for a project Xcode sets `SDKROOT` to the path of the chosen SDK, and implicitly adds that path to the search paths for frameworks, headers, and libraries. In makefile-based projects, the options you pass to the compiler and linker do this for you.

One of the files in the new search path is `$(SDKROOT)/usr/include/AvailabilityMacros.h`, which is the principal header for driving weak linking support. This header file sets the preprocessor macro `MAC_OS_X_VERSION_MAX_ALLOWED` to a constant that is equivalent to the version represented by that SDK (for example, 1028 for Mac OS X v10.2.8).

When you choose a Mac OS X deployment OS, as described in "Configuring Your Project for Cross Development" (page 17), you assign a constant value to the `MACOSX_DEPLOYMENT_TARGET` build setting that corresponds to the chosen major system version (such as 10.1, 10.2, 10.3, or 10.4). The compiler uses this build setting to enable weak linking. The compiler also uses this value to set the preprocessor symbol `MAC_OS_X_VERSION_MIN_REQUIRED`, which is used by the header `AvailabilityMacros.h` to determine which functions are weak linked.

2006-11-07 | © 2006 Apple Computer, Inc. All Rights Reserved.

Important: When you select an SDK for a project in Xcode (see "Configuring Your Project for Cross Development" (page 17)), the SDK you choose applies to all targets in the project (all are built with headers and libraries from the specified SDK). Although it is possible to specify an SDK on a per-target basis, doing so may interfere with the operation of other Xcode features that rely on a projectwide SDK setting, as described in "How SDK Settings Affect Other Features of Xcode" (page 13). You can select a deployment OS (described in "Configuring Your Project for Cross Development" (page 17)), for an individual target or for all targets in the project. The target or targets can be deployed on systems back to the specified OS version. To learn more about build settings and their precedence, see *Xcode 2.2 User Guide*.

The `MAC_OS_X_VERSION_MIN_REQUIRED` and `MAC_OS_X_VERSION_MAX_ALLOWED` preprocessor symbols are used by the `AvailabilityMacros.h` header file to define a number of other preprocessor symbols using the GCC compiler directives `__attribute__((weak_import))` and `__attribute__((deprecated))`. Those symbols are used extensively in Carbon header files (and in many, but not all, Cocoa headers) to define which function or method calls are available in different versions of the operating system.

The combination of these actions makes the following things happen:

- If your code uses a symbol that is not defined in the selected SDK, you get a compile-time error.
- If your code uses a symbol that is defined but is marked as deprecated, you get a compile-time warning.
- If your code uses a symbol that is defined in the selected SDK and in the selected deployment OS, your code builds and links normally. At run time:
  - on systems earlier than the deployment version, your code may fail to load if you use symbols unavailable in that version
  - on systems equal to or later than the deployment version, you may get weak references (that is, null function pointers) for symbols not supported in that version; it is up to your code to be prepared for this—for an example see "Check for Undefined Function Calls" (page 21).

Note: The Carbon and Cocoa framework headers in Mac OS X version 10.2 and later automatically include `AvailabilityMacros.h`, but those in version 10.1 (which pre-dates the introduction of `AvailabilityMacros.h`) do not.

To make it easier to use one set of source code across different SDKs, the SDK for Mac OS X v10.1.5 includes a copy of `AvailabilityMacros.h`. Code that uses the 10.1.5 SDK and the availability macros can compile successfully by specifically including that header, with a statement like the following (which you can place in your prefix header file):

```
#include <AvailabilityMacros.h>
```

For an example, see "Conditionally Compile for Different SDKs" (page 22).

How SDK Settings Affect Other Features of Xcode

As you would expect, changing SDK settings in Xcode can have an effect on other Xcode features. Here are some of the changes you may observe:

- When you change the selected SDK:
  - Xcode rebuilds the index for the project.

2006-11-07 | © 2006 Apple Computer, Inc. All Rights Reserved.

- Rebuilding results in the equivalent of a clean rebuild.

- Double-clicking a symbol in the Project Symbols smart group should take you to its definition in the chosen SDK. You can Command-click in the title bar of the header file to see the path, which starts with `/Developer/SDKs/MacOSX`... rather than `/System/Library/Frameworks`....
- Code completion is based on the SDK in use. For example, if you are targeting Mac OS X version 10.3.9, typing "HIAboutB…" should complete to `HIAboutBox`. However, if you are targeting version 10.2.8, `HIAboutBox` is not defined, and does not appear in the code completion interface.
- When you select a `#include<header>` directive in a source file, typing Command-D opens the header from the currently selected SDK, rather than from `/System/Library/Frameworks`.
- When debugging your code, symbolic information from headers (such as inline function body definitions) will come from the currently selected SDK, not from the root system on which you are running.

Limitations

Cross-development is most strongly supported by the Carbon APIs and to a lesser degree by Cocoa. However, given the dynamic nature of Objective-C (for example, it doesn't produce link errors, it just raises a runtime exception if you call a method that doesn't exist), cross-development support is less important for Cocoa applications.

Some OS X frameworks, most notably those derived from Open Source projects, such as OpenGL and the BSD and Kernel frameworks, do not use Apple's availability macros to support cross-development. However, code that makes kernel-specific calls is likely to be closely bound to the OS that supports those calls, and is less likely to benefit from cross-development. Code that works with OpenGL can use the mechanisms described in Technical Q&A QA1188: OpenGL Entry Points and GetProcAddress.

QuickTime has a separately distributed SDK that can be installed on any variant of Mac OS X v10.2 or later. As a result, certain QuickTime APIs are not bound to a particular version of the OS. In addition, when you install a QuickTime SDK, it's installed into your current operating system, not into any of your developer SDKs. Because a developer SDK does not necessarily include the headers from the latest software update of QuickTime, you can't currently use SDK support and the QuickTime SDK simultaneously.

Note: QuickTime and some related components, such as Core Audio, provide extensions to the availability macros that tell you which features are available in particular QuickTime software updates.

The following are some additional limitations of cross-development support:

- Weak linking is not available prior to Mac OS X v10.2. As a result, software developed with a deployment OS set for Mac OS X v10.1 will not load in v10.1 if it uses features from later system versions.

  If you want to deploy the same code in Mac OS X version 10.1.5 as in version 10.2 and later, you have to build the 10.1.5 code as a separate project with a separate executable. Set the selected SDK to 10.1.5 and the Mac OS X deployment OS to 10.1. This code will not be able to use features from the more recent OS versions.

  The compiler calling conventions for C++ code changed in Mac OS X version 10.2 with GCC 3.1, so to compile C++ code targeting Mac OS X v10.1, you must use the GCC 2.95.2 compiler.

2006-11-07 | © 2006 Apple Computer, Inc. All Rights Reserved.

> **Note:** The Mac OS X v10.1.5 SDK and GCC 2.95.2 are no longer included with the Xcode Tools. If you want to build for Mac OS X v10.1 you must install the SDK and the compiler from Xcode 1.5 or earlier.

For additional information on weak linking in this document, see "Building" (page 11), "Loading" (page 12), "How SDK Settings Affect The Build" (page 12), and "Check for Undefined Function Calls" (page 21).

- Xcode uses a native build system that performs its own dependency analysis and directly invokes the necessary build commands. Cross-development SDK support works with targets that use the native build system, but it does not work with Project Builder targets that have not been upgraded to native targets.
- In Xcode, you can't use the SDK setting in the General pane of the project inspector to alter the SDK selection based on the current build configuration.
- Cross-development support works primarily with release, not debug, builds of your software. You cannot use the ZeroLink or Fix and Continue features in Xcode if you have selected the 10.2.8 or earlier SDK, because those features require development-time frameworks that are only available in Mac OS X v10.3. However, to use ZeroLink but deploy on Jaguar, you can set the deployment OS to 10.2 in your release build configuration but not in your debug build configuration.
- Code built with GCC 4.0 does not run on versions of Mac OS X prior to Mac OS X v10.3 (10.3.9 for C++ or Objective-C++). To target these versions of Mac OS X, you must build using GCC 3.3.

2006-11-07 | © 2006 Apple Computer, Inc. All Rights Reserved.

2006-11-07 | © 2006 Apple Computer, Inc. All Rights Reserved.

Using Cross-Development

This chapter describes how to use cross-development in your projects. The provided steps cover the configuration process for both Xcode and makefile-based projects.

Note: Using SDKs in makefile-based projects requires GCC 4.0 or later.

To use cross-development, you take these steps (in addition to installing the SDKs when you install the Xcode development tools):

1. Configure your project. You choose the SDK for the version of Mac OS X you wish to target, as well as a deployment target, which specifies the earliest version of Mac OS X that your software will run on.
2. Supply a prefix file that takes into account the selected SDK.
3. If your software uses features from a newer OS version that aren't available in an earlier version it is targeted to run on, prepare your code to handle undefined function calls.
4. If you will be building the same source code using different SDKs, you may wish to compile code conditionally based on the SDK in use.

These steps are described in the sections that follow.

Configuring Your Project for Cross Development

You can configure both Xcode projects and makefile-based projects to use cross-development. For either type of project, you must make two choices:

1. Select an SDK to develop for, such as Mac OS X version 10.2.8. The software can use features available in system versions up to and including the one you select.

   If you do not select an SDK, the default is to build for the current operating system.

2. If your software must run on a range of operating system versions, choose a Mac OS X deployment OS, such as v10.2. The deployment OS identifies the earliest system version on which the software can run.

   If you do not select a deployment OS, the default is to target the current operating system.

Configuring an Xcode Project

To use cross-development, you must choose an SDK and a deployment target for your project and its targets. In Xcode, you make these choices in the project and target inspectors.

2006-11-07 | © 2006 Apple Computer, Inc. All Rights Reserved.

Select an OS SDK to Develop With

To select an SDK in Xcode, you take the following steps:

1. Select the project group in the Groups & Files list.

2. Open an Info window for the project by choosing Project > Get Info or by typing Command-I.

> **Note:** You can also use an inspector window to select an SDK. You open an inspector window by choosing Project > Show Inspector or by typing Command—Option-I. To learn more about inspector and Info windows in Xcode, see *Xcode 2.2 User Guide*.

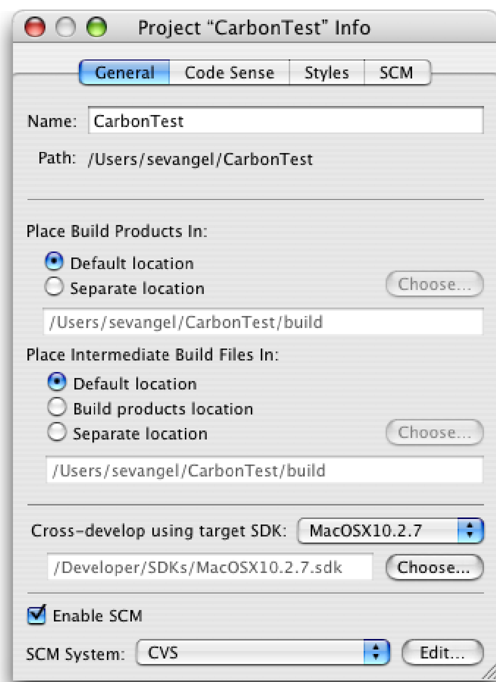
3. Choose an SDK in the Cross-Develop Using Target SDK pop-up menu.

   The result is shown in Figure 2-1 (page 18). All targets in the project are built with the selected SDK, as described in "How SDK Settings Affect The Build" (page 12). The value of `MAC_OS_X_VERSION_MAX_ALLOWED` is automatically set to correspond to the selected SDK.

   Alternatively you can use the text field or the Choose button to select an SDK other than a Mac OS X SDK, or to choose a Mac OS X SDK that is stored in a nonstandard location (for example, relative to your project).

   For each target in the project, you'll also have to perform the step described in "Set the Prefix File" (page 21).

Figure 2-1 Selecting the SDK in Xcode



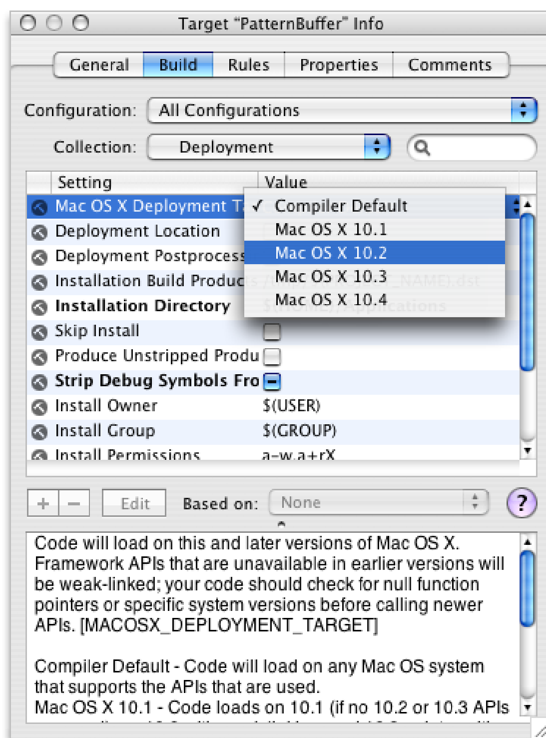
2006-11-07 | © 2006 Apple Computer, Inc. All Rights Reserved.

Select a Deployment OS

In Xcode, you can set the deployment OS for individual targets or for all targets in the project, using the project-level build setting. To select a deployment OS for a target in Xcode, you use the following steps:

1. Select a target in the Targets group in the Groups & Files list.
2. Open an Inspector window for the target.
3. Select Build to open the Build pane.
4. Find the Mac OS X Deployment Target build setting, located in the Deployment collection.

Figure 2-2 Selecting the deployment OS in Xcode



5. Use the pop-up menu in the Value column next to Mac OS X Deployment Target to choose a deployment OS value, such as 10.2 or 10.3, as shown in "Selecting the deployment OS in Xcode."

   When you build the target for which you just set the deployment OS value, the build is based on the selected version, as described in "How SDK Settings Affect The Build" (page 12). The software can run on system versions as early as the one you specify. You can use features from later system versions, but you must be prepared to check for the availability of those features, as described in "Check for Undefined Function Calls" (page 21) and "Limitations" (page 14).

   Setting the Mac OS X Deployment Target build setting automatically sets `MAC_OS_X_VERSION_MIN_REQUIRED` to the same value.

2006-11-07 | © 2006 Apple Computer, Inc. All Rights Reserved.

You can use the same steps in the Build pane of the project inspector to set the deployment OS for all targets in the project. Note that any value explicitly assigned to the Mac OS X Deployment Target build setting in the target inspector overrides, for that target, the value assigned to the setting in the project inspector.

Important: In Xcode, you should not manually override the build setting `MACOSX_DEPLOYMENT_TARGET` and set it to any value other than 10.1, 10.2, 10.3, or 10.4. If you do, it will revert to 10.0.

Configuring a Makefile-Based Project

If you have a makefile-based project, you can also take advantage of cross-development, by adding the appropriate options to your compile and link commands. These are described in the following sections.

Select an OS SDK to Develop With

To select an SDK for a makefile, you add the appropriate options to your project's compile and link commands. For the compiler, you add the `-isysroot` option. For the linker, you add the `-syslibroot` option. If you are using GCC 4.0 to compile and link, you should add both commands to the command line.

Both `-isysroot` and `-syslibroot` require that you specify the full path to the desired SDK directory. It is usually best to create a makefile directory to specify this path. The following example shows these flags assigned to makefile variables for a simple C program:

```
SDK=/Developer/SDKs/MacOSX10.4.0.sdk
CFLAGS= -isysroot ${SDK}
LDFLAGS= -isysroot ${SDK} -Wl,-syslibroot,${SDK}
```

Note: If your makefile passes `LDFLAGS` directly to `ld` instead of `gcc`, you should specify the linker flags as `LDFLAGS= -syslibroot ${SDK}`, as `ld` does not support the `-Wl` syntax.

The SDK directories are located in `/Developer/SDKs`. Your own linker flags would naturally contain any additional compiler or linker options required for your program.

Important: Support for the `-isysroot` flag is available in the GCC 4.0 compiler but is not documented in the compiler man page because support for this feature is likely to change in the future. When support for the feature is officially added to the compiler, the man page will be updated with the correct information. This flag is not supported in older versions of the compiler.

Select a Deployment OS

To set the deployment target in a makefile, use another makefile variable of the form:

```
ENVP= MACOSX_DEPLOYMENT_TARGET=10.3
```

The value you specify for `MACOSX_DEPLOYMENT_TARGET` determines the minimum target system for your software. In the preceding example, the target system is `10.3` . To use this variable in your makefile, include it in front of your compile and link commands. For example, a simple C program might use the following build commands:

```
testapp: main.o
```


2006-11-07 | © 2006 Apple Computer, Inc. All Rights Reserved.

```
    ${ENVP} ${CC} ${LDFLAGS} -o testapp main.o
main.o:
    ${ENVP} ${CC} ${CFLAGS} -c main.c -o main.o
```

Set the Prefix File

In both Xcode and makefile projects, the prefix file is typically a header file belonging to the project. However, some developers have set the prefix file to an umbrella header file, such as `/System/Library/Frameworks/Carbon.framework/Headers/Carbon.h`). This does not work with SDK Support as it defines an absolute path.

To include umbrella framework headers, it is best to define your own prefix file and add the appropriate `#include <Framework/Framework.h>` directives to your file. With this technique, the compiler always chooses the headers from the appropriate SDK directory.

For example, if your project is named CarbonTest and it has a prefix file `CarbonTest_Prefix.pch`, you could add the following line to that file:

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

Check for Undefined Function Calls

As described in "Loading" (page 12), in some situations your object code may successfully load, using weak linking, but certain function calls may be undefined because they are not available in the current system. To run successfully, your code must avoid calling those functions in system versions that do not support them. It can do this either by checking the system version at run time and globally taking a different code path based on the version, or by checking each function pointer for a null value before calling it.

For example, suppose you build your application to use features in Mac OS X version 10.3 (Panther) and to deploy back to Mac OS X version 10.2. (Jaguar). To use the `HIAboutBox` function, first available in Panther, you could use code like the following:

Listing 2-1 A function that checks for a null function pointer

```
void MyAboutBox(void)
{
    if(HIAboutBox != NULL)
    {
        HIAboutBox(NULL);
    }
    else
    {
        // Lots of code to display an about box with earlier technology.
    }
}
```


2006-11-07 | © 2006 Apple Computer, Inc. All Rights Reserved.

Important: When checking for the existence of a symbol, you must explicitly compare it to `NULL` or `nil` in your code. You cannot use the negation operator (`!`) to negate the address of the symbol.

When your code runs in Panther, it calls `HIAboutBox` to display the minimal default standard About box. When it runs in Jaguar, it displays an About box based on the code you wrote for Jaguar.

If you build this code with different settings, you should see the following results:

- If you select an SDK setting of Mac OS X 10.2.8:

  The build fails because `HIAboutBox` is not defined in those system versions.

- With an SDK setting of Mac OS X 10.3.9, if you set the deployment OS to:
  - 10.3: The software should run only in Panther and fail to launch in Jaguar or Puma (Mac OS X version 10.1).
  - 10.2: The software should run in Panther and Jaguar but fail to launch in Puma.
  - 10.1: The software should run in Panther but fail to launch in Puma because weak linking is not supported in Puma.

  Note: As described in "Limitations" (page 14), software built for Panther with a deployment OS of 10.1 will not launch in Jaguar.

Conditionally Compile for Different SDKs

If you will be building the same source code using different SDKs, you may want to compile code conditionally based on the SDK in use. You can do this by using preprocessor directives with the macros defined in `AvailabilityMacros.h`.

Important: `AvailabilityMacros.h` is included automatically by the Carbon and Cocoa frameworks in the SDKs for 10.2 and later, but not by the 10.1 SDK. If you wish to use these macros with the 10.1 SDK, you should manually include the file, typically by adding the following line to your project's prefix header:

```
#include <AvailabilityMacros.h>
```

If you fail to add this line, code that builds successfully with the other SDKs will fail silently when built with the 10.1 SDK.

For example suppose the function shown in Listing 2-1 must be compiled against the 10.2.8 SDK (or by other developers using 10.2 and no SDK). Even referring to the `HIAboutBox` function will cause a compiler failure, so you must exclude that code altogether unless the 10.3 header files are in use. You could do so with code like the following:

```
void MyAboutBox(void)
{
#if (MAC_OS_X_VERSION_MAX_ALLOWED >= MAC_OS_X_VERSION_10_3)
    if(HIAboutBox != NULL)
    {
```


2006-11-07 | © 2006 Apple Computer, Inc. All Rights Reserved.

```
        HIAboutBox(NULL);
    }
    else
    {
#endif
        // Lots of code to display an about box with earlier technology.
#if (MAC_OS_X_VERSION_MAX_ALLOWED >= MAC_OS_X_VERSION_10_3)
    }
#endif
}
```

With this technique, the source file can be built on native 10.1 or 10.2 systems (or using the 10.1 or 10.2 SDKs) without compilation errors.

Finding Deprecated API

As Mac OS X evolves, the list of APIs and technologies it encompasses are sometimes changed to meet the needs of developers. As part of this evolution, less efficient interfaces may be deprecated in favor of newer, more efficient ones. Apple makes these changes only when it is deemed absolutely necessary and uses the availability macros (defined in `/usr/include/AvailabilityMacros.h`) to help you find deprecated interfaces.

> **Note:** Deprecation does not mean the deletion of an interface from a framework or library. It is simply a way to flag interfaces for which better alternatives exists. For example, an older interface may be discouraged in favor of a newer, more efficient interface. You may still use deprecated interfaces in your code; however, Apple recommends you migrate to newer interfaces as soon as possible. Check the header files or documentation of the deprecated interface for information about any recommended replacement interfaces.

Each deprecated interface is tagged with a macro that identifies the version of Mac OS X in which it was marked as deprecated. For example, an interface that was introduced in Mac OS X version 10.0 but deprecated in version 10.3 might have the following macro after its header-file declaration:

```
AVAILABLE_MAC_OS_X_VERSION_10_0_AND_LATER_BUT_DEPRECATED_IN_MAC_OS_X_VERSION_10_3
```

If you compiled your project to run on Mac OS X version 10.3 and used an interface tagged with this macro, you would get a warning that the interface is now deprecated. The macro accomplishes this by adding the following attribute to the interface declaration:

```
__attribute__((deprecated))
```

When your code references a function that is tagged with this attribute, the compiler generates a warning. The warning includes the name of the deprecated interface and where in your code it was referenced. For example, if the `HPurge` function were deprecated, you might get an error similar to the following:

```
'HPurge' is deprecated (declared at /Users/steve/MyProject/main.c:51)
```

To locate references to deprecated interfaces, you can look for warnings of this type. If your project has a lot of warnings, you can use the search field in Xcode to filter the list based on the “deprecated” keyword.

2006-11-07 | © 2006 Apple Computer, Inc. All Rights Reserved.

2006-11-07 | © 2006 Apple Computer, Inc. All Rights Reserved.

Cross-Development and Universal Binaries

Beginning with Xcode 2.1, you can take advantage of cross-development to create universal binaries—executable files containing object code for both Intel-based and PowerPC-based Macintosh computers. For more information on creating universal binaries, see *Universal Binary Programming Guidelines, Second Edition*.

To compile code for an Intel-based Macintosh, you must compile using GCC 4.0 and target the Mac OS X 10.4 (Universal) SDK. If you can require Mac OS X v10.4 or later, you can use the same cross-development settings to build for either architecture. However, you may want to support Mac OS X v10.3 and later for PowerPC-based Macintosh computers.

If you're creating a universal binary version of a kernel extension (a generic KEXT or an I/O Kit driver), be aware that some I/O Kit classes in an Intel-based Macintosh running Mac OS X v10.4 are intentionally not binary compatible with the same classes in a PowerPC-based Macintosh running Mac OS X v10.4. If your KEXT needs to run in both versions, be sure to target the Mac OS X 10.4.0 (or earlier) SDK for the PowerPC build and the Mac OS X 10.4 (Universal) SDK for the Intel-based build. For more details on building universal I/O Kit device drivers, see Technical Note TN2163.

Note: The Mac OS X 10.4.0 SDK—which is distinct from the Mac OS X 10.4 (Universal) SDK—no longer ships with Xcode Tools. You can download it from the Downloads section of the ADC Member Site.

Xcode provides per-architecture variants for a number of build settings to allow you to specify different cross-development settings for Intel and PowerPC-based builds. These variants take the form `BUILD_SETTING_($arch)` where `BUILD_SETTING` is the usual name of the build setting and `($arch)` is a suffix denoting the architecture for which the build setting applies. The `i386` suffix indicates the architecture for Intel-based Macintosh computers, and the `ppc` suffix indicates the PowerPC architecture. For example, to specify that Xcode use GCC 4.0 when building for Intel, set the `GCC_VERSION_i386` build setting to 4.0. The build settings that have per-architecture variants related to cross-development are listed in Table 3-1 (page 25). For more on build settings in Xcode, see *Xcode 2.2 User Guide*.

Table 3-1 Xcode cross-development settings with per-architecture variants

Build setting	Description
`SDKROOT`	The SDK to develop for.
`MACOSX_DEPLOYMENT_TARGET`	The deployment target, which specifies the earliest version of Mac OS X on which your software can run.
`GCC_VERSION`	The version of GCC to use when building.

2006-11-07 | © 2006 Apple Computer, Inc. All Rights Reserved.

Important: Per-architecture build settings are supported only in Xcode 2.2 and later.

Note: By default, the Build pane of the target and project inspectors contains only the general version of each build setting, without any architecture-specific suffix. You must manually create entries in the build settings table for any per-architecture variants you wish to add. To learn more about adding build settings in Xcode, see *Xcode 2.2 User Guide*.

Using these per-architecture settings, you can create a single target that creates a universal binary and builds for Mac OS X v10.4 on Intel-based Macintosh computers while supporting Mac OS X v10.3 and later for PowerPC. The basic steps for configuring your project or target are the same as those described in "Configuring Your Project for Cross Development" (page 17):

1. Choose a target SDK.

   To target Mac OS X v10.4 for both Intel- and PowerPC-based builds, choose the Mac OS X 10.4 (Universal) SDK from the Cross-Develop Using Target SDK menu.

   Note: You can specify different target SDKs for Intel and PowerPC builds. For example, when building for PowerPC, you may want to ensure that your code doesn't use features from versions of Mac OS X later than Mac OS X v10.3.9.

   To do so, add new entries for the `SDKROOT_ppc` and `SDKROOT_i386` build settings to the table in the Build pane of the target or project inspector. Set `SDKROOT_ppc` to point to the Mac OS X 10.3.9 SDK and set `SDKROOT_i386` to point to the Mac OS X 10.4 Universal SDK. You can type the full path to each SDK or simply drag the SDK folder from the Finder to the Value column.

2. Set the deployment target.

   To support versions of Mac OS X prior to Mac OS X v10.4 for PowerPC, you can specify separate deployment OS versions for Intel and PowerPC-based builds. To do so, use the per-architecture variants of the `MACOSX_DEPLOYMENT_TARGET` build setting:

   a. In the target or project inspector, add entries for the `MACOSX_DEPLOYMENT_TARGET_ppc` and `MACOSX_DEPLOYMENT_TARGET_i386` build settings.

   b. Set `MACOSX_DEPLOYMENT_TARGET_i386` to `10.4`.

   c. Set `MACOSX_DEPLOYMENT_TARGET_ppc` to the earliest system version that you want to support for PowerPC. For example, to support Mac OS X v10.3 and later, set `MACOSX_DEPLOYMENT_TARGET_ppc` to `10.3`.

3. Select the appropriate compiler.

   As mentioned, you must use GCC 4.0 to compile for Mac OS X v10.4 and for Intel-based Macintosh computers. However, code built with GCC 4.0 will not run on versions of Mac OS X earlier than 10.3.0. Furthermore, due to changes in the C++ ABI between the GCC 4.0 and GCC 3.3 compilers, C++ or Objective-C++ code built with GCC 4.0 does not run on versions of Mac OS X prior to Mac OS X v10.3.9.

2006-11-07 | © 2006 Apple Computer, Inc. All Rights Reserved.

If you can require Mac OS X 10.3.9 or later (10.3.0 or later if you do not have C++ or Objective-C++ code), you can use the same version of GCC to build for both Intel- and PowerPC-based Macintosh computers. Otherwise, you must use different versions of the GCC compiler to build for each platform. To do so, use the per-architecture variants of the `GCC_VERSION` build setting:

a. In the Build pane of the target or project inspector, add entries to the build settings table for the `GCC_VERSION_ppc` and `GCC_VERSION_i386` build settings.

b. Set `GCC_VERSION_ppc` to `3.3`.

c. Set `GCC_VERSION_i386` to `4.0`.

> **Note:** Xcode also has per-architecture build setting variants for the `LD` and `LDPLUSPLUS` build settings; this lets you specify different linker versions for PowerPC and Intel builds.

Although it is easiest to build universal binaries from Xcode, you can also build them from the command line. For an example of how to build from the command-line targeting the same SDK for both Intel and PowerPC-based builds, see Compiling Your Code in Mac OS X in *Porting UNIX/Linux Applications to Mac OS X*.

If you need to target different SDKs for Intel and PowerPC-based builds—for example if you need to support versions of Mac OS X prior to Mac OS X v10.4 for PowerPC-based builds—you will need to compile for each architecture separately and then use the `lipo` tool to create a single executable file. To build a binary for Intel-based Macintosh computers, use the settings described in "Configuring a Makefile-Based Project" (page 20) to specify the target SDK and the deployment target. For instructions on using `lipo`, see `lipo(1)`.

2006-11-07 | © 2006 Apple Computer, Inc. All Rights Reserved.

2006-11-07 | © 2006 Apple Computer, Inc. All Rights Reserved.

Determining the Version of a Framework

Occasionally you might have the need to determine what version of the Application Kit (or Foundation) framework you're using at runtime. You might, for example, have code that uses new features in Cocoa and have been putting `#ifdef` statements in place to control these workarounds. However, others may take this code from one build and compile and run it on another where the feature may not yet be available.

There are several ways to check at runtime for new features in the versions of the Cocoa frameworks your application is linked against. You can dynamically look for a given class or method (for example, by using the `instancesRespondToSelector:` method) and, if it isn't there, follow a different code path. Another way is to use the the global framework-version constants provided by the Application Kit and Foundation frameworks.

The Application Kit (in `NSApplication.h`) declares the `NSAppKitVersionNumber` constant, which you can use to detect different versions of the Application Kit framework:

```
/* The version of the AppKit framework */

APPKIT_EXTERN double NSAppKitVersionNumber;
#define NSAppKitVersionNumber10_0 577
#define NSAppKitVersionNumber10_1 620
#define NSAppKitVersionNumber10_2 663
#define NSAppKitVersionNumber10_2_3 663.6
#define NSAppKitVersionNumber10_3 743
#define NSAppKitVersionNumber10_3_2 743.14
#define NSAppKitVersionNumber10_3_3 743.2
#define NSAppKitVersionNumber10_3_5 743.24
```

You can compare against this value to determine which version of the Application Kit your code is running against. One typical approach is to floor the value of the global constant and check the result against the constants declared in `NSApplication.h`. For example:

```
if (floor(NSAppKitVersionNumber) <= NSAppKitVersionNumber10_0) {
  /* On a 10.0.x or earlier system */
} else if (floor(NSAppKitVersionNumber) <= NSAppKitVersionNumber10_1) {
  /* On a 10.1 - 10.1.x system */
} else if (floor(NSAppKitVersionNumber) <= NSAppKitVersionNumber10_2) {
  /* On a 10.2 - 10.2.x system */
} else if (floor(NSAppKitVersionNumber) <= NSAppKitVersionNumber10_3) {
  /* On a 10.3 - 10.3.x system */
} else {
  /* Tiger or later system */
}
```

Similarly, Foundation (in `NSObjCRuntime.h`) declares the `NSFoundationVersionNumber` global constant and specific values for each version.

Some individual headers for other objects and components may also declare the versions numbers for `NSAppKitVersionNumber` where some bug fix or functionality is available in a given update.

```
#define NSAppKitVersionWithSuchAndSuchBadBugFix 582.1
```


2006-11-07 | © 2006 Apple Computer, Inc. All Rights Reserved.

2006-11-07 | © 2006 Apple Computer, Inc. All Rights Reserved.

Document Revision History

This table describes the changes to *Cross-Development Programming Guide*.

Date	Notes
2006-11-07	Added details on cross-development and universal binaries.
	Added chapter "Determining the Version of a Framework" (page 29).
	In "Limitations" (page 14), noted requirement to build with GCC 3.3 when targeting Mac OS X versions prior to Mac OS X v10.3.0.
	In "Cross-Development and Universal Binaries" (page 25), corrected Mac OS X version requirements for code compiled with GCC 4.0.
	Added links to further information on building universal binaries from the command line.
	Changed deployment target in example in "Cross-Development and Universal Binaries" (page 25) to Mac OS X version 10.3.
2006-05-23	Corrected a build-setting specification and added per-architecture-build-setting availability details.
	Corrected specification for `LDFLAGS` build setting in "Configuring a Makefile-Based Project" (page 20).
	Added availability details for the per-architecture build settings feature in "Cross-Development and Universal Binaries" (page 25).
2006-03-08	Added a link to Technical Note TN2163, which describes how to develop a universal I/O Kit driver.
2006-02-07	Made minor corrections.
	Added information on building universal binary versions of kernel extensions.
	Clarified use of `LDFLAGS` in "Select a Deployment OS" (page 20).
2005-11-09	Added a section on using cross-development to create universal binaries. Added information on identifying deprecated API. Corrected errors.
	Added "Cross-Development and Universal Binaries" (page 25).
	Added "Finding Deprecated API" (page 23).
	Reorganized content to create separate sections for configuring cross-development settings in Xcode and in makefile-based projects.

2006-11-07 | © 2006 Apple Computer, Inc. All Rights Reserved.

Date	Notes
	Corrected sample code listing in "Conditionally Compile for Different SDKs" (page 22).
2005-08-11	Fixed a bug in code that checks for the existence of a symbol. Updated steps for setting a deployment target to reflect Xcode 2.1.
2005-06-04	Updated information about checking for undefined functions. Added information about how to support SDKs from command-line programs.
2003-09-16	Made a number of changes throughout this document to reflect the final status of cross-development support as it shipped in Mac OS X version 10.3.
2003-08-21	First general release of document.

2006-11-07 | © 2006 Apple Computer, Inc. All Rights Reserved.