# SDK Compatibility Guide

**General**

2010-02-16

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



Simultaneously published in the United States and Canada.

# Contents

# Figures and Listings

# Introduction

**Note:** This document was previously titled *Cross-Development Programming Guide*.

Xcode Developer Tools includes software development kits (SDKs) that enable you to develop software that can be deployed on specified versions of Mac OS X or iOS, including versions different from the one you are developing on. This technology enables your application to be compatible with previous versions of the operating system, taking advantage of new features when they are available while gracefully degrading when running on older systems. Some Apple frameworks automatically modify their behavior based on the SDK an application was built against for improved compatibility.

**Note:** This document does not explain how to develop code that runs on both the Mac OS X and iOS platforms. Although Xcode enables you to switch platforms by simply choosing a different SDK, there are fundamental design differences between Mac OS X and iOS programs. See “Migrating from Cocoa" for more information.

You should read this document if your application needs to target a specific version or multiple versions of Mac OS X or iOS.

## Organization of This Document

This document contains the following chapters:

- “Overview of SDK-Based Development” (page 9) describes how SDK-based development works.
- “Configuring a Project for SDK-Based Development” (page 11) describes how to set up your project to use an SDK.
- “Using SDK-Based Development” (page 15) explains how to check for framework versions, deal with undefined methods and functions, and find deprecated APIs.

# Overview of SDK-Based Development

Apple makes SDKs available for specific versions of Mac OS X and iOS. Using these SDKs allows you to build against the headers and libraries of an operating system version other than the one you're running on. For example, you can build for Mac OS X version 10.4 while running on Mac OS X version 10.6. This chapter describes how SDK-based development works.

The Mac OS X SDKs are installed as part of the Xcode Essentials install package with Xcode 3.2 and later. Xcode release notes list the SDKs supported by each release. When developing for iOS, you always use an SDK downloaded from the iPhone Dev Center website.

You can take advantage of SDK-based development in these 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.

To develop software that can be deployed on, and take advantage of features from, different versions of Mac OS X or iOS, you specify which version (or SDK) of Mac OS X or iOS headers and libraries to build with. You can also specify the earliest Mac OS X or iOS system version on which the software will run. These concepts are described in "SDK Settings" (page 11).

## Behavior Selection in Frameworks

As frameworks evolve through various releases, APIs are introduced or deprecated, and behaviors of existing APIs may occasionally change. Apple makes every effort to minimize changes that may cause incompatibilities, in some cases providing alternate behaviors based on the framework version. In other cases your code may need to determine the framework version and adjust accordingly.

As a backward-compatibility mechanism, Apple frameworks sometimes check for the version of the SDK an application was built against, and, if it is an older SDK, modify the behavior to be more compatible. This is done in cases where incompatibility problems are predicted or discovered.

**Note:** Most version-related behavior changes are listed in the framework release notes, but they are not necessarily described in the reference documentation. To understand the differences from one release to another, you must carefully review the release notes.

Typically, frameworks detect how an application was built by looking at the version of the system frameworks the application was linked against. Thus, as a result of relinking your application on a newer version or its SDK, you might notice different behaviors, some of which might cause incompatibilities. In these cases, because the application is being rebuilt, you should address these issues at the same time as well. For this

reason, if you are doing a small incremental update of your application to address a few bugs, for example, it's usually best to continue building on the same build environment and libraries used originally, that is, against the original SDK.

In some cases, frameworks provide defaults (preferences) settings which can be used to get the old or new behavior, independent of what system an application was built against. Often these preferences are provided for debugging purposes only; in some cases the preferences can be used globally to modify the behavior of an application by registering the values (do it somewhere very early, with the `NSUserDefaults` method `registerDefaults:`).

# Configuring a Project for SDK-Based Development

This chapter describes the configuration of installed SDKs and explains how to set up your Xcode project to use an SDK.

## SDK Header Files and Stub Libraries

When you install the 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. The Mac OS X SDKs are named by major version, such as 10.4 and 10.5, but they include the latest minor version as well.

**iOS Note:** The iPhone SDKs are installed in the `/Developer/Platforms` directory, within which is a directory for each platform, such as `iOS.platform`. The platform directories, in turn, contain `Developer/SDKs` directories that contain the SDKs for that platform. The iOS SDK names include minor versions of OS releases.

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 Xcode Developer Tools installed.

The libraries in an SDK are stub libraries for linking only; they do not contain executable code but just the exported symbols. SDK support only works with native build targets.

## SDK Settings

To use an SDK for an Xcode project, you make two selections among your project's build settings:

- **Choose a Base SDK to build your project.** Your software can use features available in OS versions up to and including the one corresponding to the SDK you choose. If you don't select an SDK, the default is to build for the current operating system.

  The Xcode build setting name for this parameter is `SDKROOT (Base SDK)`.

- **Choose a Deployment OS version.** This identifies the earliest system version on which the software can run. By default, this is set to the version of the OS corresponding to the Base SDK version and later.

  The Xcode build setting names for this parameter are `MACOSX_DEPLOYMENT_TARGET (Mac OS X Deployment Target)` and `IPHONEOS_DEPLOYMENT_TARGET (iOS Deployment Target)`.

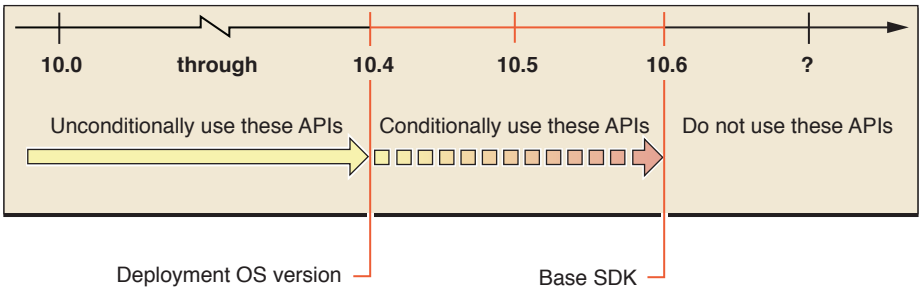
For possible values and more information about build settings in Xcode, see *Xcode Build Setting Reference* and "Running Applications" in *iOS Development Guide*.

Xcode uses information from these settings when building and linking your software. You can unconditionally use features from OS versions up to and including the system version that you have specified as your Deployment OS version. You can use features from system versions later than the Deployment OS—up to and including the system version you've selected as your Base SDK—but you must check for the availability of the feature, as described in "Checking for Undefined Method and Function Calls" (page 16).

When you build your application, your Deployment OS version selection is reflected in the `MinimumOSVersion` entry in the application's `Info.plist` file. For iOS applications, the App Store indicates the iOS release requirement for your application based on its `MinimumOSVersion` entry.

Figure 2-1 shows a timeline for a project with a Base SDK of Mac OS X version 10.6, with a Deployment OS version set to Mac OS X version 10.4. (The version numbers in the figure represent all releases of that version, including system updates.)

**Figure 2-1** SDK development timeline




In this example, the software can freely use any features from Mac OS X versions 10.0 through 10.4. It can also take advantage of features from Mac OS X versions 10.5 and 10.6, but it must first check that the feature is available.

The effect of these settings at compile time and run time is a follows:

- If your code uses a symbol that is not defined in the selected Base SDK, you get a compile-time error.
- If your code uses a symbol that is defined in the selected Base SDK but is marked as deprecated, you get a compile-time warning.
- If your code uses a symbol that is defined in the selected Base SDK and in the selected Deployment OS version, your code builds and links normally. At run time:
  - On systems earlier than the Deployment OS version, your code may fail to load if you use symbols unavailable in that version.
  - On systems equal to or later than the Deployment OS version, your code will have 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 "Checking for Undefined Method and Function Calls" (page 16).

**iOS Note:** Mac OS X v10.6 does not support using iPhone Simulator SDKs prior to version 3.0. In addition, when building with the simulator SDKs, the binary runs only on the same OS version as the SDK, not on earlier or later versions.

Of course, you should also check to see if you are using deprecated APIs; though still available, these APIs are not guaranteed to be supported in the future. The compiler can warn you about the presence of deprecated APIs, as described in "Finding Deprecated APIs" (page 18).

When you change the Base SDK setting, in addition to changing the headers and libraries your code builds against, Xcode adjusts the behavior of other features appropriately. For example, symbol lookup, code completion, and header file opening are based on the headers in the Base SDK, rather than those of the currently running system. Similarly, the Xcode Quick Help affinity mechanism ensures that documentation lookup uses the doc set corresponding to the Base SDK.

You can also set the Base SDK and Deployment OS version for any individual target in the project. Any value explicitly assigned to an individual target inspector overrides, for that target, the value assigned to that setting in the project inspector. (However, some Xcode features that attempt to correlate with the Base SDK setting, such as symbol definition and documentation lookup, may work differently.)

# Configuring a Makefile-Based Project

If you have a makefile-based project, you can also take advantage of SDK-based development, by adding the appropriate options to your compile and link commands. Using SDKs in makefile-based projects requires GCC 4.0 or later. To choose an SDK, you use the `-isysroot` option with the compiler and the `-syslibroot` option with the linker. Both options require that you specify the full path to the desired SDK directory. To set the Deployment OS version in a makefile, use a makefile variable of the form: `ENVP=MACOSX_DEPLOYMENT_TARGET=10.4`. To use this variable in your makefile, include it in front of your compile and link commands.

# Setting the Prefix File

Xcode supports prefix files, header files that are included implicitly by each of your source files when they're built. Many Xcode project templates generate prefix files automatically, including umbrella frameworks appropriate to the selected type of application. For efficiency, prefix files are precompiled and cached, so Xcode does not need to recompile many lines of identical code each time you build your project. You can add directives to import the particular frameworks on which your application depends.

If you are using SDK-based development, you must ensure that your prefix file that takes into account the selected SDK. That is, don't set the prefix file to an umbrella header file using an absolute path, such as `/System/Library/Frameworks/Cocoa.framework/Versions/A/Headers/Cocoa.h`. This absolute path does not work because the specified header is from the current system, rather than the chosen SDK.

To include umbrella framework headers, add the appropriate `#import <Framework/Framework.h>` directives to your prefix file. With this technique, the compiler always chooses the headers from the appropriate SDK directory. For example, if your project is named TestSDK and it has a prefix file `TestSDK_Prefix.pch`, add the following line to that file:

```
#import <Cocoa/Cocoa.h>
```

If you are using Objective-C, it's preferable to use the `#import` directive rather than `#include` (which you must use in procedural C programs) because `#import` guarantees that the same header file is never included more than once.

# Using SDK-Based Development

This chapter describes SDK-based development techniques to use in your projects, explaining how your code can tell what version of a framework it is using at run time, check for undefined method and functions, compile conditionally for different SDKs, and find deprecated APIs.

## Determining the Version of a Framework

There are several ways to check at run time for availability of certain features in the versions of the frameworks your application is linked against. You can dynamically look for a given class or method (for example, by using the `instancesRespondToSelector:` method, as described in "Checking for Undefined Method and Function Calls" (page 16)) and, if it isn't there, follow a different code path. Another way is to use global framework-version constants, if they are provided by the framework.

For example, the Application Kit (in `NSApplication.h`) declares the `NSAppKitVersionNumber` constant, which you can use to detect different versions of the Application Kit 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
#define NSAppKitVersionNumber10_3_7 743.33
#define NSAppKitVersionNumber10_3_9 743.36
#define NSAppKitVersionNumber10_4 824
#define NSAppKitVersionNumber10_4_1 824.1
#define NSAppKitVersionNumber10_4_3 824.23
#define NSAppKitVersionNumber10_4_4 824.33
#define NSAppKitVersionNumber10_4_7 824.41
#define NSAppKitVersionNumber10_5 949
#define NSAppKitVersionNumber10_5_2 949.27
#define NSAppKitVersionNumber10_5_3 949.33
```

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 10.3 - 10.3.x system */
} else if (floor(NSAppKitVersionNumber) <= NSAppKitVersionNumber10_4) {
  /* On a 10.4 - 10.4.x system */
} else if (floor(NSAppKitVersionNumber) <= NSAppKitVersionNumber10_5) {
  /* On a 10.5 - 10.5.x system */
} else {
  /* 10.6 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 version numbers for `NSAppKitVersionNumber` where some bug fix or functionality is available in a given update.

# Checking for Undefined Method and Function Calls

To run successfully, your code must avoid calling methods and 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 for the existence of each Objective-C method or C function before calling it.

For example, suppose you build your application to use features in Mac OS X version 10.5 (by setting the Base SDK to that version) but still run on Mac OS X version 10.4 (by setting the Deployment OS version to that version). In Objective-C, you use the `instancesRespondToSelector:` method to see if the method selector in question is available. For example, to use the `setDisplaysToolTips:` method, first available in version 10.5, you could use code like the following:

**Listing 3-1** Checking for an undefined method

```
if ([NSTextView instancesRespondToSelector:@selector(setDisplaysLinkToolTips:)])
    {
        [myTextView setDisplaysLinkToolTips:NO];
    }
    else
    {
        // Code to disable link tooltips with earlier technology
    }
```

When your code runs on Mac OS X version 10.5, it calls `setDisplaysLinkToolTips:` to disable display of help tags when the mouse hovers over links. When it runs on version 10.4, it must disable display of link help tags using code you wrote for that version.

If you were to build this code with different settings, you would see the following results:

- If you select a Base SDK setting of `Mac OS X 10.4`:

  The build would fail because `setDisplaysLinkToolTips:` is not defined in that system version.

- With a Base SDK setting of `Mac OS X 10.5`, if you set the Deployment OS version to:
  - `Mac OS X 10.5`: The software would run only on v10.5 or later and fail to launch on earlier systems.
  - `Mac OS X 10.4`: The software would run on v10.5 and v10.4 but fail to launch on earlier systems.

You can check for the existence of Objective-C properties by passing the getter method name (which is the same as the property name) to `instancesRespondToSelector:`.

In C, instead of using the `instancesRespondToSelector:` method, you compare the function pointer to `NULL`. For example, to use the `CGColorCreateGenericCMYK` function, first available in version 10.5, you could use code like the following:

**Listing 3-2** Checking for a null function pointer

```
if (CGColorCreateGenericCMYK != NULL)
{
    CGColorCreateGenericCMYK(0.1,0.5.0.0,1.0,0.1);
}
else
{
    // Code to create a color object with earlier technology
}
```

**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.

# Conditionally Compiling for Different SDKs

If you build the same source code using different SDKs, you might need to compile code conditionally based on the SDK in use. You can do this by using preprocessor directives with the macros defined in `Availability.h`.

**Note:** `Availability.h` is for iOS and Mac OS X v10.6 and later development. `AvailabilityMacros.h` is the earlier version introduced in Mac OS X 10.2. These header files reside in the `/usr/include` directory.

For example, suppose the code shown in Listing 3-2 must be compiled against the Mac OS X 10.4 SDK. Because the code refers to the `CGColorCreateGenericCMYK` function (which was introduced in Mac OS X v10.5) that portion of the code must be excluded when compiling against the earlier SDK. This is because even referring to the `CGColorCreateGenericCMYK` function in such a case causes a compiler error. The use of the `__MAC_OS_X_VERSION_MAX_ALLOWED` macro removes the code during preprocessing unless it is being compiled against the 10.5 headers. Note the use of the value `1050` instead of symbol `__MAC_10_5` in the `#if` comparison clause: if the code is loaded on a system that does not include the symbol definition, the comparison still works.

```
#ifdef __MAC_OS_X_VERSION_MAX_ALLOWED
    // code only compiled when targeting Mac OS X and not iPhone
    // note use of 1050 instead of __MAC_10_5
    #if __MAC_OS_X_VERSION_MAX_ALLOWED >= 1050
        if(CGColorCreateGenericCMYK != NULL)
        {
            CGColorCreateGenericCMYK(0.1,0.5.0.0,1.0,0.1);
        }
        else
        {
    #endif
            // code to create a color object with earlier technology
```

```
    #if __MAC_OS_X_VERSION_MAX_ALLOWED >= 1050
        }
    #endif
#endif
}
```

In addition to using preprocessor macros, the preceding code also assumes that the code could be compiled using a newer SDK but deployed on a computer running Mac OS X v10.4 and earlier. Specifically, it checks for the existence of the `CGColorCreateGenericCMYK` symbol before attempting to call it. This prevents the code from generating a runtime error, which can occur if you build the code against a newer SDK but deploy it on an older system. For more information about instituting runtime checks to determine the presence of symbols, see "Checking for Undefined Method and Function Calls" (page 16).

# Finding Deprecated APIs

As Mac OS X and iOS evolve, the APIs and technologies they encompass are sometimes changed to meet the needs of developers. As part of this evolution, less efficient interfaces are deprecated in favor of newer ones. The availability macros help you find deprecated interfaces.

**Note:** Deprecation does not mean the immediate 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 that you migrate to newer interfaces as soon as possible because deprecated APIs may be deleted from a future version of the OS. 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. If you compiled your project with a Deployment OS version of Mac OS X 10.5 and used an interface tagged in this way, you would get a warning that the interface is now deprecated. 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 would 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.

# Document Revision History

This table describes the changes to *SDK Compatibility Guide*.

| Date | Notes |
| --- | --- |
| 2010-02-16 | Corrected an example that shows how to use conditional compilation macros. |
| 2009-09-09 | Clarified build setting information. |
| 2009-05-12 | Changed the title from Cross-Development Programming Guide. Added information about iOS SDK-based development. Revised text significantly and removed obsolete information. |
| 2006-11-07 | Added details on cross-development and universal binaries. |
| | Added chapter "Determining the Version of a Framework" (page 15). Noted requirement to build with GCC 3.3 when targeting Mac OS X versions prior to Mac OS X v10.3.0. 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. |
| 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". Added availability details for the per-architecture build settings feature. |
| 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`. |
| 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" chapter. Added "Finding Deprecated APIs" (page 18). Reorganized content to create separate sections for configuring cross-development settings in Xcode and in makefile-based projects. Corrected sample code listing in "Conditionally Compiling for Different SDKs" (page 17). |
| 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. |

Document Revision History

| Date | Notes |
| --- | --- |
| 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. |