Framework Programming Guide

Tools & Languages: IDEs

2006-11-07

Contents

Figures, Tables, and Listings

Introduction to Framework Programming Guide

Mac OS X makes extensive use of frameworks to distribute shared code and resources, such as the interfaces to the system itself. You can create frameworks of your own to provide shared code and resources to one or more of your company's applications. You can also create frameworks containing class libraries or add-on modules with the intention of distributing them to other developers.

The information in this document provides the background you need to create frameworks and the steps needed to create them in Xcode. Although creating frameworks is not difficult, there are some guidelines you should follow when doing so. Xcode simplifies the creation process by helping you create the framework bundle and manage the information and placement of files in that bundle. However, this document also provides additional information about how to perform many less obvious tasks.

Organization of This Document

This document contains the following articles:

- "What are Frameworks?" (page 11) provides background information about what frameworks are and how they're used.
- "Anatomy of Framework Bundles" (page 13) describes the basic structure of frameworks, including umbrella frameworks.
- "Framework Versions" (page 19) describes the system used to manage different framework versions and how you specify version information when you create a framework.
- "Frameworks and Binding" (page 25) explains how framework symbols are bound to an application at runtime. It also explains how to improve the load time of your framework through the use of prebinding.
- "Frameworks and Weak Linking" (page 31) explains the concept of "weak-linking" for framework symbols and shows you how to use this feature with both your own frameworks and third-party frameworks.
- "Guidelines for Creating Frameworks" (page 35) provides guidelines on the best practices to use for creating frameworks.
- "Creating a Framework" (page 39) shows how to create public frameworks and private embedded frameworks using Xcode.
- "Initializing a Framework at Runtime" (page 45) shows how to create a load-time initialization routine for your framework.
- "Exporting Your Framework Interface" (page 49) shows how to limit the symbols exported by your framework to the exact set you want.
- "Installing Your Framework" (page 51) explains the conventions for where to install your custom frameworks.
- "Including Frameworks" (page 53) shows the basic ways to use frameworks in applications.

What are Frameworks?

A **framework** is a hierarchical directory that encapsulates shared resources, such as a dynamic shared library, nib files, image files, localized strings, header files, and reference documentation in a single package. Multiple applications can use all of these resources simultaneously. The system loads them into memory as needed and shares the one copy of the resource among all applications whenever possible.

A framework is also a bundle and its contents can be accessed using Core Foundation Bundle Services or the Cocoa NSBundle class. However, unlike most bundles, a framework bundle does not appear in the Finder as an opaque file. A framework bundle is a standard directory that the user can navigate. This makes it easier for developers to browse the framework contents and view any included documentation and header files.

Frameworks serve the same purpose as static and dynamic shared libraries, that is, they provide a library of routines that can be called by an application to perform a specific task. For example, the Application Kit and Foundation frameworks provide the programmatic interfaces for the Cocoa classes and methods. Frameworks offer the following advantages over static-linked libraries and other types of dynamic shared libraries:

■ Frameworks group related, but separate, resources together. This grouping makes it easier to install, uninstall, and locate those resources.

■ Frameworks can include a wider variety of resource types than libraries. For example, a framework can include any relevant header files and documentation.

■ Multiple versions of a framework can be included in the same bundle. This makes it possible to be backward compatible with older programs.

■ Only one copy of a framework's read-only resources reside physically in-memory at any given time, regardless of how many processes are using those resources. This sharing of resources reduces the memory footprint of the system and helps improve performance.

> **Note:** Frameworks are not required to provide a programmatic interface and can include only resource files. However, such a use is not common.

The Darwin layer contains many static and dynamic libraries but otherwise, most Mac OS X interfaces are packaged as frameworks. Some key frameworks—including Carbon, Cocoa, Application Services, and Core Services—provide convenient groupings of several smaller but related frameworks. These framework groups are called **umbrella frameworks** and they act as an abstraction layer between a technology and the subframeworks that implement that technology.

In addition to using the system frameworks, you can create your own frameworks and use them privately for your own applications or make them publicly available to other developers. Private frameworks are appropriate for code modules you want to use in your own applications but do not want other developers to use. Public frameworks are intended for use by other developers and usually include headers and documentation defining the framework's public interface.

Anatomy of Framework Bundles

In Mac OS X, shared resources are packaged using standard frameworks and umbrella frameworks. Both types of framework feature the same basic structure and can contain resources such as a shared library, nib files, image files, strings files, information property lists, documentation, header files, and so on. Umbrella frameworks add minor refinements to the standard framework structure, such as the ability to encompass other frameworks.

Frameworks are packaged in a bundle structure. The framework bundle directory ends with the `.framework` extension, and unlike most other bundle types, a framework bundle is presented to the user as a directory and not as a file. This openness makes it easy for developers to browse any header files and documentation included with the framework.

Framework Bundle Structure

Framework bundles use a bundle structure different from the bundle structure used by applications. The structure for frameworks is based on an earlier bundle format, and allows for multiple versions of the framework code and header files to be stored inside the bundle. This type of bundle is known as a **versioned bundle**. Supporting multiple versions of a framework allows older applications to continue running even as the framework binary continues to evolve.

The system identifies a framework by the `.framework` extension on its directory name and by the `Resources` directory at the top level of the framework bundle. Inside the `Resources` directory is the `Info.plist` file that contains the bundle's identifying information. The actual `Resources` directory does not have to reside physically at the top-level of the bundle. In fact, the system frameworks that come with Mac OS X have a symbolic link to the framework's `Resources` directory in this location. The link points to the most current version of the `Resources` directory, buried somewhere inside the bundle.

The contents of the `Resources` directory are similar to those for application bundles. (See "Anatomy of a Modern Bundle" in *Bundle Programming Guide* for more information.) Localized resources are put in language-specific subdirectories that end with the `.lproj` extension. These subdirectories hold strings, images, sounds, and interface definitions localized to the language and region represented by the directory. Nonlocalized resources reside at the top level of the `Resources` directory.

Framework Versions

When you build a new framework project in Xcode, the build environment creates a versioned bundle structure for you automatically. Listing 1 shows the basic directory structure of the resulting bundle.

Listing 1   A simple framework bundle

```
MyFramework.framework/
    MyFramework  -> Versions/Current/MyFramework
    Resources    -> Versions/Current/Resources
```

```
Versions/
    A/
        MyFramework
        Resources/
            English.lproj/
                InfoPlist.strings
            Info.plist
    Current  -> A
```

In this listing, the `Versions` directory is the only real directory at the top level of the bundle. Both `MyFramework` and `Resources` are symbolic links to items in `Versions/A`. The reason for the symbolic links is that directory `Versions/A` contains the actual contents of the framework. It contains both the executable and the resources used by the framework.

Listing 1 shows that the top-level symbolic links don't point directly to items inside the `Versions/A` directory. Instead, they point to items in the `Versions/Current` directory, which itself is a symbolic link to `Versions/A`. This additional level of indirection simplifies the process of changing the top-level links of frameworks with many resource types to point to a specific major version of the framework, because only one link, `Versions/Current`, needs to be updated.

Each real directory in `Versions` contains a specific **major version** of the framework. Earlier major versions of the framework are needed by clients created when those versions were current at the time. New major versions of a framework are required only when changes in the framework's dynamic shared library would prevent a program linked to it from running. Programs built using the earlier major version of the library must continue to use it, but programs in development should link against the current version of the library.

Listing 2 shows the MyFramework framework after a adding major revision to it.

Listing 2   A framework with multiple versions

```
MyFramework.framework/
    MyFramework  -> Versions/Current/MyFramework
    Resources    -> Versions/Current/Resources
    Versions/
        A/
            MyFramework
            Resources/
                English.lproj/
                    InfoPlist.strings
                Info.plist
        B/
            MyFramework
            Resources/
                English.lproj/
                    InfoPlist.strings
                Info.plist
        Current  -> B
```

Through the `Versions/Current` symbolic link, `MyFramework.framework/MyFramework` points to the dynamic shared library of the now current version of the framework, `Versions/B/MyFramework`. Because of the use of symbolic links, during a program's link process, the linker finds the latest version of the framework's library. This arrangement ensures that new programs are linked against the latest major version of the framework and that programs built with the earlier major version of the framework continue to work unchanged. For more on major and minor framework versions, and on versioning in general, see "Framework Versions" (page 19). For more information on using dynamic libraries, see *Dynamic Library Programming Topics*.

> **Important:** For the linker to find and link the dynamic library, the name of the framework (without the `.framework` extension), the symbolic link, and the dynamic library must be the same.

Additional Directories

Frameworks typically include more directories than just the `Resources` directory. For example, `Headers`, `Documentation`, and `Libraries`. Thus, adding a `Headers` directory to the example in Listing 2 (page 14) would result in a framework like the one shown in Listing 3.

Listing 3   A framework with additional resource types

```
MyFramework.framework/
    Headers        -> Versions/Current/Headers
    MyFramework  -> Versions/Current/MyFramework
    Resources      -> Versions/Current/Resources
    Versions/
        A/
            Headers/
                MyHeader.h
            MyFramework
            Resources/
                English.lproj/
                    Documentation
                    InfoPlist.strings
                Info.plist
        B/
            Headers/
                MyHeader.h
            MyFramework
            Resources/
                English.lproj/
                    Documentation
                    InfoPlist.strings
                Info.plist
        Current  -> B
```

To create additional directories in your framework, you must add build phases to the appropriate target in Xcode. The Copy Files build phase lets you create directories and copy selected files into those directories. Table 1 lists some of the typical directories you might add to your framework.

Table 1   Standard directories for frameworks

Directory	Description
Headers	Contains any public headers you want to make available to external developers.
Documentation	Contains HTML or PDF files describing the framework interfaces. Typically, documentation directories do not reside at the top level of your framework. Instead, they reside inside your language-specific resource directories.
Libraries	Contains any secondary dynamic libraries your framework requires.

Framework Configuration

Frameworks require the same sort of configuration as any other type of bundle. In the information property list for your framework, you should include the keys listed in Table 2. Most of these keys are included automatically when you set up the framework properties in Xcode, but you must add some manually.

Table 2   Framework configuration keys

Key	Description
`CFBundleName`	The framework display name
`CFBundleIdentifier`	The framework identifier (as a Java-style package name)
`CFBundleVersion`	The framework version
`CFBundleSignature`	The framework signature
`CFBundlePackageType`	The framework package type (which is always `'FMWK'`)
`NSHumanReadableCopyright`	Copyright information for the framework
`CFBundleGetInfoString`	A descriptive string for the Finder

Because frameworks are never associated with documents directly, you should never specify any document types. You may include display name information if you wish.

For more information on configuration and information property lists, see *Runtime Configuration Guidelines*.

Umbrella Framework Bundle Structure

The structure of an umbrella framework is similar to that of a standard framework, and applications do not distinguish between umbrella frameworks and standard frameworks when linking to them. However, two factors distinguish umbrella frameworks from other frameworks. The first is the manner in which they include header files. The second is the fact that they encapsulate subframeworks.

The Purpose of Umbrella Frameworks

The purpose of an umbrella framework is to provide all the necessary interfaces for programming in a particular application environment. Umbrella frameworks hide the complex cross-dependencies among the many different pieces of system software. Thus you do not need to know what set of frameworks and libraries you must import to accomplish a particular task. Umbrella frameworks also make faster builds possible through the use of precompiled headers.

An umbrella framework simply includes and links with constituent subframeworks and other public frameworks. An umbrella framework encompasses all the technologies and APIs that define an application environment or a layer of system software. It also provides a layer of abstraction between what outside developers link their programs with and what Apple engineering provides as implementation.

A subframework is structurally a public framework that packages a specific Apple technology, such as Apple events, Quartz, or Open Transport. However, a subframework is public with restrictions. Although the APIs of subframeworks are public, Apple has put mechanisms in place to prevent developers from linking directly with subframeworks (see "Restrictions on Subframework Linking" (page 55)). A subframework always resides in an umbrella framework installed in /System/Library/Frameworks, and within this umbrella framework, its header files are exposed.

Some umbrella frameworks include other umbrella frameworks; this is particularly the case with the umbrella frameworks for the Carbon and Cocoa application environments. For example, both Carbon and Cocoa (directly or indirectly) import and link with the Core Services umbrella framework (CoreServices.framework). This umbrella framework, in turn, imports and links with subframeworks such as Core Foundation.

The exact composition of the subframeworks within an umbrella framework is an internal implementation detail subject to change. By providing a level of indirection, umbrella frameworks insulate developers from these changes. Apple might restructure the subframeworks within an umbrella framework and might add, rename, or remove the header files within subframeworks. If you include the master header file for the subframework, these changes should not affect your programs.

The Umbrella Framework Bundle

Physically, umbrella frameworks have a similar structure to standard frameworks. One significant difference is the addition of a Frameworks directory to contain the subframeworks that make up the umbrella framework.

Listing 4 (page 17) shows a partial listing of the Core Services framework. (The contents of the subframeworks are not included since they are not referenced anyway.) As with standard frameworks, the top-level items are symbolic links to items deeper within the framework directory structure. In this case, the linked libraries and directories are located in folder A of the framework.

Listing 4   Structure of the Core Services umbrella framework

```
CoreServices.framework/
    CoreServices           -> Versions/Current/CoreServices
    CoreServices_debug     -> Versions/Current/CoreServices_debug
    CoreServices_profile   -> Versions/Current/CoreServices_profile
    Frameworks             -> Versions/Current/Frameworks
    Headers                -> Versions/Current/Headers
    Resources              -> Versions/Current/Resources
    Versions/
        A/
            CoreServices
            CoreServices_debug
            CoreServices_profile
            Frameworks/
                CarbonCore.framework
                CFNetwork.framework
                OSServices.framework
                SearchKit.framework
                WebServicesCore.framework
            Headers/
                Components.k.h
                CoreServices-gcc3.p
                CoreServices-gcc3.pp
                CoreServices.h
```

```
            CoreServices.p
            CoreServices.pp
            CoreServices.r
        Resources/
            Info-macos.plist
            version.plist
    Current             -> A
```

Unlike standard frameworks, the `Headers` directory of an umbrella framework contains a more limited set of header files. It does not contain a collection of the headers in its subframeworks. Instead, it contains only the master header file for the framework. When referring to an umbrella framework in your source files, you should include only the master header file. See "Including Frameworks" (page 53) for more information.

Framework Versions

You can create different versions of a framework based on the type of changes made to its dynamic shared library. There are two types of versions: major (or incompatible) and minor (or compatible) versions. Both have an impact on the programs linked to the framework, albeit in different ways.

Major Versions

A major version of a framework is also known as an incompatible version because it breaks compatibility with programs linked to a previous version of the framework's dynamic shared library. Any such program running under the newer version of the framework is likely to experience runtime errors because of the changes made to the framework.

The following sections describe how you designate major version information in your framework and how the system uses that information to ensure applications can run.

Major Version Numbering Scheme

Because all major versions of a framework are kept within the framework bundle, a program that is incompatible with the current version can still run against an older version if needed. The path of each major version encodes the version (see "Framework Bundle Structure" (page 13)). For example, the letter "A" in the path below indicates the major version of a hypothetical framework:

```
/System/Library/Frameworks/Boffo.framework/Versions/A/Boffo
```

When a program is built, the linker records this path in the program executable file. The dynamic link editor uses the path at runtime to find the compatible version of the framework's library. Thus the major versioning scheme enables backward compatibility of a framework by including all major versions and recording the major version for each executable to run against.

When to Use Major Versions

You should make a new major version of a framework or dynamic shared library whenever you make changes that might break programs linked to it. The following changes might cause programs to break:

- Removing public interfaces, such as a class, function, method, or structure
- Renaming any public interfaces
- Changing the data layout of a structure
- Adding, changing, or reordering the instance variables of a class
- Adding virtual methods to a C++ class

- Reordering the virtual methods of a C++ class

- Changing C++ compilers or compiler versions

- Changing the signature of a public function or method

Changes to the signature of a function include changing the order, type, or number of parameters. The signature can also change by the addition or removal of `const` labels from the function or its parameters.

When you change the major version of a framework, you typically make the new version the "current" version. Xcode automatically generates a network of symbolic links to point to the current major version of a framework. See "Framework Bundle Structure" (page 13) for details.

Avoiding Major Version Changes

Creating a major version of a framework is something that you should avoid whenever possible. Each new major version of a framework requires more storage space than a comparable minor version change. Adding new major versions is unnecessary in many cases.

Before you find yourself needing to create a new major version of your framework, consider the implementation of your framework carefully. The following list shows ways to incorporate features without requiring a new major version:

- Pad classes and structures with reserved fields. Whenever you add an instance variable to a public class, you must change the major version number because subclasses depend on the size of the superclass. However, you can pad a class and a structure by defining unused ("reserved") instance variables and fields. Then, if you need to add instance variables to the class, you can instead define a whole new class containing the storage you need and have your reserved instance variable point to it.

  Keep in mind that padding the instance variables of frequently instantiated classes or the fields of frequently allocated structures has a cost in memory.

- Don't publish class, function, or method names unless you want your clients to use them. You can freely change private interfaces because you can be sure no programs are using them. Declare any interfaces that may change in a private header.

- Don't delete interfaces. If a method or function no longer has any useful work to perform, leave it in for compatibility purposes. Make sure it returns some reasonable value. Even if you add additional arguments to a method or function, leave the old form around if at all possible.

- Remember that if you add interfaces rather than change or delete them, you don't have to change the major version number because the old interfaces still exist. The exception to this rule is instance variables.

While many of the preceding changes do not require the creation of a major version of your framework, most require changing the minor version information. See "Versioning Guidelines" (page 23) for more information.

Creating a Major Version of a Framework

When you create a major version of a framework, Xcode takes care of most of the implementation details for you. All you need to do is specify the major-version designator. A popular convention for this designator is the letters of the alphabet, with each new version designator "incremented" from the previous one. However, you can use whatever convention is suitable for your needs, for example "2.0" or "Two".

To set the major version information for a framework in Xcode, do the following:

1. Open your project in Xcode 2.4.

2. In the Groups & Files pane, select the target for your framework and open an inspector window.

3. Select the Build tab of the inspector window.

4. In the Packaging settings, set the value of the "Framework Version" setting to the designator for the new major version of your framework, for example, `B`..

You can also make major versions of standalone dynamic shared libraries (that is, libraries not contained within a framework bundle). The major version for a standalone library is encoded in the filename itself, as shown in the following example:

`libMyLib.B.dylib`

To make it easier to change the major version, you can create a symbolic link with the name `libMyLib.dylib` to point to the new major version of your library. Programs that use the library can refer to the symbolic link. When you need to change the major version, you can then update the link to point to a new library.

Minor Versions

Within a major version of a framework, you can also designate the current minor version. Minor versions are also referred to as "compatible versions" because they retain compatibility with the applications linked to the framework. Minor versions don't change the existing public interfaces of the framework. Instead, they add new interfaces or modify the implementation of the framework in ways that provide new behavior without changing the old behavior.

Within any major version of the framework, only one minor version at a time exists. Subsequent minor versions simply overwrite the previous one. This differs from the major version scheme, in which multiple major versions coexist in the framework bundle.

The following sections describe how you designate minor version information in your framework and how the system uses that information to ensure applications can run.

Minor Version Numbering Scheme

Frameworks employ two separate numbers to track minor version information. The **current version** number tracks individual builds of your framework and is mostly for internal use by your team. You can use this number to track a group of changes to your framework and you can increment it as often as seems appropriate. A typical example would be to increment this number each time you build and distribute your framework to your internal development and testing teams.

The **compatibility version** number of your framework is more important because it marks changes to your framework's public interfaces. When you make certain kinds of changes to public interfaces, you should set the compatibility version to match the current version of your framework. Thus, the compatibility version typically lags behind the current version. Only when your framework's public interfaces change do the two numbers match up.

Remember that not all changes to your framework's public interfaces can be matched by incrementing the compatibility version. Some changes may require that you release a new major version of your framework. See "Versioning Guidelines" (page 23) for a summary of the changes you can make for each version.

When to Use Minor Versions

You should update the version information of your framework when you make any of the following changes:

■ Add a class

■ Add methods to an Objective-C class

■ Add non-virtual methods to a C++ class

■ Add public structures

■ Add public functions

■ Fix bugs that do not change your public interfaces

■ Make enhancements that do not change your public interfaces

Any time you change the public interfaces of your framework, you must update its compatibility version number. If your changes are restricted to bug fixes or enhancements that do not affect the framework's public interfaces, you do not need to update the compatibility version number.

Compatibility Version Numbers at Runtime

When a program is linked with a framework during development, the linker records the compatibility version of the development framework in the program's executable file. At runtime, the dynamic link editor compares that version against the compatibility version of the framework installed on the user's system. If the value recorded in the program file is greater than the value in the user's framework, the user's framework is too old and cannot be used.

Cases where a framework is too old are uncommon, but not impossible. For example, it can happen when the user is running a version of Mac OS X older than the version required by the software. When such an incompatibility occurs, the dynamic link editor stops launching the application and reports an error to the console.

To run an application on earlier versions of a framework, you must link the application against the earliest version of the framework it supports. Xcode provides support for linking against earlier versions of the Mac OS X frameworks. This feature lets you link against specific versions of Mac OS X version 10.1 and later. For more information, see *SDK Compatibility Guide*.

Creating a Minor Version of a Framework

If you are developing a framework, you need to increment your current version and compatibility version numbers at appropriate times. You set both of these numbers from within Xcode. The linking options for Xcode framework projects include options to specify the current and compatibility version of your framework. To set these numbers for a specific target, do the following:

1. Open your project in Xcode 2.4.

2. In the Groups & Files pane, select the target for your framework and open an inspector window.

3. Select the Build tab of the inspector window.

4. In the Linking settings, set the value of the "Current Library Version" setting to the current version of your framework.

5. Update the "Compatibility version" option as needed to reflect significant changes to your framework. (See "Compatibility Version Numbers at Runtime" (page 22) for more information.)

> **Note:** In Xcode, the current and compatibility version numbers should generally be associated with the target, not the project.

Versioning Guidelines

Whenever you change the code in your framework, you should also modify the configuration information in your framework project to reflect the changes. For simple changes, such as bug fixes and most new interfaces, you may need to increment only the minor version numbers associated with your framework. For more extensive changes, you must create a new major version of your framework.

Table 1 lists the types of changes you can make to your framework code and the corresponding configuration changes you must make to your project file. Most changes that require a major or minor version update occur because the changes affect public interfaces.

Table 1   Making changes to a framework

Version type	Changes Allowed	What to do
Major version	Remove interfaces (classes, functions, methods, and so on). Rename interfaces. Change the layout of a class or structure. Add, change, or reorder instance variables of a class. Add or reorder virtual methods in a C++ class. Change the signature of a function or method. Changing C++ compilers or compiler versions.	You must change the major version designator for your project. Build your framework and incorporate the new version into your existing framework directory structure.
Minor version (public interface changes)	Add a C++ or Objective-C class. Add methods to an Objective-C class. Add non-virtual methods to a C++ class. Add structures. Add functions.	Increment your current version number and set your compatibility version number to match. Build your framework.
Minor version (no public interface changes)	Fix bugs that do not affect programmatic interfaces. Make enhancements that do not affect your programmatic interfaces. Change interfaces used only internally to the framework.	Increment your current version number. Do not change the compatibility version number.

If you don't change the framework's major version number when you need to, programs linked to it may fail in unpredictable ways. Conversely, if you change the major version number when you didn't need to, you clutter up the system with unnecessary framework versions.

Because major versions can be very disruptive to development, it is best to avoid them if possible. "Avoiding Major Version Changes" (page 20) describes techniques you can use to make major changes without releasing a new major version of your framework.

Frameworks and Binding

Dynamic binding of Mach-O libraries brings a considerable power and flexibility to Mac OS X. Through dynamic binding, frameworks can be updated transparently without requiring applications to relink to them. At runtime, a single copy of the library's code is shared among all the processes using it, thus reducing memory usage and improving system performance.

> **Note:** For a more detailed explanation of binding and dynamic linking, see *Mac OS X ABI Mach-O File Format Reference*.

Dynamic Shared Libraries

The executable code in a framework bundle is a dynamically linked, shared library—or, simply, a dynamic shared library. This is a library whose code can be shared by multiple concurrently running programs.

Dynamic shared libraries bring several benefits. One benefit is that they enable memory to be used more efficiently. Instead of programs retaining a copy of the code in memory, all programs share the same copy. Dynamic shared libraries also make it easier for developers to fix bugs in library code. Because the library is linked dynamically, the new library can be installed without rebuilding programs that rely on it.

Symbol Binding

Dynamic shared libraries have characteristics that set them apart from static linked shared libraries. For static linked shared libraries, the symbols in the library are checked at link time to make sure they exist. If they don't exist, link errors occur. With dynamic shared libraries, the binding of undefined symbols is delayed until the execution of the program. More importantly the dynamic link editor resolves each undefined symbol only when the symbol is referenced by the program. If a symbol is not referenced, it is not bound to the program.

The ability to bind symbols at runtime is made possible by the internal structure of Mach-O dynamic shared libraries. The object-code modules that make up the library are built to retain their individual boundaries; that is, the code from the source modules is not merged into a single module. At runtime, the dynamic link editor automatically loads and links modules only as they are needed. In other words, a module is linked only when a program references a symbol in that module. If the symbols in a particular module are not referenced, the module is not linked.

Figure 1 (page 26) illustrates this "lazy linking" behavior. In this example, module `a.o` is linked in the program's `main` routine when library function `a` is called. Module `b.o` is linked when library function `b` in program function `doThat` is called. Module `c.o` is never linked because its function is never called.

Figure 1   Lazy linking of dynamic shared library modules

MyApp.app

main.c

```
#include <Boffo/Boffo.h>;
#include "main.h";
#include "doThat.h";

int main() {
    a();
    doThat(1);
    return 0;
}
...
```

doThat.c

```
#include <Boffo/Boffo.h>;
#include "doThat.h";

void doThat (int n) {
    b();
    if (!n);
        c();
}
```

Boffo.framework

a.o

```
#include "a.h";

void a() {
    ...
}
```

b.o

```
#include "b.h";

void b() {
    ...
}
```

c.o

```
#include "c.h";

void c() {
    ...
}
```

links

links
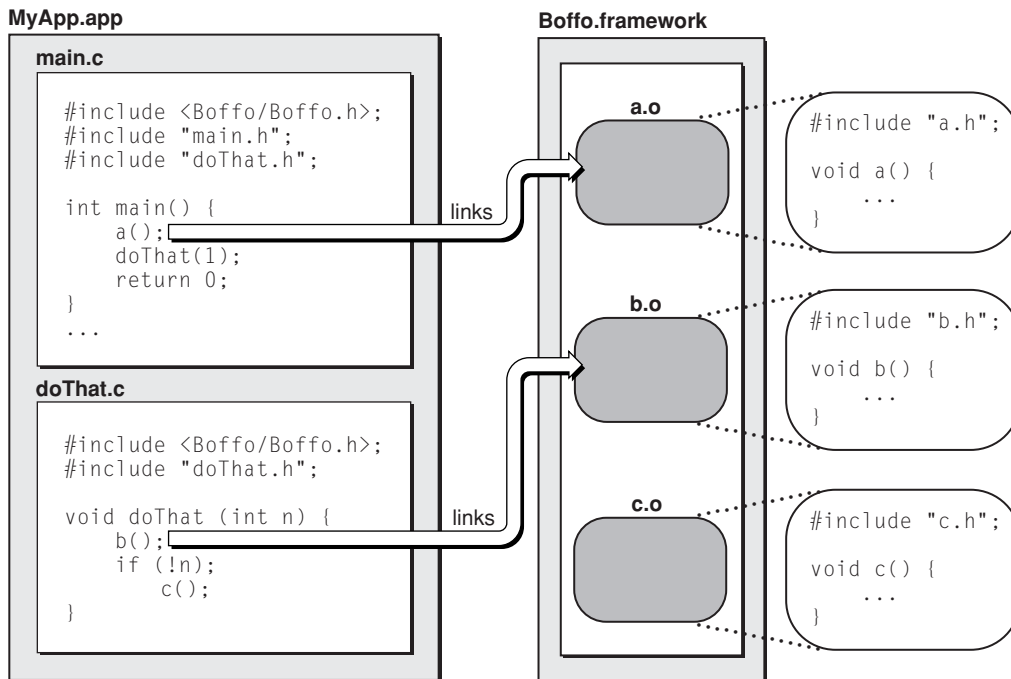
Organizing Your Framework Code

As a framework developer, you should design your dynamic shared library with this as-needed linking of separate modules in mind. Because the dynamic link editor always attempts to bind unresolved symbols within the same module before going on to other modules and other libraries, you should ensure that interdependent code is put in its own module. For example, custom allocation and deallocation routines should go in the same module. This technique prevents the wrong symbol definitions from being used. This problem can occur when definitions of a symbol exist in more than one dynamic shared library and those other symbol definitions override the correct one.

When you create a framework, you must ensure that each symbol is defined only once in a library. In addition, "common" symbols are not allowed in the library; you must use a single true definition and precede all other definitions with the `extern` keyword in C code.

When you build a program, linking it against a dynamic shared library, the installation path of the library is recorded in the program. For the system frameworks supplied by Apple, the path is absolute. For third-party frameworks, the path is relative to the application package that contains the framework. This capture of the library path improves launching performance for the program. Instead of having to search the file system, the dynamic link editor goes directly to the dynamic shared library and links it into the program. This means, obviously, that for a program to run, any required library must be installed where the recorded path indicates it can be found, or it must be installed in one of the standard fallback locations for frameworks and libraries. See "Installing Your Framework" (page 51) for more information.

Library Dependencies

Clients of dynamic shared libraries do not need to be aware of any dependencies required by the library. When a dynamic shared library is built, the static linker stores information about any dependent libraries inside the dynamic shared library executable. At runtime, the dynamic link editor reads this information and uses it to load the dependent libraries as needed.

Another important piece of information stored for each dependent library is the required version. Frameworks and dynamic shared libraries have version information associated with them. At runtime, the stored version information is compared against the actual version of the available library. If the available library is too old, the dynamic link editor may terminate the program to prevent undesirable behavior. For more information on library versioning, see "Framework Versions" (page 19).

Standalone Dynamic Shared Libraries

In addition to creating frameworks, you can create standalone dynamic shared libraries. By convention, stand-alone dynamic shared libraries have a `.dylib` extension and are typically installed in `/usr/lib`. The library file contains all the code and resources needed by the library.

Creating standalone dynamic shared libraries is an uncommon approach for most developers. In most cases, frameworks are a preferred approach. The bundle structure of frameworks makes it possible to include complex resource types such as nib files, images, and localized strings.

Frameworks and Prebinding

Prior to Mac OS X v10.3.4, Mac OS X used a feature called prebinding to eliminate the load-time delays incurred by executables linked to dynamic libraries. Prebinding involved the precalculation of symbol addresses in each framework and library on the system. The goal of this precalculation was to avoid address-space conflicts among the libraries and frameworks. Such conflicts incurred tremendous performance penalties at load-time and would noticeably slow down the launch time of an application.

Improvements to the dynamic loader in Mac OS X v10.3.4 made prebinding largely unnecessary. The dynamic loader itself was modified to handle load-time conflicts much more efficiently. Using the new dynamic loader, an application that is not prebound now usually launches at least as fast (and sometimes faster) than it did on earlier versions of the system when it was prebound.

In Mac OS X v10.4, another change was introduced to the prebinding behavior to reduce the amount of time spent "optimizing" the system after installing new software. Instead of prebinding all frameworks and libraries, now only select system frameworks are prebound. By selectively choosing which frameworks are prebound, the prebinding tools are able to tightly pack the system's most frequently-used frameworks into a smaller memory space than before. This step reduces the amount of space reserved for Apple frameworks and gives it back to third-party applications and frameworks.

If you are developing frameworks to run on versions of Mac OS X prior to 10.4, you should still enable prebinding and specify a preferred address. If you are developing frameworks for Mac OS X v10.4 or later, prebinding is not required. Prebinding your framework on later versions of the system does not decrease performance, but does require some additional configuration steps, which are described in the sections that follow.

> **Note:** Prebinding is supported only for Mach-O executables in versions of Mac OS X.

Prebinding Your Framework

If you are developing a framework that runs on Mac OS X 10.4 or earlier, you should specify your framework's preferred binding address in your Xcode project.

The following steps show you how to configure prebinding for an Xcode framework project:

1. Open your project in Xcode.

2. In the Groups & Files pane, select your target, open its Info window, and click Build.

3. Make sure the Prebinding build setting is turned on (you can enter `prebinding` in the search field to locate it).

4. To the Other Linker Flags build setting, add the `-seg1addr` flag along with the preferred address for your framework. For example, to set the preferred address of your framework to `0xb0000000`, you would enter:

   `-seg1addr 0xb0000000`

5. Build and link your framework as usual.

When prebinding frameworks, it is especially important to specify a preferred address using the `-seg1addr` option. If you enable prebinding but do not specify a preferred address, Xcode uses the default address `0x00000000`. This is a problem because that address is also the preferred address for all applications. Instead, you should set the initial address to a region of memory reserved for use by your application code and frameworks. For a list of valid address ranges, see "Prebinding Your Application" in *Launch Time Performance Guidelines*.

You can confirm the preferred address of your framework by examining the binary using the `otool` command. See "Finding the Preferred Address of a Framework" (page 29) for more information.

Caveats for Prebinding

If you are prebinding your framework so that it can run on versions of Mac OS X prior to 10.4, you should be aware of the following caveats:

- The address range occupied by your framework should not overlap the address ranges of any other libraries or frameworks you are developing. If your frameworks may have to coexist with other third-party libraries or frameworks, you can use `otool` to find the preferred addresses of those third-party products.

- Your framework must not contain references to any undefined symbols.

- Your framework must not override symbols defined in flat namespace libraries. For example, you cannot define your own `malloc` routine and then prebind using flat namespace libraries.

- Two frameworks (or libraries) cannot have circular dependencies.

- Your frameworks should always use two-level namespaces to avoid name collisions with symbols in other frameworks and libraries.

- Keep in mind that for an application to be prebound, all of its dependent frameworks must also be built prebound.

Choosing a unique preferred address for your framework can be tricky, especially if it must coexist with a number of third-party frameworks. Apple provides a set of valid ranges for your frameworks and applications to use. (See "Prebinding Your Application" in *Launch Time Performance Guidelines*.) However, you may still run into areas of overlap with frameworks developed by other groups, either inside or outside your company.

Even if an overlap does occur among frameworks, your prebinding efforts are not in vain. The dynamic linker corrects overlaps immediately at runtime, moving frameworks around as needed. In versions of Mac OS X prior to 10.4, a daemon also runs in the background to recalculate prebinding information for applications where that information is out-of-date.

Finding the Preferred Address of a Framework

To find the preferred address of a framework, use the `otool` command with the `-l` option to display the load commands for the framework's binary file. The load commands include the virtual memory address at which to load each segment of the binary. Because most segments reside at an offset from the beginning of the library, you need to look at the initial `LC_SEGMENT` command to find the library's preferred base address.

For example, suppose you create a library and assign it the preferred address `0xb0000000` in your Xcode project. Running `otool -l` on your library from a Terminal window would display an initial load command similar to the following:

```
Load command 0
      cmd LC_SEGMENT
  cmdsize 328
  segname __TEXT
   vmaddr 0xb0000000
   vmsize 0x00002000
  fileoff 0
 filesize 8192
  maxprot 0x00000007
 initprot 0x00000005
   nsects 4
    flags 0x0
```

Notice the value of the `vmaddr` field. This field indicates that the preferred address of the binary matches the address you specified in your Xcode project.

Apple Frameworks and Prebinding

In versions of Mac OS X prior to 10.4, Apple-provided frameworks are shipped prebound and are assigned to reserved regions of memory. In Mac OS X v10.4 and later, Apple system frameworks are prebound dynamically when you install the operating system. In both cases, the memory ranges reserved by Apple are listed in "Prebinding Your Application" in *Launch Time Performance Guidelines*.

Frameworks and Weak Linking

One challenge faced by developers is that of taking advantage of new features introduced in new versions of Mac OS X while still supporting older versions of the system. Normally, if an application uses a new feature in a framework, it is unable to run on earlier versions of the framework that do not support that feature. Such applications would either fail to launch or crash when an attempt to use the feature was made. Apple has solved this problem by adding support for weakly-linked symbols.

When a symbol in a framework is defined as weakly linked, the symbol does not have to be present at runtime for a process to continue running. The static linker identifies a weakly linked symbol as such in any code module that references the symbol. The dynamic linker uses this same information at runtime to determine whether a process can continue running. If a weakly linked symbol is not present in the framework, the code module can continue to run as long as it does not reference the symbol. However, if the symbol is present, the code can use it normally.

If you are updating your own frameworks, you should consider making new symbols weakly linked. Doing so can make it easier for clients of your framework to support it. You should also make sure that your own code checks for the existence of weakly-linked symbols before using them.

> **Note:** Although the Code Fragment Manager supports its own form of weak linking, the information that follows pertains solely to Mach-O executables.

For more information regarding weak linking, including additional examples, see *SDK Compatibility Guide*.

Weak Linking and Apple Frameworks

Apple frameworks use the availability macros to determine whether a symbol is weakly linked or strongly linked. Apple wraps new interfaces in its frameworks with availability macros to indicate which version of the operating system a feature first appeared. Macros are also used to indicate deprecated features and interfaces.

The availability macros defined in `/usr/include/AvailabilityMacros.h` add weak linking information to system interfaces based on the versions of Mac OS X your project supports. When you create a new project, you tell the compiler which versions of Mac OS X your project supports by setting the deployment target and target SDK in Xcode. The compiler uses these settings to assign appropriate values to the `MAC_OS_X_VERSION_MIN_REQUIRED` and `MAC_OS_X_VERSION_MAX_ALLOWED` macros, respectively. For information on how to modify these settings in Xcode, see "Setting Up Cross-Development in Xcode" in *SDK Compatibility Guide* or the Xcode help.

For example, suppose in Xcode you set the deployment target (minimum required version) to Mac OS X 10.2 and the target SDK (maximum allowed version) to Mac OS X 10.3. During compilation, the compiler would weakly link any interfaces that were introduced in Mac OS X version 10.3 while strongly linking earlier interfaces. This would allow your application to continue running on Mac OS X version 10.2 but still take advantage of newer features when they are available.

> **Important:** The deployment target setting in Xcode must be set to Mac OS X version 10.2 or later to take advantage of weak linking. If you do not set this value (or set it to an earlier version of Mac OS X), you cannot use weak linking in your project.

Before using any symbols that are introduced in a version of Mac OS X that is later than your minimum required version, make sure you check to see that the symbol exists first. See "Using Weakly Linked Symbols" (page 32) for more information.

Marking Symbols for Weak Linking

If you define your own frameworks, you can mark symbols as weakly linked using the `weak_import` attribute. Weak linking is especially appropriate if you introduce new features to an existing framework. To mark symbols as weakly linked, you must make sure your environment is configured to support weak linking:

- You must be using GCC version 3.1 or later. Weak linking is not supported in GCC version 2.95
- You must set the Mac OS X Deployment Target build option of your Xcode project to Mac OS X 10.2 or later.

The linker marks symbols as strongly linked unless you explicitly tell it otherwise. To mark a function or variable as weakly linked, add the `weak_import` attribute to the function prototype or variable declaration, as shown in the following example:

```
extern int MyFunction() __attribute__((weak_import));
extern int MyVariable __attribute__((weak_import));
```

Using Weakly Linked Symbols

If your framework relies on weakly linked symbols in any Apple or third-party frameworks, you must check for the existence of those symbols before using them. If you attempt to use a non-existent symbol without first checking, the dynamic linker may generate a runtime binding error and terminate the corresponding process.

If a weakly linked symbol is not available in a framework, the linker sets the address of the symbol to NULL. You can check this address in your code using code similar to the following:

```
extern int MyWeakLinkedFunction() __attribute__((weak_import));

int main()
{
    int result = 0;

    if (MyWeakLinkedFunction != NULL)
    {
        result = MyWeakLinkedFunction();
    }

    return result;
}
```

> **Note:** 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.

Weak Linking to Entire Frameworks

When you reference symbols in another framework, most of those symbols are linked strongly to your code. In order to create a weak link to a symbol, the framework containing the symbol must explicitly add the `weak_import` attribute to it (see "Marking Symbols for Weak Linking" (page 32)). However, if you do not maintain a framework and need to link its symbols weakly, you can explicitly tell the compiler to mark all symbols as weakly linked. To do this, you must open your project in Xcode and modify the way your targets link to the framework as follows:

1. Select the target you want to modify and reveal its build phases.

2. Expand the Link Binary With Libraries build phase to view the frameworks currently linked by the target.

3. If the framework you want to weakly link to is listed in the Link Binary With Libraries build phase, select it, and choose Edit > Delete to remove it.

    Now you can tell the linker to use weak linking for that framework.

4. Select the target, open its Info window, and click Build.

5. To the Other Linker Flags build setting, add the following command-line option specification, where `<framework_name>` is the name of the framework you want to weakly link to:

    `-weak_framework <framework_name>`

6. Build your product.

The `-weak_framework` option tells the linker to weakly link all symbols in the named framework. If you need to link to a library instead of a framework, you can use the `-weak_library` linker command instead of `-weak_framework`.

Guidelines for Creating Frameworks

The following sections provide guidance on how best to implement custom frameworks.

API Naming Guidelines

Elements in the global namespace, such as classes, functions, and global variables should all have unique names. While two-level namespaces help the dynamic-link editor find the correct symbols at runtime, the feature does not prevent static link errors from occurring because of multiply-defined symbols. For example, suppose two different frameworks define a symbol with the same name. If you were to create a project that included both frameworks, you would encounter static linking errors if you referenced the symbol in question. The static linker is responsible for selecting which framework to use and to then generate the two-level namespace hint needed by the dynamic link editor. Because both frameworks define the symbol, the static linker cannot choose and generates an error.

You should try to choose names that clearly associate each symbol with your framework. For example, consider adding a short prefix to all external symbol names. Prefixes help differentiate the symbols in your framework from those in other frameworks and libraries. They also make it clear to other developers which framework is being used. Typical prefixes include the first couple of letters or an acronym of your framework name. For example, functions in the Core Graphics framework use the prefix "CG".

If you are writing a category for an Objective-C class, you should also use some sort of unique prefix in your method names. Because categories can be loaded dynamically from multiple sources, a unique prefix helps ensure that the methods in your category don't conflict with those in other categories.

For detailed guidelines on Cocoa naming conventions, see *Coding Guidelines for Cocoa*.

Performance Impacts of Frameworks

Before you actually create a custom framework, you should carefully consider its intended use. There is a certain amount of overhead associated with loading and using frameworks at runtime. While this overhead is not high—especially if your framework and application are prebound—it may be unnecessary in some situations.

Frameworks are most effective when their code is shared by multiple applications. If you are designing a suite of applications, you may create a custom framework to store common code accessed by all applications in the suite. Similarly, you may want to create a private framework internally to separate out generic, reusable code from your application-specific code. Although you may embed this framework in each application you create, you simplify your maintenance of the reusable code.

What to Include in Your Framework

Frameworks provide shared resources for multiple applications. If you are defining a custom framework, you should keep that goal in mind. As much as possible, your framework should not contain code that is tied to a particular application. Instead, it should contain code that is reusable or that is common to multiple applications.

In addition to code, you can include other types of resources in your frameworks. In the `Resources` directory of your framework, you can include nib files, images, sound files, localized text, and any other type of resource that you might find in an application. At runtime, applications load your framework's resources using the same bundle mechanism they use for their application-specific resources.

Using C++ in Framework Code

There are problems inherent with exporting C++ class interfaces from dynamic shared libraries. These problems are the result of a lack of standardized calling conventions for the extensions that differentiate the C++ language from C.

In order to eliminate linking problems between libraries built with different compilers, compiler vendors agreed to support a set standardized calling conventions for the C language. These standards required the vendors to generate C code in a consistent way that guaranteed the code from one library could call code from another library. While such a standard has been proposed for the C++ language extensions, it has yet to be finalized. As a result, there is no guarantee that calls between two different C++ libraries would work.

If you are intent on using C++ code in your frameworks, the best way to avoid incompatibility issues is to wrap the interface for your C++ classes with ANSI C functions. By using C++ for your implementation and ANSI C for your interface, you can continue to develop in your preferred language while providing a compatible interface for clients to call.

If you need to export class interfaces from your framework, Apple recommends you use Objective-C instead of C++ to define your classes. The Objective-C language does have a standardized set of calling conventions that enable you to expose class interfaces from your frameworks.

For additional information and guidance about using C++ in shared libraries, see *C++ Runtime Environment Programming Guide*.

Don't Create Umbrella Frameworks

While it is possible to create umbrella frameworks using Xcode, doing so is unnecessary for most developers and is not recommended. Apple uses umbrella frameworks to mask some of the interdependencies between libraries in the operating system. In nearly all cases, you should be able to include your code in a single, standard framework bundle. Alternatively, if your code was sufficiently modular, you could create multiple frameworks, but in that case, the dependencies between modules would be minimal or nonexistent and should not warrant the creation of an umbrella for them.

Where to Install Your Framework

Mac OS X looks for public frameworks in several fixed locations on the system. If you are creating a framework for other developers to use, you should put it in one of these locations. If you are creating a private framework, you can either embed it inside an application or put it in a custom location. In either case, you must do some extra work to load your framework code and resources.

For details on where to install frameworks, see "Installing Your Framework" (page 51).

Creating a Framework

Once you decide that you need to create a framework for your code, you can do so easily with Xcode. As you add major versions to the framework, you also need to be able to maintain your projects. The following sections show you how perform both of these tasks.

Creating Your Framework

From Xcode, choose File > New Project to create your project. Follow the prompts to select the type of framework you want and where you want to put your project directory.

The default templates that come with Xcode let you specify whether you want to create a Carbon or Cocoa framework. The type of framework you choose determines which default files are generated for you. If you do not want to include Carbon or Cocoa headers in your framework, you can remove any references to them after you create your project.

Configuring Your Framework Project

When you create a new framework, there are several configuration options you may want to modify. These options make it easier to distribute your framework to customers and guarantee its compatibility after future development cycles. Table 1 lists some of the options you should set for your framework.

Table 1   Framework configuration options

Option	Description
Framework identifier	A Java-style package identifier that uniquely identifies the framework to the system. You should always set this option. To set this option in Xcode 2.4, open an inspector window for your framework target, select the Properties tab, and modify the Identifier field.
Framework version	The current major revision of the framework. See "Major Versions" (page 19) for more information. In Xcode 2.4, set this value for your framework target using the Framework Version build setting.
Current version	The current revision of the framework. In Xcode 2.4, set this value for your framework target using the Current Library Version build setting. See "Minor Versions" (page 21) for more information.
Compatibility version	The most recent revision of the framework that includes changes to the public interfaces. In Xcode 2.4, set this value for your framework target using the Compatibility Version build setting. See "Minor Versions" (page 21) for more information.

Option	Description
Exported symbols	The list of framework symbols you want to export to other programs. In Xcode 2.4, specify a file containing your exported symbols for your framework target using the Exported Symbols File build settings. To specify a file containing the symbols to hide, use the Unexported Symbols File build setting instead. See "Exporting Your Framework Interface" (page 49) for more information.
Installation path	The directory name in which your framework should ultimately be installed. In Xcode 2.4, set this value for your framework target using the Installation Directory build setting. See "Installing Your Framework" (page 51) for a list of standard locations.
Preferred address	For frameworks being deployed in Mac OS X 10.3.9 and earlier, specify the preferred memory address to use for prebinding operations. This value is not needed when deploying a framework in 10.4 and later. See "Frameworks and Prebinding" (page 27) for information on how to set the preferred address of a framework.

Testing Your Framework in Place

When you build a framework, Xcode places it in the `build` subdirectory of your project directory by default. Although you can tell Xcode to install your framework in its final deployment location, during development you may want to leave it where it is. If you do, you may need to tell test applications where to find your framework.

If your framework project contains additional targets for test applications, then Xcode builds those applications in the same folder as your framework. Test applications built alongside your framework find that framework automatically because of their proximity to it. However, if you build your test applications into a different build directory, those applications may be unable to find your framework unless you tell them where to find it.

The usual way for an application to find a framework is to look in the standard locations (see "Installing Your Framework" (page 51)). However, you may not want to reinstall your framework every time you make changes to it. In that case, you can tell your test applications exactly where to find the framework using the `DYLD_FRAMEWORK_PATH` environment variable. Adding this variable to your executable tells `dyld` where to look for additional frameworks if it doesn't find what it needs in the standard locations. The following steps show you how to set this variable in Xcode.

1. Open your application project in Xcode.

2. In the Groups & Files pane, open the Executables group, select the executable to configure, open its Info window, and click Arguments.

3. Add an entry to the environment variables list.

4. Set the name of the environment variable to `DYLD_FRAMEWORK_PATH`.

5. Set the value of the variable to the full pathname of the directory containing your framework.

   To specify multiple framework directories, separate the pathnames with a colon. For example, you could have a value such as the following value on this line:
   `/Users/lynn/MyFrameworks:/Volumes/Keylime/MyOtherFrameworks`.

Embedding a Private Framework in Your Application Bundle

If you need to distribute a private framework with an application, the preferred solution is to embed the framework in your application bundle. Embedding a framework inextricably links the framework to the application and ensures that the application always has the correct version of the framework needed to operate. Embedding the framework also makes it clear to other developers that they should not ever link to that framework.

> **Note:** If multiple applications must share a private framework, you should install the framework in one of the available `PrivateFrameworks` directories on the system rather than embed it in one (or all) of the applications. Sharing one framework allows for more efficient reuse of the framework's dynamic library. For information about where to put shared private frameworks, see "Locations for Private Frameworks" (page 52).

To embed a framework in an application, there are several steps you must take:

- You must configure the build phases of your application target to put the framework in the correct location.
- You must configure the framework target's installation directory, which tells the framework where it will live.
- You must configure the application target so that it references the framework in its installation directory.

It is possible to build and embed a framework in an application using a single Xcode project or multiple projects. Using a single Xcode project is somewhat easier because it requires less configuration to get both the framework and application to build. For multi-project setups, however, once the two projects are configured to build properly, the configuration steps for embedding the framework are essentially the same as those for a single Xcode project.

Using a Single Xcode Project For Both Targets

Using a single Xcode project for both your application and framework target simplifies the required setup. Once you create your project, you simply add two targets to it: one for your application and one for your framework. (Because both targets reside in the same project, there are no problems finding source files from either target at build time.) After that, you simply configure your framework and application targets with the proper runtime information for embedding.

The configuration for your framework target involves telling it where it will be installed. The framework needs this information so that it can find the resources it needs. Because frameworks are typically installed in fixed locations, you normally specify the full path to the appropriate frameworks directory. When you embed a framework inside a bundle, however, the location of the framework is not fixed, so you have to use the `@executable_path` placeholder to let the framework know its location is relative to the current executable.

1. Open an inspector for your framework target and select the Build tab.

2. Set the value of the Installation Directory build setting to `@executable_path/../Frameworks`.

At build time, Xcode builds your framework and puts the results in the build directory. Before the application can use the framework, however, you must configure the application target as follows:

- You need to copy the framework into the application's bundle.

- You need to link the application against the framework.

- You need to create a build dependency between the framework and application.

The following steps show you how to configure your application target.

1. In the Group & Files pane, open your application target to view its current build phases.

2. Drag your framework product (located in the Products folder) to the existing Link Binary With Libraries build phase of your application target. This causes the application to link against your framework.

3. Add a new Copy Files Build Phase to the application target. (This phase will be used to install the framework in the application bundle.)

4. Select the new build phase and open an inspector window.

5. In the General tab of the inspector window, set the destination for the build phase to "Frameworks".

6. Drag your framework product to the new build phase.

7. Select the application target again and open the inspector window.

8. In the General tab of the inspector window, add your framework as a dependency for the application. Adding this dependency causes Xcode to build the framework target before building the application target.

The build dependency you establish in the application target causes the framework to be built before the application. This is important because it guarantees that a built version of your framework will be available to link against and to embed in the application. Because of this dependency, you can set the active target of your Xcode project to your application and leave it there. Building the application now builds the framework and copies it to the application bundle directory, creating the necessary linkage between the two.

Using Separate Xcode Projects For Each Target

If you already have separate Xcode projects for your framework and application, you can take advantage of Xcode's cross-project references to embed the framework in your application. Cross-project references are a convenient way to create relationships between two separate Xcode projects. To set up a cross-project reference between your application and framework, you would do the following:

1. In your application project, choose Project > Add to Project and select your framework's `.xcodeproj` file. Xcode adds the framework project and displays its products in the Groups & Files pane of your application project.

2. Modify the Build Products Path setting for both the application and framework targets so that they use the same build directory. You need to modify each target in their original Xcode project file.

3. In your application project, modify the Header Search Paths setting of the application target by adding the directories containing any framework header files.

Once you have configured your Xcode projects to build properly, you can proceed with the configuration steps needed to embed the framework in your application. The remaining configuration steps for the framework and application targets are identical to the ones described in "Using a Single Xcode Project For Both Targets" (page 41). Your framework's installation directory must be configured to be relative to the executable path of the application. SImilarly, the application target must copy the framework to its bundle and set up the necessary linkage and dependencies. The only difference is that you must configure each target in its own Xcode project.

Building Multiple Versions of a Framework

After the release of a framework, you should consider how to manage your Xcode project for future releases. When you update an existing framework, the type of changes you make determines the best way to proceed with your project files. For example, major changes may warrant the copying of your project files and the maintenance of separate projects, one for each major version. On the other hand, minor changes can be folded into your existing Xcode project.

Updating the Minor Version

If you are making minor changes to your framework, there is no need to create a new Xcode project for your framework. However, you should always update the "current version" and "compatibility version" values associated with your framework. These values make it possible for the dynamic linker to determine if linking a program to your framework is possible.

For more information on how to update the minor version information of your framework and on the types of changes that constitute a minor version update, see "Minor Versions" (page 21).

Updating the Major Version

The process for updating the major version of a framework is more difficult than the process for minor versions. The recommended way to create a new major version is to make a duplicate version of your entire Xcode project folder and continue developing from there. The old project files should be archived and used to perform legacy builds. However, active development should continue with the new project.

Once you have a new project folder, you need to make several modifications to the Xcode project to identify the project as a major version. Select your framework target and open the Inspector window. In the Build pane, modify the following build options:

■ Increment the value of the Framework Version build setting to the next sequential value.

■ Increment the value of the Current Library Version build setting.

■ Update the value of the Compatibility Version build setting to match the updated Current Library Version.

■ Update any build settings whose path information includes the framework major version designator. For example, if you have a Copy Files build phase based on the product directory, you may need to update that path. Or make sure the paths are specified using the framework's `Current` symbolic link.

■ In the Properties pane of the framework target Info window, update the version number that gets stored in your framework's information property list.

Once you make the changes to your code, you can build the framework target. What you get is a new framework bundle containing only the new major version.

During installation, if a version of your framework does not already exist on the target system, have the installer copy your framework bundle over as is. However, if an existing version of the framework is present, have the installer copy the contents of your new framework directory to the old directory. Your installer script must replace symbolic links in the old framework bundle with ones that point to the new version of the framework. However, copying over the new major version should leave any old versions intact. This permits existing applications to continue running while newer versions use the updated framework.

> ⚠️ **Warning:** Do not attempt to build a new version of your framework over an existing framework bundle. Performing a clean build of your project deletes the entire contents of the bundle, including any legacy versions. It is much safer to maintain separate projects and copy the files over during installation.

You may also want the installer to remove any header files or documentation for outdated versions of your framework. This step is optional and is left to your discretion. However, it is recommended to prevent developers from accidentally including an outdated set of header files, or viewing older documentation, during development.

For more information on major versions of frameworks, see "Major Versions" (page 19).

Initializing a Framework at Runtime

When a framework is first loaded by a unique process, the system calls the framework's initialization code. Prior to Mac OS X v10.4, framework initialization typically consisted of a single routine that was called when the framework was first loaded; however, in Mac OS X v10.4 and later, the use of module initializers and finalizers became the preferred technique. The main advantage of module initializers is that they can be called after the dynamic linker has a chance to load any other libraries on which the module initializer depends. The same is not true of framework initialization routines, which are called immediately on load and may occur before other important modules (like the C++ runtime) are loaded.

Initialization Routines and Performance

Because all types of initialization routines are called at framework load-time, they are often called while an application is launching. Launch time is generally a bad time to perform large amounts of work because it can make the corresponding application feel sluggish. When writing your initialization routines, try to do as little work as possible to put your framework in a known state. For example, instead of allocating your framework data structures immediately, consider lazily initializing your data structures as they are needed. Also, avoid performing any operations that might cause a potential delay, such as accessing the network.

Remember that if your framework contains static data, that data must be initialized at load-time as well. Minimizing the number of static variables your framework uses can also help reduce initialization performance. For more information about improving the launch time of applications, see *Launch Time Performance Guidelines*.

Defining Module Initializers and Finalizers

Module initializers are the preferred way to initialize a framework. A module initializer is a static function that takes no arguments and returns no value. It is declared using the `constructor` compiler attribute as shown in Listing 1. As with any static initialization function, you should guard against the function being called twice by placing a guard variable around your initialization code.

Listing 1   Module initializer for a framework

```
__attribute__((constructor))
static void MyModuleInitializer()
{
    static initialized = 0;
    if (!initialized)
    {
        // Initialization code.
        initialized = 1;
    }
}
```

Frameworks can define multiple module initializer functions. The dynamic link editor calls any module initializer functions after initializing any static variables but before calling any other functions or methods in your framework. If the code in a module initializer function relies on other libraries, such as the C++ runtime, the dynamic linker loads those libraries prior to calling the function. Each module initializer function is called only once when the framework is loaded by a process. Module initializers are executed in the order they are encountered by the compiler.

The symbols for any module initializer functions must not be exported by your framework. By default, Xcode exports all symbols declared in your project's header files. You can restrict the set of exported symbols by explicitly exporting a list of symbols or by hiding specific symbols (and exporting everything else). For information on how to configure your framework's exports, see "Exporting Your Framework Interface" (page 49).

In Mac OS X v10.4 and later, module initializers can access the launch arguments for the current process, as shown in Listing 2. A framework might use these arguments to get information about the launch configuration of the application, such as any environment variables or flags passed in on the launch line.

Listing 2   Module initializer with launch arguments

```
__attribute__((constructor))
static void initializer(int argc, char** argv, char** envp)
{
    // Initialization code.
}
```

In addition to module initializer functions, you can also define module finalizer functions, which implement any clean up code for your framework. Module finalizers are declared using the `destructor` compiler attribute as shown in Listing 3. Like module initializers, the symbols for module finalizers must not be exported by your framework. Module finalizers execute in the reverse order that they are encountered by the compiler.

Listing 3   Module finalizer function

```
__attribute__((destructor))
static void finalizer()
{
    // Clean up code.
}
```

Using Initialization Routines

If your framework must run in versions of Mac OS X prior to 10.4, you can still use a framework initialization function as needed to initialize your framework data structures. When implementing your initialization routine, however, it is important to do as little work as possible. Because your initialization routine runs immediately at load time, other code modules may not be loaded and available for use, so it is important that you do not perform any complex initialization involving other libraries.

The signature of an initialization function is the same as that for a standard module initializer:

```
void MyFrameworkInit()
```

To load your routine, you must pass the name of your routine to the linker using the `INIT_ROUTINE` flag. From the command line you set this flag using the `-init` option, followed by the name of your initialization routine. In Xcode, you set this flag by doing the following:

1. Select your framework target and open an inspector window.

2. In the Build pane, find the Initialization Routine build setting. It is with the Linking options.

3. Set the value of this setting to the symbol name of your initialization routine.

For ANSI C–based routines, the symbol name is simply the function name with a leading underscore character. For example, the symbol name for the `MyFrameworkInit` function is `_MyFrameworkInit`. You should not use C++ routines for initialization functions.

If you have trouble building your framework with the specified initialization routine, there are a few things you can try to fix the problem. Use the `nm` command-line tool to confirm the symbol name of your routine. You can also use `nm` with the `-g` option to make sure that a global symbol for your routine is exported by the library. If it isn't, check to see if you have an exports file and make sure your routine is included in it.

> **Important:** In Mac OS X v10.3.9 and later, it is especially important not to use C++ code from a framework initialization function. Prior to 10.3.9, the C++ runtime was packaged as a static library and linked into the application executable. In Mac OS X v10.3.9 and later, the library is packaged as a dynamic shared library and loaded only as needed, which might be after your framework is loaded. Framework initialization functions that try to use C++ features may subsequently fail or cause the application to terminate.

Exporting Your Framework Interface

When you build a framework or application using Xcode, the linker exports all of the symbols defined in your code by default. For a shipping framework with many symbols, this can lead to performance problems at runtime. When a framework is loaded, the dynamic link editor loads the symbols associated with the framework. If a framework contains a number of private functions, the symbols for those private functions are not going to be used but are still loaded along with symbols for the public functions. Loading these extra symbols not only wastes memory, it also requires more work to walk the list during a symbol lookup.

In Xcode, you can limit the symbols exported by your executable by specifying an exports file in your linker options.

Creating Your Exports File

An exports file is a simple text file (`.txt` or other text file extension) that contains the list of symbols you want to export. To create the file, add a new empty file to your Xcode project. To this file, add the list of symbols you want to export, one symbol per line.

For ANSI C-based code, you can usually just prefix an underscore character to the name of a function or variable to get the symbol name. For languages like C++, which uses mangled symbol names, you may need to run the `nm` tool to get the list of existing symbol names. Run `nm` with the `-g` option to see the currently exported symbols. You can then copy the output from the tool and paste it into your exports file, removing any extraneous information. The following text shows some sample output for a Cocoa framework generated by `nm`:

```
 U .objc_class_name_NSDate
b000ad54 T _InitCocoaFW
b000aea8 T _addNumbers
b000ade8 T _getInitDate
         U _objc_msgSend
```

To export the framework functions specified in this output, you would create a text file with this text:

```
_InitCocoaFW
_addNumbers
_getInitDate
```

You can temporarily remove a symbol from your exports file by putting a pound sign at the beginning of the line containing the symbol. For example, the following text temporarily removes the `_getInitDate` function from the exports list:

```
_InitCocoaFW
_addNumbers
#_getInitDate
```

Specifying Your Exports File

To specify an exports file for your framework in Xcode, do the following:

1. Open your project in Xcode.

2. Add your exports file to the project, and place it in the Resources group.

3. Open the framework target's Info window and click Build.

4. Set the Exported Symbols File build setting to the name of your exports file.

   You can locate this build setting by choosing All from the Collections pop-up menu and entering its name in the search field.

If you want to export all symbols except for a restricted subset, you can use the Unexported Symbol Files build setting to do so. Create your symbols file as before, but this time include the list of symbols you do not want to export. In the Linking build settings for the target, find the Unexported Symbol Files setting and set its value to the name of your file.

If the Unexported Symbol Files build setting is not present, as it might not be on versions of Xcode prior to v2.2, you can use the "Other linker flags" build setting instead. To hide a set of symbols, set the value of that build setting to the following text, replacing *exports_filename* with the name of your exports file:

`-unexported_symbols_list`*exports_filename*

Installing Your Framework

Once your framework is ready to go, you need to decide where to install it. Where you install a framework also helps determine how to install the framework.

Locations for Public Frameworks

Third-party frameworks can go in a number of different file-system locations, depending on certain factors.

■   Most public frameworks should be installed at the local level in `/Library/Frameworks`.

■   If your framework should only be used by a single user, you can install it in the `~/Library/Frameworks` subdirectory of the current user; however, this option should be avoided if possible.

■   If they are to be used across a local area network, they can be installed in `/Network/Library/Frameworks`; however, this option should be avoided if possible.

For nearly all cases, installing your frameworks in `/Library/Frameworks` is the best choice. Frameworks in this location are discovered automatically by the compiler at compile time and the dynamic linker at runtime. Applications that link to frameworks in other directories, such as `~/Library/Frameworks` or `/Network/Library/Frameworks`, must specify the exact path to the framework at build time so that the dynamic linker can find it. If the path changes (as it might for a user home directory), the dynamic linker may be unable to find the framework.

Another reason to avoid installing frameworks in `~/Library/Frameworks` or `/Network/Library/Frameworks` is the potential performance penalties. Frameworks installed in network directories or in network-based user home directories can cause significant delays in compilation times. Loading the framework across the network can also slow down the launch of the target application.

Third-party frameworks should never be installed in the `/System/Library/Frameworks` directory. Access to this directory is restricted and is reserved for Apple-provided frameworks only.

When you build an application or other executable, the compiler looks for frameworks in `/System/Library/Frameworks` as well as any other location specified to the compiler. The compiler writes path information for any required frameworks in the executable file itself, along with version information for each framework. When the application is run, the dynamic link editor tries to find a suitable version of each framework using the paths in the executable file. If it cannot find a suitable framework in the specified location (perhaps because it was moved or deleted), it looks for frameworks in the following locations, in this order:

1.   The explicit path to the framework that was specified at link time.

2.   The `/Library/Frameworks` directory.

3.   The `/System/Library/Frameworks` directory.

If the dynamic link editor cannot locate a required framework, it generates a link edit error, which terminates the application.

Locations for Private Frameworks

Custom frameworks intended for internal use should be installed inside the application that uses them. Frameworks embedded in an application are stored in the `Frameworks` directory of the application bundle. The advantage of embedding frameworks in this manner is that it guarantees the application always has the correct version of the framework to run against. See "Embedding a Private Framework in Your Application Bundle" (page 41) for information on how to embed a custom framework in your application.

The limitation of embedding frameworks is that you cannot share the framework among a suite of applications. If your company develops a suite of applications that rely on the same framework, you might want to install one copy of that framework that all of the applications can share. In such a situation, you should install the frameworks in the `/Library/Frameworks` directory and make sure the frameworks bundle does not contain any public header information.

Installing Frameworks

How you install frameworks depends on your framework. If your framework is bundled inside of a particular application, there is nothing special you need to do. The user can drag the application bundle to a local system and use the application without any need for additional installation steps.

If your framework is external to an application, you should use an installation package to make sure it is put in the proper location. You should also use an installation package in situations where an older version of your framework might be in place. In that case, you might want to write some scripts to update an existing framework bundle rather than replace it entirely. For example, you may want to install a new major version of your framework without disturbing any other versions. Similarly, if you have multiple applications that rely on the same framework, your installation package should check for the existence of the framework and install it only as needed.

For more information on creating installation packages, see *Software Delivery Guide*.

Including Frameworks

For Mac OS X software developers the guideline for including header files and linking with system software is straightforward: add the framework to your project and include only the top-level header file in your source files. For umbrella frameworks, include only the umbrella header file.

Including Frameworks in Your Project

To include a framework in your Xcode project, choose Project > Add to Project and select the framework directory. Alternatively, you can control-click your project group and choose Add Files > Existing Frameworks from the contextual menu. When you add an existing framework to your project, Xcode asks you to associate it with one or more targets in your project. Once associated, Xcode automatically links the framework against the resulting executable.

> **Note:** If you are not using Xcode to build your project, you must use the `-framework` option of GCC and ld to build and link against the specified framework. See the `gcc` and `ld` man pages for more information.

You include framework header files in your code using the `#include` directive. If you are working in Objective-C, you may use the `#import` directive instead of the `#include` directive. The two directives have the same basic results. but the `#import` directive guarantees that the same header file is never included more than once. There are two ways for including framework headers:

```
#include <Framework_name/Header_filename.h>
#import <Framework_name/Header_filename.h>
```

In both cases, `Framework_name` is the name of the framework and `Header_filename` is the name of a header file in that framework or in one of its subframeworks.

When including framework header files, it is traditional to include only the master framework header file. The master header file is the header file whose name matches the name of the framework. For example, the Address Book framework has a master header file with the name `AddressBook.h`. To include this header in your source, you would use the following directive:

```
#include <AddressBook/AddressBook.h>
#import <AddressBook/AddressBook.h>
```

For most frameworks, you can include header files other than the master header file. You can include any specific header file you want as long as it is available in the framework's `Headers` directory. However, if you are including an umbrella framework, you must include the master header file. Umbrella frameworks do not allow you to include the headers of their constituent subframeworks directly. See "Restrictions on Subframework Linking" (page 55) for more information.

Locating Frameworks in Non-Standard Directories

If your project links to frameworks that are not included in any of the standard locations, you must explicitly specify the location of that framework before Xcode can locate its header files. To specify the location of such a framework, add the directory containing the framework to the "Framework Search Paths" option of your Xcode project. Xcode passes this list of directories to the compiler and linker, which both use the list to search for the framework resources.

> **Note:** The standard locations for frameworks are the `/System/Library/Frameworks` directory and the `/Library/Frameworks` directory on the local system.

Headers and Performance

If you are worried that including a master header file may cause your program to bloat, don't worry. Because Mac OS X interfaces are implemented using frameworks, the code for those interfaces resides in a dynamic shared library and not in your executable. In addition, only the code used by your program is ever loaded into memory at runtime, so your in-memory footprint similarly stays small.

As for including a large number of header files during compilation, once again, don't worry. Xcode provides a precompiled header facility to speed up compile times. By compiling all the framework headers at once, there is no need to recompile the headers unless you add a new framework. In the meantime, you can use any interface from the included frameworks with little or no performance penalty.

Including the Flat Carbon Headers

For Carbon developers porting their source code from Mac OS 9 to Mac OS X, including only the `Carbon.h` header file may require changes to many source files that might be difficult to make right away. For this situation, Apple provides a "flat header" alternative that lets you continue to use your present `#include` commands.

In `/Developer/Headers/FlatCarbon` are stub files for all public Mac OS 9 header files. These stub files redirect the compiler to the appropriate umbrella header file or contain warnings if the API is not valid on Mac OS X. To make use of the stub files, use the compiler's `-I` flag (that is capital "I", not lowercase "l") to include the files in the `FlatCarbon` directory, as shown here:

```
-I/Developer/Headers/FlatCarbon
```

When using this option, make sure that you include both `MacWindows.h` and `MacTypes.h` in your source files.

> **Note:** Apple provides scripts for converting a flat header project to one that uses the new framework headers. These scripts are available in the `/Developer/Headers/FlatHeaderConversion` directory.

Once you are compiling code only for Mac OS X, you should use the native syntax for including framework header files. The book *Carbon Porting Guide* in Carbon Porting Documentation contains a more detailed discussion of the flat-header `#include` technique.

Restrictions on Subframework Linking

Mac OS X includes two mechanisms for ensuring that developers link only with umbrella frameworks. One mechanism is an Xcode feature that prevents you from selecting subframeworks. The other mechanism is a compile-time error that occurs when you attempt to include subframework header files.

In Xcode, the Add Frameworks command displays the available frameworks in `/System/Library/Frameworks`. However, when you select one of these frameworks, the Open dialog displays information about the framework and not a list of subdirectories.

If you try to include a header file that is in a subframework, Xcode generates a compile-time error message. The umbrella header files and the subframework header files contain preprocessor variables and checks to guard against the inclusion of subframework header files. If you compile your project with an improper `#include` statement, the compiler generates an error message.

Document Revision History

This table describes the changes to *Framework Programming Guide*.

Date	Notes
2006-11-07	Updated framework initialization advice. Revised prebinding guidelines. Updated steps for embedding private frameworks in an application bundle.
2006-03-08	Updated for Xcode 2.2 and added naming requirements for versioned frameworks.
	Updated "Framework Bundle Structure" (page 13) to specify the naming requirements for versioned frameworks.
2005-07-07	Updated linking and binding information. Updated steps for Xcode 2.1. Changed title from "Mac OS X Frameworks".
2004-11-02	Updated the list of recommended the guidelines and locations for installing frameworks.
2004-04-15	Added information on how to test a framework in place without installing it in its deployment location.
	Added information relating to weak linking and frameworks.
	Fixed minor bugs.
2003-12-08	First version of *Mac OS X Frameworks*. The information in this document replaces information about frameworks that was previously published in *System Overview*.