Loading Code at Runtime

You may need to use *dynamic shared libraries*, which store reusable code, in your applications to take advantage of functionality used by more than one application. For example, when you develop Cocoa applications, at a minimum your application links against the Foundation and Application Kit frameworks. Through this practice, your program can automatically take advantage of improvements in those frameworks as your application's users update the system software in their computers. See The Products—Types of Mach-O Files You Can Build for more information on dynamic shared libraries.

If you develop shared libraries and distribute them as framework bundles to be used by other developers in their applications, you should ensure changes you make to the libraries (to implement new features, for example) don't break current versions of the applications that use them. You maintain compatibility by exposing the same programming interface to client applications between upgrades of the shared libraries.

When a change to your framework's API is required to implement a feature, you should make available in the same framework bundle the last version of the framework that exposes the API current client applications expect, in addition to the version of the framework that exposes the new, incompatible API. If you follow this guideline, developers of client applications don't have to revise them every time your framework's API changes. And developers that choose to update their applications can take advantage of the features you added to the framework. To ensure that earlier versions of your framework's client applications don't break when the framework is updated, you must package the shared libraries and their resources within the framework bundle in a way that allows earlier versions of client applications to continue using the versions of the framework that they understand.

This article describes how you can load code at runtime. It shows the benefits of using shared libraries and explains how they are packaged inside frameworks to maintain client-application compatibility across updates to the frameworks. It also describes how the OS X runtime environment takes advantage of bundles to allow you to load plug-in code at runtime.

Using Shared Libraries and Frameworks

Programmers often refer to dynamic shared libraries using different names, such as *dynamically linked shared libraries*, *dynamic libraries*, *DLLs*, *dylibs*, or just *shared libraries*. In OS X, all these names refer to the same thing: a library of code dynamically loaded into a process at runtime.

Dynamic shared libraries allow the operating system as a whole to use memory efficiently. Each process in OS X has its own virtual address space. The OS X kernel allows regions of logical memory to be mapped into multiple processes at different addresses. The dynamic linker takes advantage of this feature by mapping the same read-only copy of the shared library code into the address space of each process. The result is that only one physical copy of a shared library is in memory at any time, even though many processes may use it at the same time. Data, such as variables and constants, contained by a shared library is mapped into each client process using the kernel's copy-on-write optimization capability. With copy-on-write, the data is shared among processes until one of the processes attempts to change the data. At that point, the kernel creates a writable copy of the data private to that process. The other processes continue to use the read-only shared copy. Thus, additional memory for data is allocated only when absolutely necessary.

Dynamic shared libraries also provide a way for programs to seamlessly benefit from system upgrades. When the system is upgraded, the shared libraries are updated, but the programs need not be. Since they are dynamically bound to the shared libraries, the programs can continue to call the same functions and the updated implementation of the shared libraries is executed. For more information on dynamic-library development and usage, see *Dynamic Library Programming Topics*.

Maintaining Client Application Compatibility

This is an overview of various parameters that affect compatibility with client applications. You can set these parameters at build time. For a more detailed discussion of this topic, see Dynamic Library Design Guidelines in *Dynamic Library Programming Topics*.

Shared libraries have two version numbers, which allow you to create versions of a shared library that are *binary compatible* (that is, they do not require client programs to be recompiled) with the functions exported by the earlier versions of a library:

- The *current version* of the library specifies the current version number of the library's implementation. A client program can examine this version number to find out the exact version of the library, which can be useful for checking for bug fixes and feature additions. The shared library can also examine the version number that the client program originally linked against, which can be useful for maintaining backwards compatibility.
- The *compatibility version* of the library specifies the version of the library's API that the shared library claims to be backward-compatible with. If the compatibility version of the shared library is more recent than the version recorded with the client program, the program fails to link and an error occurs.

The *install name* is the pathname used by the dynamic linker to find a shared library at runtime. The install name is defined by the shared library and recorded into the client program by the static linker.

Packaging a Shared Library as a Framework

A *framework* is a shared library packaged with associated resources, such as headers, localized strings, and documentation, installed in a standard folder hierarchy, usually in a standard location in the file system. The folders usually contain related header files, documentation in any format, and resource files. A framework may contain multiple versions of itself, and each version may have its own set of resources, documentation, and header files.

From a tools perspective, a framework is a shared library whose install name ends in the form *frameworkName*`.framework/Versions/`*versionName*`/`*frameworkName* or the form *frameworkName*`.framework/`*frameworkName*.

You create a framework by building a normal dynamic shared library into a folder with the same name and the `.framework` extension. For example, to create a framework named Chaos, place a dynamic shared library named `Chaos` in a folder called `Chaos.framework`. You can create other folders inside this folder to store related resources, such as header files, documentation, and images (the standard folder names for these are called, respectively, `Headers`, `Documentation`, and `Resources`).

You can locate private frameworks and shared libraries in an application package using a relative-path install name beginning with `@executable_path`, such as `@executable_path/../Frameworks/MyFramework.framework`. This is useful for sharing functionality with plug-ins (bundles).

Apple follows a standard framework versioning convention, different from the shared library version numbering system. By versioning your framework, you can ship older versions of your framework alongside newer versions, to allow older clients to continue functioning, while still allowing you to advance the design of the framework in ways not compatible with older clients.

To version your framework, create a parent folder inside the framework called `Versions`, create a subfolder in `Versions` using a naming scheme of your choice, and build the framework shared library and other folders in this subfolder. Then create symbolic links in the framework's root folder to point to the shared library and folders. When you need to create a version of your framework that is not compatible with the previous version, build it into a new directory in the versions directory and update the symlinks to point to the new version. When a client links to a versioned framework, the install name recorded in the client executable includes the full path to the shared library executable, and the dynamic linker, thus, loads only that version.

For example, a client links to a framework called `Peace.framework`, and the symlinks in `Peace.framework` point to the latest version, which is named `B`.The install name of the framework ends

with `Peace.framework/Versions/B/Peace`. The static linker records this install name in the client. When the client is loaded, the dynamic linker attempts to load the shared library with this install name. Note that, while frameworks that ship with the system usually name successive versions with consecutive letters of the English alphabet (A through Z), you can use any name you want.

A framework developer can build a simple, versioned framework in four steps:

1. Create the framework version directory.
2. Compile the framework executable into the framework version directory.
3. Create a symbolic link named `Current` that points to the framework version directory.
4. Create a symbolic link to the framework executable in the parent framework directory.

The shell commands in Listing 1 builds a framework named `Bliss` from the C source files `Peace.c` and `Love.c`. The resulting framework has the install name `Bliss.framework/Versions/A/Bliss`.

Listing 1 Building a framework

```
gcc -c -o Peace.o Peace.c
gcc -c -o Love.o Love.c
mkdir -p Bliss.framework/Versions/A
gcc -dynamiclib -o Bliss.framework/Versions/A/Bliss Peace.o Love.o
cd ./Bliss.framework/Versions && ln -sf A Current
cd ./Bliss.framework && ln -sf Versions/Current/Bliss Bliss
```

Listing 2 demonstrates how to create a *private framework*—that is, a framework located in an application package. Specify the install name explicitly during the linking phase and prefix it with `@executable_path`. The install name of the resulting framework is `@executable_path/../Frameworks/Bliss.framework/Versions/A/Bliss`.

Listing 2 Building a private framework

```
mkdir -p Bliss.framework/Versions/A
gcc -c Peace.c Love.c
libtool -dynamic
  -install_name @executable_path/../Frameworks/Bliss.framework/Versions/A/Bliss
  -o Bliss.framework/Versions/A/Bliss Peace.o Love.o
  -framework System
```

For detailed information about designing and using frameworks, see *Framework Programming Guide*.

Packaging Frameworks and Libraries Under an Umbrella Framework

An *umbrella framework* is a framework that serves as the “parent” of a group of frameworks and shared libraries that implement related functionality. Umbrella frameworks help manage extremely large development projects with complex interdependencies, such as subsystems of OS X itself. For all other projects, a single framework should suffice (and is better for load-time performance).

To create an umbrella framework, you can take a normal framework and designate a subset of its imported frameworks as subframeworks. The subframeworks themselves need not be aware that they are part of the umbrella. With `ld`, you can use the `-sub_umbrella` option to designate a subframework.

When your program links against an umbrella framework, it also implicitly links against all the subframeworks. Symbols located in subframeworks of umbrella frameworks are recorded in the client program as if they were implemented directly in the umbrella framework. This feature allows the

contents of the umbrella framework to change over time while preserving compatibility with older client programs.

To ensure that clients link to the “parent” umbrella framework and not one of the subframeworks, the subframework can be built with a special load command to prevent unauthorized linking. When a client tries to link directly to such a subframework, the static linker produces an error. However, the subframework can authorize specific clients to link against it, and all subframeworks of an umbrella framework are implicitly authorized to link against each other. (Load commands are explained in *OS X ABI Mach-O File Format Reference*. The particular load commands referenced here are documented as `sub_framework_command` and `sub_client_command`; `ld` generates them if given the `-sub_framework <parent_umbrella_name>` and `-sub_client <client_name>` options.) Note that these conventions are enforced at build time by the static linker but ignored by the dynamic linker at runtime.

You can also include libraries in umbrella frameworks. For example the Foundation framework includes both the Objective-C runtime library (`libobjc`) as a sublibrary and the Core Foundation framework as a subframework. You may build Foundation using a variation on the commands listed in Listing 3.

Listing 3 Building a simple umbrella framework

```
mkdir Foundation.framework
gcc -dynamiclib -o Foundation.framework/Foundation -sub_umbrella CoreFoundation
   -sub_library libobjc -framework CoreFoundation -lobjc Foundation.o
```

By convention, subframeworks of an umbrella framework live within the `Frameworks` directory in the root directory of the umbrella framework, although this is obviously not a technical requirement. For example, the Cocoa framework is an umbrella framework that includes the AppKit framework; the AppKit framework is itself an umbrella framework that includes Foundation and Application Services as subframeworks.

Loading Plug-in Code With Bundles

Bundles provide the OS X mechanism for loading extension (or plug-in) code into an application at runtime. Typically, a bundle links against the application binary to gain access to the application’s exported API. Bundles can be—but are not required to be—packaged with resources, using the same folder hierarchy as that of an application package. In some cases (depending on the code in the bundle), bundles can also be unloaded.

OS X supports several schemes that allow third-party developers to extend the capabilities of your application by writing plug-in code that your program can load at runtime. Although you can use any one of these plug-in schemes in any type of application, some are more suited to particular situations than others. For example:

- To load Objective-C classes at runtime, use the Foundation framework class `NSBundle`. `NSBundle` provides general services for referring to a packaged program, whether the program is an application or a plug-in.
- To load C functions at runtime, use the Core Foundation framework object `CFBundle`, which, like the `NSBundle` class, provides general services for referring to a packaged program, whether the program is an application or a plug-in.
- The Core Foundation framework object `CFPlugin` implements a small subset of the Microsoft Component Object Model (COM) standard. COM allows you to instantiate C functions and data in an object-oriented manner at runtime.
- Carbon developers can also use *Code Fragment Manager (CFM)* to load code fragments updated for Carbon from PEF files. For more information, see the Code Fragment Manager documentation.
-

In general, for applications or libraries targeted for OS X v10.4 or later, use the dynamic loader compatibility functions, defined in `/usr/include/dlfcn.h`, to load and link bundle files. These functions are the preferred way to load code at runtime. They are particularly helpful when porting UNIX tools that support plug-ins to OS X. See “Dynamic Loader Compatibility Functions” in *OS X ABI Dynamic Loader Reference* for more information.

Note: The dynamic linker in OS X v10.0 causes your program to crash if you ask it to load programs that are built with a two-level namespace hint table. By default, the static linker in Mach OS X v10.0 creates bundles that do not include the two-level namespace hint table. If you want your program to run in OS X v10.0 and are developing in OS X v10.1 or later, use the `-flat_namespace` flag to ask the static linker to create the program using a flat namespace.

The `CFBundle` and `CFPlugin` objects can both be used from Carbon applications running in both Mac OS 9 and OS X. Both the `NSBundle` class and the `CFPlugin` object allow you to package plug-in code with the resources associated with the plug-in (such as graphics files and documentation), similar to the packaging for an application. To load COM objects in Mac OS 9, `CFPlugin` uses Code Fragment Manager, and in OS X, `CFPlugin` uses the object file image `dyld` library functions.

For more information on loading resources using the `NSBundle` class, see *Resource Programming Guide*. For more information on Code Fragment Manager, see *Mac OS Runtime Architectures*. For more information on `CFPlugin` and COM, see *Plug-in Programming Topics*.