# Plug-ins

**Core Foundation > Process Management**

**2005-03-03**

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



Simultaneously published in the United States and Canada.

# Contents

# Figures and Listings

# Introduction to Plug-ins

Plug-in architectures are an attractive solution for developers seeking to build applications that are modular, customizable, and easily extensible. What began as a clever way to allow third parties to add features to an application without access to source code has, for many developers, evolved into a full-blown component architecture. Core Foundation Plug-ins uses the basic code-loading facility of Core Foundation bundles to provide a standard plug-in architecture for Mac OS applications. Although this section is addressed primarily to host application developers, plug-in developers also need to read it in order to fully understand and make use of the CFPlugIn opaque type.

## Organization of This Document

The examples in this section demonstrate how to create and work with CFPlugIn objects. The error-checking code has been removed for clarity. In practice, it is *vital* that you check for errors because passing bad parameters into Core Foundation routines can cause your application to crash.

These articles discuss the plug-in architecture and how they work:

- "About Plug-ins" (page 9)
- "Plug-in Architecture" (page 11)
- "Anatomy of a Plug-in" (page 15)
- "Conceptual Building Blocks" (page 17)
- "Plug-in Registration" (page 21)

These articles contain examples on how to create and use plug-ins:

- "Defining Types and Interfaces" (page 23)
- "Implementing a Plug-in" (page 25)
- "Loading and Using a Plug-in" (page 31)

To generate UUIDs programmatically see:

- "Generating a UUID Programmatically" (page 33)

# About Plug-ins

Structuring an application as a well-designed host framework and a set of plug-ins has many benefits to you as an application developer:

- You can implement and incorporate application features very quickly.
- Because plug-ins are separate modules with well-defined interfaces, you can quickly isolate and solve problems.
- You can create custom versions of an application without source code modifications.
- Third parties can develop additional features without any effort on the part of the original application developer.
- Because plug-ins are language independent, you can reuse legacy code.

End-users also benefit from using applications with a plug-in architecture:

- They can customize feature sets to particular workflows.
- They can disable unwanted features, potentially simplifying the application's interface, reducing memory footprint, and improving performance.

If you are a host application developer who is designing a new plug-in model for your applications you should consider using the Core Foundation plug-in model for all but the simplest situations. Adopting the plug-in architecture can save both you and your plug-in developers a great deal of time and effort. Using plug-ins frees you from having to design, implement, and test a new plug-in model yourself. Because plug-ins is already documented, you won't have to develop and publish documentation for an entire plug-in architecture, only the interfaces your host application supports. And finally, your plug-in developers won't have to learn yet another plug-in model.

If you have a plug-in model in place, you may still want to consider converting to Core Foundation plug-ins simply because your plug-in developers need Carbonize their plug-ins if they want them to work on Mac OS X. If the conversion to plug-ins is a small enough task, it probably makes sense to have them convert at the same time. The level of difficulty involved in the conversion depends primarily on how close your existing model is to the plug-in model.

If you have an existing plug-in model that represents too great an investment to give up, you probably can (and should) make use of the more primitive code-loading features of bundles to add support for multiple binary formats. See *Bundle Programming Guide* if you aren't familiar with Core Foundation bundles.

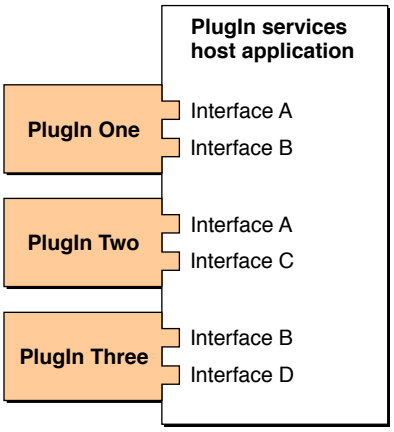
The Core Foundation concepts and API documented here are specific to Carbon 1.1 and later. Carbon 1.0, which was first shipped with Mac OS 9, included an early version of plug-ins, but that programming interface is now considered deprecated. Core Foundation 1.3 (released with Carbon 1.1) maintains basic compatibility with the previous plug-in model. If you have code written to the older API and you don't want to re-write it for the new API, you will still have to modify the code slightly to compile against the new header file. There are many differences between the 1.0 and 1.1 plug-in APIs, but only one you must worry about if you want

to keep using the 1.0 plug-in model—the use of UUIDs instead of unique strings to identify factories, types, interfaces and so on. Because Core Foundation uses opaque types, you can simply cast the type `CFStringRef` to `CFUUIDRef` and the functions will still work properly.

# Plug-in Architecture

All plug-in models require two basic entities—the plug-in host and the plug-in itself. The host could be an application, operating system, or even another plug-in. The plug-in host's code is structured such that certain well-defined areas of functionality can be provided by an external module of code—a plug-in. Plug-ins are written and compiled entirely separately from the host, typically by another developer. When the host code is executed, it uses whatever mechanism is provided by the plug-in architecture to locate compatible plug-ins and load them, thus adding capabilities to the host that were not previously available.

**Figure 1** A CFPlugIn host with three plug-ins.




The plug-in model is flexible enough to be used in at least two fundamentally different ways. The first approach is to use plug-ins to support "variations on a theme" features wherein each plug-in implements very similar functionality and uses an identical interface. This methodology is frequently employed to add special processing support to image-editing and audio-editing applications.

An audio-editing application, for instance, might ship with only a few simple processing options like equalization and normalization. If this application has a plug-in architecture, a third party could add support for additional processing functions—perhaps reverberation or flanging—without access to the application's source code. This is possible because the host application's developer has provided a clearly defined interface that all of its audio processing plug-ins must use. By requiring the host and plug-in to communicate only through this well-specified interface, the plug-in architecture allows the audio application to remain entirely ignorant of the details of processing.

Using this approach, the host application developer designs a plug-in interface with one function for processing data. To identify the interface, the host developer gives it a unique ID. The interface also contains a place to put a string describing the type of processing so that the host application can distinguish between plug-ins implementing the interface.

When the host application is launched, it searches for all plug-ins with the appropriate identifier. For each plug-in found, the host uses the plug-in architecture to obtain the processing description string and a pointer to the processing function. The host can then construct a menu of available audio processing techniques and present them to the user. When a user chooses a processing type from the menu, the host calls the

associated function to do the work. The host knows nothing about the details of the plug-in's implementation, and the plug-in knows nothing about the application's implementation. Either one might be completely rewritten, but as long as the interface is honored by both parties, everything will continue to work.

The plug-in model can also be used as a component architecture wherein each component (a plug-in) implements very different functionality. In this approach, you would structure your application as a plug-in "shell" and a set of plug-ins, each of which takes care of a major area of application function—user interface, file system interaction, network communications, and so on.

This model offers benefits similar to those of the more literal plug-in approach outlined above. Because a component's implementation details are hidden from other components, they can be modified at will so long as the component interfaces continue to behave as specified. An added benefit of this approach is that components can be easily shared among different applications. Note that you can use both approaches in a single application.

# Plug-ins and Microsoft's COM

The plug-in model is compatible with the basics of Microsoft's COM (Component Object Model) architecture. This means that the plug-in interface is laid out according to the COM guidelines and that all interfaces must inherit from COM's IUnknown interface. These are the only elements plug-ins share with COM. Other COM concepts such as the IClassFactory interface, aggregation, out-of-process servers, and Windows registry are not mapped. This document only minimally covers COM concepts as necessary to explain the way they are used in Core Foundation plug-ins. For additional information, you are encouraged to seek out the wealth of documentation already published about COM. A good place to start is the COM area of Microsoft's web site, `http://www.microsoft.com/com/tech/com.asp`.

# Anatomy of a Plug-in

On disk, a CFPlugIn is laid out as a file package just like a CFBundle. What makes a CFPlugIn special is the addition of a few keys in the plug-in's information property list. These keys are documented in "Plug-in Registration" (page 21) Because bundles and plug-ins share the same structure on disk, it is tempting—though incorrect—to think of a CFPlugIn as a CFBundle. At runtime, however, it is correct to say that a CFPlugIn *has* a CFBundle.

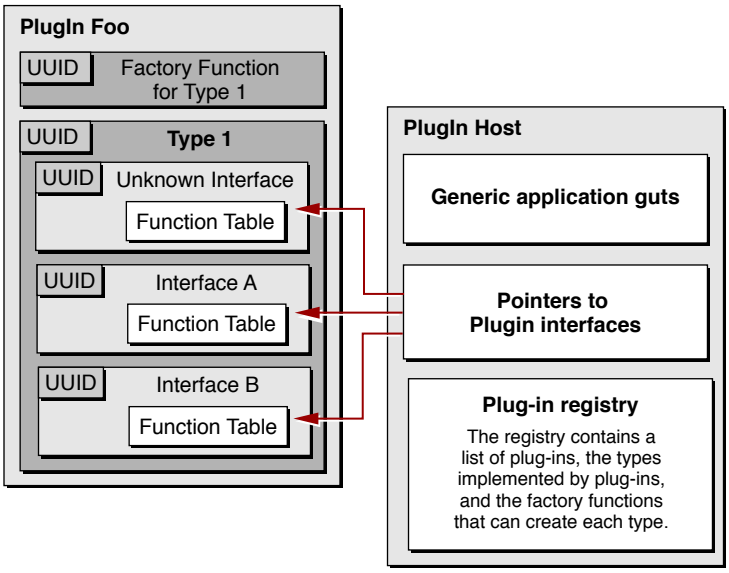
CFPlugIns and CFBundles come in pairs. Every CFPlugIn has a CFBundle, but each CFBundle (an application or framework bundle for instance) does not necessarily correspond to a CFPlugIn. You should *never* attempt to directly access a plug-in as a CFBundle. You should use the function `CFPlugInGetBundle` if you need to access a plug-in's resources with the CFBundle API.

If your plug-in will be manually installed by users it is a good idea to define Mac OS style creator/type codes (and perhaps a filename extension as well, though this is not required) for your plug-ins so that they will display the appropriate icon in file system views.

# Conceptual Building Blocks

In the plug-in model, the host application defines one or more types that each consist of one or more interfaces. A plug-in implements all of the functions in all of the interfaces for each type the plug-in supports. A plug-in also provides creation functions called factories for each type it wants to implement. When the host application loads a plug-in, a factory function is registered with the host for each of the plug-in's types. The host can then use a type's factory function to instantiate the type and obtain a pointer to its IUnknown interface. The host uses IUnknown to query the type for its interface function tables. The function tables give the host access to the plug-in's interface implementations. Figure 1 (page 17) illustrates the relationships between these various components.

**Figure 1** Building blocks of the Core Foundation plug-in model.




The following sections describe each aspect of the plug-in model in detail. In addition, the tasks in this topic provide a complete example of how to build a simple plug-in host and plug-in.

## UUIDs (Universally Unique Identifiers)

UUIDs are used by plug-ins to uniquely identify types, interfaces, and factories. When creating a new type, host developers must generate UUIDs to identify the type as well as its interfaces and factories.

UUIDs (Universally Unique Identifiers), also known as GUIDs (Globally Unique Identifiers) or IIDs (Interface Identifiers), are 128-bit values guaranteed to be unique. A UUID is made unique over both space and time by combining a value unique to the computer on which it was generated—usually the Ethernet hardware address—and a value representing the number of 100-nanosecond intervals since October 15, 1582 at 00:00:00.

To create a UUID for a plug-in you typically use the commandline utility `uuidgen`. If you need to generate UUIDs programmatically you can do so using the functions defined in `CFUUID.h` (see "Generating a UUID Programmatically" (page 33)). The standard format for UUIDs represented in ASCII is a string punctuated by hyphens, for example `68753A44-4D6F-1226-9C60-0050E4C00067`. The hex representation looks, as you might expect, like a list of numerical values preceded by `0x`. For example, `0xD7, 0x36, 0x95, 0x0A, 0x4D, 0x6E, 0x12, 0x26, 0x80, 0x3A, 0x00, 0x50, 0xE4, 0xC0, 0x00, 0x67`. In order to use a UUID, you simply create it and then copy the resulting strings into your header and C language source files. Note that Core Foundation's UUID API is available only on Mac OS X.

# Interfaces

An interface is the fundamental abstraction a host uses to define an area of functionality to be implemented by plug-in developers. All interactions between a host and plug-in occur through an interface. In programming terms, an interface represents a function table where each function has a specific semantic meaning.

An interface constitutes a contract between the host and plug-in. This contract includes

- the order of the functions in the interface
- the interface functions' parameter types and return types
- the expected behavior of each function

A group of functions cannot rightly be considered an interface unless there is a specific definition of exactly what each function is supposed to do.

For example, consider an interface with one function that takes a single integer and returns an integer. Without a definition of behavior, any function that takes and returns an integer might be considered an implementation of that interface. To address this ambiguity, the expected behavior of each function in the interface is also "part" of that interface. A plug-in host developer must provide a header file and documentation to potential plug-in developers describing the functions that make up the interface, and exactly how the functions are expected to behave. It is critical that the plug-in developer know exactly what is required to correctly implement the interface.

Interfaces in the plug-in model are identical to the interfaces defined by Microsoft's COM architecture. When implemented in C++, all plug-in interfaces inherit the IUnknown interface, and in C, all plug-in interface function tables must include the IUnknown functions. At runtime, the host uses the IUnknown interface to find and obtain access to other interfaces implemented by the plug-in. See "Defining Types and Interfaces" (page 23) for more information about IUnknown and implementing a plug-in interface.

# Types

A type represents an aggregation of interfaces. Like interfaces, types are also defined by the host and implemented by the plug-in. A type is an entirely abstract entity that groups related interfaces together, providing a conceptual umbrella for the different APIs represented by the type's constituent interfaces. Types allow for a high-level representation of a feature to be implemented or problem to be solved. Interfaces express the way you intend to factor the problem represented by the type. You can think of interfaces as the implementation of a type.

For example, say you're developing an application called ImageViewer and you want to allow third parties to add file format support for different types of images using plug-ins. You might define the type `ImageAdaptorType`, which would consist of two interfaces, `ImageIOInterface` and `ImageSnifferInterface`. `ImageIOInterface` might consist of two functions. The first function would take a parameter of type `CFURLRef` and return a special `ImageViewerImage` type defined by ImageViewer to represent an image. The second function would take parameters of type `CFURLRef` and `ImageViewerImage` and write the image to the location described by the URL. `ImageSnifferInterface` might define a function that would take a `CFURLRef` parameter and return a `Boolean` value indicating whether the file pointed to could be read.

Because an interface cannot ever change once it has been published, types must support multiple interfaces. Instead of changing a type's interface, you simply add a new interface to the type with the changed or additional functionality. In this way, a single plug-in might support different versions of a host application by implementing a different interface for each version. The type would remain the same; it would be up to the host application to locate and use the appropriate interface.

It is also possible to define optional interfaces for a type. For example, a host application might define a type with an optional interface specific to a piece of add-on hardware. The host application could test for the presence of that hardware at runtime and, if it exists, the host could request the hardware-specific plug-in interface. If a plug-in implementor doesn't want to support the hardware specific functionality, they need not implement that interface.

# Factories

A factory represents a function that can create an instance of one or more types. Calling a type's factory function is analogous to the calling operator `new` on a class in Java or C++. When called by the plug-in host, the factory function allocates memory for an instance of the type being requested, sets up the function tables for its interfaces, and returns a pointer to the type's IUnknown interface. The plug-in host can then use the IUnknown interface to search for other interfaces supported by the type.

When a CFPlugin is created, the system registers all types the plug-in supports along with their associated factory functions. When the plug-in host wants to create an instance of a given type, it uses the type's UUID to search for any registered factory functions. It can then use a factory function to create an instance of the type.

Note that it is possible for a plug-in to have multiple factory functions for the same type. It is up to the developer to define appropriate usage for the different functions.

# Instances

Because the Core Foundation 1.3 and later plug-in APIs use the COM model, `CFPlugInInstanceRefs` are no longer necessary. In the post-1.2 API an instance is not a Core Foundation data type that can be directly manipulated, it is rather a generic term referring to the resources used by a plug-in to implement a type.

# Plug-in Registration

For the plug-in host to know what types are available, each plug-in must register with the host. Registration consists of making the host aware of the types a plug-in implements and their associated factory functions. This information can be declared statically using a few special keys in the plug-in's information property list or it can be registered dynamically by code in the plug-in. See "Implementing a Plug-in" (page 25) for an example CFPlugIn's information property list.

When a plug-in is found by a host, plug-ins use the value of the `CFPlugInDynamicRegistration` key in the plug-in's information property list to determine whether a plug-in requires static or dynamic registration. If a plug-in uses dynamic registration, the plug-in's code must be loaded immediately so the dynamic registration can take place. If the plug-in uses static registration, its code need not be loaded until the application actually needs to instantiate a type. For this reason, static registration is preferred when there's no overriding reason for using dynamic registration.

For static registration, the information property list contains the `CFPlugInFactories` key whose value is a dictionary whose keys are factory UUIDs (expressed in the standard string format) and whose values are function names. Each key-value pair in this dictionary registers a factory function when the plug-in's CFBundle is created. The information property list also contains a `CFPlugInTypes` key. The value of this key is a dictionary whose keys are type UUIDs and whose values are arrays of factory UUIDs. For each type then, there is a list of the factories within the plug-in that can create that type. See "Implementing a Plug-in" (page 25) for an example plug-in's information property list.

For dynamic registration, plug-ins load the plug-in's code and allows it to do its dynamic registration as soon as the associated CFBundle is created. For plug-ins that do dynamic registration, the optional `CFPlugInDynamicRegisterFunction` key can be set to the name of the function that should be called to do the dynamic registration. If this key is not provided, plug-ins attempt to call a function named `CFPlugInDynamicRegister`. When you implement the `CFPlugInDynamicRegister` function (or your custom version) you use dynamic registration functions of plug-ins to create the type/factory associations that would otherwise be declared in the plug-in's information property list.

A code entity can also register built-in types and factories using the dynamic registration functions. The code with built-in types could then use the plug-in model internally, or allow plug-ins to query the host for the built-in types and interfaces. This might be necessary if plug-ins need to discover some information about the host's state before being used.

## Information Property List Keys Defined by Plug-ins

The keys described in this section are defined by plug-ins and are used either to statically register the types supported by the plug-in or to define the plug-in's dynamic registration behavior.

`kCFPlugInDynamicRegistration`

Used to determine the method of registration required by the plug-in. Its value is a string—`YES` for dynamic registration, `NO` for static registration.

`CFPlugInDynamicRegisterFunction`

The name of a custom function to be called to perform dynamic registration. If dynamic registration is enabled and this key is not present, the function `CFPlugInDynamicRegister` is called.

`CFPlugInUnloadFunction`

The name of a custom function to be called to when a plug-in's code is to be unloaded.

`CFPlugInFactories`

Used for static registration. Its value should be a dictionary whose keys are factory UUIDs (expressed in the standard string format) and whose values are function names.

`CFPlugInTypes`

Used for static registration. Its value should be a dictionary whose keys are type UUIDs and whose values are arrays of factory UUIDs.

# Defining Types and Interfaces

Your first task as a plug-in host developer is to define the types and interfaces the host supports.

To define a type, all you really need is a UUID. To create a UUID for a plug-in you typically use the commandline utility `uuidgen`. If you need to generate UUIDs programmatically you can do so using the functions defined in `CFUUID.h` (see "Generating a UUID Programmatically" (page 33)). In the case of types and factories, you need the UUID in two forms, the hexadecimal representation for the header file, and the ASCII representation for the information property list. The standard format for UUIDs represented in ASCII is a string punctuated by hyphens, for example: `D736950A-4D6E-1226-803A-0050E4C00067`. The hex representation looks, as you might expect, like a list of numerical values preceded by `0x`. For example, `0xD7, 0x36, 0x95, 0x0A, 0x4D, 0x6E, 0x12, 0x26, 0x80, 0x3A, 0x00, 0x50, 0xE4, 0xC0, 0x00, 0x67`.

In addition to its UUID, an important bit of information about a type is what interfaces the type is expected to implement and which—if any—are optional. This information is not needed at runtime and is not expressed as code in the header, but should be expressed as a comment, just to be clear.

To define an interface you need a UUID for the interface and a structure for the function table for that interface. This means you have to define the function pointers and expected behavior for each function in the interface.

Listing 1 (page 23) shows the contents of a header file that declares a type and the interface used to implement the type. This header is typically created by the plug-in host developer and made available to plug-in writers.

**Listing 1** Defining a type and interface

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

// Define the UUID for the type.
#define kTestTypeID (CFUUIDGetConstantUUIDWithBytes(NULL, 0xD7, 0x36, 0x95,
0x0A,
0x4D, 0x6E, 0x12, 0x26, 0x80, 0x3A, 0x00, 0x50, 0xE4, 0xC0, 0x00, 0x67))

// Define the UUID for the interface.
// TestType objects must implement TestInterface.
#define kTestInterfaceID (CFUUIDGetConstantUUIDWithBytes(NULL, 0x67, 0x66, 0xE9,
0x4A, 0x4D, 0x6F, 0x12, 0x26, 0x9E, 0x9D, 0x00, 0x50, 0xE4, 0xC0, 0x00, 0x67))

// The function table for the interface.
typedef struct TestInterfaceStruct
{
    IUNKNOWN_C_GUTS;
    void (*fooMe)( void *this, Boolean flag );
} TestInterfaceStruct;
```

Notice that the structure that defines the interface's function table includes `IUNKNOWN_C_GUTS` as its first element. This macro—shown in Listing 2 (page 24)— expands into the structure definition for the COM IUnknown interface. The COM specification requires that all interfaces inherit from IUnknown. Practically speaking, this means every interface function table must begin with three functions—`QueryInterface`,

`AddRef`, and `Release`. This also means that any interface can be treated polymorphically as an IUnknown interface. At runtime, the host uses `QueryInterface` to find and gain access to all other interfaces the type supports. `AddRef` and `Release` are used for reference counting.

**Listing 2** The IUnknown interface in C

```
#define IUNKNOWN_C_GUTS \
 void *_reserved; \
 HRESULT (STDMETHODCALLTYPE *QueryInterface) \
            (void *thisPointer, REFIID iid, LPVOID *ppv); \
 ULONG (STDMETHODCALLTYPE *AddRef)(void *thisPointer); \
 ULONG (STDMETHODCALLTYPE *Release)(void *thisPointer)
```

In C++ , using a compiler that supports COM, this would be accomplished by deriving your interface class from the IUnknown class. Listing 3 (page 24) shows what the IUnknown interface would look like in C++.

**Listing 3** The IUnknown interface in C++

```
interface IUnknown
{
    virtual HRESULT __stdcall QueryInterface(const IID& iid
                                    void **ppv) = 0;
    virtual ULONG __stdcall AddRef() = 0;
    virtual ULONG __stdcall Release() = 0;
}
```

Finally, notice that the function `fooMe` in `TestInterfaceStruct` takes a `this` argument as its first parameter. This is not required, but it is a nice thing to do to assist plug-in writers. By passing a `this` pointer to each interface function, you allow the plug-in writer to implement in C++ and to have access to the plug-in object when the function executes in any language.

# Implementing a Plug-in

This section details all of the steps necessary to actually implement a plug-in that supports the type declared in "Defining Types and Interfaces" (page 23)

## Registering Types and Interfaces

Now that we have a type and some interfaces, let's look at how a plug-in that supported this type would be implemented. First, consider the information property list for the plug-in in Listing 1 (page 25)

**Listing 1** An Info.plist file for a plug-in

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>CFBundleDevelopmentRegion</key>
    <string>English</string>
    <key>CFBundleExecutable</key>
    <string>CFTestPlugin</string>
    <key>CFBundleIconFile</key>
    <string></string>
    <key>CFBundleIdentifier</key>
    <string>com.apple.yourcfbundle</string>
    <key>CFBundleInfoDictionaryVersion</key>
    <string>6.0</string>
    <key>CFBundlePackageType</key>
    <string>BNDL</string>
    <key>CFBundleSignature</key>
    <string>????</string>
    <key>CFBundleVersion</key>
    <string>1.0</string>
    <key>CFPlugInDynamicRegisterFunction</key>
    <string></string>
    <key>CFPlugInDynamicRegistration</key>
    <string>NO</string>
    <key>CFPlugInFactories</key>
    <dict>
        <key>68753A44-4D6F-1226-9C60-0050E4C00067</key>
        <string>MyFactoryFunction</string>
    </dict>
    <key>CFPlugInTypes</key>
    <dict>
        <key>D736950A-4D6E-1226-803A-0050E4C00067</key>
        <array>
            <string>68753A44-4D6F-1226-9C60-0050E4C00067</string>
        </array>
```

```
    </dict>
    <key>CFPlugInUnloadFunction</key>
    <string></string>
</dict>
</plist>
```

The information property list defines various aspects of the plug-in's runtime behavior and contains optional static registration information for the various types the plug-in supports. For more information about static and dynamic registration, see "Plug-in Registration" (page 21)

In this example, the `CFBundleExecutable` key tells CFBundle the name of the executable and is used by the primitive code-loading API of bundles. The rest of the keys are specific to the plug-in model.

The `CFPlugInDynamicRegistration` key indicates whether this plug-in requires dynamic registration. In this example, static registration is used, so the dynamic registration key is set to `NO`.

The `CFPlugInFactories` key is used to statically register factory functions, and the `CFPlugInTypes` key is used to statically register the factories that can create each supported type.

# Implementing the Types, Factories, and Interfaces

When implementing a plug-in, you must provide

- the implementation for any factory functions registered for the plug-in
- implementations for all the functions in all the interfaces for any types supported by your plug-in
- an interface function table for each interface you implement
- the required COM functions, `QueryInterface`, `AddRef` and `Release`

Listing 2 (page 26) contains the code for a plug-in that implements the type `kTestTypeID` and its interface.

**Listing 2** Example plug-in implementation

```
#include <CoreFoundation/CoreFoundation.h>
#include "TestInterface.h"

// The UUID for the factory function.
#define kTestFactoryID (CFUUIDGetConstantUUIDWithBytes(NULL,
 0x68, 0x75, 0x3A, 0x44, 0x4D, 0x6F, 0x12, 0x26, 0x9C, 0x60,
0x00, 0x50, 0xE4, 0xC0, 0x00, 0x67))

// The layout for an instance of MyType.
typedef struct _MyType {
    TestInterfaceStruct *_testInterface;
    CFUUIDRef _factoryID;
    UInt32 _refCount;
 } MyType;

// Forward declaration for the IUnknown implementation.
static void _deallocMyType( MyType *this );

// Implementation of the IUnknown QueryInterface function.
static HRESULT myQueryInterface( void *this, REFIID iid, LPVOID *ppv )
```

```
{
    // Create a CoreFoundation UUIDRef for the requested interface.
    CFUUIDRef interfaceID = CFUUIDCreateFromUUIDBytes( NULL, iid );

    // Test the requested ID against the valid interfaces.
    if (CFEqual(interfaceID, kTestInterfaceID))
    {
        // If the TestInterface was requested, bump the ref count,
        // set the ppv parameter equal to the instance, and
        // return good status.
        ((MyType *)this)->_testInterface->AddRef( this );
        *ppv = this;
        CFRelease(interfaceID);
        return S_OK;
    }

    if (CFEqual(interfaceID, IUnknownUUID))
    {
        // If the IUnknown interface was requested, same as above.
        ((MyType *)this)->_testInterface->AddRef( this );
        *ppv = this;
        CFRelease( interfaceID );
        return S_OK;
    }

    // Requested interface unknown, bail with error.
    *ppv = NULL;
    CFRelease( interfaceID );
    return E_NOINTERFACE;
}

// Implementation of reference counting for this type.
// Whenever an interface is requested, bump the refCount for
// the instance. NOTE: returning the refcount is a convention
// but is not required so don't rely on it.
static ULONG myAddRef( void *this )
{
    ((MyType *)this)->_refCount += 1;
    return ((MyType *)this)->_refCount;
}

// When an interface is released, decrement the refCount.
// If the refCount goes to zero, deallocate the instance.
static ULONG myRelease( void *this )
{
    ((MyType *)this)->_refCount -= 1;
    if (((MyType *)this)->_refCount == 0)
    {
        _deallocMyType( (MyType *)this );
        return 0;
    }
    return ((MyType *)this)->_refCount;
}

// The implementation of the TestInterface function.
static void myFooMe( void *this, Boolean flag )
{
    printf("myFooMe: instance 0x%x: I've been fooed.  %s\n",
```

```
            (unsigned)this, (flag ? "YES" : "NOPE"));
}

// The TestInterface function table.
static TestInterfaceStruct testInterfaceFtbl =
{
        NULL,               // Required padding for COM
        myQueryInterface,   // These three are the required COM functions
        myAddRef,
        myRelease,
        myFooMe
}; // Interface implementation

// Utility function that allocates a new instance.
static MyType *_allocMyType( CFUUIDRef factoryID )
{
    // Allocate memory for the new instance.
    MyType *newOne = (MyType *)malloc( sizeof(MyType) );

    // Point to the function table
    newOne->_testInterface = &testInterfaceFtbl;

    // Retain and keep an open instance refcount
    // for each factory.
    newOne->_factoryID = CFRetain( factoryID );
    CFPlugInAddInstanceForFactory( factoryID );

    // This function returns the IUnknown interface
    // so set the refCount to one.
    newOne->_refCount = 1;
    return newOne;
}

// Utility function that deallocates the instance
// when the refCount goes to zero.
static void _deallocMyType( MyType *this )
{
    CFUUIDRef factoryID = this->_factoryID;
    free(this);
    if (factoryID)
    {
        CFPlugInRemoveInstanceForFactory( factoryID );
        CFRelease( factoryID );
    }
}

// Implementation of the factory function for this type.
void *MyFactory(CFAllocatorRef allocator, CFUUIDRef typeID)
{
    // If correct type is being requested, allocate an
    // instance of TestType and return the IUnknown interface.
    if (CFEqual(typeID, kTestTypeID))
    {
        MyType *result = _allocMyType( kTestFactoryID );
        return result;
    }
    // If the requested type is incorrect, return NULL.
    return NULL;
```

```
}
```

As illustrated in Listing 2 (page 26) the first step in implementing a plug-in to is define the UUID for the factory you are going to supply. This is the same UUID that was used in the `CFPlugInFactories` key in the information property list. Next, the data structure for instances of the `TestType` implementation is defined.

After defining the instance structure, you implement the IUnknown interface functions required for every plug-in. In this example, `QueryInterface`, relies on the fact that the first pointer in the instance structure is an interface, so returning a pointer to the `MyType` structure is the same as returning a pointer to a pointer to `TestInterface`. Types that implement more than one interface would be more complicated. In C++, this can be accomplished using multiple inheritance and static casting. In C, you would have to keep track of the interface pointers by hand.

After the IUnknown functions, there is the implementation for the `fooMe` function from `TestInterface`. In this example it just prints a message. Next comes the static definition of the actual `TestInterface` function table. This table is filled in with the IUnknown and `TestInterface` functions.

Following the function table are two utility functions that allow easy creation and freeing of `MyType` structures. The allocator fills in the pointer to the interface function table and sets the initial reference count to 1. It also takes care of registering the instance with the factory so plug-ins knows not to unload the plug-in's code while there are still instances active. The deallocator function frees the memory for `MyType` and unregisters the instance from the factory.

Finally, there is the actual factory function that creates a new instance and returns a pointer to it. This pointer is also a pointer to the IUnknown interface. The `MyFactory` function must conform to the `CFPlugInFactoryFunction` prototype. Factory functions take allocators and type UUIDs as parameters.

Listing 2 (page 26) contains a lot of glue code that would be unnecessary for C++ developers using a compiler with built-in support for generating COM interface layouts. If you wish to implement CFPlugIns in C++, refer to the wealth of COM documentation by Microsoft and others.

# Loading and Using a Plug-in

The code example in Listing 1 (page 31) shows how a plug-in host can load and use a plug-in. It searches registered plug-ins for one that implements a test interface. If found, it invokes functions using the interface.

- `CFPlugInFindFactoriesForPlugInType` searches all the registered plug-ins and returns an array of all factories that can create the requested type.
- For each factory, `CFPlugInInstanceCreate` creates an instance of the test type and obtains a pointer to its `IUnknown` interface, if that exists.
- The `IUnknown` interface is tested to determine if it is the test interface. If it is, one of its functions is invoked and a flag is set to stop the search.

**Listing 1** Loading and using a plug-in

```
Boolean foundInterface = false;

// Create a URL that points to the plug-in using a CFString path 'aPath'.
CFURLRef url =
    CFURLCreateWithFileSystemPath(NULL, aPath, kCFURLPOSIXPathStyle, TRUE);

// Create a CFPlugin using the URL.
// This causes the plug-in's types and factories to be registered with the
system.
// The plug-in's code is not loaded unless it is using dynamic registration.
CFPlugInRef plugin = CFPlugInCreate(NULL, url);

if (!plugin)
{
    printf("Could not create CFPluginRef.\n");
}
else
{
    // Test whether this plug-in implements the Test type.
    CFArrayRef factories = CFPlugInFindFactoriesForPlugInType(kTestTypeID);

    // If there are factories for the requested type, attempt to
    // get the IUnknown interface.
    if (factories != NULL)
    {
        CFIndex factoryCount = CFArrayGetCount(factories);

        if (factoryCount <= 0)
        {
            printf("Could not find any factories.\n");
        }
        else
        {
            CFIndex index;
```

```
            for (index = 0;
                (index < factoryCount) && (!foundInterface);
                index++)
            {
                // Get the factory ID at the current index.
                CFUUIDRef factoryID =
                            CFArrayGetValueAtIndex(factories, index);
                // Use the factory ID to get an IUnknown interface.
                // The code for the PlugIn is loaded here.
                IUnknownVTbl **iunknown =
                    CFPlugInInstanceCreate(NULL, factoryID, kTestTypeID);
                // If this is an IUnknown interface,
                // query for the Test interface.
                if (iunknown)
                {
                    TestInterfaceStruct **interface = NULL;
                    (*iunknown)->QueryInterface(iunknown,
                                CFUUIDGetUUIDBytes(kTestInterfaceID),
                                (LPVOID *)(&interface));
                    // Finished with IUnknown.
                    (*iunknown)->Release(iunknown);

                    // If this is a Test interface, try to call its function.
                    if (interface)
                    {
                        (*interface)->fooMe(interface, TRUE);
                        (*interface)->fooMe(interface, FALSE);
                        // Finished with test interface.
                        // This causes the plug-in's code to be unloaded.
                        (*interface)->Release(interface);

                        foundInterface = true;
                    }
                }
                // end of iunknown
            }
            // end of index loop
        }
        // factoryCount > 0

        // Release the factories array.
        CFRelease(factories);
    }
    if (!foundInterface)
    {
        printf("Failed to get interface.\n");
    }
    // Release the CFPlugin -- memory for the plug-in is deallocated here.
    CFRelease(plugin);
}
```

# Generating a UUID Programmatically

Listing 1 (page 33) shows you how to generate a UUID programmatically using CFUUID functions. Plug-ins use UUIDs to uniquely identify types, interfaces, and factories.

**Listing 1** Generating a UUID programmatically

```
CFUUIDRef     myUUID;
CFStringRef   myUUIDString;
char          strBuffer[100];

myUUID = CFUUIDCreate(kCFAllocatorDefault);
myUUIDString = CFUUIDCreateString(kCFAllocatorDefault, myUUID);

// This is the safest way to obtain a C string from a CFString.
CFStringGetCString(myUUIDString, strBuffer, 100, kCFStringEncodingASCII);

CFStringRef outputString =
    CFStringCreateWithFormat(kCFAllocatorDefault,
                             NULL,
                             CFSTR("My UUID is: %s!\n"),
                             strBuffer);
CFShow(outputString);
```

# Document Revision History

This table describes the changes to *Plug-ins*.

| Date | Notes |
| --- | --- |
| 2005-03-03 | Added reference to uuidgen command-line utility for generating UUIDs. |
| 2004-08-31 | Updated info.plist example to XML format; made UUID examples equivalent. |
| 2003-10-24 | Code sample for "Loading and Using a Plug-in" (page 31) corrected. |
| 2003-03-18 | Both pieces of art replaced with cleaner versions. |
| 2003-01-17 | Converted existing Core Foundation documentation into topic format. Added revision history. |