Daemons and Services Programming Guide

Developer

Contents

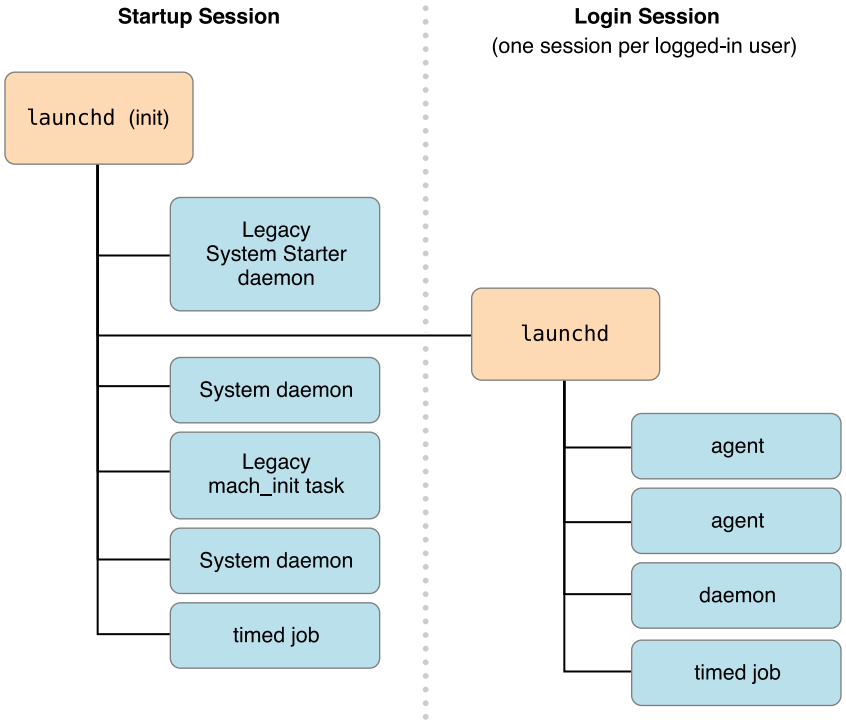
Figures and Tables

About Daemons and Services

Many kinds of tasks that do not require user interaction are most effectively handled by a process that runs in the background. You can use a daemon or service to:

- Provide server functionality, such serving web pages.
- Coordinate access to of a shared resource, such as a database.
- Perform work for a foreground application, such as file system access.

Figure I-1 Daemons and services are started by launchd in two separate session contexts



Note: This document was previously titled System Startup Programming Topics.

At a Glance

This document provides information that developers of daemons and other low-level system services need to write their code and incorporate it into the startup process. It also provides some useful information for system administrators who must manage the startup process on the computers they manage.

Terminology Note: The terms "service" and "daemon" have several meanings in different contexts, with further variation over time and from one development community to another.

In this document, *service* refers to a background process that supports a full GUI application in some way, for example by registering a global hotkey or by performing network communication. *Daemon* refers to all other types of background processes, especially those that don't present any kind of user interface.

Design your Background Process

OS X provides a variety of background process types with different characteristics, designed for a different situations. There are also several ways for other processes to communicate with background processes. Choosing the appropriate design for a background process is an important first step.

Relevant Chapters: Designing Daemons and Services (page 9)

Implement your Background Process

Having made the design decisions, you are ready to begin writing code. These chapters guide you through the process of creating specific types of background jobs.

Relevant Chapters: Adding Login Items (page 18), Creating XPC Services (page 20), Creating Launch Daemons and Agents (page 39)

Logging Errors and Warnings

Logging is an effective way for background processes to record unusual behavior and error conditions. Making appropriate use of the logging mechanisms provided by OS X can assist during debugging and end-user troubleshooting.

Relevant Chapters: Logging Errors and Warnings (page 53)

Running Jobs on a Timed Schedule

Although it is recommended that background jobs be launched on demand, in some cases running the job on a timed schedule is the most appropriate solution.

Relevant Chapters: Scheduling Timed Jobs (page 62)

See Also

Daemons and Agents provides additional details about implementing launch daemons and agents.

Kernel Programming Guide and *Kernel Extension Programming Topics* describe how to write kernel extensions and other kernel-level background processes.

Networking Overview describes the APIs available for sending and receiving data across the network.

Designing Daemons and Services

Two of the most important design decisions to consider when creating a background process are how it will be run and how other processes will communicate with it. These two considerations interact with each other: different types of background processes have different forms of communication available to them.

Types of Background Processes

There are four types of background processes in OS X. The differences are summarized in Table 1-1 (page 9) and described in detail in the following subsections. To select the appropriate type of background process, consider the following:

- Whether it does something for the currently logged in user or for all users.
- Whether it will be used by single application or by multiple applications.
- Whether it ever needs to display a user interface or launch a GUI application.

> **Note:** The descriptions in this section focus on recommended best practices, but some additional kinds of behavior are also available. For these details, see the respective chapters for each kind of daemon and service.

Table 1-1 Types of Background Process

Type	Managed by launchd?	Run in which context?	Can present UI?
Login item	No*	User	Yes
XPC service	Yes	User	No (Except in a very limited way using IOSurface)
Launch Daemon	Yes	System	No
Launch Agent	Yes	User	Not recommended

* Login items are started by the per-user instance of `launchd`, but it does not take any actions to manage them.

For more information about the system and user contexts, see *Multiple User Environment Programming Topics* .

Login items

Login items are launched when the user logs in, and continue running until the user logs out or manually quits them. Their primary purpose is to allow users to open frequently-used applications automatically, but they can also be used by application developers. For example, a login item can be used to display a menu extra or to register a global hotkey.

For example, many to-do applications use a login item that listens for a global hotkey and presents a minimal UI allowing the user to enter a new task. Login items are also commonly used to display user interface items, such as a floating clock or a timer, or to display an icon in the menu bar. .

Another example is a calendaring application with a helper application launched as a login item. The helper application runs in the background and launches the main GUI application when appropriate to remind the user of upcoming appointments.

For information about how to create a login item, see Adding Login Items (page 18).

XPC Services

XPC services are managed by `launchd` and provide services to a single application. They are typically used to divide an application into smaller parts. This can be used to improve reliability by limiting the impact if a process crashes, and to improve security by limiting the impact if a process is compromised.

With traditional single-executable applications, if any part of the application crashes, the entire application terminates. By restructuring the application into a main process and services, the impact of a crash in a service is significantly less. The user can continue to work; the service that crashed gets relaunched. For example, an email application can use an XPC service to handle communication with the mail server. Even if the service crashes, temporarily interrupting communication with the server, the rest of the application remains usable.

Sandboxing allows you to specify a list of things your program is expected to do during normal operation. The operating system enforces that list, limiting the damage that can by done by an attacker. For example, a text editor might need to edit files on disk that have been opened by the user, but it probably does not need to open arbitrary files in other locations or communicate over the network.

You can combine sandboxing with XPC services to provide privilege separation by splitting a complex application, tool, or daemon into smaller pieces with well-defined functionality. Because of the reduced privileges of of each individual piece, any flaws are less exploitable by an attacker: none of the pieces run with

the full capabilities of the user. For example, an application that organizes and edits photographs does not usually need network access. However, if it also allows users to upload photos to a photo sharing website, that functionality can be implemented as an XPC service with network access and mediated access (or no access) to the file-system.

For information about how to create an XPC service, see Creating XPC Services (page 20).

Launch Daemons

Daemons are managed by `launchd` on behalf of the OS in the system context, which means they are unaware of the users logged on to the system. A daemon cannot initiate contact with a user process directly; it can only respond to requests made by user processes. Because they have no knowledge of users, daemons also have no access to the window server, and thus no ability to post a visual interface or launch a GUI application. Daemons are strictly background processes that respond to low-level requests.

Most daemons run in the system context of the system—that is, they run at the lowest level of the system and make their services available to all user sessions. Daemons at this level continue running even when no users are logged into the system, so the daemon program should have no direct knowledge of users. Instead, the daemon must wait for a user program to contact it and make a request. As part of that request, the user program usually tells the daemon how to return any results.

For information about how to create a launch daemon, see Creating Launch Daemons and Agents (page 39).

Launch Agents

Agents are managed by `launchd`, but are run on behalf of the currently logged-in user (that is, in the user context). Agents can communicate with other processes in the same user session and with system-wide daemons in the system context. They can display a visual interface, but this is not recommended.

If your code provides both user-specific and user-independent services, you might want to create both a daemon and an agent. Your daemon would run in the system context and provide the user-independent services while an instance of your agent would run in each user session. The agents would coordinate with the daemon to provide the services to each user.

For information about how to create a launch daemon, see Creating Launch Daemons and Agents (page 39).

Protocols for Communicating with Daemons

There are four major communication mechanisms commonly used between daemons and their clients: XPC, traditional client-server communications (including Apple events, TCP/IP, UDP, other socket and pipe mechanisms), remote procedure calls (including Mach RPC, Sun RPC, and Distributed Objects), and memory mapping (used underneath the Core Graphics APIs, among others).

XPC is the easiest way to launch and communicate with your daemon. For details on implementing this mechanism, see Creating XPC Services (page 20) and *XPC Services API Reference*.

Other RPC (remote procedure call) mechanisms such as Distributed Objects should be avoided for communication across security domain boundaries, for example a user process communicating with a system-level daemon, because this creates a security risk. They are appropriate only when you can be certain that both processes involved have the same level of privileges.

In most other cases, you should use a traditional client-server communication API. Code based on these APIs tends to be easier to understand, debug, and maintain than RPC or memory mapping designs. It also tends to be more portable to other platforms than RPC-based code. For details on implementing using TCP/IP, read *Networking Overview*.

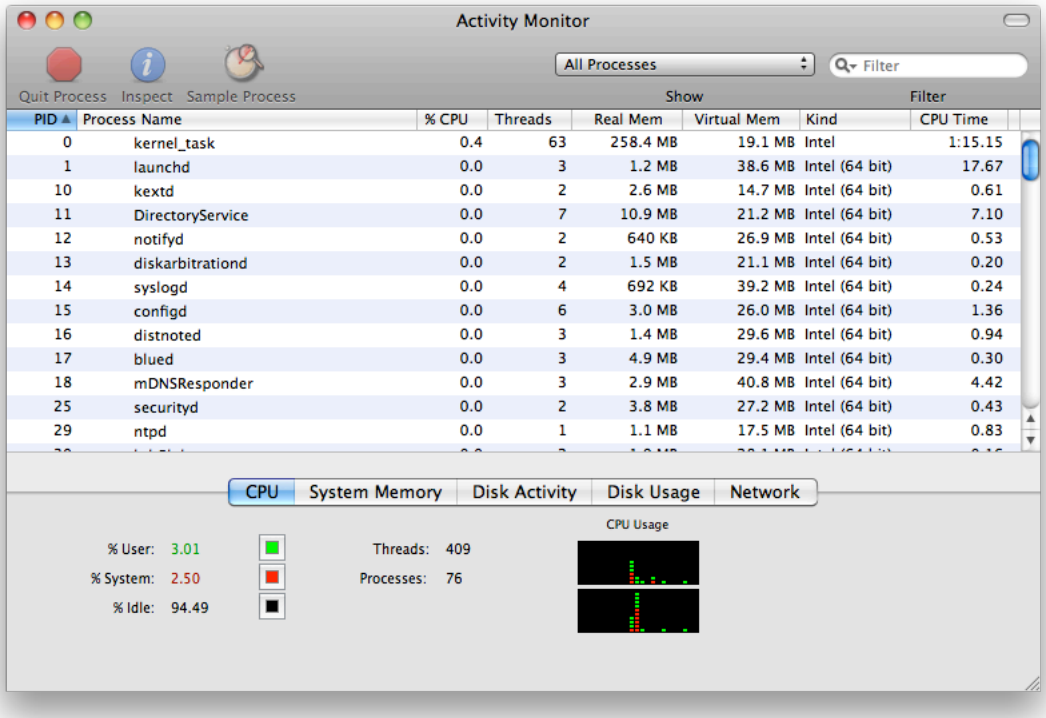
Memory mapping requires complex management and represents a security risk if you are not careful about what memory pages you share or if you do not sufficiently validate the shared data. You should use memory mapping only if your client and daemon require a large amount of shared state with low latency, such as passing audio or video in real time. For details on implementing this mechanism, see *SharedMemory*.

Viewing the Currently Running Daemons

If you want to see the daemons currently running on your system, use the Activity Monitor application (located in `/Applications/Utilities`). This application lets you view information about all processes including their resource usage. Figure 1-1 shows the Activity Monitor window and the process information.

Note: If you want to know more about the services provided by a particular daemon, consult the `man` page for that daemon. You can also view the manual pages online by reading *OS X Man Pages*.

Figure 1-1 Processes shown in Activity Monitor



The Life Cycle of a Daemon

Starting the User Environment

The root process on OS X is `launchd` (which replaces the `mach_init` and `init` processes used in previous versions of OS X and many traditional Unix systems). In addition to initializing the system, the `launchd` process coordinates the launching of system daemons in an orderly manner. Like the `inetd` process, `launchd` launches daemons on-demand. Daemons launched in this manner can be shut down during periods of inactivity and relaunched as needed. When a service request comes in, if the daemon is not running, `launchd` automatically starts the daemon to process the request.

Launching daemons on demand frees up memory and other resources associated with the daemon, which is worthwhile if the daemon is likely to be idle for extended periods of time. More importantly, however, this guarantees that runtime dependencies between daemons are satisfied without the need for manual lists of dependencies.

As the final part of system initialization, `launchd` launches `loginwindow`. The `loginwindow` program controls several aspects of user sessions and coordinates the display of the login window and the authentication of users.

For information about earlier portions of the boot process, see The Early Boot Process in *Kernel Programming Guide*.

Authenticating Users

OS X requires users to authenticate themselves prior to accessing the system. The `loginwindow` program coordinates the visual portion of the login process (as manifested by the window where users enter name and password information) and the security portion (which handles the user authentication). Once a user has been authenticated, `loginwindow` begins setting up the user environment.

In two key situations, `loginwindow` bypasses the usual login prompt and begins the user session immediately. The first situation occurs when the system administrator has configured the computer to automatically log in as a specified user. The second occurs during software installation when the installer program is to be launched immediately after a reboot.

Configuring User Sessions

Immediately after the user is successfully authenticated, `loginwindow` sets up the user environment and records information about the login. As part of this process, it performs the following tasks:

- Secures the login session from unauthorized remote access.
- Records the login in the system's `utmp` and `utmpx` databases.
- Sets the owner and permissions for the console terminal.
- Resets the user's preferences to include global system defaults.
- Configures the mouse, keyboard, and system sound using the user's preferences.
- Sets the user's group permissions (`gid`).
- Retrieves the user record from Directory Services and applies that information to the session.
- Loads the user's computing environment (including preferences, environment variables, device and file permissions, keychain access, and so on).
- Launches the Dock, Finder, and SystemUIServer.
- Launches the login items for the user.

Once the user session is up and running, `loginwindow` monitors the session and user applications in the following ways:

- It manages the logout, restart, and shutdown procedures. See Logout Responsibilities (page 15) for more information.
- It manages the Force Quit window, which includes monitoring the currently active applications and responding to user requests to force-quit applications and relaunch the Finder. (Users open this window from the Apple menu or by pressing Command-Option-Escape.)
- It arranges for any output on standard error to be logged using the ASL API. (See Log Messages Using the ASL API (page 55).)

Logout Responsibilities

The procedures for logging out, restarting the system, or shutting down the system are similar. A typical logout/restart/shutdown takes place as follows:

1. The user selects Log Out, Restart, or Shut Down from the Apple menu.
2. The foreground application initiates the user request by sending an Apple event to `loginwindow`. (See Initiating a Logout, Restart, or Shutdown (page 17) for a list of events.) For Cocoa applications, this is done by the Application Kit.

3. The `loginwindow` program displays an alert to the user asking for confirmation of the action.
4. If the user confirms the action, `loginwindow` quits every foreground and background user process.
5. Once all processes have quit, `loginwindow` ends the user session and performs the logout, restart or shutdown.

Terminating Processes

As part of a log out, restart, or shutdown sequence, `loginwindow` attempts to terminate all foreground and background user processes.

Your process should support sudden termination for the best user experience. See *NSProcessInfo Class Reference* for information on how to adopt this technology. If your process supports sudden termination, it is just sent a `SIGKILL` signal. If you have temporarily disabled sudden termination, the normal process below applies.

For Cocoa applications, termination is partly handled by the Application Kit, which calls the `applicationShouldTerminate:` delegate method. To abort the termination sequence, implement this method and return `NSTerminateCancel`; otherwise, termination of your application continues normally.

Non-Cocoa applications receive a "Quit Application" Apple event (`kAEQuitApplication`), as a courtesy, to give them a chance to shut down gracefully. The process should terminate itself immediately or post an alert dialog if a user confirmation is required (for example, if there is an unsaved document). As soon as that condition is resolved, the application should terminate. If the user decides to abort the termination sequence (by clicking Cancel in a Save dialog, for example) the application should respond to the event by returning a `userCanceledErr` error (`–128`).

If a foreground application fails to reply or terminate itself after 45 seconds, `loginwindow` automatically aborts the termination sequence. This safeguard is to protect data in various situations, such as when an application is saving a large file to disk and is unable to terminate in the allotted time. If a foreground application is unresponsive and not doing anything, the user must use the Force Quit window to quit it before proceeding.

For background processes, the procedure is a little different. The `loginwindow` program notifies the process that it is about to be terminated by sending it a Quit Application Apple event (`kAEQuitApplication`). Unlike foreground processes, however, `loginwindow` does not wait for a reply. It proceeds to terminate any open background processes by sending a `SIGKILL` signal, regardless of any returned errors.

If the system is being shut down or restarted, it sends a `SIGTERM` signal to all daemons, followed a few seconds later by `SIGKILL` signal.

Initiating a Logout, Restart, or Shutdown

To initiate a logout, restart, or shutdown sequence programmatically, the foreground application must send the appropriate Apple event to `loginwindow`. Upon receipt of the event, `loginwindow` begins the process of shutting down the user session.

The following list shows the preferred Apple events for logout, restart, and shutdown procedures. These events have no required parameters.

- `kAELogOut`
- `kAEShowRestartDialog`
- `kAEShowShutdownDialog`

In addition to the preferred Apple events, there are two additional events that tell `loginwindow` to proceed immediately with a restart or shutdown sequence:

Warning: If you send one of these events to `loginwindow`, the user does not have an opportunity to cancel the action, and unsaved data can be lost. These events should be used very seldom, if at all.

- `kAERestart`
- `kAEShutDown`

Adding Login Items

There are two ways to add a login item: using the Service Management framework, and using a shared file list

Login items installed using the Service Management framework are not visible in System Preferences and can only be removed by the application that installed them.

Login items installed using a shared file list are visible in System Preferences; users have direct control over them. If you use this API, your login item can be disabled by the user, so any other application that communicates with it it should have reasonable fallback behavior in case the login item is disabled.

Adding Login Items Using the Service Management Framework

Applications can contain a helper application as a full application bundle, stored inside the main application bundle in the `Contents/Library/LoginItems` directory. Set either the `LSUIElement` or `LSBackgroundOnly` key in the `Info.plist` file of the helper application's bundle.

Use the `SMLoginItemSetEnabled` function (available in OS X v10.6.6 and later) to enable a helper application. It takes two arguments, a `CFStringRef` containing the bundle identifier of the helper application, and a `Boolean` specifying the desired state. Pass `true` to start the helper application immediately and indicate that it should be started every time the user logs in. Pass `false` to terminate the helper application and indicate that it should no longer be launched when the user logs in. This function returns `true` if the requested change has taken effect; otherwise, it returns `false`. This function can be used to manage any number of helper applications.

If multiple applications (for example, several applications from the same company) contain a helper application with the same bundle identifier, only the one with the greatest bundle version number is launched. Any of the applications that contain a copy of the helper application can enable and disable it.

Adding Login Items Using a Shared File List

This method is available in OS X v10.5 and later. For specific details, see the appropriate functions in *Launch Services Reference*.

Deprecated APIs

In previous versions of OS X, it is possible to add login items by sending an Apple event, by using the CFPreferences API, and by manually editing a property list file. These approaches are deprecated.

If you need to maintain compatibility with versions of OS X prior to v10.5, the preferred approach is to use Apple events; for details, see *LoginItemsAE*. Using the CFPreferences API is an acceptable alternative. You should not directly edit the property list file on any version of OS X.

Creating XPC Services

The XPC Services API, part of libSystem, provides a lightweight mechanism for basic interprocess communication integrated with Grand Central Dispatch (GCD) and `launchd`. The XPC Services API allows you to create lightweight helper tools, called XPC services, that perform work on behalf of your application.

There are two main reasons to use XPC services: privilege separation and stability.

Stability:

Let's face it; applications sometimes crash. We don't want it to happen, but it does anyway. Often, certain parts of an application are more prone to crashes than others. For example, the stability of any application with a plug-in API is inherently at the mercy of the authors of plug-ins.

When one part of an application is more at risk for crashes, it can be useful to separate out the potentially unstable functionality from the core of the application. This separation lets you ensure that that, if it crashes, the rest of the application is not affected.

Privilege Separation:

Modern applications increasingly rely on untrusted data, such as web pages, files sent by email, and so on. This represents a growing attack vector for viruses and other malware.

With traditional applications, if an application becomes compromised through a buffer overflow or other security vulnerability, the attacker gains the ability to do anything that the user can do. To mitigate this risk, Mac OS X provides sandboxing—limiting what types of operations a process can perform.

In a sandboxed environment, you can further increase security with privilege separation—dividing an application into smaller pieces that are responsible for a part of the application's behavior. This allows each piece to have a more restrictive sandbox than the application as a whole would require.

Other mechanisms for dividing an application into smaller parts, such as `NSTask` and `posix_spawn`, do not let you put each part of the application in its own sandbox, so it is not possible to use them to implement privilege separation. Each XPC service has its own sandbox, so XPC services can make it easier to implement proper privilege separation.

For more information about sandboxing, see *App Sandbox Design Guide*.

Understanding the Structure and Behavior

XPC services are managed by `launchd`, which launches them on demand, restarts them if they crash, and terminates them (by sending `SIGKILL`) when they are idle. This is transparent to the application using the service, except for the case of a service that crashes while processing a message that requires a response. In that case, the application can see that its XPC connection has become invalid until the service is restarted by `launchd`. Because an XPC service can be terminated suddenly at any time, it must be designed to hold on to minimal state—ideally, your service should be completely stateless, although this is not always possible.

By default, XPC services are run in the most restricted environment possible—sandboxed with minimal filesystem access, network access, and so on. Elevating a service's privileges to root is not supported. Further, an XPC service is private, and is available only to the main application that contains it.

Choosing an XPC API

Beginning in Mac OS X v10.8, there are two different APIs for working with XPC services: the XPC Services API and the `NSXPCConnection` API.

The `NSXPCConnection` API is an Objective-C-based API that provides a remote procedure call mechanism, allowing the client application to call methods on proxy objects that transparently relay those calls to corresponding objects in the service helper and vice-versa.

The XPC Services API is a C-based API that provides basic messaging services between a client application and a service helper.

The XPC Services API is recommended if you need compatibility with Mac OS X v10.7, or if your application and its service are not based on the Foundation framework. The `NSXPCConnection` API is recommended for apps based on the Foundation framework in Mac OS X v10.8 and later.

The NSXPCConnection API

The `NSXPCConnection` API is part of the Foundation framework, and is described in the `NSXPCConnection.h` header file. It consists of the following pieces:

- `NSXPCConnection`—a class that represents the bidirectional communication channel between two processes. Both the application and the service helper have at least one connection object.
- `NSXPCInterface`—a class that describes the expected programmatic behavior of the connection (what classes can be transmitted across the connection, what objects are exposed, and so on).
- `NSXPCListener`—a class that listens for incoming XPC connections. Your service helper must create one of these and assign it a delegate object that conforms to the `NSXPCListenerDelegate` protocol.

- `NSXPCListenerEndpoint`—a class that uniquely identifies an `NSXPCListener` instance and can be sent to a remote process using `NSXPCConnection`. This allows a process to construct a direct communication channel between two other processes that otherwise could not see one another.

In addition, the header contains a few constants, described in *NSXPCConnection Constants Reference*.

The XPC Services API

The (C-based) XPC Services API consists of two main pieces:

- `xpc.h`—APIs for creating and manipulating property list objects and APIs that daemons use to reply to messages.

  Because this API is at the libSystem level, it cannot depend on Core Foundation. Also, not all CF types can be readily shared across process boundaries. XPC supports connection types such as file descriptors in its object graphs, which are not supported by CFPropertyList because it was designed as a persistent storage format, whereas XPC was designed as an IPC format. For these reasons, the XPC API uses its own container that supports only the primitives that are practical to transport across process boundaries.

  Some higher-level types can be passed across an XPC connection, although they do not appear in the `xpc.h` header file (because referencing higher level frameworks from libSystem would be a layering violation). For example, the `IOSurfaceCreateXPCObject` and `IOSurfaceLookupFromXPCObject` functions allow you to pass an IOSurface object between the XPC service that does the drawing and the main application.

- `connection.h`—APIs for connecting to an XPC service. This service is a special helper bundle embedded in your app bundle.

  A connection is a virtual endpoint; it is independent of whether an actual instance of the service binary is running. The service binary is launched on demand.

  A connection can also be sent as a piece of data in an XPC message. Thus, you can pass a connection through XPC to allow one service to communicate with another service (for example).

Note: The underlying encoding used by XPC is opaque to the user, and so is the communication channel. You should not attempt to interact directly with either, as they are subject to change without notice.

You should also not try to archive the bytes of a message or the objects that contain it to disk; the encoding is not considered an ABI contract, and may change at any time.

Creating the Service

An XPC service is a bundle in the `Contents/XPCServices` directory of the main application bundle; the XPC service bundle contains an `Info.plist` file, an executable, and any resources needed by the service. The XPC service indicates which function to call when the service receives messages by calling `xpc_main(3) Mac OS X Developer Tools Manual Page` from its `main` function.

To create an XPC service in Xcode, do the following:

1. Add a new target to your project, using the XPC Service template.
2. Add a Copy Files phase to your application's build settings, which copies the XPC service into the `Contents/XPCServices` directory of the main application bundle.
3. Add a dependency to your application's build settings, to indicate it depends on the XPC service bundle.
4. If you are writing a low-level (C-based) XPC service, implement a minimal `main` function to register your event handler, as shown in the following code listing. Replace `my_event_handler` with the name of your event handler function.

```
int main(int argc, const char *argv[]) {
    xpc_main(my_event_handler);

    // The xpc_main() function never returns.
    exit(EXIT_FAILURE);
}
```

If you are writing a high-level (Objective-C-based) service using `NSXPCConnection`, first create a connection delegate class that conforms to the `NSXPCListenerDelegate` protocol. Then, implement a minimal `main` function that creates and configures a listener object, as shown in the following code listing.

```
int main(int argc, const char *argv[]) {
    MyDelegateClass *myDelegate = ...
```

```
    NSXPCListener *listener =
        [NSXPCListener serviceListener];

    listener.delegate = myDelegate;
    [listener resume];

    // The resume method never returns.
    exit(EXIT_FAILURE);
}
```

The details of this class are described further in Using the Service (page 24).

5. Add the appropriate key/value pairs to the helper's `Info.plist` to tell launchd the name of the service. These are described in XPC Service Property List Keys (page 37).

Note: If you want to sandbox your service, it must also be signed; entitlements are stored as part of the code signature.

Using the Service

The way you use an XPC service depends on whether you are working with the C API (XPC Services) or the Objective-C API (`NSXPCConnection`).

Using the Objective-C NSXPCConnection API

The Objective-C `NSXPCConnection` API provides a high-level remote procedure call interface that allows you to call methods on objects in one process from another process (usually an application calling a method in an XPC service). The `NSXPCConnection` API automatically serializes data structures and objects for transmission and deserializes them on the other end. As a result, calling a method on a remote object behaves much like calling a method on a local object.

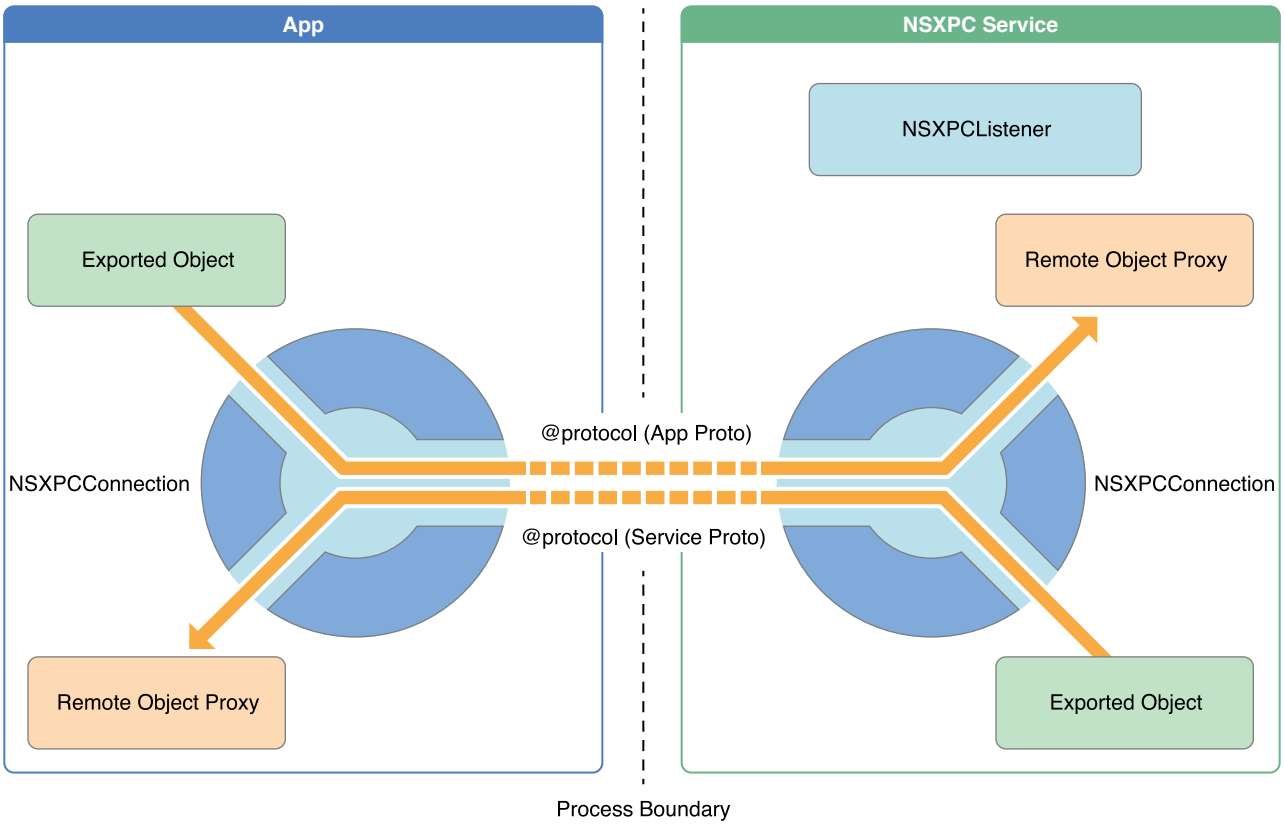
To use the `NSXPCConnection` API, you must create the following:

- An interface. This mainly consists of a protocol that describes what methods should be callable from the remote process. This is described in Designing an Interface (page 26)
- A connection object on both sides. On the service side, this was described previously in Creating the Service (page 23). On the client side, this is described in Connecting to and Using an Interface (page 27).

- A listener. This code in the XPC service accepts connections. This is described in Accepting a Connection in the Helper (page 29).
- Messages.

Figure 4-1 shows how these pieces fit together.

Figure 4-1 The NSXPC architecture



In some cases, you may need to further tweak the protocol to whitelist additional classes for use in collection parameters or to proxy certain objects instead of copying them. This is described further in Working with Custom Classes (page 30).

Overall Architecture

When working with `NSXPCConnection`-based helper apps, both the main application and the helper have an instance of `NSXPCConnection`. The main application creates its connection object itself, which causes the helper to launch. A delegate method in the helper gets passed its connection object when the connection is established. This is illustrated in Figure 4-1.

Each `NSXPCConnection` object provides three key features:

- An `exportedInterface` property that describes the methods that should be made available to the opposite side of the connection.
- An `exportedObject` property that contains a local object to handle method calls coming in from the other side of the connection.
- The ability to obtain a proxy object for calling methods on the other side of the connection.

When the main application calls a method on a proxy object, the XPC service's `NSXPCConnection` object calls that method on the object stored in its `exportedObject` property.

Similarly, if the XPC service obtains a proxy object and calls a method on that object, the main app's `NSXPCConnection` object calls that method on the object stored in its `exportedObject` property.

Designing an Interface

The `NSXPCConnection` API takes advantage of Objective-C protocols to define the programmatic interface between the calling application and the service. Any instance method that you want to call from the opposite side of a connection must be explicitly defined in a formal protocol. For example:

```
@protocol FeedMeACookie
    - (void)feedMeACookie: (Cookie *)cookie;
@end
```

Because communication over XPC is asynchronous, all methods in the protocol must have a return type of `void`. If you need to return data, you can define a reply block like this:

```
@protocol FeedMeAWatermelon
    - (void)feedMeAWatermelon: (Watermelon *)watermelon
        reply:(void (^)(Rind *))reply;
@end
```

A method can have only one reply block. However, because connections are bidirectional, the XPC service helper can also reply by calling methods in the interface provided by the main application, if desired.

Each method must have a return type of `void`, and all parameters to methods or reply blocks must be either:

- Arithmetic types (`int`, `char`, `float`, `double`, `uint64_t`, `NSUInteger`, and so on)
- `BOOL`
- C strings

- C structures and arrays containing only the types listed above
- Objective-C objects that implement the `NSSecureCoding` protocol.

Important: If a method (or its reply block) has parameters that are Objective-C collection classes (`NSDictionary`, `NSArray`, and so on), and if you need to pass your own custom objects within a collection, you must explicitly tell XPC to allow that class as a member of that collection parameter. For details, read Working with Custom Classes (page 30).

Connecting to and Using an Interface

Once you have defined the protocol, you must create an interface object that describes it. To do this, call the `interfaceWithProtocol:` method on the `NSXPCInterface` class. For example:

```
NSXPCInterface *myCookieInterface =
    [NSXPCInterface interfaceWithProtocol:
        @protocol(FeedMeACookie)];
```

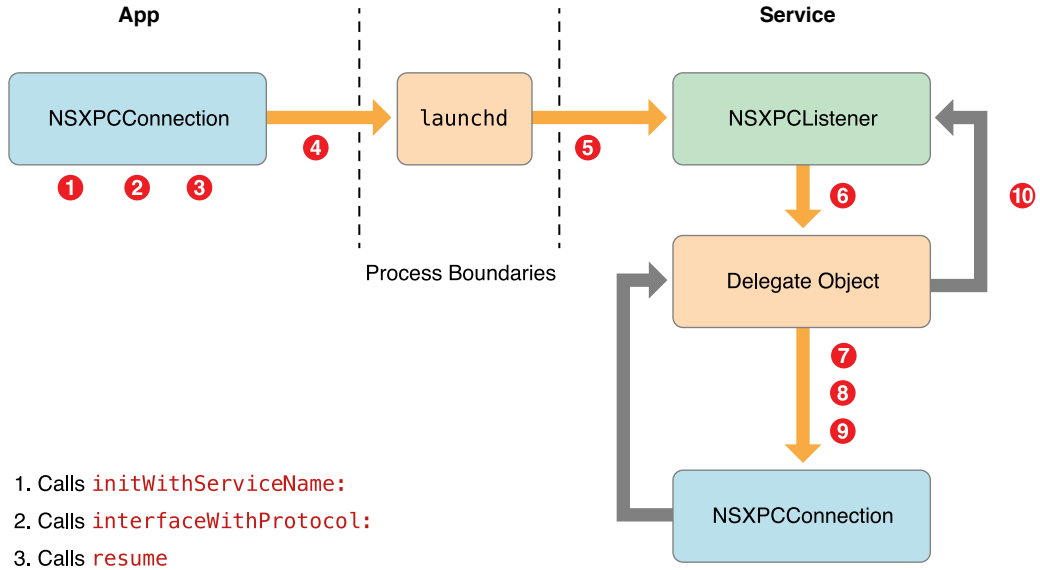
Once you have created the interface object, within the main app, you must configure a connection with it by calling the `initWithServiceName:` method. For example:

```
NSXPCConnection *myConnection =    [[NSXPCConnection alloc]
     initWithServiceName:@"com.example.monster"];
myConnection.remoteObjectInterface = myCookieInterface;
[myConnection resume];
```

Note: For communicating with XPC services outside your app bundle, you can also configure an XPC connection with the `initWithMachServiceName:` method. For details, see the documentation for that method.

Figure 4-2 shows the basic steps in this process. Note that only the first four steps are described in this section.

Figure 4-2 The NSXPC connection process



1. Calls `initWithServiceName:`
2. Calls `interfaceWithProtocol:`
3. Calls `resume`
4. Connects to `launchd`
5. `launchd` starts service
6. Listener object calls `listener:shouldAcceptNewConnection:` on delegate
7. Sets `exportedObject`
8. Sets `exportedInterface`
9. Calls `resume`
10. Returns `YES`

At this point, the main application can call the `remoteObjectProxy` or `remoteObjectProxyWithErrorHandler:` methods on the `myConnection` object to obtain a proxy object. This object acts as a proxy for the object that the XPC service has set as its exported object (by setting the `exportedObject` property). This object must conform to the protocol defined by the `remoteObjectInterface` property.

When your application calls a method on the proxy object, the corresponding method is called on the exported object inside the XPC service. When the service's method calls the reply block, the parameter values are serialized and sent back to the application, where the parameter values are deserialized and passed to the reply block. (The reply block executes within the application's address space.)

> **Note:** If you want to allow the helper process to call methods on an object in your application, you must set the `exportedInterface` and `exportedObject` properties before calling `resume`. These properties are described further in the next section.

Accepting a Connection in the Helper

As shown in Figure 4-2, when an `NSXPCConnection`-based helper receives the first message from a connection, the listener delegate's `listener:shouldAcceptNewConnection:` method is called with a listener object and a connection object. This method lets you decide whether to accept the connection or not; it should return `YES` to accept the connection or `NO` to refuse the connection.

> **Note:** The helper receives a connection request when the first actual message is sent. The connection object's `resume` method does not cause a message to be sent.

In addition to making policy decisions, this method must configure the connection object. In particular, assuming the helper decides to accept the connection, it must set the following properties on the connection:

- `exportedInterface`—an interface object that describes the protocol for the object you want to export. (Creating this object was described previously in Connecting to and Using an Interface (page 27).)
- `exportedObject`—the local object (usually in the helper) to which the remote client's method calls should be delivered. Whenever the opposite end of the connection (usually in the application) calls a method on the connection's proxy object, the corresponding method is called on the object specified by the `exportedObject` property.

After setting those properties, it should call the connection object's `resume` method before returning `YES`. Although the delegate may defer calling `resume`, the connection will not receive any messages until it does so.

Sending Messages

Sending messages with NSXPC is as simple as making a method call. For example, given the interface `myCookieInterface` (described in previous sections) on the XPC connection object `myConnection`, you can call the `feedMeACookie` method like this:

```
Cookie *myCookie = ...

[[myConnection remoteObjectProxy] feedMeACookie: myCookie];
```

When you call that method, the corresponding method in the XPC helper is called automatically. That method, in turn, could use the XPC helper's connection object similarly to call a method on the object exported by the main application.

Handling Errors

In addition to any error handling methods specific to a given helper's task, both the XPC service and the main app should also provide the following XPC error handler blocks:

- Interruption handler—called when the process on the other end of the connection has crashed or has otherwise closed its connection.

  The local connection object is typically still valid—any future call will automatically spawn a new helper instance unless it is impossible to do so—but you may need to reset any state that the helper would otherwise have kept.

  The handler is invoked on the same queue as reply messages and other handlers, and it is always executed after any other messages or reply block handlers (except for the invalidation handler). It is safe to make new requests on the connection from an interruption handler.

- Invalidation handler—called when the invalidate method is called or when an XPC helper could not be started. When this handler is called, the local connection object is no longer valid and must be recreated.

  This is always the last handler called on a connection object. When this block is called, the connection object has been torn down. It is not possible to send further messages on the connection at that point, whether inside the handler or elsewhere in your code.

In both cases, you should use block-scoped variables to provide enough contextual information—perhaps a pending operation queue and the connection object itself—so that your handler code can do something sensible, such as retrying pending operations, tearing down the connection, displaying an error dialog, or whatever other actions make sense in your particular app.

Working with Custom Classes

> **Note:** Before you read this section, you should read the chapters Serializations and Serializing Property Lists in *Archives and Serializations Programming Guide* to learn the basics of object serialization in Mac OS X.

The `NSXPCConnection` class limits what objects can be passed over a connection. By default, it allows only known-safe classes—Foundation collection classes, `NSString`, and so on. You can identify these classes by whether they conform to the `NSSecureCoding` protocol.

Only classes that conform to this protocol can be sent to an `NSXPCConnection`-based helper. If you need to pass your own classes as parameters, you must ensure that they conform to the `NSSecureCoding` protocol, as described below.

However, this is not always sufficient. You need to do extra work in two situations:

- If you are passing the object inside a collection (dictionary, array, and so on).
- If you need to pass the object by proxy instead of copying the object.

All three cases are described in the sections that follow.

Conforming to NSSecureCoding

All objects passed over an NSXPC connection must conform to `NSSecureCoding`. to do this, your class must do the following:

- **Declare support for secure coding.** Override the `supportsSecureCoding` method, and make it return `YES`.
- **Decode singleton class instances safely.** If the class overrides its `initWithCoder:` method, when decoding any instance variable, property, or other value that contains an object of a non-collection class (including custom classes) always use `decodeObjectOfClass:forKey:` to ensure that the data is of the expected type.
- **Decode collection classes safely.** Any non-collection class that contains instances of collection classes *must* override the `initWithCoder:` method. In that method, when decoding the collection object or objects, always use `decodeObjectOfClasses:forKey:` and provide a list of any objects that can appear within the collection.

When generating the list of classes to allow within a decoded collection class, you should be aware of two things.

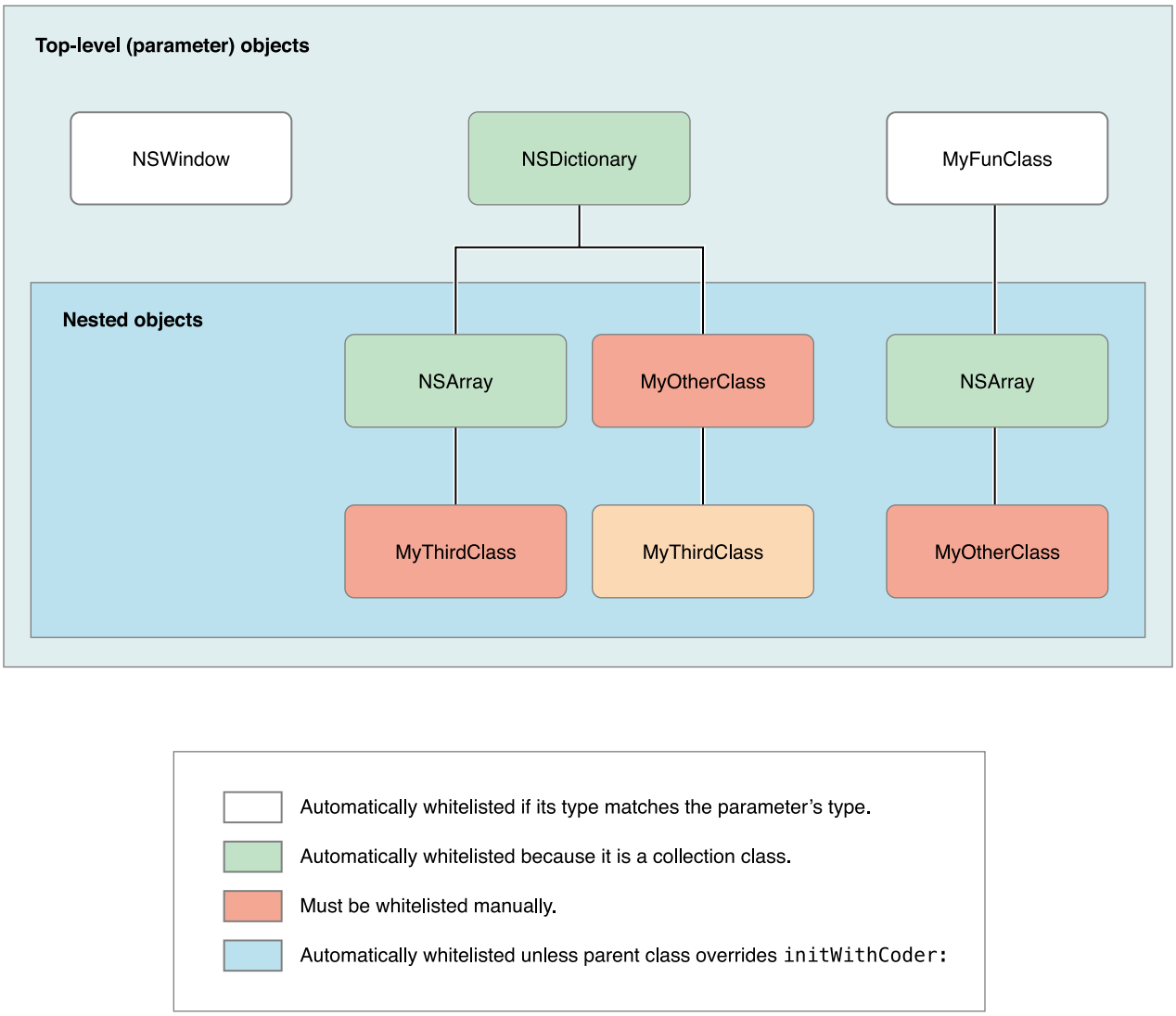
First, Apple collection classes are *not* automatically whitelisted by the `decodeObjectOfClasses:forKey:` method, so you must include them explicitly in the array of class types.

Second, you should list only classes that are direct members of the collection object graph that you are decoding without any intervening non-collection classes.

For example, if you have an array of dictionaries, and one of those dictionaries might contain an instance of a custom class called `OuterClass`, and `OuterClass` has an instance variable of type InnerClass, you must include `OuterClass` in the list of classes because it is a direct member of the collection tree. However, you do not need to list `InnerClass` because there is a non-collection object between it and the collection tree.

Figure 4-3 (page 32) shows some examples of when whitelisting is required and shows when classes must provide overridden `initWithCoder:` methods.

Figure 4-3 Whitelisting and secure coding examples



Whitelisting a Class for Use Inside Containers

Most of the time, custom classes passed through method parameters can be automatically whitelisted based upon the method's signature. However, when a method specifies a collection class (`NSArray`, `NSDictionary`, and so on) as the parameter type, the compiler has no way to determine what classes should be allowed to appear within that container. For this reason, if your methods take collection class instances as parameters, you must explicitly whitelist any classes that can appear within those containers.

For every method in your interface that takes a collection class as a parameter, you must determine what classes should be allowed as members.

You should whitelist *only* classes that can be members of any top-level collection objects. If you whitelist classes at the top level unnecessarily, those objects are allowed to appear within the top-level collection objects, which is not what you want. In particular:

- Apple-provided classes that support property list serialization (such as other collection classes) are automatically whitelisted for you. It is never necessary to whitelist these classes at the top level.

  For the most up-to-date list of classes supported by property list serialization, read Serializing Property Lists in *Archives and Serializations Programming Guide*.
- Don't whitelist custom classes if they are allowed only inside instances of other custom classes. If the enclosing class correctly conforms to `NSSecureCoding`, such whitelisting is not required. For more details, read Conforming to NSSecureCoding (page 31).

For example, suppose you have the following class and protocol:

```
@interface FrenchFry
...
@end

@protocol FeedMeABurgerAndFries
    - (void)feedMeFries: (NSArray *)pommesFrites;
    - (void)feedMeABurger: (Burger *)aBurger;
@end
```

Assuming the `pommesFrites` array is an array of `FrenchFry` objects (as the name implies), you would whitelist the array of `FrenchFry` objects as follows:

```
// Create the interface
NSXPCInterface *myBurgerInterface =
    [NSXPCInterface interfaceWithProtocol:
        @protocol(FeedMeABurgerAndFries)];

// Create a set containing the allowed
// classes.
NSSet *expectedClasses =
    [NSSet setWithObjects:[NSArray class], [FrenchFry class], nil];
```

```
[myBurgerInterface
      setClasses: expectedClasses
     forSelector: @selector(feedMeFries:)
   argumentIndex: 0  // the first parameter
         ofReply: NO // in the method itself.
];
```

The first parameter to a method is parameter 0, followed by parameter 1, and so on.

In this case, the value `NO` is passed for the `ofReply` parameter because this code is modifying the whitelist for one of the parameters of the method itself. If you are whitelisting a class for a parameter of the method's reply block, pass `YES` instead.

Passing an Object By Proxy

Most of the time, it makes sense to copy objects and send them to the other side of a connection. Passing objects by copying is the most straightforward way to use NSXPC, and should be used wherever possible.

However, copying objects is not always desirable. In particular:

- If you need to share a single instance of the data between the client application and the helper, you must pass the objects by proxy.
- If an object needs to call methods on other objects within your application that you cannot or do not wish to pass across the connection (such as user interface objects), then you must pass an object by proxy—either the caller, the callee (where possible), or a relay object that you construct specifically for that purpose.

The downside to passing objects by proxy is that performance is significantly reduced (because every access to the object requires interprocess communication). For this reason, you should pass objects by proxy *only* if it is not possible to pass them by copying.

You can configure additional proxy objects similarly to the way you configured the `remoteObjectInterface` property of the initial connection. First, identify which parameter to a method should be passed by proxy, then specify an `NSXPCInterface` object that defines the interface for that object.

Suppose, for example, that you have a class conforming to the following protocol:

```
@protocol FeedSomeone
    - (void)feedSomeone:
```

```
        (id <FeedMeACookie>)someone;
@end

...

NSXPCInterface *myFeedingInterface =
    [NSXPCInterface interfaceWithProtocol:
        @protocol(FeedSomeone)];
```

If you want to pass the first parameter to that method by proxy, you would configure the interface like this:

```
// Create an interface object that describes
// the protocol for the object you want to
// pass by proxy.
NSXPCInterface *myCookieInterface =
    [NSXPCInterface interfaceWithProtocol:
        @protocol(FeedMeACookie)];

// Create an object of a class that
// conforms to the FeedMeACookie protocol
Monster *myMonster = ...

[myFeedingInterface
     setInterface: myCookieInterface
      forSelector: @selector(sendOtherProxy:)
    argumentIndex: 0  // the first parameter of
          ofReply: NO // the feedSomeone: method
];
```

The first parameter to a method is parameter 0, followed by parameter 1, and so on.

In this case, the value `NO` is passed for the `ofReply` parameter because this code is modifying the whitelist for one of the parameters of the method itself. If you are whitelisting a class for a parameter of the method's reply block, pass `YES` instead.

Using the C XPC Services API

Typical program flow is as follows:

1. An application calls `xpc_connection_create(3) Mac OS X Developer Tools Manual Page` to create an an XPC connection object.
2. The application calls some combination of `xpc_connection_set_event_handler(3) Mac OS X Developer Tools Manual Page` or `xpc_connection_set_target_queue(3) Mac OS X Developer Tools Manual Page` as needed to configure connection parameters prior to actually connecting to the service.
3. The application calls `xpc_connection_resume(3) Mac OS X Developer Tools Manual Page` to begin communication.
4. The application sends messages to the service using `xpc_connection_send_message(3) Mac OS X Developer Tools Manual Page`, `xpc_connection_send_message_with_reply(3) Mac OS X Developer Tools Manual Page`, or `xpc_connection_send_message_with_reply_sync(3) Mac OS X Developer Tools Manual Page`.
5. When you send the first message, the `launchd` daemon searches your application bundle for a service bundle whose `CFBundleIdentifier` value matches the specified name, then launches that XPC service daemon on demand.
6. The event handler function (specified in the service's `Info.plist` file) is called with the message. The event handler function runs on a queue whose name is the name of the XPC service.
7. If the original message was sent using `xpc_connection_send_message_with_reply(3) Mac OS X Developer Tools Manual Page` or `xpc_connection_send_message_with_reply_sync(3) Mac OS X Developer Tools Manual Page`, the service must reply using `xpc_dictionary_create_reply(3) Mac OS X Developer Tools Manual Page`, then uses `xpc_dictionary_get_remote_connection(3) Mac OS X Developer Tools Manual Page` to obtain the client connection and `xpc_connection_send_message(3) Mac OS X Developer Tools Manual Page`, `xpc_connection_send_message_with_reply(3) Mac OS X Developer Tools Manual Page`, or `xpc_connection_send_message_with_reply_sync(3) Mac OS X Developer Tools Manual Page` to send the reply dictionary back to the application.

   The service can also send a message directly to the application with `xpc_connection_send_message(3) Mac OS X Developer Tools Manual Page`.
8. If a reply was sent by the service, the handler associated with the previous message is called upon receiving the reply. The reply can be put on a different queue than the one used for incoming messages. No serial relationship is guaranteed between reply messages and non-reply messages.

9. If an error occurs (such as the connection closing), the connection's event handler (set by a previous call to `xpc_connection_set_event_handler(3) Mac OS X Developer Tools Manual Page`) is called with an appropriate error, as are (in no particular order) the handlers for any outstanding messages that are still awaiting replies.
10. At any time, the application can call `xpc_connection_suspend(3) Mac OS X Developer Tools Manual Page` when it needs to suspend callbacks from the service. All suspend calls must be balanced with resume calls. It is not safe to release the last reference to a suspended connection.
11. Eventually, the application calls `xpc_connection_cancel(3) Mac OS X Developer Tools Manual Page` to terminate the connection.

    Note: Either side of the connection can call `xpc_connection_cancel(3) Mac OS X Developer Tools Manual Page`. There is no functional difference between the application canceling the connection and the service canceling the connection.

XPC Service Property List Keys

XPC requires you to specify a number of special key-value pairs in the `Info.plist` file within the service helper's bundle. These keys are listed below.

`CFBundleIdentifier`

String. The name of the service in reverse-DNS style (for example, `com.example.myapp.myservice`). (This value is filled in by the XPC Service template.)

`CFBundlePackageType`

String. Value must be `XPC!` to identify the bundle as an XPC service. (This value is filled in by the XPC Service template.)

`XPCService`

Dictionary. Contains the following keys:

Key	Value
`EnvironmentVariables`	Dictionary. The variables which are set in the environment of the service.

Key	Value
`JoinExistingSession`	Boolean. Indicates that your service runs in the same security session as the caller. The default value is `False`, which indicates that the service is run in a new security session. Set the value to `True` if the service needs to access to the user's keychain, the pasteboard, or other per-session resources and services.
`RunLoopType`	String. Indicates the type of run loop used for the service. The default value is `dispatch_main`, which uses the `dispatch_main(3) Mac OS X Developer Tools Manual Page` function to set up a GCD-style run loop. The other supported value is `NSRunLoop`, which uses the `NSRunLoop` class to set up a run loop.

Creating Launch Daemons and Agents

If you are developing daemons to run on OS X, it is highly recommended that you design your daemons to be `launchd` compliant. Using `launchd` provides better performance and flexibility for daemons. It also improves the ability of administrators to manage the daemons running on a given system.

If you are running per-user background processes for OS X, `launchd` is also the preferred way to start these processes. These per-user processes are referred to as user agents. A user agent is essentially identical to a daemon, but is specific to a given logged-in user and executes only while that user is logged in.

Unless otherwise noted, for the purposes of this chapter, the terms "daemon" and "agent" can be used interchangeably. Thus, the term "daemon" is used generically in this section to encompass both system-level daemons and user agents except where otherwise noted.

There are four ways to launch daemons using `launchd`. The preferred method is on-demand launching, but `launchd` can launch daemons that run continuously, and can replace `inetd`for launching `inetd`-style daemons. In addition, `launchd` can start jobs at timed intervals.

Although `launchd` supports non-launch-on-demand daemons, this use is not recommended. The `launchd` daemon was designed to remove the need for dependency ordering among daemons. If you do not make your daemon be launched on demand, you will have to handle these dependencies in another way, such as by using the legacy startup item mechanism.

Launching Custom Daemons Using launchd

With the introduction of `launchd` in OS X v10.4, an effort was made to improve the steps needed to launch and maintain daemons. What `launchd` provides is a harness for launching your daemon as needed. To client programs, the port representing your daemon's service is always available and ready to handle requests. In reality, the daemon may or may not be running. When a client sends a request to the port, `launchd` may have to launch the daemon so that it can handle the request. Once launched, the daemon can continue running or shut itself down to free up the memory and resources it holds. If a daemon shuts itself down, `launchd` once again relaunches it as needed to process requests.

In addition to the launch-on-demand feature, `launchd` provides the following benefits to daemon developers:

- Simplifies the process of making a daemon by handling many of the standard housekeeping chores normally associated with launching a daemon.

- Provides system administrators with a central place to manage daemons on the system.
- Supports `inetd`-style daemons.
- Eliminates the primary reason for running daemons as root. Because `launchd` runs as root, it can create low-numbered TCP/IP listen sockets and hand them off to the daemon.
- Simplifies error handling and dependency management for inter-daemon communication. Because daemons launch on demand, communication requests do not fail if the daemon is not launched. They are simply delayed until the daemon can launch and process them.

The launchd Startup Process

After the system is booted and the kernel is running, `launchd` is run to finish the system initialization. As part of that initialization, it goes through the following steps:

1. It loads the parameters for each launch-on-demand system-level daemon from the property list files found in `/System/Library/LaunchDaemons/` and `/Library/LaunchDaemons/`.
2. It registers the sockets and file descriptors requested by those daemons.
3. It launches any daemons that requested to be running all the time.
4. As requests for a particular service arrive, it launches the corresponding daemon and passes the request to it.
5. When the system shuts down, it sends a `SIGTERM` signal to all of the daemons that it started.

The process for per-user agents is similar. When a user logs in, a per-user `launchd` is started. It does the following:

1. It loads the parameters for each launch-on-demand user agent from the property list files found in `/System/Library/LaunchAgents`, `/Library/LaunchAgents`, and the user's individual `Library/LaunchAgents` directory.
2. It registers the sockets and file descriptors requested by those user agents.
3. It launches any user agents that requested to be running all the time.
4. As requests for a particular service arrive, it launches the corresponding user agent and passes the request to it.
5. When the user logs out, it sends a `SIGTERM` signal to all of the user agents that it started.

Because `launchd` registers the sockets and file descriptors used by all daemons before it launches any of them, daemons can be launched in any order. If a request comes in for a daemon that is not yet running, the requesting process is suspended until the target daemon finishes launching and responds.

If a daemon does not receive any requests over a specific period of time, it can choose to shut itself down and release the resources it holds. When this happens, `launchd` monitors the shutdown and makes a note to launch the daemon again when future requests arrive.

> **Important:** If your daemon shuts down too quickly after being launched, `launchd` may think it has crashed. Daemons that continue this behavior may be suspended and not launched again when future requests arrive. To avoid this behavior, do not shut down for at least 10 seconds after launch.

Creating a launchd Property List File

To run under `launchd`, you must provide a configuration property list file for your daemon. This file contains information about your daemon, including the list of sockets or file descriptors it uses to process requests. Specifying this information in a property list file lets `launchd` register the corresponding file descriptors and launch your daemon only after a request arrives for your daemon's services. Table 5-1 lists the required and recommended keys for all daemons.

The property list file is structured the same for both daemons and agents. You indicate whether it describes a daemon or agent by the directory you place it in. Property list files describing daemons are installed in `/Library/LaunchDaemons`, and those describing agents are installed in `/Library/LaunchAgents` or in the `LaunchAgents` subdirectory of an individual user's `Library` directory. (The appropriate location for executables that you launch from your job is `/usr/local/libexec`.)

Table 5-1 Required and recommended property list keys

Key	Description
`Label`	Contains a unique string that identifies your daemon to `launchd`. (required)
`ProgramArguments`	Contains the arguments used to launch your daemon. (required)
`inetdCompatibility`	Indicates that your daemon requires a separate instance per incoming connection. This causes `launchd` to behave like `inetd`, passing each daemon a single socket that is already connected to the incoming client. (required if your daemon was designed to be launched by `inetd`; otherwise, must not be included)

Key	Description
`KeepAlive`	This key specifies whether your daemon launches on-demand or must always be running. It is recommended that you design your daemon to be launched on-demand.

For more information: For a complete listing of the keys, see the `launchd.plist` manual page.

For sample configuration property lists, look at the files in `/System/Library/LaunchDaemons/`. These files are used to configure many daemons that run on OS X.

Writing a "Hello World!" launchd Job

The following simple example launches a daemon named `hello`, passing `world` as a single argument, and instructs launchd to keep the job running:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>Label</key>
    <string>com.example.hello</string>
    <key>ProgramArguments</key>
    <array>
        <string>hello</string>
        <string>world</string>
    </array>
    <key>KeepAlive</key>
    <true/>
</dict>
</plist>
```

In this example, there are three keys in the top level dictionary. The first is `Label`, which uniquely identifies the job. when. The second is `ProgramArguments` which has a value of an array of strings which represent the tokenized arguments and the program to run. The third and final key is `KeepAlive` which indicates that this job needs to be running at all times, rather than the default launch-on-demand behavior, so launchd should always try to keep this job running.

Listening on Sockets

You can also include other keys in your configuration property list file. For example, if your daemon monitors a well-known port (one of the ports listed in `/etc/services`), add a `Sockets` entry as follows:

```
<key>Sockets</key>
<dict>
    <key>Listeners</key>
    <dict>
        <key>SockServiceName</key>
        <string>bootps</string>
        <key>SockType</key>
        <string>dgram</string>
        <key>SockFamily</key>
        <string>IPv4</string>
    </dict>
</dict>
```

The string for `SockServiceName` typically comes from the leftmost column in `/etc/services`. The `SockType` is one of `dgram` (UDP) or `stream` (TCP/IP). If you need to pass a port number that is not listed in the well-known ports list, the format is the same, except the string contains a number instead of a name. For example:

```
<key>SockServiceName</key>
<string>23</string>
```

Debugging launchd Jobs

There are some options that are useful for debugging your launchd job.

The following example enables core dumps, sets standard out and error to go to a log file, and instructs launchd to temporarily increase the debug level of its logging while acting on behalf of your job (remember to adjust your syslog.conf accordingly):

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>Label</key>
    <string>com.example.sleep</string>
    <key>ProgramArguments</key>
    <array>
        <string>sleep</string>
        <string>100</string>
    </array>
    <key>StandardOutPath</key>
    <string>/var/log/myjob.log</string>
    <key>StandardErrorPath</key>
    <string>/var/log/myjob.log</string>
    <key>Debug</key>
    <true/>
    <key>SoftResourceLimits</key>
    <dict>
        <key>Core</key>
        <integer>9223372036854775807</integer>
    </dict>
    <key>HardResourceLimits</key>
    <dict>
        <key>Core</key>
        <integer>9223372036854775807</integer>
    </dict>
</dict>
</plist>
```

Running a Job Periodically

The following example creates a job that is run every five minutes (300 seconds):

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>Label</key>
    <string>com.example.touchsomefile</string>
    <key>ProgramArguments</key>
    <array>
        <string>touch</string>
        <string>/tmp/helloworld</string>
    </array>
    <key>StartInterval</key>
    <integer>300</integer>
</dict>
</plist>
```

Alternately, you can specify a calendar-based interval. The following example starts the job on the 7th day of every month at 13:45 (1:45 pm). Like the Unix cron subsystem, any missing key of the `StartCalendarInterval` dictionary is treated as a wildcard—in this case, the month is omitted, so the job is run every month.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>Label</key>
    <string>com.example.touchsomefile</string>
    <key>ProgramArguments</key>
    <array>
        <string>touch</string>
        <string>/tmp/helloworld</string>
    </array>
    <key>StartCalendarInterval</key>
    <dict>
        <key>Minute</key>
        <integer>45</integer>
```

```
        <key>Hour</key>
        <integer>13</integer>
        <key>Day</key>
        <integer>7</integer>
    </dict>
</dict>
</plist>
```

Monitoring a Directory

The following example starts the job whenever any of the paths being watched have changed:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>Label</key>
    <string>com.example.watchhostconfig</string>
    <key>ProgramArguments</key>
    <array>
        <string>syslog</string>
        <string>-s</string>
        <string>-l</string>
        <string>notice</string>
        <string>somebody touched /etc/hostconfig</string>
    </array>
    <key>WatchPaths</key>
    <array>
        <string>/etc/hostconfig</string>
    </array>
</dict>
</plist>
```

An additional file system trigger is the notion of a queue directory. The launchd daemon starts your job whenever the given directories are non-empty, and it keeps your job running as long as those directories are not empty:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>Label</key>
    <string>com.example.mailpush</string>
    <key>ProgramArguments</key>
    <array>
        <string>my_custom_mail_push_tool</string>
    </array>
    <key>QueueDirectories</key>
    <array>
        <string>/var/spool/mymailqdir</string>
    </array>
</dict>
</plist>
```

Emulating inetd

The launchd daemon emulates the older `inetd`-style daemon semantics if you provide the `inetdCompatibility` key:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>Label</key>
    <string>com.example.telnetd</string>
    <key>ProgramArguments</key>
    <array>
        <string>/usr/libexec/telnetd</string>
```

```
    </array>
    <key>inetdCompatibility</key>
    <dict>
        <key>Wait</key>
        <false/>
    </dict>
    <key>Sockets</key>
    <dict>
        <key>Listeners</key>
        <dict>
            <key>SockServiceName</key>
            <string>telnet</string>
            <key>SockType</key>
            <string>stream</string>
        </dict>
    </dict>
</dict>
</plist>
```

Behavior for Processes Managed by launchd

Processes that are managed by `launchd` must follow certain requirements so that they interact properly with `launchd`. This includes launch daemons and launch agents.

Required Behaviors

To support `launchd`, you must obey the following guidelines when writing your daemon code:

- You must provide a property list with some basic launch-on-demand criteria for your daemon. See Creating a launchd Property List File (page 41).
- You must not daemonize your process. This includes calling the `daemon` function, calling `fork` followed by `exec`, or calling `fork` followed by `exit`. If you do, `launchd` thinks your process has died. Depending on your property list key settings, `launchd` will either keep trying to relaunch your process until it gives up (with a “respawning too fast” error message) or will be unable to restart it if it really does die.

- Daemons and agents that are installed globally must be owned by the root user. Agents installed for the current user must be owned by that user. All daemons and agents must not be group writable or world writable. (That is, they must have file mode set to `600` or `400`.)

Recommended Behaviors

To support `launchd`, it is recommended that you obey the following guidelines when writing your daemon code:

- Wait until your daemon is fully initialized before attempting to process requests. Your daemon should always provide a reasonable response (rather than an error) when processing requests.
- Register the sockets and file descriptors used by your daemon in your `launchd` configuration property list file.
- If your daemon advertises a socket, check in with `launchd` as part of your daemon initialization. For an example implementation of the check-in process, see *SampleD*.
- During check-in, get the launch dictionary from `launchd`, extract and store its contents, and then discard the dictionary. Accessing the data structure used for the dictionary is very slow, so storing the whole dictionary locally and accessing it frequently could hurt performance.
- Provide a handler to catch the `SIGTERM` signal.

In addition to the preceding list, the following is a list of things it is recommended you *avoid* in your code:

- Do not set the user or group ID for your daemon. Include the `UserName`, `UID`, `GroupName`, or `GID` keys in your daemon's configuration property list instead.
- Do not set the working directory. Include the `WorkingDirectory` key in your daemon's configuration property list instead.
- Do not call `chroot` to change the root directory. Include the `RootDirectory` key in your daemon's configuration property list instead.
- Do not call `setsid` to create a new session.
- Do not close any stray file descriptors.
- Do not change `stdio` to point to `/dev/null`. Include the `StandardOutPath` or `StandardErrorPath` keys in your daemon's configuration property list file instead.
- Do not set up resource limits with `setrusage`.
- Do not set the daemon priority with `setpriority`

Although many of the preceding behaviors may be standard tasks for daemons to perform, they are not recommended when running under `launchd`. The reason is that `launchd` configures the operating environment for the daemons that it manages. Changing this environment could interfere with the normal operation of your daemon.

Deciding When to Shut Down

If you do not expect your daemon to handle many requests, you might want to shut it down after a predetermined amount of idle time, rather than continue running. Although a well-written daemon does not consume any CPU resources when idle, it still consumes memory and could be paged out during periods of intense memory use.

The timing of when to shut down is different for each daemon and depends on several factors, including:

- The number and frequency of requests it receives
- The time it takes to launch the daemon
- The time it takes to shut down the daemon
- The need to retain state information

If your daemon does not receive frequent requests and can be launched and shut down quickly, you might prefer to shut it down rather than wait for it to be paged out to disk. Paging memory to disk, and subsequently reading it back, incurs two disk operations. If you do not need the data stored in memory, your daemon can shut down and avoid the step of writing memory to disk.

Special Dependencies

While `launchd` takes care of dependencies between daemons, in some cases, your daemon may depend on other system functionality that cannot be addressed in this manner. This section describes many of these special cases and how to handle them.

Network Availability

If your daemon depends on the network being available, this cannot be handled with dependencies because network interfaces can come and go at any time in OS X. To solve this problem, you should use the network reachability functionality or the dynamic store functionality in the System Configuration framework. This is

documented in *System Configuration Programming Guidelines* and *System Configuration Framework Reference*. For more information about network reachability, see Determining Reachability and Getting Connected in *System Configuration Programming Guidelines*.

Disk or Server Availability

If your daemon depends on the availability of a mounted volume (whether local or remote), you can determine the status of that volume using the Disk Arbitration framework. This is documented in *Disk Arbitration Framework Reference*.

Non-launchd Daemons

If your daemon has a dependency on a non-`launchd` daemon, you must take additional care to ensure that your daemon works correctly if that non-`launchd` daemon has not started when your daemon is started. The best way to do this is to include a loop at start time that checks to see if the non-`launchd` daemon is running, and if not, sleeps for several seconds before checking again.

Be sure to set up handlers for `SIGTERM` prior to this loop to ensure that you are able to properly shut down if the daemon you rely on never becomes available.

User Logins

In general, a daemon should not care whether a user is logged in, and user agents should be used to provide per-user functionality. However, in some cases, this may be useful.

To determine what user is logged in at the console, you can use the System Configuration framework, as described in Technical Q&A QA1133.

Kernel Extensions

If your daemon requires that a certain kernel extension be loaded prior to executing, you have two options: load it yourself, or wait for it to be loaded.

The daemon may manually request that an extension be loaded. To do this, run `kextload` with the appropriate arguments using `exec` or variants thereof. I/O Kit kernel extensions should not be loaded with `kextload`; the I/O Kit will load them automatically when they are needed.

> **Note:** The `kextload` executable must be run as root in order to load extensions into the kernel. For security reasons, it is not a setuid executable. This means that your daemon must either be running as the root user or must include a helper binary that is setuid root in order to use `kextload` to load a kernel extension.

Alternatively, our daemon may wait for a kernel service to be available. To do this, you should first register for service change notification. This is further documented in *I/O Kit Framework Reference*.

After registering for these notifications, you should check to see if the service is already available. By doing this after registering for notifications, you avoid waiting forever if the service becomes available between checking for availability and registering for the notification.

> **Note:** In order for your kernel extension to be detected in a useful way, it must publish a node in the I/O registry to advertise the availability of its service. For I/O Kit drivers, this is usually handled by the I/O Kit family.
>
> For other kernel extensions, you must explicitly register the service by publishing a nub, which must be an instance of `IOService`.

For more information about I/O Kit services and matching, see *IOKit Fundamentals*, *I/O Kit Framework Reference* (user space reference), and *Kernel Framework Reference* (kernel space reference).

For More Information

The manual pages for `launchd` and `launchd.plist` are the two best sources for information about `launchd`.

In addition, you can find a source daemon accompanying the `launchd` source code (available from http://www.macosforge.org/). This daemon is also provided from the Mac Developer Library as the SampleD sample code project.

The *Daemons and Agents* technical note provides additional information about how `launchd` daemons and agents work under the hood.

Finally, many Apple-provided daemons support `launchd`. Their property list files can be found in `/System/Library/LaunchDaemons`. Some of these daemons are also available as open source from http://www.opensource.apple.com/ or http://www.macosforge.org/.

Logging Errors and Warnings

You can use two interfaces in OS X to log messages: ASL and Syslog. You can also use a number of higher-level approaches such as `NSLog`. However, because most daemons are not linked against Foundation or the Application Kit, the low-level APIs are often more appropriate.

For software specific to OS X, you should use the ASL interface because it provides more functionality. The Syslog API is the most commonly used logging API on UNIX and Linux systems, so you should consider using it when writing cross-platform software.

Structure Messages with Keys and Hashtags

Historically, log messages have consisted of a text message accompanied by information such as where the message came from, how important it was, and when it was logged. The ASL API structures the entire message as keys and values. There are a number of standard keys listed in `/usr/include/asl.h`, but you can also create your own keys. To prevent key collisions, you should name your keys using a reverse DNS style. For example, `com.example.myCustomKey`.

You are also encouraged to include hashtags in your log messages, regardless of what API you use. A hashtag is composed of a hash (#) symbol, followed by at least four non-whitespace characters, terminated by whitespace or the end of the message. Hashtags may not begin with a number. Additionally, any custom ASL key that begins with a hash is treated as a hashtag. The Console application and the logging APIs understand hashtags on OS X v10.6 and later. They do not interfere with normal logging on previous versions of OS X, because they are just part of the message text.

Some suggested hashtags are:

Hashtag	Meaning
`#System`	Message in the context of a system process.
`#User`	Message in the context of a user process.
`#Developer`	Message in the context of software development. For example, deprecated APIs and debugging messages.

Hashtag	Meaning
`#Attention`	Message that should be investigated by a system administrator, because it may be a sign of a larger issue. For example, errors from a hard drive controller that typically occur when the drive is about to fail.
`#Critical`	Message in the context of a critical event or critical failure.
`#Error`	Message that is a noncritical error.
`#Comment`	Message that is a comment.
`#Marker`	Message that marks a change to divide the messages around it into those before and those after the change.
`#Clue`	Message containing extra key/value pairs with additional information to help reconstruct the context.
`#Security`	Message related to security concerns.
`#Filesystem`	Message describing a file system related event.
`#Network`	Message describing a network-related event.
`#Hardware`	Message describing a hardware-related event.

The following log messages illustrate the use of hashtags.

```
The volume "Macintosh HD" is almost full. #System #Critical #Filesystem

Reloading configuration files. #Marker

Too many failed logins from user "Mouse". Possible brute force attack? #Attention
 #Security #Network

Startup items are deprecated. Use a launchd job instead. #Developer

While organizing orchestra, expected cowbell but found drums. #Comment #Developer
```

Set a Log Level

The system logger supports logging at a number of priority levels, as is traditional on Unix-based and Unix-like systems. The priority levels and suggested uses for these levels are:

Log Level	Suggested Usage
Emergency (level 0)	The highest priority, usually reserved for catastrophic failures and reboot notices.
Alert (level 1)	A serious failure in a key system.
Critical (level 2)	A failure in a key system.
Error (level 3)	Something has failed.
Warning (level 4)	Something is amiss and might fail if not corrected.
Notice (level 5)	Things of moderate interest to the user or administrator.
Info (level 6)	The lowest priority that you would normally log, and purely informational in nature.
Debug (level 7)	The lowest priority, and normally not logged except for messages from the kernel.

The system logger in OS X determines where to log messages at any given priority level based on the file `/etc/syslog.conf`.

It is important to choose the appropriate level for log messages. The system logger discards most low-priority messages, depending on the facility specified. To find out how the system logger decides which facilities and priority levels to log in a given log file, look at the files `/etc/asl.conf` and `/etc/syslog.conf`. By default, messages logged at Info and Debug are discarded.

Log Messages Using the ASL API

The ASL (Apple System Logger) API is very similar to the historic Syslog API, but provides additional functionality:

- Uses a text string for the facility identifier for more precise filtering of log messages. You should use a reverse-DNS style name for your facility, such as `com.example.myproduct`.
- Provides functions for querying the log files.
- Is safe for use in a multithreaded environment because it provides functions for obtaining a separate communication handle for each thread.

- Allows a file to be attached to a log message. This is an effective way to capture extensive state or backtrace information without polluting the logs.

The following code example shows how to log a simple error message:

```
#include <fcntl.h>
#include <asl.h>
#include <unistd.h>

main()
{
        aslclient log_client;
        int cause_an_error = open("/fictitious_file", O_RDONLY, 0);

        log_client = asl_open("LogIt", "The LogIt Facility", ASL_OPT_STDERR);
        asl_log(log_client, NULL, ASL_LEVEL_EMERG, "This is a silly test: Error
%m: %d", 42);
        asl_close(log_client);
}
```

A complete explanation of the features of the ASL API is beyond the scope of this document. For more information, see the `asl` manual page.

Log Messages Using the Syslog API

To use the Syslog API, first call the `openlog` function. This function associates with a particular facility. You can find a list of facilities in the manual page for `syslog`.

> **Note:** Although you can call `syslog` without calling `openlog`, as a rule, you *should* always call `openlog` to specify a facility and logging options. If you do not call `openlog` before you call `syslog`, the API uses the default facility, `LOG_USER`.

If you need to write a wrapper function for the `syslog` function, use the function `vsyslog` instead. This function is identical to `syslog` except that it takes a variable argument list parameter instead of a series of individual parameters.

Before your program exits, or when you need to specify a different facility, call `closelog`. This function resets the facility and logging options to the default settings as though you had never called `openlog`.

The following code example shows how to log a simple error message:

```
#include <fcntl.h>
#include <syslog.h>


main()
{
        int cause_an_error = open("/fictitious_file", O_RDONLY, 0); // sets errno
 to ENOENT
        openlog("LogIt", (LOG_CONS|LOG_PERROR|LOG_PID), LOG_DAEMON);
        syslog(LOG_EMERG, "This is a silly test: Error %m: %d", 42);
        closelog();
}
```

The flags passed to `openlog` affect the behavior of the `syslog` function as follows:

- **LOG_CONS:** Prints the message to the console if the `syslogd` daemon is not running.
- **LOG_PERROR:** In addition to normal logging, also prints the message to standard error.

  Note that anything printed to standard error from a `launchd` job gets logged, so using this option from a `launchd` job results in messages being logged *twice*.
- **LOG_PID:** Appends the process ID after the process name at the beginning of the log message.

These and other flags are described in more detail in the `syslog` manual page.

In addition to the usual `printf` format flags, this command supports an additional flag, `%m`. If this flag appears in the log string, it is replaced by a string representation of the last error stored in `errno`. This is equivalent to what would be reported if you called `perror` or `strerror` directly.

Thus, the code sample above prints the following message to standard output:

```
LogIt[165]: This is a silly test: Error No such file or directory: 42
```

Then, the code sample tells the system logger to log that message. As a result, assuming you have not changed `/etc/syslog.conf`, the system logger broadcasts this message to all users:

```
Broadcast Message from user@My-Machine-Name.mycompany.com
        (no tty) at 13:28 PDT...


Jul 24 13:28:46 My-Machine-Name LogIt[601]: This is a silly test: Error No such
file or directory: 42
```

In this example, the process ID was `601`, and the process name was `LogIt`.

For additional control over what gets logged, you can use the function `setlogmask` to quickly enable or disable logging at various levels. For example, the following code disables logging of any messages below the `LOG_EMERG` level (which is one higher than the `LOG_ALERT` level):

```
setlogmask(LOG_UPTO(LOG_ALERT));
```

You might, for example, use this function to disable logging of debug messages without recompiling your code or adding conditional statements.

Messages are Filtered

The logging APIs filter log messages at three points:

- The global filter (also called the master filter) applies to all log messages from all facilities. By default, it is disabled. To set it, use `asl_set_filter(3) Mac OS X Developer Tools Manual Page` (the ASL API), `setlogmask(3) Mac OS X Developer Tools Manual Page` (the syslog API), or the `syslog` command-line tool.
- The per-client filter. By default, it does not filter out any messages.
- The database filter controls what messages get saved to the log database. By default, it discards debug- and info-level log messages. However, any message that was specifically allowed by the global or per-client filter bypasses this filter.

The functions in the Syslog API consult the process's filter to decide which messages to pass to `syslogd` and which messages to discard. The default filter passes all messages. Use the `setlogmask(3) Mac OS X Developer Tools Manual Page` function to set the filter.

The functions in the ASL API consult the filter of the ASL client passed to them. Each client can have a different value, which means it is possible to have multiple filters in the same process, although uncommon. The default filter discards debug- and info-level messages. Use the `asl_set_filter(3) Mac OS X Developer Tools Manual Page` function to set the filter.

There are two levels at which you can override filters—global and per-client—using the `syslog` command. Both override mechanisms are off by default. Overriding the global filter replaces the filter for the entire system with the one you provide. Overriding per-client replaces the filter for a specific logger client. If both overrides are in effect, the per-client filters replace the global filter for their clients.

The `syslogd` server receives messages form various sources through many different communications channels. It processes each message and may take many actions, including saving messages in one or more files or databases, forwarding messages to other servers over the network, sending notifications, and so on.

There are two main processing modules. The BSD module reads the `/etc/syslog.conf` file and follows rules it finds there. The rules in that file have two parts: a selector and an action. The selector is composed of message facility names and log levels. When the facility and/or log level of a message matches the specification of a selector, the module performs the associated action. Typically, the action is to format the message and write it to a log file.

A second output module, the ASL module, performs a very similar sorting and filtering function. The rules for this module are found in `/etc/asl.conf`. This configuration file is more general-purpose than the `syslog.conf` file. It allows an administrator to set various parameters to control the functions of the `syslogd` server. It also typically contains matching rules and actions. When a message matches one of these rules, the module performs the associated action. Many of these actions specify that a message should be saved in the ASL database.

The following code listing shows some example ASL-style rules, with comments explaining them:

```
# Store everything from emergency to notice.
# This means messages with debug and info log
# level are not saved in the database.
? [<= Level notice] store

# All messages from the kernel (PID 0) and launchd
# (PID 1) are stored, no matter what log level.
? [<= PID 1] store

# Likewise, all messages from the mail, ftp, local0,
# and local1 facilities are stored.
```

```
? [= Facility mail] store
? [= Facility ftp] store
? [= Facility local0] store
? [= Facility local1] store

# All non-debug messages from the lpr facility
# are stored.
? [<= Level info] [= Facility lpr] store

# All messages from the internal facility are
# ignored, regardless of their log level.
? [= Facility internal] ignore
```

View and Search Log Messages

Use the Console application (found in `/Applications/Utilities`) to view log messages from the GUI, or the `syslog` command to view and search log messages from the command line.

Use the `asl` API to search log messages programmatically. For details, see the `asl_search` manual page.

Adopt Best Practices for Logging

Treat your log messages as a potentially customer-facing portion of your application, not as purely an internal debugging tool. Follow good logging practices to make your logs as useful as possible:

- Provide the right amount of information; no more, no less. Avoid creating clutter.
- Avoid logging messages that the user can't do anything about.
- Use hashtags and log levels to make your log messages easier to search and filter.

In order for the log files to be useful (to developers and users), they must contain the right level of detail. If applications log too much, the log fils quickly fill up with uninteresting and meaningless messages, overwhelming any useful content. If they log too little, the logs lack the details needed to identify and diagnose issues.

Logging excessively makes the log files much harder to use, and decreases the value of the logs to your user (who can't easily find the important log messages), to you (who can't easily use the log messages to aid in debugging), and to other developers (whose applications' log messages are buried under yours).

You should avoid writing your own log files. Using the APIs instead of creating your own logging system is the recommended best practice, and provides several advantages:

- Your log messages are easier to find—users only have to look in one place.
- The system doesn't get cluttered with numerous log files from different programs.
- The logging APIs manage log rotation and filtering for you.
- The logging APIs allow you to search logs as well as log messages.

If you log debugging information, you should either disable these messages by default or log them at the Debug level. This ensures that your debugging messages don't clutter up your (and your users') logs.

Scheduling Timed Jobs

In OS X, you can run a background job on a timed schedule in two ways: `launchd` jobs and `cron` jobs. (Older approaches, such as `at` jobs and `periodic` jobs are deprecated and should not be used.) This section explains these methods briefly and provides links to manual pages that provide additional details.

Timed Jobs Using launchd

The preferred way to add a timed job is to use `launchd`. Each `launchd` job is described by a separate file. This means that you can manage `launchd` timed jobs by simply adding or removing a file.

To create a `launchd` timed job, you should create a configuration property list file similar to those described in Creating a launchd Property List File (page 41) except that you specify a `StartCalendarInterval` key containing a dictionary of time values.

For example, the following property list runs the program `happybirthday` at midnight every time July 11 falls on a Sunday.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
  "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>Label</key>
    <string>com.example.happybirthday</string>
    <key>ProgramArguments</key>
    <array>
        <string>happybirthday</string>
    </array>
    <key>StartCalendarInterval</key>
    <dict>
        <key>Day</key>
        <integer>11</integer>
```

```
        <key>Hour</key>
        <integer>0</integer>
        <key>Minute</key>
        <integer>0</integer>
        <key>Month</key>
        <integer>7</integer>
        <key>Weekday</key>
        <integer>0</integer>
    </dict>
</dict>
</plist>
```

For more information on these values, see the manual page for `launchd.plist`.

Timed Jobs Using cron

> **Note:** Although it is still supported, `cron` is not a recommended solution. It has been deprecated in favor of `launchd`.

Systemwide `cron` jobs can be installed by modifying `/etc/crontab`. Per-user `cron` jobs can be installed using the `crontab` tool. The format of these `crontab` files is described in the man page for the `crontab` file format.

Because installing `cron` jobs requires modifying a shared resource (the `crontab` file), you should not programmatically add a `cron` job.

Effects of Sleeping and Powering Off

If the system is turned off or asleep, `cron` jobs do not execute; they will not run until the next designated time occurs.

If you schedule a `launchd` job by setting the `StartCalendarInterval` key and the computer is asleep when the job should have run, your job will run when the computer wakes up. However, if the machine is off when the job should have run, the job does not execute until the next designated time occurs.

All other `launchd` jobs are skipped when the computer is turned off or asleep; they will not run until the next designated time occurs.

Consequently, if the computer is always off at the job's scheduled time, both `cron` jobs and `launchd` jobs never run. For example, if you always turn your computer off at night, a job scheduled to run at 1 A.M. will never be run.

Startup Items

Deprecation Note: Startup items are a deprecated technology. Launching of daemons through this process may be removed or eliminated in a future release of OS X.

Unless your software requires compatibility with OS X v10.3 or earlier, use the `launchd` facility instead. For more information, see Creating Launch Daemons and Agents (page 39).

A startup item is a specialized bundle whose code is executed during the final phase of the boot process, and at other predetermined times (see Managing Startup Items (page 70)). The startup item typically contains a shell script or other executable file along with configuration information used by the system to determine the execution order for all startup items.

The `/System/Library/StartupItems` directory is reserved for startup items that ship with OS X. All other startup items should be placed in the `/Library/StartupItems` directory. Note that this directory does not exist by default and may need to be created during installation of the startup item.

Anatomy of a Startup Item

Unlike many other bundled structures, a startup item does not appear as an opaque file in the Finder. A startup item is a directory whose executable and configuration property list reside in the top-level directory. The name of the startup item executable must match the name of the startup item itself. The name of the configuration property list is always `StartupParameters.plist`. Depending on your needs, you may also include other files in your startup item bundle directory.

```
MyStartupItem/
    |
    |- MyStartupItem
    \- StartupParameters.plist
```

To create your startup item:

1. Create the startup item directory. The directory name should correspond to the behavior you're providing.

   Example: `MyDBServer`

2. Add your executable to the directory. The name of your executable should be exactly the same as the directory name. For more information, see Creating the Startup Item Executable (page 66).
3. Create a property list with the name `StartupParameters.plist` and add it to the directory. For information on the keys to include in this property list, see Specifying the Startup Item Properties (page 68).

   Example: `MyDBServer/StartupParameters.plist`
4. Create an installer to place your startup item in the `/Library/StartupItems` directory of the target system. (Your installer may need to create this directory first.)

   Your installer script should set the permissions of the startup item directory to prevent non-root users from modifying the startup item or its contents. For more information, see Startup Item Permissions (page 72).

Important: A startup item is *not* an application. You cannot display windows or do anything else that you can't do while logged in via ssh. If you want to launch an application after login, you should install a login item instead. For more information, see Adding Login Items (page 18).

Creating the Startup Item Executable

The startup item executable can be a binary executable file or an executable shell script. Shell scripts are more commonly used because they are easier to create and modify.

If you are implementing your startup item executable as a shell script, OS X provides some code to simplify the process of creating your script. The file `/etc/rc.common` defines routines for processing command-line arguments and for gathering system settings. In your shell script, source the `rc.common` file and call the `RunService` routine, passing it the first command-line argument, as shown in the following example:

```
#!/bin/sh
. /etc/rc.common

# The start subroutine
StartService() {
    # Insert your start command below.  For example:
    mydaemon -e -i -e -i -o
    # End example.
}
```

```
# The stop subroutine
StopService() {
    # Insert your stop command(s) below.  For example:
    killall -TERM mydaemon
    sleep 10
    killall -9 mydaemon
    # End example.
}

# The restart subroutine
RestartService() {
    # Insert your start command below.  For example:
    killall -HUP mydaemon
    # End example.
}

RunService "$1"
```

The `RunService` routine looks for `StartService`, `StopService`, and `RestartService` functions in your shell script and calls them to start, stop, or restart your services as needed. You must provide implementations for all three routines, although the implementations can be empty for routines whose commands your service does not support.

If your startup-item executable contains code that might take a long time to finish, consider spawning off a background process to run that code. Performing lengthy startup tasks directly from your scripts delays system startup. Your startup item script should execute as quickly as possible and then exit.

For more information about writing shell scripts, see *Shell Scripting Primer*.

Note: Most Apple-provided startup items have a test in the script to check to see if a particular variable is set to prevent automatic starting of daemons unless they are enabled (usually in the System Preference sharing pane).

To enable or disable a daemon that does not have a GUI checkbox, you must add or modify these variables directly by editing the file `/etc/hostconfig`. Note that this file is writable only by the `root` user, so this technique is discouraged. (Use a `launchd` daemon instead, if at all possible.)

Specifying the Startup Item Properties

The configuration property list of a startup item provides descriptive information about the startup item, lists the services it provides, and lists its dependencies on other services. OS X uses the service and dependency information to determine the launch order for startup items. This property list is stored in ASCII format (as opposed to XML) and can be created using the Property List Editor application.

Note: In OS X v10.4 and later, the dependency information for many stubbed-out system startup items is still present in case other startup items depend on it.

Table A-1 (page 68) lists the key-value pairs you can include in your startup item's `StartupParameters.plist` file. Each of the listed arrays contains string values. You can use the Property List Editor application that comes with the Xcode Tools to create this property list. When saving your property-list file, be sure to save it as an ASCII property-list file.

Table A-1 StartupParameters.plist key-value pairs

Key	Type	Value
Description	String	A short description of the startup item, used by administrative tools.
Provides	Array	The names of the services provided by this startup item. Although a startup item can potentially provide multiple services, it is recommended that you limit your startup items to only one service each.
Requires	Array	The services provided by other startup items that must be running before this startup item can be started. If the required services are not available, this startup item is not run.
Uses	Array	The services provided by other startup items that should be started before this startup item, but which are not required. The startup item should be able to start without the availability of these services.

For example, here is an old-style plist:

```
{
  Description     = "Software Update service";
  Provides        = ("SoftwareUpdateServer");
  Requires        = ("Network");
  Uses            = ("Network");
  OrderPreference = "Late";
  Messages =
  {
    start = "Starting Software Update service";
    stop  = "Stopping Software Update service";
  };
}
```

And here is an XML plist example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist SYSTEM "file://localhost/System/Library/DTDs/PropertyList.dtd">
<plist version="0.9">
    <dict>
        <key>Description</key>
        <string>Apple Serial Terminal Support</string>
        <key>OrderPreference</key>
        <string>Late</string>
        <key>Provides</key>
        <array>
                <string>Serial Terminal Support</string>
        </array>
        <key>Uses</key>
        <array>
                <string>SystemLog</string>
        </array>
    </dict>
</plist>
```

The service names you specify in the `Requires` and `Uses` arrays may not always correspond directly to the name of the startup item that provides that service. The `Provides` property specifies the actual name of the service provided by a startup item, and while this name usually matches the name of the startup item, it is not required to do so. For example, if the startup item launches multiple services, only one of those services can have the same name as the startup item.

If two startup items provide a service with the same name, the system runs only the first startup item it finds with that name. This is one of the reasons why your own startup items should launch only one service. If the name of only one of the services matches the name of another service, the entire startup item might not be executed and neither service would be launched.

The values of the `Requires` and `Uses` keys do not guarantee a particular launch order.

In OS X v10.4 and later, most low-level services are started with `launchd`. By the time your startup item starts executing, `launchd` is running, and any attempt to access any of the services provided by a `launchd` daemon will result in that daemon starting. Thus, you can safely assume (or at least pretend) that any of these services are running by the time your startup item is called.

For this reason, with few exceptions, the `Requires` and `Uses` keys are largely irrelevant after OS X v10.3 except to support service dependencies between two or more third-party startup items.

Managing Startup Items

During the boot process, the system launches the available startup items, passing a `start` argument to the startup item executable. After the boot process, the system may run the startup item executable again, this time passing it a `restart` or `stop` argument. Your startup item executable should check the supplied argument and act accordingly to start, restart, or stop the corresponding services.

> **Note:** In general, with the exception of daemons provided with OS X, the system will only run your startup script with `start` or `stop` arguments (at boot and shutdown, respectively). Users, however, may elect to use the `restart` argument.
>
> You should not make any assumptions about the order in which daemons will be shut down.

If you want to start, restart, or stop startup items from your own scripts, you can do so using the `SystemStarter` program. To use `SystemStarter`, you must execute it with two parameters: the desired action and the name of the service provided by the startup item. For example, to restart the Apache Web server (prior to OS X v10.4), you would execute the following command:

```
/sbin/SystemStarter restart "Web Server"
```

Important: You must have root authority to start, restart, or stop startup items.

Startup items should always respect the arguments passed in by `SystemStarter`. However, the response to those arguments is dependent on the startup item. The stop and restart options may not make sense in all cases. Your startup item could also support the restart option using the existing code for stopping and starting its service.

Displaying and Localizing a Startup Message

When your startup item starts at boot time, you may (if desired) display a message to the user. To do this, use the `ConsoleMessage` command. (You can use this command even if the computer is not starting up, but the user will not see it unless the Console application is running.)

For example:

```
ConsoleMessage "MyDaemon is running.  Better go catch it."
```

If you want to localize the message displayed when a startup item starts, you must create a series of property list files with localized versions of the strings. Each of these files must be named `Localizable.strings`, and must be in a localized project directory whose name is based on the name of a language or locale code for the desired language. These folders, in turn, must be in a folder called `Resources` in the startup item folder.

For example, you might create a tree structure that looks like this:

```
MyDaemon
    |
    |- MyDaemon
    |- StartupParameters.plist
    \- Resources
            |
            |- English.lproj
            |- French.lproj
            |- German.lproj
            \- zh_CN.lproj
```

Within each of these localizable strings files, you must include a dictionary whose keys map an English string to a string in another language. For example, the French version of the `PrintingServices` localization file looks like this:

```
<?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>Starting printing services</key>
        <string>Démarrage des services d'impression</string>
    </dict>
</plist>
```

Whenever the `ConsoleMessage` command is passed the string "Starting printing services", if the user's language preference is French, the user will instead see "Démarrage des services d'impression" at startup. C'est très bien!

The value of the *key* field must precisely match the English string printed by your startup item using `ConsoleMessage`.

See the manual page for `locale` for more information about locale codes.

Startup Item Permissions

Because startup items run with root authority, you must make sure your startup item directory permissions are set correctly. For security reasons, your startup item directory should be owned by root, the group should be set to wheel, and the permissions for the directory should be 755 (rwxr-xr-x). This means that only the root user can modify the directory contents; other users can examine the directory and view its contents, but cannot modify them. The files inside the directory should have similar permissions and ownership. Thus, the file listing for the Apache startup item directory is as follows:

```
./Apache:
total 16
drwxr-xr-x    4 root  wheel   136 Feb 14 14:33 .
drwxr-xr-x   21 root  wheel   714 Feb 14 15:03 ..
-rwxr-xr-x    1 root  wheel  1253 Feb 10 19:31 Apache
```

```
-rw-r--r--    1 root  wheel   152 Feb 10 19:31 StartupParameters.plist
```

Important: In OS X version 10.4 and later, the system asks the user what to do about startup items with incorrect permissions. At this point, the user may choose to disable the startup item, which could have unexpected results for your software. To avoid this, be sure to set the permissions during installation.

Customizing Login and Logout

This appendix describes technologies that fill very specific roles. As a rule, if your goal is to have a process running while the user is logged in, you should almost always use either a launch daemon or agent, as described in Creating Launch Daemons and Agents (page 39).

Running Agents Before Login

Most software that displays a user interface does not run prior to the user logging in. However, in some rare cases, it may be necessary to create a graphical agent that does.

By default, OS X does not allow any application to draw content prior to login. If you need to do so, your agent must call the `setCanBecomeVisibleWithoutLogin:` method on its windows. For more information, see the documentation for that method and the *PreLoginAgents* sample code.

Authentication Plug-Ins

Authentication plug-ins are the recommended way to perform tasks during the login process. An authentication plug-in executes while the user is logging in, and is guaranteed to complete before the user is allowed to actually interact with his or her account.

You might write an authentication plug-in if you need to programmatically reset an account to a predetermined state, perform some administrative task such as deleting caches to reduce server utilization, and so on.

To learn more about writing an authentication plug-in, read *Running At Login*.

Login and Logout Scripts

Important: There are numerous reasons to *avoid* using login and logout scripts:

- Login and logout scripts are a deprecated technology. In most cases, you should use `launchd` jobs instead, as described inCreating Launch Daemons and Agents (page 39).
- Login and logout scripts are run as root, which presents a security risk.
- Only one of each script can be installed at a time. They are intended for system administrators; application developers should not use them in released software.

One way to run applications at login time is to launch them using a custom shell script. When creating your script file, keep the following in mind:

- The permissions for your script file should include execute privileges for the appropriate users.
- In your script, the variable `$1` returns the short name of the user who is logging in.
- Other login actions wait until your hook finishes executing. Therefore, your script needs to run quickly.

Use the `defaults` tool to install your login script. Create the script file and put it in a directory that is accessible to all users. In Terminal, use the following command to install the script (where `/path/to/script` is the full path to your script file):

```
sudo defaults write com.apple.loginwindow LoginHook /path/to/script
```

To remove this hook, delete the property:

```
sudo defaults delete com.apple.loginwindow LoginHook
```

Use the same procedure to add or remove a logout hook, but type `LogoutHook` instead of `LoginHook`.

Note: If no `plist` file exists for `com.apple.loginwindow`, this method will not work. This file (`/var/root/Library/Preferences/com.apple.loginwindow.plist`) does not exist on a fresh installation until the user changes a login window setting (such as turning on fast user switching).

If you must install startup scripts programmatically, you should consider providing a copy of this file containing the default configuration options. Then, if the file does not exist, copy that default configuration file into place before running `defaults`. Again, application developers are strongly discouraged from using login or logout scripts, because only one such script may be installed.

Bootstrap or "mach_init" Daemons

In OS X v10.3, a mechanism similar to `launchd` was supported to allow the launching of programs either at system startup or on a per-user basis. The process involved placing a specially formatted property list file in either the `/etc/mach_init.d` or the `/etc/mach_init_per_user.d` directory. Such daemons also are sometimes referred to as `mach_init` daemons.

The use of bootstrap daemons is deprecated and should be avoided entirely. Launching of daemons through this process may be removed or eliminated in a future release of OS X.

If you need to launch daemons, use the `launchd` facility. If you need to launch daemons on versions of OS X that do not support `launchd`, use a startup item.

Document Revision History

This table describes the changes to *Daemons and Services Programming Guide*.

Date	Notes
2014-07-15	Corrected code listing to whitelist both NSArray and FrenchFry. See Whitelisting a Class for Use Inside Containers (page 32).
2013-10-22	Improved documentation for NSXPCConnection.
2012-07-23	Added conceptual documentation for NSXPCConnection.
2012-01-09	Minor updates.
2011-08-26	Updated coverage of XPC services.
2011-06-30	Updated for OS X v10.7, including discussion of XPC services. Changed the title from "System Startup Programming Topics." Moved material related to kernel extensions and the early boot process to *Kernel Programming Guide*. Various structural and organizational changes throughout.
2008-11-19	Miscellaneous edits.
2007-02-08	Clarified the explanation of launchd daemons versus startup items.
2006-11-07	Added mention of prelinked kernels and helper partitions.

Date	Notes
2006-10-03	Added information about Intel booting, launchd, and asl (logging).
2005-08-11	Updated guidance for customizing login and logout. Updated information pertaining to Startup Item property-list files.
2005-04-29	Updated for OS X v10.4. Updated login/logout customization information. Added information about launch-on-demand daemons. Updated information related to the use of startup items.
2003-08-07	First revision of this programming topic. Some information in this programming topic originally appeared in *System Overview*.

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