

Mac Notification Overview

Contents

Introduction to OS X Notification Overview 4

Organization of This Document 4

Notification Basics 5

What Is a Notification? 6

What Is Coalescing? 7

What Types of Shared Store Can Be Used? 7

What Message-Passing Mechanisms Can Be Used? 8

How Should Notifications Be Named? 8

Choosing a Notification Technology 9

Applications Communicating with Daemons 9

Mixed Carbon and Cocoa Applications 10

System-Provided Notifications 10

Darwin Notification Concepts 11

Including Relevant Headers 11

Sending Notifications 12

Receiving Notifications 12

Receiving Notifications Using Core Foundation 13

Receiving Notifications Using File Descriptors 14

Receiving Notifications Using Signals 15

Receiving Notifications Using Mach Messages 17

Receiving Notifications Manually 19

Alternatives to Notification 21

Message Passing and Remote Procedure Call APIs 21

Apple Events 21

Distributed Objects 21

Other Message-Passing and Remote Procedure Call Technologies 21

Memory Mapping and Shared Memory 22

Document Revision History 23

Figures

Notification Basics 5

Figure 1-1 Notification with a shared data store 6

Introduction to OS X Notification Overview

Notifications are a means of sharing state information between two applications, daemons, or other processes in a way that is robust while maintaining good performance.

You should read this document if you are writing an application that uses interprocess communication. You should also read this document if you need to learn about Darwin notifications (including kernel event notifications).

Organization of This Document

This document is organized into four chapters:

- [Notification Basics](#) (page 5)—describes notifications at a high level.
- [Choosing a Notification Technology](#) (page 9)—highlights the differences between notification technologies and provides guidance about what notification API you should use under different circumstances.
- [Darwin Notification Concepts](#) (page 11)—describes the Darwin notification mechanism.
- [Alternatives to Notification](#) (page 21)—describes alternatives to notifications and provides guidelines for their use.

Notification Basics

In modern computing, developers often need to find out when external information (such as configuration files, the current time zone or time of day, and so on) changes.

There are many ways to solve this problem, of which notifications are just one. The main ways are:

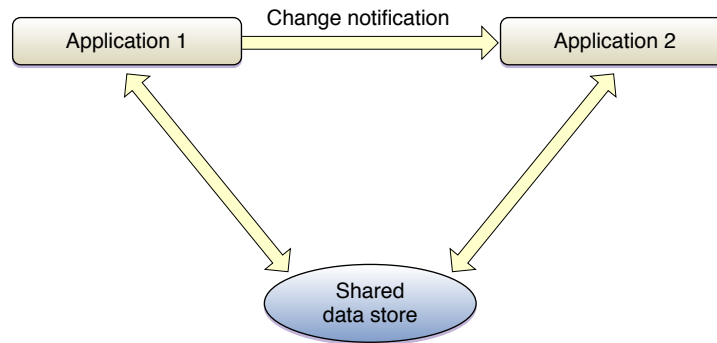
- **Notifications**—the subject of this document. Notifications are a combination of shared memory and message passing.
- **Polling**—pulling the needed information from another application or from a file on a regular basis. This technique is wasteful of CPU resources and is thus strongly discouraged for general use. However, when combined with shared memory, polling is useful in certain specialized environments.
- **Shared memory**—providing a common repository for shared information that can be accessed through pointers in multiple processes. This technique is used as part of notification mechanisms. However, by itself, it is not a sufficient solution because of poor responsiveness. It is generally combined with either message passing or polling.
- **Message passing**—a “push” method in which the sending application tells the receiving application that the information has changed. By itself, this is a good technique in terms of responsiveness, but it can be problematic in terms of robustness because when a message is lost, the receiving application cannot obtain the new value.

Although notification mechanisms differ greatly in the details of their implementation, they share a common design goal: allowing the recipient of the notification to monitor a piece of information in such a way that minimizes overhead while enabling designs that are robust even if a notification gets lost.

What Is a Notification?

At a high level, a notification is a message indicating that something has changed. Notification schemes typically combine a shared data store with a message-passing mechanism.

Figure 1-1 Notification with a shared data store



It is this combination that makes notification schemes powerful. With message passing by itself, you face a tradeoff between performance and robustness. The UDP networking protocol, for example, is relatively efficient but does not provide delivery guarantees. By contrast, TCP provides delivery guarantees, but the extra overhead involved in doing so reduces performance.

Similarly, with shared storage by itself, you face a tradeoff between responsiveness and overall system performance. If you check the shared storage frequently, the application would appear responsive to changes in the shared state, but it would also hog CPU resources. By contrast, if you check less frequently, the CPU load would be lessened, but the application would be less responsive.

By separating the data from the message, you can get robustness, responsiveness, and performance. Because the message is sent to the receiving application, the receiving application does not have to constantly check for a state change. This results in responsiveness without the performance penalties associated with polling.

Similarly, because the data store is shared, the receiving application can request the current information and act on it at any time. Thus, that application could occasionally check the shared store to make certain the data has not changed behind its back. Alternatively, it could do such a check before performing any particularly critical operations to ensure correctness.

Thus, a notification can be compared to a bulletin board. If you are the keeper of the board, you can post information on the board. You can tell other people that new information has been posted. Others can then look at the board whenever it is convenient for them to do so. However, if someone fails to get such a message, he or she can still look at the bulletin board before making a critical decision. If desired, people can even wander by and look at the board without signing up to receive notices.

What Is Coalescing?

The word coalesce means “to unite”. In the context of notifications, if two notifications have identical content, this means two things:

- Identical messages can be reduced to a single message. You only need to receive one notification that “variable X has changed” for correctness.
- Nonidentical messages with small payloads can be sent to the receiving process as a group.

Coalescing notifications can have a significant effect on performance when a large number of notifications are sent in a short period of time. By combining notifications, the extra context switching required to send those additional notifications to the receiving application is eliminated. For this reason, you should try to maximize coalescing as much as possible.

To maximize coalescing of messages, you should minimize the payload as much as possible. Darwin notifications, for example, which carry no payload other than the name of the notification, can be coalesced very easily.

What Types of Shared Store Can Be Used?

The shared data store used for a notification can be a file, POSIX or System V shared memory, a database record, or something else entirely, at the discretion of the developer who creates the notification scheme. The only hard and fast requirement for a shared store is that it must be accessible to both the sending and the receiving application—that is, that it must actually be a shared store.

For notifications generated by OS X components, the storage is usually a file on disk or a commpage location (date and time changes, for example). However, the location need not be a single physical location at all. Instead, it could be an application that gathers the information from throughout the file system or an API for obtaining the data from lower levels of the operating system. For example, in response to a network change notification, your application might ask the operating system for updated network interface information through any number of APIs.

Applications often implement the shared store as part of one of the applications using POSIX or System V shared memory. This has the advantage of not creating unnecessary files on disk, but it may make state recovery after an application crash impossible. This may or may not be an issue, depending on the application and the nature of the shared data.

Another common form of shared storage is a memory-mapped file. To memory map a file, you must first create the file (of an appropriate length), then call `mmap(2)` on that file. When you are through with the file, you delete it as you would any other temporary file. Memory mapping has the advantage of making state recovery

possible in the event of a crash. However, it has the disadvantage of polluting the buffer cache. If your application depends on rapid disk I/O (such as an audio application) for correctness, you should probably use shared memory instead.

What Message-Passing Mechanisms Can Be Used?

Notifications can be passed using any number of mechanisms, from Apple Events to UDP (and everything in between). Most notifications, however, are sent with Darwin notifications (described in [Darwin Notification Concepts](#) (page 11)) or a technology built on top of Darwin notifications, such as Core Foundation notifications (described in *CFNotificationCenter Reference*) or Cocoa notifications (described in *NSNotificationCenter Class Reference*).

How Should Notifications Be Named?

Notifications should be named using a reverse-DNS-style naming. For example, if the Mklinux team released a daemon called `kernel_daemon` that provided a notification called `kernel_loaded`, that notification would be named `org.mklinux.kernel_daemon.kernel_loaded`.

The reason for this naming convention is to avoid collisions; the notification namespace is shared across all applications in the system, both at the Darwin notification level and at higher levels with Core Foundation or Cocoa notifications.

Note: Notifications that begin with `com.apple` are reserved.

Choosing a Notification Technology

For the most part, the notification technology you choose should be appropriate for the type of programming you are doing. Command-line tools and daemons should generally use Darwin notifications. High-level applications should generally use either Core Foundation or Cocoa notifications.

This is not always true, however. In some cases, it may be more appropriate to choose a low-level notification scheme even in a high-level application. This chapter describes some environments in which you should choose a different technology than the most obvious choice.

Applications Communicating with Daemons

When applications and daemons must communicate with notifications, the best choice is not always obvious.

- If the daemon is written with Cocoa or Core Foundation, you can use the Cocoa, Core Foundation, or Darwin notification mechanism, at your option.
- If the daemon is not based on Core Foundation (for example, most cross-platform open source software), it is usually much easier to use Darwin notifications in the daemon because you can do so without creating a run loop.
- Darwin notifications are also easier to tie into existing UNIX/Linux daemons because these daemons often already use signal handlers for event handling, and Darwin offers signal delivery as one of its supported delivery methods.
- For daemons that use file descriptors (`select(2)` loops), Darwin notifications can be integrated more easily because it offers file descriptors as a supported delivery method. Using Darwin notifications directly is significantly easier than rewriting the main program loop as a `CFRunLoop` (and, for open source projects, is much more likely to be accepted into their official source tree).

Of course, you do not have to use the same API in your application as in the daemon. As long as you limit your use of the Core Foundation or Cocoa notification APIs to empty messages (that is, messages with only a name), you can use those APIs in your application and still use the Darwin notification API in your daemon.

Mixed Carbon and Cocoa Applications

Cocoa notifications (`NSNotificationCenter`) and Core Foundation notifications (`CFNotificationCenterRef`) can communicate with each other, making it easy to provide notifications between Carbon and Cocoa applications. However, because these types are not toll-free bridged, you cannot cast between them.

For the most part, this is not a problem. However, if you need to post notifications from portions of your application written in Carbon, it may be easier to use Core Foundation notifications throughout. This will make it easier to share data structures between C and Objective-C portions of your code without introducing redundancy.

Similarly, if you are adding notifications to Carbon and Cocoa applications simultaneously and need to write any glue code that is common to both your Carbon and Cocoa applications, you may find it more convenient to write a single notification module based on Core Foundation notifications in order to avoid maintaining multiple versions of your glue code.

System-Provided Notifications

Many parts of OS X provide notifications in other ways. Examples include the I/O Kit, Disk Arbitration, System Configuration (`configd`), and kernel queues. To learn how to receive these notifications, you should read Apple's documentation about those technologies.

Kernel queues and kernel event notifications are a much better alternative to polling for file changes. Kernel event notifications also provide a way to find out about a number of other kernel-related events. The kernel queues mechanism is described in *File System Events Programming Guide* and in the manual pages for `kevent(2)` and `kqueue(2)`.

The I/O Kit notification mechanism is based around the `IOService` class. Registering for and posting notifications is described in *IOKit Fundamentals*.

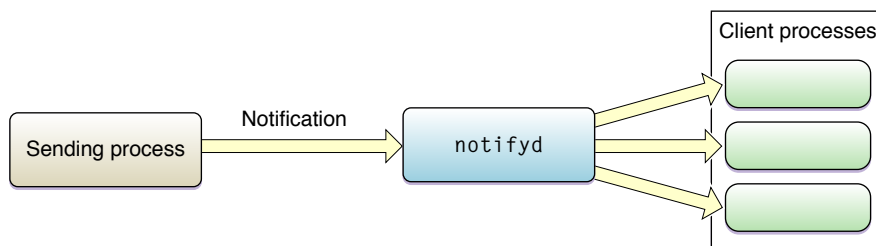
System Configuration notifications are provided through the System Configuration framework. You can learn more about this framework by reading *System Configuration Programming Guidelines* and *System Configuration Framework Reference*.

Disk Arbitration notifications can tell you when a volume is mounted or unmounted. (You can also learn when volumes are mounted or unmounted using the File System Events API in OS X v10.5 and later.) You can learn more about Disk Arbitration notifications in *Disk Arbitration Framework Reference*.

Darwin Notification Concepts

This chapter describes how to send and receive Darwin notifications. The Darwin notification system is relatively straightforward for developers familiar with programming in other UNIX/Linux operating systems. It ties into commonly used systems such as file descriptors and signals to provide delivery of messages to the client process.

Darwin notifications are supported by the `notifyd(8)` daemon, a process that listens for incoming notifications and redelivers those notifications to interested processes in a variety of ways.



Including Relevant Headers

When writing a tool that uses Darwin notifications, the following headers are commonly used:

```
#include <unistd.h>      // good idea in general
#include <stdlib.h>       // good idea in general

#include <strings.h>      // for bcopy, used by FD_COPY macro
                        // (file descriptor delivery)
#include <sys/select.h>   // for select (file descriptors delivery)
#include <stdio.h>        // for read (file descriptor delivery)

#include <signal.h>       // for signal names (signal delivery)

#include <mach/message.h> // For mach message functions and types
                        // (mach message delivery)
```

```
#include <notify.h>          // for all notifications
```

Sending Notifications

Sending a Darwin notification is very simple. Just call the function `notify_post`. The function takes a single argument that contains the name of the notification. (See [How Should Notifications Be Named?](#) (page 8) for information about notification naming.)

For example, if you integrate notifications into the Apache web server and want to notify it that its configuration file has changed, you might send the notification with a call like the following:

```
if (notify_post("org.apache.httpd.configFileChanged")) {  
    printf("Notification failed.\n"); exit(-1);  
}
```

The function `notify_post` returns zero on success or an error code on failure. The possible error codes are described in *Status Codes* in *Darwin Notification API Reference*.

Receiving Notifications

Darwin notifications provide five mechanisms for receiving notifications: Core Foundation, file descriptors, signals, Mach messages, and manual polling. This section describes these delivery mechanisms and explains when you should choose each one.

When adding notification support to existing applications, you should choose whichever mechanism is easiest to integrate into the application or tool. If your application already uses signal handling, you should use signals. If your application already uses sockets or file descriptors, you should use file descriptors. And so on.

If you are writing a new application from scratch, Core Foundation notifications are recommended if your application is based on Core Foundation. For non-Core Foundation applications, file descriptors are the preferred notification transport because they are more robust against message loss than signals and are easier to use than Mach messages. Again, you should choose the mechanism that most closely fits the architecture of the tool you are writing.

As a general rule, you should avoid using the polling interface (described in [Receiving Notifications Manually](#) (page 19)). You should use polling *only* if you need to check for a status change infrequently.

Receiving Notifications Using Core Foundation

To register for Darwin notifications using Core Foundation, first create a notification center by calling `CFNotificationCenterGetDarwinNotifyCenter`. This creates an object that interacts with the notification daemon on your behalf.

Next, call `CFNotificationCenterAddObserver`. This function instructs the notification center to register for notifications with the notification daemon.

The following snippet shows how to register for the notification described in [Sending Notifications](#) (page 12):

```
/* This function is called when a notification is received. */
void MyNotificationCenterCallBack(CFNotificationCenterRef center,
    void *observer,
    CFStringRef name,
    const void *object,
    CFDictionaryRef userInfo)
{
    printf("Notification center handler called\n");
}

...

/* Create a notification center */
CFNotificationCenterRef center = CFNotificationCenterGetDarwinNotifyCenter();

/* Tell notifyd to alert us when this notification
   is received. */
if (center) {
    CFNotificationCenterAddObserver(center,
        NULL,
        MyNotificationCenterCallBack,
        CFSTR("org.apache.httpd.configFileChanged"),
        NULL,
        CFNotificationSuspensionBehaviorDeliverImmediately);
    ...
}
```

Receiving Notifications Using File Descriptors

Receiving notifications with file descriptors is relatively straightforward. First, you must call `notify_register_file_descriptor` to register for notifications. For example, the following snippet shows how to register for the notification sent in [Sending Notifications](#) (page 12):

```
int fd; /* file descriptor---one per process if
        NOTIFY_REUSE is set, else one per name */
int notification_token; /* notification token---one per name */

...

if (notify_register_file_descriptor("org.apache.httpd.configFileChanged",
                                   &fd,
                                   0,
                                   &notification_token)) {
    /* Something went wrong. Bail. */
    printf("Registration failed.\n"); exit(-1);
}
```

The function `notify_register_file_descriptor` returns zero on success or an error code on failure. The possible error codes are described in Status Codes in *Darwin Notification API Reference*.

Note: The flag field is set to zero (0) in this call. You *must* not set the flag `NOTIFY_REUSE` until the second and subsequent calls to `notify_register_file_descriptor`. This flag tells the notification library code to reuse the existing file descriptor. If you use this flag, your code will not work correctly unless the file descriptor value has been initialized by a previous call to `notify_register_file_descriptor`.

After you register for notification, you will begin receiving data through the returned file descriptor. You can detect the arrival of new data using `select(2)` or `poll(2)`, as shown in the following snippet:

```
fd_set receive_descriptors, receive_descriptors_copy;
```

```
FD_SET(fd, /* from call to notify_register_file_descriptor */
        &receive_descriptors);
FD_COPY(&receive_descriptors, &receive_descriptors_copy);
while (select(fd + 1, &receive_descriptors_copy,
             NULL, NULL, NULL) >= 0) {
    /* Data was received. */
    if (FD_ISSET(fd, &receive_descriptors_copy)) {
        /* Data was received on the right descriptor.
           Do something. */
        int token;

        /*! Read four bytes from the file descriptor. */
        if (read(fd, &token, sizeof(token)) != sizeof(token)) {
            /* An error occurred.  Panic. */
            printf("Read error on descriptor.  Exiting.\n");
            exit(-1);
        }

        /* At this point, the value in token should match one of the
           registration tokens returned through the fourth parameter
           of a previous call to notify_register_file_descriptor. */
    }

    FD_COPY(&receive_descriptors, &receive_descriptors_copy);
}
```

For more information about the `select` system call, see the manual page for `select(2)`.

Receiving Notifications Using Signals

To receive a signal when a new notification is posted, call the function `notify_register_signal`. This function tells the notification daemon to send a signal to your process whenever it posts new messages.

Note: Although you can choose what signal `notifyd` sends to your process, some signals have special purposes and should not be used for notifications.

The `SIGALRM`, `SIGVTALRM`, and `SIGTHR` (`P_SIGTHR`) signals are used for other purposes related to thread scheduling and `sleep(1)`/`usleep(3)`/`nanosleep(2)` timers. If you use these signals for notifications, unexpected behavior may result.

The `SIGKILL` signal cannot be trapped and thus cannot be used for notifications (unless your goal is to terminate your application when the event occurs).

The following snippet shows how to register for the notification described in [Sending Notifications](#) (page 12):

```
int notification_token;

...

/* Set up a signal handler for SIGHUP */
signal(SIGHUP, &my_sighup_handler);

/* Tell notifyd to send SIGHUP when this notification
   is received. */

if (notify_register_signal(
    "org.apache.httpd.configFileChanged", SIGHUP,
    &notification_token)) {
    printf("Registration failed.\n"); exit(-1);
}
```

The function `notify_register_signal` returns zero on success or an error code on failure. The possible error codes are described in Status Codes in *Darwin Notification API Reference*.

The value of `notification_token` is set to an integer value specific to the name for this notification. It is your responsibility to keep track of these notification token values so that you can find out which notification was posted (unless you are registering only for a single notification and don't care about false positives caused by signals not generated by `notifyd`).

Portability Note: In OS X, signal handlers are not cleared when a signal handler returns. However, for maximum portability, you should still get in the habit of calling `signal` at the end of your signal-handler function. This will ensure that signal-handling code is portable across operating system environments.

After you register to receive a signals, you must then call `notify_check` to determine which (if any) notification triggered the signal, as shown in the next snippet:

```
int was_posted;

if (notify_check(notification_token, &was_posted)) {
    printf("Call to notify_check failed.\n"); exit(-1);
}

if (was_posted) {
    /* The notification org.apache.httpd.configFileChanged
       was posted. */
}
```

The function `notify_check` returns zero on success or an error code on failure. The possible error codes are described in Status Codes in *Darwin Notification API Reference*. The function returns a value through the second parameter to indicate whether the notification has been posted since the last time you called `notify_check`.

Receiving Notifications Using Mach Messages

If your tool or application uses Mach messages for communication, you may find it convenient to use Mach messages for receiving notification. Communication based on Mach messaging is not recommended for use in new designs because it is relatively easy to corrupt your application's memory if used incorrectly.

Note: This section assumes that you are already familiar with Mach IPC concepts such as ports and port rights. Thus, it does not cover these concepts in any great detail. For more information, see Mach Overview in *Kernel Programming Guide*.

You can register for Mach message-based notification on a Mach port by calling `notify_register_mach_port`, as shown in the snippet below:

```
mach_port_name_t port; /* mach port---one per process if
    NOTIFY_REUSE is set, else one per name */
int notification_token; /* notification token---one per name */

/* Allocate a mach port.  If you already have receive rights on a
    port and would prefer to use that, you can do so, of course. */
if (mach_port_allocate (mach_task_self(),
    MACH_PORT_RIGHT_RECEIVE, &port) != KERN_SUCCESS) {
    printf("Could not allocate mach port.\n"); exit(-1);
}

if (notify_register_mach_port("org.apache.httpd.configFileChanged",
    &port,
    NOTIFY_REUSE,
    &notification_token)) {
    /* Something went wrong.  Bail. */
    printf("Registration failed.\n"); exit(-1);
}
```

The function `notify_register_mach_port` returns zero on success or an error code on failure. The possible error codes are described in Status Codes in *Darwin Notification API Reference*.

If desired, you can let Mach allocate the port for you by skipping the call to `mach_port_allocate` and passing in 0 instead of `NOTIFY_REUSE`.

Note: You *must* not set the flag `NOTIFY_REUSE` without allocating a port first. If you do not allocate a port in advance, you must not use the `NOTIFY_REUSE` flag until the second and subsequent calls to `notify_register_file_descriptor`. If you do, your application will probably crash.

This flag tells the notification library code to reuse the existing Mach port. The call will fail unless the Mach port has been initialized and your Mach task holds receive rights on the port.

After you have registered for notification, you can receive messages on the port with the `mach_msg_overwrite` call, as shown in the following snippet:

```
struct {
    mach_msg_header_t hdr;
    int token;
} mydatastructure;

while (1) {
    mach_msg_overwrite(NULL, MACH_RCV_MSG,
        0, sizeof(mydatastructure), port, MACH_MSG_TIMEOUT_NONE,
        MACH_PORT_NULL, (mach_msg_header_t *)&mydatastructure,
        sizeof(mydatastructure.token));
    printf("Data received : %d (compare to %d).\n", mydatastructure.token,
notification_token);
}
```

Receiving Notifications Manually

Although it is not common to do so, you may sometimes find it useful to poll to see (on an occasional basis) whether a particular notification has occurred. To do this, you must request a token corresponding to the notification name by calling `notify_register_check`, as shown in the following snippet:

```
if (notify_register_check(
    "org.apache.httpd.configFileChanged", &notification_token)) {
    printf("Registration failed.\n"); exit(-1);
}
```

You can then check for notifications using `notify_check` (just as you would for signal delivery), as shown in the following snippet:

```
int was_posted;
while (1) {
    sleep(1);

    if (notify_check(notification_token, &was_posted)) {
        printf("Call to notify_check failed.\n"); exit(-1);
    }
    if (was_posted) {
```

```
    /* The notification org.apache.httpd.configFileChanged
       was posted. */
    printf("Notification %d was posted.\n", notification_token);
}
}
```

Note: Due to limitations in the underlying architecture, you may get a "false positive" result on the initial call to `notify_check` when using this method.

The function `notify_check` returns zero on success or an error code on failure. The possible error codes are described in Status Codes in *Darwin Notification API Reference*. The function returns a value through the second parameter to indicate whether the notification has been posted since the last time you called `notify_check`.

Alternatives to Notification

Notifications are just one possible solution. They are usually a good solution, but in some cases, you may want to use a different means of interprocess data sharing. This chapter describes some alternative mechanisms.

Message Passing and Remote Procedure Call APIs

Distributed objects, Apple events, and other similar technologies are good solutions when you need to receive a response to indicate that the client has received a state change notification. This section briefly describes these technologies and provides pointers to further documentation.

Apple Events

Apple events is a Carbon message passing API. The Apple Events API enables you to send an event notification to another application and receive a response message. You can learn more in *Apple Events Programming Guide*.

Distributed Objects

Distributed objects is a Cocoa remote procedure call API. Distributed objects enable one application to call Objective-C methods in another application and receive the return value. You can learn more in *Distributed Objects Programming Topics*.

Other Message-Passing and Remote Procedure Call Technologies

Other message-passing techniques in OS X include Mach messaging, sockets, pipes, and standard input and output. These techniques are explained in Cross-Architecture Plug-in Support in *64-Bit Transition Guide*.

Memory Mapping and Shared Memory

Memory mapping and other shared memory techniques provide a good way to move large quantities of data between two applications. When two applications need to move data on a continuous basis, polling can be more efficient than notifications. For example, two audio applications connected by a buffer would be a poor match for notifications because the receiving application must read data on a regular basis even in the absence of a notification.

OS X provides several ways to share memory, depending on your needs.

For audio, you should use the Audio Queue API (part of the Core Audio framework). This API is described in *Audio Queue Services Programming Guide* and *Audio Queue Services Reference*.

For general memory sharing, the easiest mechanism to use is the `mmap(2)` system call. This system call allows you to map a file or portion thereof into the memory space of your process, effectively giving you a read-only or read-write pointer into the contents of the file itself. By mapping a file simultaneously into multiple processes, you can easily create shared memory between these processes. (Note that before calling this system call, you must first create the file, then extend it to an appropriate size.)

Two other ways to share memory are the POSIX and System V shared memory APIs. Because the POSIX shared memory API is newer and more flexible, you should favor the POSIX shared memory API for new applications unless you need to support other computing platforms where it is not available.

Note: Although you can create shared memory regions using Mach APIs (`vm_allocate`, `vm_map`, and so on) directly, this is strongly discouraged.

You can learn more about POSIX shared memory in the `shm_open(2)` and `shm_unlink(2)` manual pages.

You can learn more about System V shared memory in the `shmat(2)`, `shmctl(2)`, `shmget(2)`, and `shmdt(2)` manual pages.

You can learn how to memory map files in the `mmap(2)` manual page. To find a simple example of this technique, see Cross-Architecture Plug-in Support in *64-Bit Transition Guide*.

Document Revision History

This table describes the changes to *Mac Notification Overview*.

Date	Notes
2009-05-01	Added example of using Core Foundation APIs to receive Darwin notifications.
2007-05-15	New document that introduces Apple's notification technologies.



Apple Inc.
Copyright © 2009 Apple Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer or device for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-branded products.

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

Apple, the Apple logo, Carbon, Cocoa, Mac, Objective-C, and OS X are trademarks of Apple Inc., registered in the U.S. and other countries.

UNIX is a registered trademark of The Open Group.

APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT, ERROR OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

Some jurisdictions do not allow the exclusion of implied warranties or liability, so the above exclusion may not apply to you.