Low-Level File Management Programming Topics

Cocoa > File Management

2009-03-05

Contents

Listings

Introduction to Low-Level File Management Programming Topics

This document describes the various Foundation methods and functions for manipulating files and directories.

Organization of This Document

This document contains the following articles:

- "Portable File-System Operations" (page 9) explains the general approach for composing paths to locations in the file system.
- "HFS File Types" (page 11) describes the APIs that allow a Cocoa application to retrieve and create HFS type and creator codes.
- "File Manager" (page 13) describes the class that enables you to perform many generic file-system operations.
- "File Handle" (page 15) describes the class that provides a wrapper for accessing open files or communications channels.
- "Enumerating A Directory" (page 17) describes how to retrieve the contents of a directory using an `NSDirectoryEnumerator` object.
- "Resolving Aliases" (page 19) describes how to resolve aliases in a file path.
- "Locating Directories on the System" (page 21) explains how to use both Cocoa and Carbon functions to compose paths used in file-system operations.

Portable File-System Operations

When you find, move, or delete files, do not assume that files are in set locations. If you include hard-coded paths, the portability and longevity of your code could suffer. The reasons behind this caution range from the obvious to the not-so-obvious.

As an example of the obvious, consider the user's home directory. There is no way your program can "know" (before doing the appropriate queries, that is) the name of the logged-in user's home directory and the location of that directory. The home directory can be named anything, the volume of the home directory can be named anything (and does not have to be the root volume), and the path between the volume and the home directory can be anything.

Even hard-coding system directories may cause a problem. For example, Apple's public frameworks are currently located at `/System/Library/Frameworks` on the root volume. But this location could change at a future date (who knows?), and any legacy code that relies on hard-coded paths to these frameworks would break.

For these reasons, before you perform many file-system operations, you should use one of the APIs that help you to locate standard directories in the file system. The first API is a Cocoa one: some of the functions and related constants defined in `NSPathUtilities.h`. The second is a Carbon API: the Folder Manager's `FSFindFolder` function, which is defined in the Core Services umbrella framework (Carbon Core subframework). Both APIs allow you to search for frameworks, applications, documents, and other resources across file-system domains. The two APIs are also similar in that both use constants to specify well-known directories and domains.

However, there are some differences in the domains and directories covered. Although you may use one or the other, the Cocoa API is recommended. See the task "Locating Directories on the System" (page 21) for the procedures (and examples) of using these APIs in a Cocoa program.

HFS File Types

Classic HFS file type and creator codes are specified by a 32-bit unsigned integer (`OSType`), usually represented as four characters, such as `GIFf` or `MooV`. Compilers in Mac OS X automatically convert the 4-character representation to the integer representation when they encounter four characters within single quotes, such as `'GIFf'`.

```
OSType aFileType = 'GIFf';
```

The `NSFileManager` class provides methods for setting and getting the attributes of documents, including HFS type and creator codes. To retrieve HFS type and creator codes, use the `fileHFSTypeCode` and `fileHFSCreatorCode` methods. To set type and creator codes, use the `changeFileAttributes:atPath:` method, passing a dictionary that contains the `NSFileHFSCreatorCode` and `NSFileHFSTypeCode` keys. The values of these keys should be `NSNumber` objects that hold `OSType` values.

```
NSNumber *aFileType = [NSNumber numberWithUnsignedLong:'GIFf'];
```

Other parts of Cocoa, such as `NSOpenPanel`, have traditionally been restricted to specifying file types by filename extension, such as `"gif"` or `"mov"`. Because of the conflicting data types—strings versus integers—these filename-extension APIs cannot use HFS file types directly. These methods can instead accept HFS file types that have been properly encoded into `NSString`.

The Foundation Kit defines the following functions to convert between a classic HFS file type and an encoded string:

`NSFileTypeForHFSTypeCode`	Returns a string encoding a file type code.
`NSHFSTypeCodeFromFileType`	Returns a file type code.
`NSHFSTypeOfFile`	Returns a string encoding a file type.

The first two functions convert between an HFS file type and an encoded HFS string. The final function returns the encoded HFS string for a specific file.

As an example, when specifying the file types that can be opened by an `NSOpenPanel` object, you can create the array of file types as follows to include any text documents with the appropriate HFS type code:

```
NSArray *fileTypes = [NSArray arrayWithObjects: @"txt", @"text",
                        NSFileTypeForHFSTypeCode('TEXT'), nil];
```

When you need to find out whether the HFS file type or extension of a particular file is part of a set of HFS file types and extensions, you can use a function similar to this one:

```
BOOL FileValid(NSString *fullFilePath)
{
    // Create an array of strings specifying valid extensions and HFS file types.
    NSArray *fileTypes = [NSArray arrayWithObjects:
                                        @"txt",
                                        @"text",
                                        NSFileTypeForHFSTypeCode('TEXT'),
                                        nil];

    // Try to get the HFS file type as a string.
    NSString *fileType = NSHFSTypeOfFile(fullFilePath);

    if ([fileType isEqualToString:@"''"])
    {
        // No HFS type; get the extension instead.
        fileType = [fullFilePath pathExtension];
    }

    return [fileTypes containsObject:fileType];
}
```

For information about setting HFS file types with `NSDocument` subclasses, see Saving HFS Type and Creator Codes.

File Manager

This article describes the `NSFileManager` class and how you use it.

Overview

`NSFileManager` enables you to perform many generic file-system operations. With it you can:

- Create directories and files.
- Extract the contents of files (as `NSData` objects).
- Change your current working location in the file system.
- Copy, move, and link files and directories.
- Remove files, links, and directories.
- Determine the attributes of a file, a directory, or the file system.
- Set the attributes of a file or directory.
- Make and evaluate symbolic links.
- Determine the contents of directories.
- Compare files and directories for equality.

Besides offering a useful range of generic functionality, the `NSFileManager` API insulates an application from the underlying file system. An important part of this insulation is the encoding of file names (in, for example, Unicode, ISO Latin1, and ASCII). There is a default `NSFileManager` object for the file system; this object responds to all messages that request a operation on the associated file system.

The pathnames specified as arguments to `NSFileManager` methods can be absolute or relative to the current directory (which you can determine with `currentDirectoryPath` and set with `changeCurrentDirectoryPath:`). However, pathnames cannot include wildcard characters.

Note: An absolute pathname starts with the root directory of the file system, represented by a slash (/), and ends with the file or directory that the pathname identifies. A relative pathname is relative to the current directory, the directory in which you are working and in which saved files are currently stored (if no pathname is specified). Relative pathnames start with a subdirectory of the current directory—without an initial slash—and end with the name of the file or directory the pathname identifies.

Broken Links

Constructing a pathname to a file does not guarantee that the file exists at that path. Specifying a path results in one of the following possibilities:

- A file exists at that path
- A link to a file exists at that path
- A broken link exists at that path
- No file exists at that path

If the pathname specifies a valid file or link, you can obtain information about the file using the methods of this class. If the pathname specifies a broken link, you can still use `fileAttributesAtPath:traverseLink:` to obtain attributes for the link itself (by specifying `NO` for the *`traverseLink`* argument). However, the methods `fileExistsAtPath:` and `fileAttributesAtPath:traverseLink:` (with `YES` specified for the *`traverseLink`* argument) return `nil` when the pathname specifies a broken link. Other methods return appropriate errors—see the method descriptions for specific information. Regardless of whether a link is broken or valid, the link still appears in directory listings.

Path Utilities

`NSFileManager` methods are commonly used together with path-utility methods implemented as a category on `NSString`. These methods extract the components of a path (directory, file name, and extension), create paths from those components, “translate” path separators, clean up paths containing symbolic links and redundant slashes, and perform similar tasks. Where your code manipulates strings that are part of file-system paths, it should use these methods. See the specification of the `NSString` class for details.

File Handle

`NSFileHandle` objects provide an object-oriented wrapper for accessing open files or communications channels. The objects you create using this class are called file handle objects. Because of the nature of class clusters, file handle objects are not actual instances of the `NSFileHandle` class but of one of its private subclasses. Although a file handle object's class is private, its interface is public, as declared by the abstract superclass `NSFileHandle`.

Generally, you instantiate a file handle object by sending one of the `fileHandle...` messages to the `NSFileHandle` class object. These methods return a file handle object pointing to the appropriate file or communications channel. As a convenience, `NSFileHandle` provides class methods that create objects representing files and devices in the file system and that return objects representing the standard input, standard output, and standard error devices. You can also create file handle objects from file descriptors (such as found on BSD systems) using the `initWithFileDescriptor:` and `initWithFileDescriptor:closeOnDealloc:` methods. If you create file handle objects with these methods, you "own" the represented descriptor and are responsible for removing it from system tables, usually by sending the file handle object a `closeFile` message.

An `NSFileHandle` is an object that represents an open file or communications channel. It enables programs to read data from or write data to the represented file or channel. You can use other Cocoa methods for reading from and writing to files—`NSFileManager`'s `contentsAtPath:` and `NSData`'s `writeToFile:options:error:` are but a couple of examples. Why would you use `NSFileHandle` then? What are its advantages?

- `NSFileHandle` gives you greater control over input/output operations on files. It allows more manipulative operations on and within open files, such as seeking, truncating. and reading and writing at an exact position within a file (the file pointer). Other methods read or write a file in its entirety; with `NSFileHandle`, you can range over an open file and insert, extract, and delete data.
- The scope of `NSFileHandle` is not limited to files. It provides the only Foundation object that can read and write to communications channels such as those implemented by sockets, pipes, and devices.
- `NSFileHandle` makes possible asynchronous background communication. With it a program can connect to, and read from, a socket in a separate thread. (See "Background Inter-Process Communication Using Sockets" (page 15) below for details on how this is done.)

Instances of `NSPipe`, a class closely related to `NSFileHandle`, represent pipes: unidirectional interprocess communication channels. See the `NSPipe` reference for details.

Background Inter-Process Communication Using Sockets

Sockets are full-duplex communication channels between processes either local to the same host machine or where one process is on a remote host. Unlike pipes, in which data goes in one direction only, sockets allow processes both to send and receive data. `NSFileHandle` facilitates communication over stream-type sockets by providing mechanisms run in background threads that accept socket connections and read from sockets.

`NSFileHandle` currently handles only communication through stream-type sockets. If you want to use datagrams or other types of sockets, you must create and manage the connection using native system routines.

The process on one end of the communication channel (the server) starts by creating and preparing a socket using system routines. These routines vary slightly between BSD and non-BSD systems, but consist of the same sequence of steps:

1. Create a stream-type socket of a certain protocol.

2. Bind a name to the socket.

3. Adding itself as an observer of `NSFileHandleConnectionAcceptedNotification`.

4. Sending `acceptConnectionInBackgroundAndNotify` to this file handle object. This method accepts the connection in the background, creates a new `NSFileHandle` object from the new socket descriptor, and posts an `NSFileHandleConnectionAcceptedNotification`.

In a method implemented to respond to this notification, the server extracts the `NSFileHandle` object representing the "near" socket of the connection from the notification's *`userInfo`* dictionary; it uses the `NSFileHandleNotificationFileHandleItem` key to do this.

Typically the other process (the client) then locates the named socket created by the first process. Instead of accepting a connection to the socket by calling the appropriate system routine, the client creates an `NSFileHandle` object using the socket identifier as argument to `initWithFileDescriptor:`.

The client can now send data to the other process over the communications channel by sending `writeData:` to the `NSFileHandle` instance. (Note that `writeData:` can block.) The client can also read data directly from the `NSFileHandle`, but this would cause the process to block until the socket connection was closed, so it is usually better to read in the background. To do this, the process must:

1. Add itself as an observer of `NSFileHandleReadCompletionNotification` or `NSFileHandleReadToEndOfFileCompletionNotification`.

2. Send `readInBackgroundAndNotify` or `readToEndOfFileInBackgroundAndNotify` to this `NSFileHandle` object. The former method sends a notification after each transmission of data; the latter method accumulates data and sends a notification only after the sending process shuts down its end of the connection.

3. In a method implemented to respond to either of these notifications, the process extracts the transmitted or accumulated data from the notification's *`userInfo`* dictionary by using the `NSFileHandleNotificationDataItem` key.

4. If you wish to keep getting notified you'll need to again call `readInBackgroundAndNotify` in your observer method.

You close the communications channel in both directions by sending `closeFile` to the `NSFileHandle` object; either process can partially or totally close communication across the socket connection with a system-specific shutdown command.

Enumerating A Directory

You use an `NSDirectoryEnumerator` object to enumerate the contents of a directory. It returns the pathnames of all files and directories contained within that directory. The pathnames are relative to the directory. This enumeration is recursive, including the files of all subdirectories, and crosses device boundaries. It does not resolve symbolic links or attempt to traverse symbolic links that point to directories.

`NSDirectoryEnumerator` is an abstract class, a cover for a private concrete subclass tailored to the file system's directory structure. You cannot create an `NSDirectoryEnumerator` object directly—you use `NSFileManager`'s `enumeratorAtPath:` method to retrieve a suitable instance.

To get the next item from the `NSDirectoryEnumerator`, invoke the `NSEnumerator` method `nextObject`. The methods declared by `NSDirectoryEnumerator` return attributes—both of the parent directory and the current file or directory—and allow you to control recursion into subdirectories.

The following example enumerates the contents of the user's home directory and processes files; if, however, it comes across RTFD file packages, it skips recursion into them:

```
NSDirectoryEnumerator *direnum = [[NSFileManager defaultManager]
        enumeratorAtPath:NSHomeDirectory()];
NSString *pname;
while (pname = [direnum nextObject])
{
    if ([[pname pathExtension] isEqualToString:@"rtfd"])
    {
         /* don't enumerate this directory */
         [direnum skipDescendents];
    }
    else
    {
        /* ...process file here... */
    }
}
```

Resolving Aliases

Unlike symbolic links, Mac OS X aliases are not handled automatically by Cocoa. This article explains how to resolve aliases in a path.

Paths returned by Cocoa classes such as `NSOpenPanel` may contain alias references that you must resolve before using. To resolve aliases in a path contained in an `NSString`, you need to first convert the string to a URL, convert the URL to an FSRef, resolve the alias, and reverse the series of conversions to yield another `NSString`. To perform the necessary conversions, you need to use the URL and alias services provided in `<CoreServices/CoreServices.h>`. The following code fragment uses `FSResolveAliasFile` to resolve any aliases in `path` and stores the resolved path in `resolvedPath`:

```
NSString *path;    // Assume this exists.
NSString *resolvedPath = nil;

CFURLRef url = CFURLCreateWithFileSystemPath
                (kCFAllocatorDefault, (CFStringRef)path, kCFURLPOSIXPathStyle,
 NO);
if (url != NULL)
{
    FSRef fsRef;
    if (CFURLGetFSRef(url, &fsRef))
    {
        Boolean targetIsFolder, wasAliased;
        OSErr err = FSResolveAliasFile (&fsRef, true, &targetIsFolder,
&wasAliased);
        if ((err == noErr) && wasAliased)
        {
            CFURLRef resolvedUrl = CFURLCreateFromFSRef(kCFAllocatorDefault,
&fsRef);
            if (resolvedUrl != NULL)
            {
                resolvedPath = (NSString*)
                        CFURLCopyFileSystemPath(resolvedUrl,
kCFURLPOSIXPathStyle);
                CFRelease(resolvedUrl);
            }
        }
    }
    CFRelease(url);
}

if (resolvedPath == nil)
{
    resolvedPath = [[NSString alloc] initWithString:path];
}
```

The second argument to `FSResolveAliasFile` specifies whether you want the function to resolve all aliases in a chain (for example, an alias file that refers to an alias file and so on), stopping only when it reaches the target file.

Locating Directories on the System

When you programmatically locate files and directories in the file system—such as for copy, move, and delete operations—you should always try to avoid hard-coded paths in your code. (The document "Portable File-System Operations" (page 9) gives the reasons for this recommendation.) As much as possible, obtain the directory locations used in file-system operations through functions provided by Cocoa or other application environments.

Ideally, you should use the Cocoa functions and constants defined in `NSPathUtilities.h`. They are easier to use and more efficient in Cocoa code. The primary function for obtaining paths to standard directories is `NSSearchPathForDirectoriesInDomains`. This function takes three parameters:

- A constant that identifies the name or type of directory (for instance, `Library`, `Documents`, `Applications`, the directory used for demo applications)
- A constant that identifies the file system domain (User, System, Local, Network) or indicating all domains
- A flag that indicated whether to expand any tildes (identifying home directories) in the returned path or paths

The `NSSearchPathForDirectoriesInDomains` function returns an array of paths (as `NSString` objects). In general this function returns multiple values if the parameters imply multiple locations. However, don't make any assumptions as to the number of paths returned. Some domains or locations might be made obsolete over time, or other new domains or locations might be added (while preserving the older ones); in either case, the number of paths in the returned array might increase or decrease. Simply look at all of the returned values if you want to enumerate all of the files. If you just want to copy or move a file to a location and multiple paths are returned, use the first one in the array.

Listing 1 illustrates how you might use `NSSearchPathForDirectoriesInDomains` before moving a file with `NSFileManager`'s `copyItemAtPath:toPath:error:` (prior to Mac OS X v10.5, you would use `copyPath:toPath:handler:`).

Listing 1 Using NSSearchPathForDirectoriesInDomains

```
- (BOOL)makeUserCopyOfFile(NSString* sourcePath) error:(NSError **)outError
{
    NSArray *paths;
    NSFileManager *mgr = [NSFileManager defaultManager];

    paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
NSUserDomainMask, YES);
    if ([paths count] > 0)
    {
        // only copying one file
        NSString *destinationPath = [[paths objectAtIndex:0]
                    stringByAppendingPathComponent:[sourcePath
lastPathComponent]];
        if (![mgr copyItemAtPath:toPath:destinationPath error:outError])
        {
            return NO;
        }
        return YES;
    }
    // create a suitable NSError object to return in outError
    return NO;
}
```

The NSPathUtilities.h header file also defines other functions that you can use to obtain the current logged-in user's home directory (NSHomeDirectory), the home directory of a specified user (NSHomeDirectoryForUser), and the directory used for temporary storage (NSTemporaryDirectory).

You can also use functions of the Carbon Folder Manager to get paths to standard directories in the file system. (The Folder Manager is defined in Folders.h in the Carbon Core framework, which is a subframework of the Core Services umbrella framework.)

Important: Carbon Folder Manager functions are not available on iPhone.

Perhaps the most useful function for Cocoa programs is FSFindFolder. This function (like other Folder Manager functions) returns the found path typed as a FSRef. To use this value in Cocoa, you must first convert it to a Core Foundation CFURL object, then to a CFString object, and finally to an NSString object. "Using FSFindFolder" shows how you might call FSFindFolder and convert the returned value to obtain a path used in moving a file to the user's desktop.

Listing 2 Using FSFindFolder

```
- (BOOL)moveFileToUserDesktop:(NSString *filePath) error:(NSError **)outError
{

    CFURLRef        desktopURL;
    FSRef           desktopFolderRef;
    CFStringRef     desktopPath;
    OSErr           err;
    NSFileManager   *mgr = [NSFileManager defaultManager];

    err = FSFindFolder(kUserDomain, kDesktopFolderType, kDontCreateFolder,
&desktopFolderRef);
    if (err == noErr)
    {
        desktopURL = CFURLCreateFromFSRef(kCFAllocatorSystemDefault,
&desktopFolderRef);
        if (desktopURL)
        {
            desktopPath = CFURLCopyFileSystemPath (desktopURL,
kCFURLPOSIXPathStyle);
            NSString *destinationPath = [(NSString *)desktopPath
                                 stringByAppendingPathComponent:[filePath
lastPathComponent]];
            if (![mgr moveItemAtPath:filePath toPath:destinationPath
error:outError])
            {
                return NO;
            }
            if (desktopPath)
            {
                CFRelease(desktopPath);
            }
            CFRelease(desktopURL);
            return YES;
        }
        else
        {
            // create a suitable NSError object to return in outError
            return NO;

        }
    }
    // create a suitable NSError object to return in outError
    return NO;
}
```

Similar to Cocoa's `NSSearchPathForDirectoriesInDomains` function, the `FSFindFolder` function includes as parameters constants specifying standard directories and file-system domains.

Moving files to the Trash: To move items to the Trash, you should use `NSWorkspace`'s `performFileOperation:source:destination:files:tag` method using the operation `NSWorkspaceRecycleOperation`.

Document Revision History

This table describes the changes to *Low-Level File Management Programming Topics*.

Date	Notes
2009-03-05	Corrected a broken link.
2008-05-05	Added note that FSFindFolder is not available on iPhone.
2008-02-08	Removed references to deprecated Java classes.
2006-07-24	Added code sample showing how to inquire about the HFS file type of a file.
	Added code sample to "HFS File Types" (page 11).
2005-08-11	Corrected example code. Changed title from "Low-Level File Management."
2004-12-02	Corrected typo in "Locating Directories on the System."
2004-08-31	Removed link to missing article and made minor edits.
2002-11-12	Revision history was added to existing document.