Technical Note TN2015

Locating Application Support Files under Mac OS X

CONTENTS

Applications may need additional files, such as dictionaries, plug-ins, scripts, etc., in order to run properly. Apple recommends that such files are stored in special folders such as the Application Support, Scripting Additions, Scripts, and Internet Plug-Ins folders, which can be located using the `FindFolder` API that works transparently on Mac OS 8, 9, and Mac OS X, but some developers prefer to store those files in folders located adjacent to the related application. In order to locate those files, the application has to locate itself first. The different methods used in the past by developers to do accomplish this must be revised on Mac OS X.

If your application is running as a Classic application on Mac OS X, then the current method used to locate itself is still going to work. That's the Classic application compatibility; you don't need to read this Technote further.

If you carbonized your application to benefit from the multiple advantages of running as a separate modern process on Mac OS X, then you need to read this Technote.

Updated: [Mar 27 2001]

Using Packages

If you already carbonized your application but you are not delivering it as a package, then we strongly recommend that you do so. Packages have been supported since Mac OS 9.0 and are the preferred way of distributing software. Even if you still are building your application as a single file containing a CFM-based data fork and a resource fork, such as is generated by the popular MetroWerks CodeWarrior, you can still deliver it as a minimal package that will run on both Mac OS 9.x and Mac OS X:

```
folder MyCFMPkgAppl.app
alias     MyCFMPkgAppl alias
folder    Contents
folder       MacOSClassic
appl.           MyCFMPkgAppl
```

The advantage of using the package structure is that you can then use the `CFBundle` APIs to locate the package and/or its elements, and your code will be compatible when you eventually adopt the Mach-O code format. Furthermore, you will gain the full benefits of running on Mac OS X, which is only distributed in a package structure by Apple's ProjectBuilder:

```
folder MyMachOPkgAppl.app
alias     MyMachOPkgAppl alias
folder    Contents
folder       MacOS
appl.           MyMachOPkgAppl
folder       Resources
```

```
file            MyMachOPkgAppl.rsrc
…               …
…           …
```

Note:
For more details on packages, read Technote TN1188, "Packages in Mac OS 9"

If you don't use the package structure now, then you will have to change your code again when you switch from CFM to Mach-O in the future even though you would be using the same Carbon APIs.

CFBundle APIs

There are 3 popular methods used by developers to locate the running application. There are three equivalent modern CFBundle APIs to replace them and one extra Process Manager API available on Mac OS X only.

Here are the 3 CFBundle APIs:

```
OSErr GetApplicationPackageFSSpecFromBundle(FSSpecPtr theFSSpecPtr)
 {
 OSErr err = fnfErr;
 CFBundleRef myAppsBundle = CFBundleGetMainBundle();
 if (myAppsBundle == NULL) return err;
 CFURLRef myBundleURL = CFBundleCopyBundleURL(myAppsBundle);
 if (myBundleURL == NULL) return err;


 FSRef myBundleRef;
 Boolean ok = CFURLGetFSRef(myBundleURL, &myBundleRef);
 CFRelease(myBundleURL);
 if (!ok) return err;


 return FSGetCatalogInfo(&myBundleRef, kFSCatInfoNone,
      NULL, NULL, theFSSpecPtr, NULL);
 }

OSErr GetApplicationResourceFSSpecFromBundle(FSSpecPtr theFSSpecPtr)
 {
 OSErr err = fnfErr;
 CFBundleRef myAppsBundle = CFBundleGetMainBundle();
 if (myAppsBundle == NULL) return err;
 CFURLRef myBundleURL = CFBundleCopyResourcesDirectoryURL(myAppsBundle);
 if (myBundleURL == NULL) return err;


 FSRef myBundleRef;
 Boolean ok = CFURLGetFSRef(myBundleURL, &myBundleRef);
 CFRelease(myBundleURL);
 if (!ok) return err;


 return FSGetCatalogInfo(&myBundleRef, kFSCatInfoNone,
      NULL, NULL, theFSSpecPtr, NULL);
 }

OSErr GetExecutableParentFSSpecFromBundle(FSSpecPtr theFSSpecPtr)
 {
 OSErr err = fnfErr;
 CFBundleRef myAppsBundle = CFBundleGetMainBundle();
```

```
 if (myAppsBundle == NULL) return err;
 CFURLRef myBundleURL = CFBundleCopyExecutableURL(myAppsBundle);
 if (myBundleURL == NULL) return err;


 FSRef myBundleRef;
 Boolean ok = CFURLGetFSRef(myBundleURL, &myBundleRef);
 CFRelease(myBundleURL);
 if (!ok) return err;


 return FSGetCatalogInfo(&myBundleRef, kFSCatInfoNone,
      NULL, NULL, theFSSpecPtr, NULL);
 }
```

Listing 1 - CFBundle APIs

The Process Manager extra API available only on Mac OS X:

```
OSErr GetApplicationBundleFSSpec(FSSpecPtr theFSSpecPtr)
 {
 OSErr err;
 ProcessSerialNumber psn;
 err = GetCurrentProcess(&psn);
 if (err != noErr) return err;


 FSRef location;
 err = GetProcessBundleLocation(&psn, &location);
 if (err != noErr) return err;


 return FSGetCatalogInfo(&location, kFSCatInfoNone,
      NULL, NULL, theFSSpecPtr, NULL);
 }
```

Listing 2 - Process Manager API

Older Methods

And here are the three older methods used by developers in the past:

```
OSErr GetApplicationPackageFSSpec(FSSpecPtr theFSSpecPtr)
 {
 OSErr err;
 Str255 applName;
 ProcessSerialNumber psn;
 ProcessInfoRec info;
 info.processInfoLength = sizeof(ProcessInfoRec);
 info.processName = applName;
 info.processAppSpec = theFSSpecPtr;


 err = GetCurrentProcess(&psn);
 if (err != noErr) return err;


 err = GetProcessInformation(&psn, &info);
 return err;
 }
```

```
OSErr GetApplicationResourceFSSpec(OSType creator,
     FSSpecPtr theFSSpecPtr)
 {
 Handle creatorHandle;
 OSErr err;
 Str255 applName;
 FCBPBRec theFCBPBRec;
 // creator is the 4-byte creator code of your application
 // necessarily unique and sure to belong to your application
 // resource fork only.
 creatorHandle = GetResource(creator, 0);
 if (creatorHandle == NULL) return resNotFound;
 theFCBPBRec.ioCompletion = nil;
 theFCBPBRec.ioNamePtr = applName;
 theFCBPBRec.ioRefNum = HomeResFile(creatorHandle);
 theFCBPBRec.ioFCBIndx = 0;


 err = PBGetFCBInfoSync(&theFCBPBRec);
 if (err != noErr) return err;


 err = FSMakeFSSpec(theFCBPBRec.ioFCBVRefNum,
      theFCBPBRec.ioFCBParID,
      applName, theFSSpecPtr);
 return err;
 }

OSErr GetExecutableParentFSSpec(OSType creator,
     FSSpecPtr theFSSpecPtr)
 {
 Handle creatorHandle;
 OSErr err;
 Str255 volName;
 short vRefNum;
 long dirID;
 // creator is the 4-byte creator code of your application
 // necessarily unique and sure to belong to your application
 // resource fork only.
 creatorHandle = GetResource(creator, 0);
 if (creatorHandle == NULL) return resNotFound;


 err = HGetVol(volName, &vRefNum, &dirID);
 if (err != noErr) return err;
 // the vRefNum returned by HGetVol is not
 // really a volume Reference Number
 // so we need the following extra call
 // which assumes that the executable and
 // the resources are on the same volume


 err = GetVRefNum(HomeResFile(creatorHandle), &vRefNum);
 if (err != noErr) return err;


 err = FSMakeFSSpec(vRefNum, dirID, nil, theFSSpecPtr);
 return err;
 }
```

Listing 3 - Older methods

As an anecdote, it is worthwhile to mention that there was a fourth way which used undocumented side effects (read: unreliable) of a File Manager routine. Specifically, calling `FSMakeFSSpec(0, 0, nil, theFSSpecPtr)` very early

after the application launch would fill the FSSpec with the default directory of the default volume, and the folder containing the application.

The Results

To illustrate the advantages of the packaging and CFBundle APIs, let's see what each method returns in all the following cases:

case 1: Carbon CFM Application distributed as a file running on Mac OS 9.

case 2: Carbon CFM Application distributed as a file running on Mac OS X.

case 3: Carbon CFM Application distributed as a package running on Mac OS 9.

case 4: Carbon CFM Application distributed as a package running on Mac OS X.

the ultimate goal:

case 5: Carbon Mach-O Application distributed as a package running on Mac OS X.

and let's call the CFBundle methods: 1.a, 1.b, and 1.c, the Process Manager method: 2, and the older methods: 3.a, 3.b, and 3.c.

We fill the following table with the object pointed by the FSSpec:

	1.a	1.b	1.c	2	3.a	3.b	3.c
case 1	appl folder	N/A	appl file	N/A	appl file	appl file	appl folder
case 2	appl folder	appl folder	appl file	N/A	appl file	appl file	appl folder
case 3	.app "folder"	N/A	appl binary	N/A	appl binary	appl binary	MacOSClassic
case 4	.app "folder"	N/A	appl binary	N/A	appl binary	appl binary	MacOSClassic
case 5	.app "folder"	Resources	appl binary	.app "folder"	appl binary	rsrc file	MacOS

When the application is delivered as a package, locating the application binary (the file containing the object code) is not a great help since it would be dangerous to assume that the structure of a package will never change. Thus, the only safe way to locate the application as the user sees it (its icon, a.k.a., the .app "folder") is to use either method 1.a or method 2.

Depending on your current distribution constraints, i.e., which platforms you're targeting, you will have to choose whichever method is the most appropriate for you, but as you can see in the table above, the best way to pave the route to case 5 is to adopt the package structure and use the CFBundle APIs even if you still temporarily use the CFM format.

Back to top

References

Technote TN1188, "Packages in Mac OS 9"

Back to top

Downloadables

Acrobat version of this Note (K). Download

Back to top