Technical Note TN1085
Using the Drag Manager to Interact with and Manipulate File System Entities

CONTENTS

The Drag Manager defines two data flavors for interacting with and manipulating file system entities. While *The Drag Manager Programmer's Guide* explains these flavors, it does not provide sufficient detail for a complete understanding of how to use them.

Developers who are interested in "teaching" (or even those who have already taught) their applications to interact with and manipulate file system entities via the Drag Manager should read this Technote.

This Technote assumes you are familiar with the material in the *Drag Manager Programmer's Guide* and that, in particular, you have read pages 2-36 and 2-37 and understand the operation of a `DragSendDataProc`, which is documented starting on the bottom of page 2-72. Also, some familiarity with the AppleEvent Manager data structure `AEDesc` is assumed. (The AppleEvent Manager is documented in chapter 3 of *Inside Macintosh: Interapplication Communication*.) Finally, familiarity with the File Manager call `PBGetCatInfo` is recommended.

You can download a complete version of the code snippets in this Technote, FinderDragPro Metrowerks Project, as well as the Drag Manager Programmer's Guide, by clicking on the item here or by clicking on the appropriate icon in the Downloadables section at the end of this Note.

Updated: [Feb 06 1997]

Introducing `flavorTypeHFS` and `flavorTypePromiseHFS`

There are two data flavors for interacting with and manipulating file system entities: `flavorTypeHFS` and `flavorTypePromiseHFS`. Despite similar names, their meanings are quite different. The key difference is that for one, the file exists, while for the other the file does not yet exist.

Putting `flavorTypeHFS` data, which refers to an existing file, into a `DragReference` is like saying "I know of an existing file (which I may or may not have created myself) in which drag receivers might be interested." Putting `flavorTypePromiseHFS` data into a `DragReference` is like saying "I'm willing to create a new file as soon as somebody (a drag receiver) tells me where to put it."

Note:
The Drag Manager has the concept of "promising" data to a `DragReference`. Do not confuse this with `flavorTypePromiseHFS`. The two kinds of promise are different, and promising `flavorTypeHFS` data to a `DragReference` has nothing to do with `flavorTypePromiseHFS`.

This can be especially confusing when an application promises `flavorTypePromiseHFS` data to a `DragReference`; the promised data is in turn a promise to the receiving application to provide data which refers to a newly created file - a triple indirection.

In this Technote, I make an effort to avoid using the word "promise" in more than one sense at a time; nevertheless, read carefully.

Important:
The format of Drag Manager flavor data is only conventional. This means that nothing in the API forces senders and receivers to use it correctly. As always, you need to be careful to implement these flavors strictly by the book.

The situation may even be worse. This document came into existence several years after the release of the Drag Manager. As a result, it's been difficult for early adopters of the Drag Manager to implement these flavors properly. And there may be applications which aren't as conscientious as yours. You should be extra careful to check error return values and build assertions into your code, so that your app is ready to cope with other apps which unexpectedly deviate from the conventions documented in this Note.

Back to top

Using `flavorTypeHFS`

In theory, using`flavorTypeHFS` data appears simple, but in practice there are a few tricks you need to know. The following sections document a few of those tricks for you. For quick reference, here's a copy of the `HFSFlavor` declaration from <`Drag.h`>:

```
struct HFSFlavor
{
    OSType          fileType;       // file type
    OSType          fileCreator;    // file creator
    unsigned short  fdFlags;        // Finder flags
    FSSpec          fileSpec;       // file system specification
};

typedef struct HFSFlavor HFSFlavor;
```

Back to top

Sending flavorTypeHFS

To originate a drag containing `flavorTypeHFS` data, you must first declare an `HFSFlavor` record. This record contains an `FSSpec` and a few other fields which allow some potential drag receivers to avoid calling `FSpGetFInfo`.

The next step is to initialize the `fileSpec` field appropriately, then decide whether the data refers to a file. If so, simply set the `fileType`, `fileCreator`, and `fdFlags` fields to match the appropriate information for the file. If the `flavorTypeHFS` data refers to a directory or volume, set the `fileType` and `fileCreator` fields in the `HFSFlavor` record according to Table 1:

Table 1. *The `fileType` and `fileCreator` fields for the `HFSFlavor` record*

entity type	fileCreator	fileType
directory (folder)	MACS	fold
volume (disk)	MACS	disk

These values are a hint to potential drag receivers that they are dealing with something other than a file. They are the same as the ones you would use in your application's bundle resource to let Finder know your app will accept folders and disks dropped onto your application's icon.

```
pascal OSErr MakeHFSFlavor
    (short vRefNum, long dirID, ConstStr255Param path,
        HFSFlavor register *hfsFlavorP)
{
    OSErr err = noErr;

    if (!(err = FSMakeFSSpec
        (vRefNum,dirID,path,&(hfsFlavorP->fileSpec))))
    {
        CInfoPBPtr cipbp =
            (CInfoPBPtr) NewPtrClear (sizeof (*cipbp));
        if (!(err = MemError ( )))
        {
            cipbp->hFileInfo.ioVRefNum  =
                hfsFlavorP->fileSpec.vRefNum;
            cipbp->hFileInfo.ioDirID    =
                hfsFlavorP->fileSpec.parID;
            cipbp->hFileInfo.ioNamePtr  =
                hfsFlavorP->fileSpec.name;

            if (!(err = PBGetCatInfoSync (cipbp)))
            {
                hfsFlavorP->fdFlags =
                    cipbp->hFileInfo.ioFlFndrInfo.fdFlags;

                if (hfsFlavorP->fileSpec.parID == fsRtParID)
                {
                    hfsFlavorP->fileCreator     = 'MACS';
                    hfsFlavorP->fileType        = 'disk';
                }
                else if (cipbp->hFileInfo.ioFlAttrib & ioDirMask)
                {
                    hfsFlavorP->fileCreator     = 'MACS';
                    hfsFlavorP->fileType        = 'fold';
                }
                else
                {
                    hfsFlavorP->fileCreator     =
                        cipbp->hFileInfo.ioFlFndrInfo.fdCreator;
                    hfsFlavorP->fileType        =
                        cipbp->hFileInfo.ioFlFndrInfo.fdType;
                }
            }

            DisposePtr ((Ptr) cipbp);
            if (!err) err = MemError ( );
        }
    }

    return err;
}
```

Snippet 1. Deciding how to set the `fileCreator` and `fileType` fields

Back to top

Coping with Finder Bugs

Dragging `flavorTypeHFS` data from your application to Finder has always supposed to have been possible. However, Finder bugs have prevented most applications from successfully using this feature.

From Finder's perspective, there are two cases for receiving `flavorTypeHFS` data. The second case is more interesting.

1. If the drop location is on the same volume as the `flavorTypeHFS` data, Finder simply moves the file to the drop location.
2. If the the drop location is on a different volume, Finder needs to copy the file to the new volume.

Finder is AppleEvent-intensive. It sends itself AppleEvents to order itself to do all sorts of things, including displaying the progress window for copying files. However, Finder's drag-receiving code mistakenly sends these particular AppleEvents to the front process instead of the current process. The front process is generally the application which originated the drag. Since the application does not have handlers for these events, AppleEvent Manager returns an error to Finder's `AESend` call and Finder cancels the entire operation.

Until this bug is fixed, your application can work around the problem by "handling" these AppleEvents. On systems under which Finder has been fixed, the handler will simply lie dormant in your app, because the AppleEvents will be sent to Finder, not your app. Unfortunately, it doesn't do any good to "reflect" these events back to the Finder; trust us, we've tried. This means you'll have to do without the progress dialog, but this is better than abject failure.

```
pascal OSErr BogusFinderEventHandler
    (const AppleEvent *, AppleEvent *, long)
{
    return noErr; // just drop that bad boy on the floor
}

pascal OSErr InstallBogusFinderEventHandler (void)
{
    OSErr err = noErr;

    static AEEventHandlerUPP bogusFinderEventHandlerUPP;

    if (!bogusFinderEventHandlerUPP)
    {
        bogusFinderEventHandlerUPP =
            NewAEEventHandlerProc (BogusFinderEventHandler);

        if (!bogusFinderEventHandlerUPP)
            err = nilHandleErr;
        else
        {
            err = AEInstallEventHandler
                ('cwin','****',bogusFinderEventHandlerUPP,0,false);

            if (err)
            {
                DisposeRoutineDescriptor (bogusFinderEventHandlerUPP);
                bogusFinderEventHandlerUPP = nil;
            }
        }
    }

    return err;
}
```

Snippet 2. Receiving bogus AppleEvents from Finder

Back to top

Receiving `flavorTypeHFS`

Receiving `flavorTypeHFS`, often from Finder, is much like receiving any other flavor of data. However, be aware that some applications will offer you a truncated record; they do not provide the unused bytes at the end of the name field of the `fileSpec` field of the `HFSFlavor` record. (This is a bug in the sending application, but if it's not your app, you probably don't have an opportunity to fix it.)

```
static pascal Size MinimumBytesForFSSpec (const FSSpec *fss)
{
    // callers can and do assume this does not move memory
    return sizeof (*fss) - sizeof (fss->name) + *(fss->name) + 1;
}
```

Snippet 3. Calculating minimum bytes for `FSSpec`

(called from snippets #4 , #6, and #14)

```
pascal OSErr GetHFSFlavorFromDragReference
    (DragReference dragRef, ItemReference itemRef,
        HFSFlavor *hfsFlavor)
{
    OSErr err = noErr;

    Size size = sizeof (*hfsFlavor);
    err = GetFlavorData
        (dragRef,itemRef,flavorTypeHFS,hfsFlavor,&size,0);

    if (!err)
    {
        Size minSize = sizeof (*hfsFlavor) -
            sizeof (hfsFlavor->fileSpec);
        minSize += MinimumBytesForFSSpec (&(hfsFlavor->fileSpec));
            // see snippet 3 for MinimumBytesForFSSpec
        if (size < minSize)
            err = cantGetFlavorErr;
    }

    return err;
}
```

Snippet 4. Extracting `flavorTypeHFS` data

Back to top

Using `flavorTypePromiseHFS`

Using `flavorTypePromiseHFS` data is significantly more complicated than using `flavorTypeHFS` data. The chief area of confusion is centered on the multi-part nature of `flavorTypePromiseHFS` data. For quick reference, here's a copy of the `PromiseHFSFlavor` declaration from <`Drag.h`>:

```
struct PromiseHFSFlavor
{
    OSType          fileType;           // file type
    OSType          fileCreator;        // file creator
    unsigned short  fdFlags;            // Finder flags
    FlavorType      promisedFlavor;     // promised flavor
};

typedef struct PromiseHFSFlavor PromiseHFSFlavor;
```

Back to top

Sending `flavorTypePromiseHFS`

Promising to Create a File

Before calling `TrackDrag`, your application should call `AddDragItemFlavor` twice, passing the same `ItemReference` value both times, once for each part of the data.

For the first call, declare a record of type PromiseHFSFlavor and put 'fssP' (0x66737350, kDragPromisedFlavor) in the promisedFlavor field. The *Drag Manager Programmer's Guide* tells you to put any value you like into promisedFlavor, but we're now recommending this specific value. (Details can be found below; if your application already uses something else, don't worry too much right now unless it's 'rWm1'.) Fill in the other fields of the PromiseHFSFlavor record appropriately and add the record to the DragReference, passing flavorTypePromiseHFS for the FlavorType parameter.

With the second call to AddDragItemFlavor, pass 'fssP' for the FlavorType parameter. Pass 0 for the dataPtr and dataSize parameters to set up a promise to be kept later.

```
pascal OSErr AddDragItemFlavorTypePromiseHFS
    (DragReference dragRef, ItemReference itemRef,
        OSType fileType, OSType fileCreator,
            UInt16 fdFlags, FlavorType promisedFlavor)
{
    OSErr err = noErr;

    PromiseHFSFlavor phfs;

    phfs.fileType           = fileType;
    phfs.fileCreator        = fileCreator;
    phfs.fdFlags            = fdFlags;
    phfs.promisedFlavor     = kDragPromisedFlavor;

    if (!(err = AddDragItemFlavor
        (dragRef,itemRef,flavorTypePromiseHFS,
            &phfs,sizeof(phfs),flavorNotSaved)))
    {
        err = AddDragItemFlavor
            (dragRef,itemRef,kDragPromisedFlavor,nil,0,flavorNotSaved);
    }

    return err;
}
```

Snippet 5. Adding flavorTypePromiseHFS data

Important:
Due to a bug in some versions of Finder, your application should add flavorTypePromiseHFS flavor data before any other, followed immediately by the flavor data for the promisedFlavor field. If your application does not add these flavors in this order, Finder will position the file's icon incorrectly.

Note:
If your application hasn't already attached a DragSendDataProc to the DragReference with a call to SetDragSendDataProc, you'll need to add this functionality.

Add any other flavors you might want to provide in this DragReference, and you're ready to call TrackDrag.

Keeping the Promise

When Drag Manager requests a FlavorType equal to the promisedFlavor field of your flavorTypePromiseHFS data, it's your cue to keep your promise by delivering the file. Keeping the promise involves finding out where the drag receiver wants the file to end up, deciding where to create the file, and creating the file. You'll do this in your DragSendDataProc associated with the DragReference.

Getting the Drop Location

First, your DragSendDataProc will need to find out where the drag receiver wants the file. You'll need to call GetDropLocation, which will produce an AEDesc record. The type of the data found in this record is defined by the drag receiver. Finder, for example, provides typeAlias data. To convert this data to an FSSpec, coerce its type to typeFSS and copy the FSSpec data out of the resulting descriptor.

```
pascal OSErr GetDropDirectory (DragReference dragRef, FSSpecPtr fssOut)
{
    OSErr err = noErr;

    AEDesc dropLocAlias = { typeNull, nil };

    if (!(err = GetDropLocation (dragRef,&dropLocAlias)))
    {
        if (dropLocAlias.descriptorType != typeAlias)
            err = paramErr;
        else
        {
            AEDesc dropLocFSS = { typeNull, nil };

            if (!(err = AECoerceDesc
                (&dropLocAlias,typeFSS,&dropLocFSS)))
            {
                // assume MinimumBytesForFSSpec does not move memory
                FSSpecPtr fss = (FSSpecPtr) *(dropLocFSS.dataHandle);
                BlockMoveData (fss,fssOut,MinimumBytesForFSSpec(fss));
                // see snippet 3 for MinimumBytesForFSSpec
                err = AEDisposeDesc (&dropLocFSS);
            }
        }

        if (dropLocAlias.dataHandle)
        {
            OSErr err2 = AEDisposeDesc (&dropLocAlias);
            if (!err) err = err2;
        }
    }

    return err;
}
```

Snippet 6. Extracting the drop folder

Note:
The `FSSpec` data describes a directory; it is not an `FSSpec` you can use for creating your file. To get the directory ID for the file you want to create, use `PBGetCatInfo`, as is done in the function in Appendix C.

If the drop location data is not of `typeAlias`, the call to `AECoerceDesc` will fail. Your `DragSendDataProc` will probably want to provide no data and return an error in this situation. However, be aware that applications other than Finder are free to provide a drop location of `typeAlias` (and some even do), so don't rely on `typeAlias` signifying that Finder is the drop receiver.

Note:
Finder currently has a few bugs having to do with deciding where to allow `flavorTypePromiseHFS` drops. Aliases to folders, aliases to the Trash, and applications which accept the file type presented in the `PromiseHFSFlavor` record will highlight as if they are going to accept a drag. However, they reject the drag when the mouse button is released. In the latter case (applications), the drop location will be an alias to the application file itself. There is no good workaround for this problem.

Note:
Don't try to create the file on a volume other than the one specified by the drop location. Finder will not copy the file to the drop location.

Back to top

Creating the File

Once you've decided where to put the file, you can create it by calling a function like this one:

```
pascal OSErr CreatePromisedFileOrFolder
   (const PromiseHFSFlavor *phfs, const FSSpec *fss,
      ScriptCode scriptTag)
{
   OSErr err = noErr;

   if (phfs->promisedFlavor == kPromisedFlavorFindFile)
      err = paramErr;
   else if (phfs->fileType == 'disk')
      err = paramErr;
   else if (phfs->fileType == 'fold')
      err = CreatePromisedFolder (phfs,fss,scriptTag); // see Snippet 9
   else
      err = CreatePromisedFile (phfs,fss,scriptTag); // see Snippet 8

   return err;
}
```

Snippet 7. Creating the promised file or folder

```
static pascal CreatePromisedFile
    (const PromiseHFSFlavor *phfs, const FSSpec *fss,
        ScriptCode scriptTag)
{
    OSErr err = noErr;

    if (!(err = FSpCreate
        (fss,phfs->fileCreator,phfs->fileType,scriptTag)))
    {
        if (phfs->fdFlags)
        {
            FInfo finderInfo;

            if (!(err = FSpGetFInfo (fss,&finderInfo)))
            {
                finderInfo.fdFlags = phfs->fdFlags;
                err = FSpSetFInfo (fss,&finderInfo);
            }
        }
    }

    return err;
}
```

Snippet 8. Called by snippet #7

```
static pascal CreatePromisedFolder
    (const PromiseHFSFlavor *phfs, const FSSpec *fss,
        ScriptCode scriptTag)
{
    OSErr err = noErr;

    long newDirID; // scratch
    if (!(err = FSpDirCreate (fss,scriptTag,&newDirID)))
    {
        if (phfs->fdFlags)
        {
            DInfo finderInfo;

            // see Appendix B for FSpGetDInfo and FSpSetDInfo

            if (!(err = FSpGetDInfo (fss,&finderInfo)))
            {
                finderInfo.frFlags = phfs->fdFlags;
                err = FSpSetDInfo (fss,&finderInfo);
            }
        }
    }

    return err;
}
```

Snippet 9. Called by Snippet #7

Back to top

Deferring Writing the File

Once the file is created, you may or may not want to write its contents in your `DragSendDataProc`. If the file is large or your app needs some time to generate the data that will be in the file, you may want to defer writing the file. Since Process Manager context switches are disabled during Drag Manager callbacks, other applications would get no execution time if you were to spend time writing the file, even if it were safe to periodically call `WaitNextEvent`, which it is not.

In this situation, you'll want to open the file in your `DragSendDataProc` and leave it open. In addition, set a flag to tell another part of your application it needs to write the file. After `TrackDrag` returns, have that part of your app write the file with periodic calls to `WaitNextEvent`.

Finishing the Drag

Once (and only if) the file has been successfully created, you should let the drag receiver know what the filename was and where the file was created. To do this, call `SetItemFlavorData`. For the `FlavorType` parameter, pass the value of the `promisedFlavor` field of the `PromiseHFSFlavor` record. For the flavor data, pass an `FSSpec` record describing the name and location of the file. The `promisedFlavor` data should always be an `FSSpec`, not an `HFSFlavor`. This snippet consists of simple glue which adds the data correctly:

```
pascal OSErr SetPromisedHFSFlavorData
    (DragReference dragRef, ItemReference itemRef,
        const PromiseHFSFlavor *phfs, const FSSpec *fss)
{
    return SetDragItemFlavorData
        (dragRef,itemRef,phfs->promisedFlavor,fss,sizeof(*fss),0);
}
```

Snippet 10. Adding the promised `FSSpec`

Impersonating Find File

If you need to provide a `DragReference` which refers to an existing file or files, then if at all possible you should be sending `flavorTypeHFS`. But if you discover a compelling reason to send `flavorTypePromiseHFS` instead, make sure you:

- Set the `promisedFlavor` field of your `PromiseHFSFlavor` record to '`rWm1`' (`0x72576D31`).
- When Drag Manager asks your `DragSendDataProc` for `'rWm1'` data and `GetDropLocation` produces an `AEDesc` whose `descriptorType` field contains `typeNull`, provide the original location of the file.
- If `GetDropLocation` produces an `AEDesc` whose `descriptorType` field contains `typeAlias`, copy the file into the drop location. `'rWm1'` is only a hint to the drag receiver, and the drag receiver may not take the hint.

Important:
Perform these steps for all drag items or none; don't mix and match.

The section Coping with Find File elsewhere in this Note details why these steps are necessary. The following snippet implements a decision tree which tells its caller whether to copy a file the caller is dropping:

```
pascal OSErr ShouldCopyToDropLoc
    (DragReference dragRef, FlavorType promisedFlavor,
        Boolean *shouldCopy)
{
    OSErr err = noErr;

    AEDesc dropLoc = { typeNull, nil };

    *shouldCopy = false;

    if (!(err = GetDropLocation (dragRef,&dropLoc)))
    {
        if (dropLoc.descriptorType == typeAlias)
        {
            // no hint or receiver missed it
            *shouldCopy = true;
        }
        else if (dropLoc.descriptorType != typeNull)
        {
            // unknown drop location descriptor type
            err = paramErr;
        }
        else if (promisedFlavor != kPromisedFlavorFindFile)
        {
            // null descriptor but no hint intended (DragPeeker)
            err = dirNFErr;
        }

        if (dropLoc.dataHandle)
        {
            OSErr err2 = AEDisposeDesc (&dropLoc);
            if (!err) err = err2;
        }
    }

    return err;
}
```

Snippet 11. Deciding whether to copy a dropped file

Back to top

Receiving `flavorTypePromiseHFS`

Most applications have no need to receive `flavorTypePromiseHFS` data; `flavorTypeHFS` should suffice for most needs. More senders provide `flavorTypeHFS`, although there is at least one important application (Find File) which provides `flavorTypePromiseHFS`. In any case, seriously consider `flavorTypeHFS` before investing effort in `flavorTypePromiseHFS`.

Getting the Two Flavors

In your drag tracking handler, you may retrieve the `flavorTypePromiseHFS` data, which is a `PromiseHFSFlavor`, but don't try to retrieve the `promisedFlavor` data. Your drag tracking handler can't know whether a given window in your application will be the ultimate receiver of the data - the ultimate receiver might be another window in your app or one of the windows of another app. If your drag tracking handler were to ask for the `promisedFlavor` data, Drag Manager would call the sender's `SendDataProc`, and the data would thereafter be cached in the `DragReference`. Consequently, other potential receivers would get the cached data and the sender would not have a chance to adjust it according to the receiver's drop location.

In your drag receive handler, it's safe to retrieve both the `flavorTypePromiseHFS` data and the `promisedFlavor` data. Before requesting the `promisedFlavor` data, however, make sure to call `SetDropLocation`. The next snippet is a function which administrates this process. Note that the folder parameter can be `NIL`; this means the caller supports Find File; we'll explain how this works and why you'd want to do it a little later.

```
pascal OSErr ReceivePromisedFile
    (DragReference dragRef, ItemReference itemRef,
        HFSFlavor *hfsFlavor, const FSSpec *folder)
{
    OSErr err = noErr;

    if (folder)
        // see Snippet 13 for SetDropFolder
        err = SetDropFolder (dragRef,folder);

    if (!err)
    {
        // we'll explain 'isSupposedlyFromFindFile' later
        Boolean isSupposedlyFromFindFile = (folder == nil);
        err = GetHFSFlavorFromPromise // see snippet 14
            (dragRef, itemRef, hfsFlavor, isSupposedlyFromFindFile);
    }

    return err;
}
```

Snippet 12. Receiving `flavorTypeHFS`

Setting the Drop Location

This part of receiving `flavorTypePromiseHFS` is relatively easy. First, create an alias to the drop location, which for `flavorTypePromiseHFS` should always be a directory. Next, copy the alias into an `AEDesc`. Finally, call `SetDropLocation`. This procedure is demonstrated in the next snippet.

```
static pascal OSErr SetDropFolder
    (DragReference dragRef, const FSSpec *folder)
{
    OSErr err = noErr;

    AliasHandle aliasH;

    if (!(err = NewAliasMinimal (folder,&aliasH)))
    {
        HLockHi ((Handle) aliasH);
        if (!(err = MemError ( )))
        {
            Size size = GetHandleSize ((Handle) aliasH);
            if (!(err = MemError ( )))
            {
                AEDesc dropLoc;

                if (!(err = AECreateDesc
                    (typeAlias,*aliasH,size,&dropLoc)))
                {
                    OSErr err2;

                    err = SetDropLocation (dragRef,&dropLoc);

                    err2 = AEDisposeDesc (&dropLoc);
                    if (!err) err = err2;
                }
            }
        }

        DisposeHandle ((Handle) aliasH);
        if (!err) err = MemError ( );
    }

    return err;
}
```

Snippet 13. Called by Snippet #12

Coping with Find File

Many drag receivers would like to be able to receive data dragged from a Find File results window. The first flavor most developers would look for in the `DragReference` would be `flavorTypeHFS`. However, Find File provides `flavorTypePromiseHFS` instead, in an attempt to work around Finder bugs mentioned elsewhere in this Technote.

Back to top

The True Nature of Find File's Evil: A Sidebar

Find File's workaround works pretty well within the scope of Finder, but it doesn't work very well with many other applications which receive `flavorTypePromiseHFS`. You'll remember that `flavorTypePromiseHFS` is a promise to create a file which doesn't exist yet, but Find File's results window contains only existing files. Right away there's semantic conflict. Let's look at a concrete example to see how this conflict can cause problems:

If an email application were to accept `flavorTypePromiseHFS` as an enclosure to a message and assumed that the drag sender were honoring the semantics of `flavorTypePromiseHFS` as documented in this Technote, the email app would probably want to set the drop location to its outgoing spool folder and delete the file when the associated message were successfully sent. After all, the semantics of `flavorTypePromiseHFS` are to create a file expressly for the exclusive use of the receiving app.

However, if instead Find File were merely to move a pre-existing file into that spool folder, the email app might well be deleting the user's only copy of that data, and at the very least Find File would be moving a file to a place the user isn't likely to expect or understand. This is in fact what Find File does.

Why? Well, since Finder is buggy, Find File convinces Finder a drop has occurred and then proceeds to delete the dropped file and send AppleEvents to Finder to induce it to do what it should have done with `flavorTypeHFS` on its own. The only data Find File really wants from Finder is the drop location.

Regardless of any of the background information in this sidebar, your application should conform as strictly as possible to the rest of this Technote.

Back to top

Working Around Find File

The Find File engineers didn't just bludgeon the Finder into working the way they wanted; they also provided a way for other applications to receive HFS-related drags sensibly. It just hasn't been documented until now.

In your drag tracking handler, retrieve the `flavorTypePromiseHFS` data and compare its `promisedFlavor` field to `'rWm1'` (0x72576D31). This is the value which Find File always uses. If `promisedFlavor` has this value, set a flag to remind you not to call `SetDropLocation` later.

In your drag receive handler, you'd normally call `SetDropLocation` before asking for the `promisedFlavor` data. However, if you're receiving `flavorTypePromiseHFS` data from Find File, skip this step before asking the Drag Manager for the `promisedFlavor` data (and, of course, in this case `promisedFlavor` will always have the value `'rWm1'`). This will produce `FSSpec` data without inducing Find File to move or copy the file.

And now we can see why the value of `promisedFlavor` is important; if it's `'rWm1'`, the data comes from Find File, and if the value is anything else (we've recommended `'fssP'` [0x66737350]; but if your program already uses something else, don't worry about it), the data comes from some other application. Applications other than Find File should conform to the semantics of `flavorTypePromiseHFS` as documented in this Technote.

The next snippet shows how to retrieve both flavors, with some extra checking thrown in to make sure nobody is confused about Find File.

```
static pascal OSErr GetHFSFlavorFromPromise
    (DragReference dragRef, ItemReference itemRef,
        HFSFlavor *hfs, Boolean isSupposedlyFromFindFile)
{
    OSErr             err = noErr;
    PromiseHFSFlavor  phfs;
    Size              size = sizeof (phfs);

    err = GetFlavorData
        (dragRef,itemRef,flavorTypePromiseHFS,&phfs,&size,0);

    if (!err)
    {
        if (size != sizeof (phfs))
            err = cantGetFlavorErr;
        else
        {
            Boolean isFromFindFile =
                phfs.promisedFlavor == kPromisedFlavorFindFile;

            if (isSupposedlyFromFindFile != isFromFindFile)
                err = paramErr;
            else
            {
                size = sizeof (hfs->fileSpec);
                err = GetFlavorData
                    (dragRef,itemRef,phfs.promisedFlavor,
                        &(hfs->fileSpec),&size,0);

                if (!err)
                {
                    Size minSize = MinimumBytesForFSSpec
                        (&(hfs->fileSpec));
                    // see snippet 3 for MinimumBytesForFSSpec

                    if (size < minSize)
                        err = cantGetFlavorErr;
                    else
                    {
                        hfs->fileType     = phfs.fileType;
                        hfs->fileCreator  = phfs.fileCreator;
                        hfs->fdFlags      = phfs.fdFlags;
                    }
                }
            }
        }
    }

    return err;
}
```

Snippet 14. Called by Snippet #12

Back to top

Summary

There are two file system-oriented flavor types associated with the Drag Manager. One, `flavorTypeHFS`, is a relatively simple flavor which can be handled like most others except for some simple workarounds for bugs in Finder. The other, `flavorTypePromiseHFS`, is probably the most complex flavor type developers will encounter and requires a high degree of care, attention to detail, and tolerance for intrusive workarounds to implement correctly.

Here are some important lessons worth repeating:

- For existing files, use `flavorTypeHFS`. For files which don't yet exist but you're willing to create, use `flavorTypePromiseHFS`.
- Don't confuse the Drag Manager's concept of promising flavor data with `flavorTypePromiseHFS`. They're both promises, but they are significantly different kinds of promises.
- Check all error codes and build assertions into your code to avoid being surprised by applications which don't conform to the behavior you expect.
- `GetDropLocation` and `SetDropLocation` are your friends.
- When receiving `flavorTypePromiseHFS` for a file you plan to delete, make sure you do the right thing with Find File to avoid destroying data the user wanted to keep.

Back to top

References

The Drag Manager Programmer's Guide , available on the Developer CD Series Mac OS SDK disc. In addition, you can download it here.

`AEDesc` is an AppleEvent Manager data structure documented starting on page 3-12 of *Inside Macintosh: Interapplication Communication* .

Back to top

Appendices

The Appendices to this Technote contain code snippets which are necessary for a full understanding of other snippets in the Technote but would have obstructed the flow of the main text stream.

Appendix A

This is a utility function called by the functions in Appendices B and C. It allocates and populates a `CInfoPBRec` so that it contains information on the given directory. The caller is expected to dispose the `CInfoPBRec` if the function does not return an error.

```
static pascal OSErr FSpGetDirInfo
    (const FSSpec *spec, CInfoPBPtr *cipbpp)
{
    OSErr err = noErr;

    CInfoPBPtr pbp = (CInfoPBPtr) NewPtrClear (sizeof (*pbp));

    *cipbpp = nil;

    if (!(err = MemError ( )))
    {
        pbp->dirInfo.ioVRefNum = spec->vRefNum;
        pbp->dirInfo.ioDrDirID = spec->parID;
        pbp->dirInfo.ioNamePtr = (StringPtr) spec->name;

        err = PBGetCatInfoSync (pbp);

        if (!err && !(pbp->hFileInfo.ioFlAttrib & ioDirMask))
            err = dirNFErr;

        if (err)
            DisposePtr ((Ptr) pbp);
        else
            *cipbpp = pbp;
    }

    return err;
}
```

Appendix B

These functions are intended to follow the same API as `FSpGetFinfo` and `FSpSetFInfo`. They both call `FSpGetDirInfo`, which can be found in Appendix A.

```
static pascal OSErr FSpGetDInfo
    (const FSSpec *spec, DInfo *fndrInfo)
{
    OSErr err = noErr;

    CInfoPBPtr cipbp;

    if (!(err = FSpGetDirInfo (spec,&cipbp)))
    {
        *fndrInfo = cipbp->dirInfo.ioDrUsrWds;

        DisposePtr ((Ptr) cipbp);
        if (!err) err = MemError ( );
    }

    return err;
}

static pascal OSErr FSpSetDInfo
    (const FSSpec *spec, const DInfo *fndrInfo)
{
    OSErr err = noErr;

    CInfoPBPtr cipbp;

    if (!(err = FSpGetDirInfo (spec,&cipbp)))
    {
        cipbp->dirInfo.ioDrUsrWds    = *fndrInfo;
        cipbp->dirInfo.ioDrDirID    = spec->parID;

        err = PBSetCatInfoSync (cipbp);

        DisposePtr ((Ptr) cipbp);
        if (!err) err = MemError ( );
    }

    return err;
}
```

Appendix C

This function returns the directory ID of a given folder. It calls `FSpGetDirInfo`, which can be found in Appendix A.

```
pascal OSErr GetDirectoryID (const FSSpec *spec, long *dirID)
{
    OSErr err = noErr;

    CInfoPBPtr cipbp;

    if (!(err = FSpGetDirInfo (spec,&cipbp)))
    {
        *dirID = cipbp->dirInfo.ioDrDirID;

        DisposePtr ((Ptr) cipbp);
        if (!err) err = MemError ( );
    }

    return err;
}
```

Back to top

Downloadables

	Acrobat version of this Note (92K).	Download
	Acrobat version of Drag Manager Programmer's Guide (378K)	Download
	Binhexed FinderDragPro Metrowerks Project (220K)	Download

Back to top