Technical Note FL515

File Manager File Handling Q&As

CONTENTS

This Technical Note contains a collection of archived Q&As relating to a specific topic--questions sent the Developer Support Center (DSC) along with answers from the DSC engineers. Current Q&As can be found on the Macintosh Technical Q&As web site.

[Oct 01 1990]

Locking and unlocking a Macintosh file

Date Written: 3/12/92

Last reviewed: 6/14/93

Is there any way to lock or protect a file in such a way that the user cannot unlock it with the Get Info window? I realize it's possible to set a resource map to be `readOnly`, and to (sort of) `resProtect` individual resources, and that you can lock the file name as part of the Finder attributes, but what about the File is Locked bit?

There really isn't a way to lock a file on the Macintosh so that it can't be unlocked by a user with the right tools. All you can do is prevent someone from deleting or changing something by accident (they'll have to unlock the file or resource). It's kind of like locking your house or car - it prevents the casual bypasser from getting in, but not someone who *really* wants in.

The Finder locks or unlocks a file (via the Get Info window) with the File Manager `PBHSetFLock` or `PBHRstFLock` routines. `PBHSetFLock` and `PBHRstFLock` manipulate bit 0 (the file locked bit) in the `flFlags` byte of the file's directory entry. When you call the `PBGetFInfo` or `PBGetCatInfo` routines on a file, those routines return the state of the file locked bit in the `ioFlAttrib` field of the parameter block. The documentation for `PBSetFInfo` and `PBSetCatInfo` in *Inside Macintosh* Volume IV is wrong; you cannot set the attributes of a file or a folder (this is unrelated to your question, but you also cannot set a file's clump size with `PBSetCatInfo`). The only file attribute you can manipulate through the File Manager is the locked bit. Some bits in a directory's attributes can be manipulated when our file server software is running (see the next paragraph).

If Macintosh File Sharing or AppleShare 3.0 are running and a volume is sharable (it can be seen by the owner remotely), then the file server software adds additional privileges to the File Manager privileges. Under both versions of the file server software, you can lock folders so that they cannot be moved, renamed or deleted (see the Macintosh Technote "File Sharing and Shared Folders" for more information in this area) by remote users. With AppleShare 3.0, you can also "copy protect" (or unprotect) files with one of two server control calls. The copy protection only keeps the Finder from copying the file; any other copy utility can still copy the file.

You already know about resource locking, so that probably covers everything.

Allocate, AllocContig, and Macintosh file allocation

Date Written: 5/20/92

Last reviewed: 6/14/93

When we allocate space for a new file using `AllocContig` with an argument in multiples of clump size, we should be grabbing whole clumps at a time so that file length (and physical EOF) will be a multiple of clump size. What happens if

we truncate a file by moving the logical EOF somewhere inside a clump? *Inside Macintosh* says disk sectors are freed at the allocation block level, so we could have a file whose physical EOF isn't a multiple of clump size, right? Does `AllocContig` guarantee that the new bytes added are contiguous with the end of the existing file, or only that the newly added bytes are contiguous among themselves? If the logical and physical EOFs aren't the same, does AllocContig subtract the difference before grabbing the new bytes, or do we get the extra bytes (between EOFs) as a bonus?

———

You can create a file whose physical size isn't a multiple of the clump size, if you try. When the file shrinks, the blocks are freed at the allocation level, without regard for the clump size. Therefore, if you set the logical EOF to a smaller value, you can create a file of any physical length.

There's no guarantee that the allocated bytes will be contiguous with the current end of the file. The decisions that file allocation makes are as follows:

* It always attempts to allocate contiguously, regardless of whether you're explicitly doing a contiguous allocation. (If it can't, it fails rather than proceeding if doing an AllocContig.)

* It always attempts to keep the added space contiguous with the existing space, but it will forgo this before it will fragment the current allocation request (regardless of whether you're calling Allocate or AllocContig).

So these are the actions that file allocation will take:

1. Allocate contiguous space immediately after the current physical end of file.
2. Allocate contiguous space separated from the current physical EOF.
3. Fail here if allocating contiguously.
4. Allocate fragmented space, where the first fragment follows the physical EOF.
5. Allocate fragmented space somewhere on the volume.

You don't get "extra" space with AllocContig. It just does a basic allocation but makes sure any added blocks are contiguous. PBAllocContig does not guarantee that the space requested will be allocated contiguously. Instead, it first grabs all the room remaining in the current extent, and then guarantees that the remaining space will be contiguous. For example, if you have a 1-byte file with a chunk size of 10K and you try to allocate 20K, 10K-1 bytes will be added to the current file; the remaining 10K+1 bytes are guaranteed to be contiguous.

Use OpenDF to open files with driver-like names

Date Written: 9/25/92

Last reviewed: 11/1/92

Our program has a problem with filenames that start with a period. During an Open call, if the filename starts with a period, the Open code calls the Device Manager (for drivers and DAs) instead of the File Manager. However, we've seen other applications that can successfully open these files. What's the secret? How do we open files that otherwise look (from the name) like drivers?

———

The Open trap is shared between the Device Manager and the File Manager. When Open is called, it checks first to see whether you're trying to open a driver. Driver names always start with a period. If you can, avoid using filenames that begin with a period. DTS Technote 1089 HFS Elucidations Revisited discusses this conflict. The secret to opening those files is using the new Open Data Fork functions available with System 7 -- FSpOpenDF, HOpenDF, and PBHOpenDF. These functions bypass the driver name check and go right to the File Manager. Here's the code we use to open a file:

```
err := HOpenDF(vRefNum, dirID, fileName, permission, refNum);
IF (err = paramErr) THEN {HOpenDF call isn't available}
  err := HOpen(vRefNum, dirID, fileName, permission, refNum);
                     {try again with old HOpen call}
```

Try this and your problem should go away under System 7. The code retries with the regular Open call (which uses the same input parameters), so this code can be used in programs that run under both System 6 and System 7.

Should ioNamePtr point to Str255 or Str63?

Date Written: 7/15/92

Last reviewed: 9/15/92

The Macintosh Technical Note "Setting ioNamePtr in File Manager Calls" says that ioNamePtr needs to point either to nil or to storage for a Str255. This contradicts the Technical Note "Searching Volumes--Solutions and Problems," which gives an example of a recursive indexed search using PBGetCatInfo. The example uses a Str63. Which Technical Note is correct?

To be generically correct, ioNamePtr should point to a Str255. However, in the case of PBGetCatInfo and other calls that return a filename (or a directory name), a Str63 is sufficient. The reasons are tied to the history of the Macintosh file system.

MFS, the original Macintosh file system, supported filename lengths of up to 255 characters. However, the Finder on those systems supported filename lengths up to only 63 characters and, in fact, developers were warned to limit filename lengths to fewer than 64 characters (see page II-81 of *Inside Macintosh* Volume II).

HFS, the hierarchical file system (in every Macintosh ROM since the Macintosh Plus), further limited filename lengths to 31 characters. If you mount an MFS disk while running HFS, the old MFS code is called to handle the operation. So, the file system can still create and use files with long filenames on MFS volumes.

When the System 7 file system was being designed, Engineering had to decide what size string to use in FSSpec records. The decision was to use a Str63 instead of a Str31 to be able to support long MFS filenames, and to use a Str63 instead of a Str255 because there were probably very few filenames with over 63 characters (remember, the old Finder limited filenames to 63 characters). Using a Str63 instead of a Str255 saves 192 bytes per FSSpec record.

So, we recommend that you use at least a Str63 for filenames, as in "Searching Volumes--Solutions and Problems." If you need to manipulate the filename in any way after you've gotten the name--for example, to concatenate it with another string--you might want to use a Str255.

Note: Even though the System 7 file system supports filenames longer than 31 characters on MFS volumes, the System 7 Finder does not. In fact, the System 7 Finder currently crashes if you try to open an MFS volume (that is, open the volume window) that has files with names longer than 31 characters.

Macintosh file reference number (refNum) range

Date Written: 11/13/90

Last reviewed: 05-October-1999

Can the refNum returned by FSOpen ever be 1? What is the range or format of legal refNums?

File reference numbers are defined to be positive SInt16's, that is in the range from 1 through 32767. You should not rely on special characteristics of file reference numbers, such as the fact that they are currently always even, or that the System file's file reference number is currently always 2. These characteristics may change on future systems.

> **Note:**
> The number 0 (zero) is suitable as both a nil file reference number and a nil device reference number.

X-Ref:

DTS Technote 1184 "FCBs, Now and Forever"

Steps for duplicating a Macintosh file

Date Written: 12/6/90

Last reviewed: 1/16/91

Is there a routine in the Macintosh operating system to duplicate a file? (Something that is similar to doing "File - Duplicate" or dragging a file from one disk to another.) We tried PBHCopyFile but, as *Inside Macintosh* Volume V implied, it did not work.

Unfortunately there isn't a routine in the Macintosh OS to duplicate a file. The only way to duplicate a file is by doing the following:

1. Create a new file,
2. Open the old file,
3. Open the new file,
4. Check to see how long the old file is,
5. Read the old file,
6. Write everything you read to the new file, then
7. Close both files.

The DTS sample code MoreFiles contains an example of this code.

Macintosh filename cannot start with a period (.)

Date Written: 12/18/90

Last reviewed: 5/21/90

Why do I get a bomb when I create a Macintosh filename starting with a period (.)?

———

Macintosh filenames are not allowed to begin with a period, to avoid possible confusion with driver names, which must begin with a period. (This restriction does not apply to folder names.) Ideally, the Finder should catch this possible error and require the file to be renamed, but it doesn't. Future versions of the Finder should catch this potential problem, but until then users must remember not to begin a filename with a period. See the DTS Technote 1089, "HFS Elucidations Revisited" for details.

Code for reading from a non-Macintosh-formatted floppy disk

Date Written: 6/5/91

Last reviewed: 6/14/93

My Macintosh application needs to recognize, list the files of, and read files from a DOS disk in a Macintosh application, while running under either System 6 or 7. Something akin to the way Apple File Exchange works would be fine--starting the application and then having the application recognize the DOS disk and listing its contents.

———

Assuming you want to start from scratch and write your own, instead of using a third-party software package such as DosMounter, you'll probably need to check out technical references on DOS floppy formats. Here's some code for the hard part--reading an arbitrary floppy in non-Macintosh format:

```
/* pass driver name, e.g. ".Sony"  */
/* returns driver Reference Number */
OSErr
Open(fn, RefNum)
StringPtr    fn;
short        *RefNum;
{
    OSErr            result;
    ParamBlockRec    pb;

    pb.ioParam.ioNamePtr = fn;
    pb.ioParam.ioCompletion = 0;
    pb.ioParam.ioPermssn = fsRdPerm;
    result = PBOpen(&pb, false);
    *RefNum = pb.ioParam.ioRefNum;

    return result;
}
/* pass refNum gotten from Open    */
/*      pointer to destination     */
/*      bytes to read              */
/*      offset from byte 0 of disk */
OSErr
Read(refNum, dest, count, offset)
short    refNum;
Ptr        dest;
long    count;
long    offset;
{
    OSErr result;
    ParamBlockRec    io;

    io.ioParam.ioCompletion = NULL;
    io.ioParam.ioVRefNum = 1;
    io.ioParam.ioRefNum = refNum;
    io.ioParam.ioBuffer = dest;
    io.ioParam.ioReqCount = count;
    io.ioParam.ioPosMode = fsFromStart;
    io.ioParam.ioPosOffset = offset;
    result = PBRead(&io, false);

    if (result != noErr)
        printf("PBRead failed. ioActCount = %d\n", io.ioParam.ioActCount);

    return result;
}
```

System 7 Finder and file duplication time

Date Written: 6/13/91

Last reviewed: 6/14/93

Duplicating a large file under the System 7 Finder took four times as long as under System 6. How can I make System 7 copy more blocks and larger sizes, to reduce duplication time?

———

Duplicating a file in the Finder is slower in System 7.0 than in previous systems because the new system allows you to switch out of the Finder to another application for those really big file duplications. This is a feature that was installed into the Finder. The Finder is constantly checking to see if it should be switched out and this takes time. Since you can't get

something for nothing, even in the Macintosh world, there is a speed loss when copying files in the Finder.

This speed change will not happen from within an application. In fact, you may notice a small increase in the speed at which applications can duplicate files. Try using the "duplicate" command in MPW and you'll see that it's considerably faster than the Finder.

Copying a file or folder dragged to Macintosh application

Date Written: 6/13/91

Last reviewed: 10/8/91

What do I need to do to have my Macintosh application copy any folder or file that is dragged to it?

———

One method is to create an application and have it reside on the desktop. The user can then just drag files to that application, which will in turn receive an open event on the file and duplicate the file. This is possible because when a document's icon is dragged on top of an application's icon, the Finder will tell the application to open the file if the application knows about the file. The only catch is that you'll need to use the special 'FREF' type that will enable the Finder to send you an 'odoc' for ANY file. The key 'FREF's you'll need are:

```
**** -- any file,
fold -- any folder,
disk -- any disk.
```

The characters should be all lowercase letters. This should allow your application to receive an open for anything that's dragged to it. It will be up to your application to handle possible error conditions such as not enough disk space.

Simulating PBExchangeFiles for System 6

Date Written: 6/19/91

Last reviewed: 10/22/91

Here's how to do the equivalent of PBExchangeFiles (new in System 7) for System 6:

Create a new file. Copy the old files to the new one, do a `PBHGetFileInfo` on the old and new, and copy the Finder and creation date from the old to the new. Then do a `PBHSetFileInfo`, renaming the new and deleting the old. The File Manager implements `PBExchangeFiles` as shown below:

Catmove file 1 to file 2's directory

Catmove file 2 to file 1's directory

rename file 2 to 1

rename file 1 to 2

Renaming and deleting require the filenames instead of refnums. The filename may have been changed by the user since the file was opened because the Finder doesn't disallow changing names and moving files that are open. To get around this problem, use `GetFCBInfo` to recover the filename, `DirID` and `VRefNum` of an open file. In the case of files, the FCB's `ioFCBParID` field is the `ioDirID` of the file, and `ioFCBVRefNum` is the `ioVRefNum`.

You may need to do an intermediate rename if there is file already exists with the name of the moved file in the destination directory. The File Manager is able to pull this procedure off a little more gracefully because it manipulates the B-Tree entries instead of going through the APIs.

The DTS sample code MoreFiles contains an example of this code.

Macintosh open file maximums & how to alter

Date Written: 7/11/91

Last reviewed: 05-October-1999

This topic is now covered in depth in DTS Technote 1184, "FCBs, Now and Forever."

The MPW linker tends to stretch the open file maximum. If you're running into problems trying to link too many files at once, you might consider using the lib utility to combine some of these files, thus requiring fewer file control blocks to be open at once during the link process.

Where to get Macintosh third-party file formats

Date Written: 8/30/91

Last reviewed: 8/30/91

How can I get the file format details on the likes of MacWrite, MacWrite II, and Microsoft Word?

Macintosh file formats are not published like Apple II file formats. You must contact the developers of the Macintosh software in order to obtain their file formats.

FSSpec and SFReply information blocks

Date Written: 8/19/91

Last reviewed: 10/8/91

How can I make FSSpec file information comply with what was an SFReply information block? Is there a way to convert FSSpec information--as passed, for example, via an Open Documents Apple event--to a vRefNum as understood by an SFReply record? We want to keep our tried-and-true non-System 7 file management logic and convert from FSSpec to SFReply-type format.

Not wanting to make a good bit of file system code obsolete is understandable; however, while it's unlikely that Apple will dispense with support for old SFGetFile or SFPutFile functions in the near future, the use of SFReply-style data structures in internal calls has no development future.

The vRefNum field of the SFReply record was originally (in *Inside Macintosh* Volume II days) a volume reference number; later, with the creation of HFS in 1986, it became a working directory reference number for purposes of backward compatibility. In HFS, a file or directory entity on a volume is specified with a volume reference number, a directory ID, and a name. An FSSpec contains this latter information.

Converting from FSSpec to SFReply requires that your application manage the manipulation of working directory entities, which has disadvantages from the point of view of the system and compatibility. There are several difficulties with working directory references:

* There's a system-wide limit on their number.

* If you have a working directory reference to which no file buffers are open and some other application closes that working directory without your knowledge of it, your internally stored reference number is invalid and you have no way of knowing about it.

* The documentation about where, when, and how to close a working directory is somewhat ambiguous.

* An FSSpec can refer to either a file or a directory while an SFReply can refer only to a file.

Developer Technical Support urges you to take the time to remove dependencies on SFReply data structures as soon as is feasible.

How to search only nonserver mounted volumes

Date Written: 8/29/91

Last reviewed: 6/14/93

If I don't want to search any AppleShare or File Sharing volumes, how can I tell which mounted volume to search?

You should be able to use the field vMServerAdr in the GetVolParmsInfo attributes buffer of PBHGetVolParms to determine whether to search a volume. Since the vMServerAdr field specifies the internet address of the server that manages an AppleTalk server volume, checking for a zero internet address before searching the volume would seem the way to go for you.

X-Ref:

Inside Macintosh Volume VI, pages 25-37 to 25-40

Using the Macintosh file system asynchronously at interrupt time

Date Written: 11/19/91

Last reviewed: 6/14/93

Is it true that calls in the file system like PBOpen can move memory under all conditions? Can I create, open, write, and close a file completely at interrupt time? If not, which calls must be called at a driver's accRun time? I need to be compatible with both 6.0.x and 7.0.x.

———

The answer to your question is (drum roll...) 42. Oops, wrong question.

The answer to your question is yes, all this (and more) can be done completely at interrupt time. Any call that can be made asynchronously can be safely made at interrupt time, provided it is made asynchronously. Glancing though *Inside Macintosh* Volume IV, this includes just about all of the File Manager, except for the ability to mount and unmount volumes.

One caveat: making a call asynchronously here means really making it asynchronously; making the call and then sitting in a little loop waiting for the ioResult field to change does not qualify. You must either use completion routines to determine when a call has completed, or you must check the ioResult from time to time, never waiting for it at interrupt time. (And in this case, a deferred task does qualify as being at interrupt time).

Using Macintosh PBRead call asynchronously

Date Written: 11/26/91

Last reviewed: 05-October-1999

Can PBRead be called asynchronously? I would like to start the "next" file read while processing the "previous" read's data.

———

You can certainly call PBReadAsync; however, the results you get may be a little unexpected. For example, if the device you're reading from is accessed via the classic SCSI Manager (or any other synchronous-only technology, such as early versions of the ATA Manager), in certain respects the effect will be synchronous. Classic SCSI Manager does not support asynchronous operations, so once the driver actually starts a classic SCSI Manager operation, it won't return until that operation has completed.

It sounds from your question that you want to be able to do processing *while* the read is being serviced. This works as long as the underlying device driver is asynchronous. You can determine this by querying the driver with the kdgSync Driver Gestalt selector. Most modern Mac OS systems include asynchronous disk drivers.

Partition Macintosh volumes to work around 2 GB size limit

Date Written: 11/26/91

Last reviewed: 05-October-1999

How do I create a Macintosh volume that is >2 gigabytes in size? Up to 2 GB everything works OK. Since my driver and the File Manager work with disk/allocation blocks, 2 GB falls well within a longint. Is this limitation imposed by the Finder?

———

System 7.5 raised the maximum volume size limit to 4 GB, and System 7.5.2 raised it again to 2 TB. The following Q&A contain the full story, from an application perspective.

DTS Q&A FL 08, "Determining the Volume Size"

DTS Q&A FL 09, "PBXGetVolInfo Glue"

The situation for disk device driver writers is somewhat more complex. Please write to DTS for a copy of the pre-release technote that explains the full story.

Accessing files in a folder dropped onto an application

Date Written: 12/5/91

Last reviewed: 6/14/93

How can I get to the files in a folder which was dropped onto my System 7 application? PBGetCatInfo doesn't work with the dirID from the FSSpec.

———

To determine if the FSSpec returned by AEGetNthPtr points to a folder (and if so, to find its dirID), call PBGetCatInfo and check the ioFlAttrib field of the CInfoPBRec after the call. If bit 4 is set, the item is a folder.

Remember that the FSSpec for a folder indicates its parent's dirID, not its own. Before the call to PBGetCatInfo, the ioDrDirID field of the CInfoPBRec should contain the folder's parent's dirID, ioNamePtr must point to the name of the folder, and ioFDirIndex must be zero. Also assign the ioVRefNum and ioCompletion fields appropriately. When PBGetCatInfo returns, ioDrDirID will contain the folder's own dirID.

Use the folder's own dirID in calls to PBHGetFInfo (or PBGetCatInfo) with an increasing index parameter to identify all of the files (or files and directories) contained in the folder.

A sample function showing how to do this is pasted below. PBGetCatInfo is documented in the File Manager chapter of *Inside Macintosh* Volume IV, and in the Macintosh Technical Note "Setting ioFDirIndex in PBGetCatInfo Calls." An application must also have a 'fold' FREF resource included in its bundle to allow folders to be dropped onto it, as discussed in the Finder Interface chapter of *Inside Macintosh* Volume VI.

```
FUNCTION DoAEOpenDoc(theAEvent: AppleEvent; reply: AppleEvent;
                     refcon: LONGINT): OSErr;
 { handle each item in each folder opened (only one folder deep - not
   recursive) }
  VAR
   retCode: OSErr;
   docList: AEDescList;
   odocFSSpec: FSSpec;
   index, itemsInList: LONGINT;
   actualSize: Size;
   keywd: AEKeyword;
   retdType: DescType;

   fileName: Str255;
   folderContentsFSSpec: FSSpec;
   folderDirID: LONGINT;
   folderIndex: INTEGER;

   myCInfoPBRec: CInfoPBRec;

  BEGIN

   retCode := AEGetParamDesc (theAEvent, keyDirectObject,
                              typeAEList, docList);
   IF retCode <> noErr THEN
    BEGIN
     { never ExitToShell from within an AE handler }
     PostWarning('cannot get parameter descriptor', retCode);
     DoAEOpenDoc := retCode;
     EXIT(DoAEOpenDoc)
```

```
 END;

retCode := AECountItems(docList, itemsInList);

FOR index := 1 TO itemsInList DO
 BEGIN
  retCode := AEGetNthPtr(docList, index, typeFSS, keywd,
                         retdType, @odocFSSpec,
   sizeof(odocFSSpec), actualSize);
  IF retCode <> noErr THEN
   BEGIN
    PostWarning('cannot get nth document', retCode);
    DoAEOpenDoc := retCode;
    EXIT(DoAEOpenDoc)
   END;

  fileName := odocFSSpec.name;

  myCInfoPBRec.ioCompletion := NIL;
  myCInfoPBRec.ioNamePtr := @fileName;
  myCInfoPBRec.ioVRefNum := odocFSSpec.vRefNum;
  myCInfoPBRec.ioDrDirID := odocFSSpec.parID;
  myCInfoPBRec.ioFDirIndex := 0; { use name and dirID }
  retCode := PBGetCatInfoSync(@myCInfoPBRec);
  IF retCode <> noErr THEN
   BEGIN
    PostWarning('cannot get cat info', retCode);
    DoAEOpenDoc := retCode;
    EXIT(DoAEOpenDoc)
   END;

  IF BTST(myCInfoPBRec.ioFlAttrib, 4) THEN { it's a directory }
   BEGIN
    { index through all items in the directory }

    folderIndex := 0;

    { myCInfoPBRec.ioDrDirID now contains the dirID of the folder
     pointed to by odocFSSpec }
    folderDirID := myCInfoPBRec.ioDrDirID;
    REPEAT
     folderIndex := folderIndex + 1;
     fileName := ''; { reset name string }

     myCInfoPBRec.ioCompletion := NIL;
     myCInfoPBRec.ioNamePtr := @fileName;
     myCInfoPBRec.ioVRefNum := odocFSSpec.vRefNum;
     myCInfoPBRec.ioDrDirID := folderDirID;
     myCInfoPBRec.ioFDirIndex := folderIndex;
     retCode := PBGetCatInfoSync(@myCInfoPBRec);
     IF retCode = noErr THEN
     BEGIN
      retCode := FSMakeFSSpec(odocFSSpec.vRefNum, folderDirID, fileName,
       folderContentsFSSpec);
      IF retCode <> noErr THEN
       BEGIN
        PostWarning('cannot make FSSpec', retCode);
        DoAEOpenDoc := retCode;
        EXIT(DoAEOpenDoc)
       END;
```

```
        { now do something with the item }
        DoSomethingWithTheItem(folderContentsFSSpec);
       END;

      UNTIL retCode <> noErr; { exhausted folder contents }
     END

    ELSE { odoc entry for a file, not for a folder }
     DoSomethingWithTheItem(odocFSSpec);

   END; { for all items in docList }

  retCode := AEDisposeDesc(docList);

  DoAEOpenDoc := noErr;
END; { DoAEOpenDoc }
```

Ignore asynchronous low-level File Manager function results

Date Written: 2/18/92

Last reviewed: 6/14/93

I'm making an asynchronous low-level File Manager call from inside a completion routine (for example, "error := PBxxx(@PB, TRUE);"). Occasionally on some machines, the call immediately returns an error in the function result even though everything appears to work correctly. Do I need to worry about the function result when I make the call?

———

It sounds like you're making the mistake of testing the function result of an asynchronous File Manager call (the value of register D0 is returned in the function result). There is no useful information in the function result of an asynchronous call made to the File Manager; the call might not even have been looked at by the File Manager yet. It's only `ioResult` after the call completes, or either DO or `ioResult` at the entry to the completion routine that contains the call's result status. If you're polling to check for the call's completion, `ioResult` will indicate the call has completed when it is less than or equal to 0.

In general, when making asynchronous I/O calls (reads or writes) there are only two types of function result errors that are of any possible consequence: a driver not open error (`notOpenErr`) or a driver reference number error (`badUnitErr` or `unitEmptyErr`), which indicate the call was not successfully unqueued by the driver and the ioCompletion routine will not be called. Neither one of these error conditions makes any sense for the File Manager (which isn't a driver); the File Manager will *always* call the completion routine (if any) of a given asynchronous call.

Your program should just always ignore the function result of an asynchronous low-level File Manager call and leave it up to the completion routine or the routine polling ioResult to check for and handle any errors that may have happened on the call.

How to get Macintosh file label & color strings

Date Written: 3/2/92

Last reviewed: 4/7/92

We'd like to search for files by label or color, getting the actual string for the label/color field, so that the user can select from a menu that looks like what they'd see in the Finder or ResEdit.

———

In the Icon Utilities package is a call that will get you the RGB color and string for the Finder's labels. Information from the Macintosh Technote "Drawing Icons the System 7 Way" draft is pasted below. It includes the glue code for the call in MPW C format.

```
Function GetLabelColor(labelNumber:Integer; var labelColor:RGBColor;
                       VAR LabelString:str255):OSErr;
INLINE $303C, $050B, ABC9;
```

This call returns the actual color and string used in the label menu of the Finder and the label's Control Panel. This information is provided in case you wish to include the label text or color when displaying a file's icon in your application.

In C the call looks like:

```
pascal OSErr GetLabelColor(short labelNumber,RGBColor *labelColor,
                           Str255 labelString)={0x303C, 0x050B, 0xABC9};

The assembly being:
;         Push the usual stuff
          Move.W  #$050B,D0
          _IconDispatch                        ; $ABC9
```

You can use this call to get both the string and the color.

X-Ref: Macintosh Technical Note "Drawing Icons the System 7 Way"

_CatMove vs _HReName

Date Written: 11/30/90

Last reviewed: 6/14/93

When using the Macintosh _CatMove trap, can I pass the ioNewName field and leave the ioNewDirID field nil? When tracing through a few calls to this trap, it seems that _CatMove works just fine when the reverse is true (ioNewName is nil).

It sounds like you are trying to get _CatMove to behave like _HRename. Leaving the ioNewName field nil is reasonable if you are just changing the position of the file in the directory structure without affecting its name. In the case of filling in ioNewName and leaving ioNewDirID blank (if it were allowed), this would be like saying "change the name and leave the directory unchanged." This is definitely a job for _HRename.

Macintosh verified read error produces dataVerErr (-68)

Date Written: 11/30/90

Last reviewed: 6/14/93

If I call _Read with "44-ioPosMode" ORed with $40 for a Macintosh verify, what error code is returned if the verify is not successful?

An error in a verified read should produce an dataVerErr (-68).

Determining the amount of free space on a Macintosh disk

Date Written: 11/17/89

Last reviewed: 05-October -1999

How can I determine the amount of free space on my Macintosh disk?

This is covered in detail in DTS Q&A FL 08, "Determining Volume Size."

How to determine if a Mac file or resource file is already open

Date Written: 5/3/89

Last reviewed: 05-October -1999

How can I tell if a Macintosh file or a resource file is already open?

Use the File Manager routines PBGetFInfo (IM IV: 148), PBHGetFInfo (IM IV:149), and PBGetCatInfo (IM IV: 155) to determine if a file is open, and which forks (resource and data) of the file are open. All these routines return ioFlAttrib in the parameter block, which has bits set or cleared to indicate if the file is locked, open, or a directory, and which forks are open (IM IV:125).

Here's a fragment in C:

```
/* uses a full pathname rather than vRefNum/dirId  */

HFileInfo  fParams;
...
fParams.ioCompletion = NIL;
fParams.ioVRefNum = 0;
fParams.ioFDirIndex = 0;
fParams.ioDirID = 0L;
fParams.ioNamePtr := "\pVolume:Folder:Filename";  /* Pascal string */
err = PBGetCatInfo(fParams, FALSE);
if (fParams.ioFlAttrib & 0x10)
  /* pathname is a directory */
else if (fParams.ioFlAttrib & 0x04)
  /* resource fork is open */
else ...
```

X-Refs:

"The File Manager," *Inside Macintosh* Volumes I, II, IV, and VI

Macintosh Technical Note "New High-Level File Manager Calls"

If you're trying to determine whether a file is open because you need to know whether OpenResFile actually opened the resource file and therefore you should close it, you should DTS Technote 1120, "Opening Resource Files Twice Considered Hard?" which explains the full story.

How to get the correct size of a Macintosh file

Date Written: 12/13/90

Last reviewed: 05-October -1999

How do I determine the size of a Macintosh file? I want to get exactly what the Finder reports when you do Get Info on a selected file. How do I use the FileParam fields, ioFlLgLen, ioFlPyLen, ioFlRLgLen, and ioFlRPyLen to find out the size of a disk file after a call to PBGetFInfo?

I presume you want to determine the amount of disk space that the file is consuming, in which case you need to calculate the physical size of the file. Prior to Mac OS 9.0, you should do this by calling PBGetCatInfo (or PBHGetFInfo) and then summing the returned ioFlPyLen and ioFlRPylen fields. In Mac OS 9.0 or later, you should iterate through the file's forks (using FSIterateForks) and sum the returned forkPhysicalSize results.

Retained Macintosh file reference requirements

Date Written: 4/8/91

Last reviewed: 05-October-1999

How do I retain a persistent reference to a file? What can I store in a configuration file so I can find a specific file again at a later launch?

———

In System 7.0 and higher, you should maintain a persistent reference to a file using an alias. See *Inside Macintosh: Files* , Chapter 4 Alias Manager for details. The following discussion is only relevant to System 6 or earlier.

The Macintosh has two native file systems: the original (64K ROM) Macintosh File System (MFS) and the Hierarchical File System (HFS) (Macintosh Plus and newer). In an attempt to simplify this discussion, I'll assume you know the differences between them and base my explanation upon the HFS system as it is the current one.

To retain a reference to a file location that will persist across application launch and restarts, store its file name, volume name, and directory ID. With this information, you will be able to obtain the volume reference number (a dynamic value based on mount order, subject to change because the addition of another drive can change the mount order) and then use either the new high-level File Manager calls or the parameter block-based calls to operate on these files. If you happen to encounter an MFS volume, the HFS File Manager will do the appropriate thing given that you use the calls correctly.

Only if you are supporting MFS for an old system will you need to call the MFS functions for obtaining the correct reference values; MFS was a flat-file system.

Standard File's `reply.vRefNum` value when used with HFS is actually a working directory reference number, not a fixed reference; therefore it will usually change between launches of the application or across restarts.

Documentation for the File Manager is extensive and spread out because it was updated when the Macintosh Plus was released and has been added to subsequently by Technical Notes. *Inside Macintosh* Volume II deals with the MFS File Manager, *Inside Macintosh* Volumes IV and V deal with the HFS File Manager through System 6.0.x, and *Inside Macintosh* Volume VI (on your System 7.0 CD-ROM) lists the new File Manager services available under System 7.0. The following Technical Notes should prove useful to you:

File Manager:

"Available Volumes"

"Determining Which File System is Active"

"HFS Ruminations"

Technote 1089, HFS Elucidations Revisited

"Why PBHSetVol is Dangerous"

"Problem with GetVInfo"

"Setting ioNamePtr in File Manager Calls"

"Working Directories and MultiFinder"

"HFS Tidbits"

"New High-Level File Manager Calls"

"Mixing HFS and C File I/O"

Standard File:

"Customizing Standard File"

"Standard File Tips"

PBHCopyFile and Macintosh file copying

Date Written: 3/18/91

Last reviewed: 05-October-1999

I was just about to write a Macintosh file copying function when I decided to look up a definition in HFS.h (THINK C 4.0.2) and stumbled across the following definition:

```
typedef struct {
   STANDARD_PBHEADER
   int ioDstVRefNum;
   int filler8;
   Ptr ioNewName;
   Ptr ioCopyName;
   longioNewDirID;
   longfiller14;
   longfiller15;
   longioDirID;
} CopyParam;
```

Isn't this a PB for a file copying function?

The ParamBlock structure that you found is indeed for the PBHCopyFile call. PBHCopyFile is documented in *Inside Macintosh* Volume V, the File Manager chapter. The hitch is that it is an optional call for AppleShare file servers, and it works only intraserver (but can work across volumes that are on the same server). You can determine whether a particular volume supports PBHCopyFile by calling PBHGetVolParms on that volume and checking the bHasCopyFile flag (bit 14) of the VMAttrib field. Also, you can determine whether two volumes are on the same server by calling PBHGetVolParms on them and comparing their internet addresses. If they are the same, then they are on the same server.

Unfortunately, if what you are looking for is a single call to copy a file on the Macintosh, there isn't one (oddly enough). To handle the general case of copying a file on the Macintosh, you still have to copy the file's data fork, resource fork, and Finder Info flags, except for the INITed bit. The DTS sample code MoreFiles contains an example of this code.

Copying a file from application folder to System Folder

Date Written: 3/11/91

Last reviewed: 05-October-1999

My installer application should automatically copy known files from either a floppy or a folder in the same directory or folder as the application. How can I determine a reference number for the source file in the directory without using the Standard File Package SFGetFile?

The best way to do this is to include the following code early in your application's startup sequence.

```
// This snippet uses FSpGetFileLocation, which is part of
// the DTS sample "MoreFiles".

#include "MoreFilesExtra.h"

static FSSpec gApplicationFSSpec;

extern void main(void)
{
    err = FSpGetFileLocation(CurResFile(), &gApplicationFSSpec);
}
```

This gives you an FSSpec for your application. You can then navigate your way to the other items in your folder. Once you have an FSSpec for the source items, you can use the code in the DTS sample MoreFiles to copy them using standard File Manager calls.

Your application should not rely on the fact that the Process Manager creates a working directory for your application and sets the current "volume" to be that working directory. While this should continue to occur on traditional Mac OS, working directories are not supported under Carbon.

Macintosh Read calls at interrupt time

Date Written: 9/26/91

Last reviewed: 6/14/93

Read calls at interrupt time often result in a "hang," waiting for the parameter block to show "done." This happens if the interrupt occurred during another Read call. I've tried checking the low-memory global FSBusy, and that decreases the occurrence of this problem but does not eliminate it. When is it safe to make the Read call?

The problem you're experiencing is a common one known as "deadlock." The good news is that you can *always* make Read calls at interrupt time! The only requirement is that you make them *asynchronously* and provide a completion routine, rather than loop, waiting for the ioResult field to indicate the call has completed. This will require that you use the lower-level PBRead call, rather than the high-level FSRead.

The low-memory global FSBusy is *not* a reliable indicator of the state of the File Manager. The File Manager's implementation has changed over time, and new entities patch it and use the hooks it offers to do strange and wonderful things. File Sharing really turns it on its ear. The result is that when FSBusy is set, you can be sure the File Manager is busy, but when it's clear you can't be sure it's free. Therefore, it would be best if you ignore its existence.

If you need to have the Read calls execute in a particular order, you'll have to chain them through their completion routine. The basic concept is that the completion routine for the first Read request initiates the next Read request, and so on until you're done reading.

By the way, never make synchronous calls at interrupt time (and, contrary to the popular misconception, deferred tasks are still considered to be run at interrupt time) or from ioCompletion routines, which may get called at interrupt time.

Should ioNamePtr point to Str255 or Str63?

Date Written: 7/15/92

Last reviewed: 6/14/93

The Macintosh Technical Note "Setting ioNamePtr in File Manager Calls" says that ioNamePtr needs to point either to nil or to storage for a Str255. This contradicts the Technical Note "Searching Volumes--Solutions and Problems," which gives an example of a recursive indexed search using PBGetCatInfo. The example uses a Str63. Which Technical Note is correct?

To be generically correct, ioNamePtr should point to a Str255. However, in the case of PBGetCatInfo and other calls that return a filename (or a directory name), a Str63 is sufficient. The reasons are tied to the history of the Macintosh file system.

MFS, the original Macintosh file system, supported filename lengths of up to 255 characters. However, the Finder on those systems supported filename lengths up to only 63 characters and, in fact, developers were warned to limit filename lengths to fewer than 64 characters (see page II-81 of *Inside Macintosh* Volume II).

HFS, the hierarchical file system (in every Macintosh ROM since the Macintosh Plus), further limited filename lengths to 31 characters. If you mount an MFS disk while running HFS, the old MFS code is called to handle the operation. So, the file system can still create and use files with long filenames on MFS volumes.

When the System 7 file system was being designed, engineering had to decide what size string to use in FSSpec records. The decision was to use a Str63 instead of a Str31 to be able to support long MFS filenames, and to use a Str63 instead of a Str255 because there were probably very few filenames with over 63 characters (remember, the old Finder limited filenames to 63 characters). Using a Str63 instead of a Str255 saves 192 bytes per FSSpec record.

So, we recommend that you use at least a Str63 for filenames, as in "Searching Volumes--Solutions and Problems." If you need to manipulate the filename in any way after you've gotten the name--for example, to concatenate it with another string--you might want to use a Str255.

Note:
Even though the System 7 file system supports filenames longer than 31 characters on MFS volumes, the System 7 Finder does not. In fact, the System 7 Finder currently crashes if you try to open an MFS volume (that is, open the volume window) that has files with names longer than 31 characters.

Back to top

Downloadables

Acrobat version of this Note (K) Download

Back to top
