Software Applications in a Shared Environment

Final Draft: 8/6/87

Communications and Networking Technical Publications

APPLE COMPUTER, INC.

20525 Mariani Avenue
Cupertino, California 95014
(408) 996-1010

Simultaneously published in the United States and Canada.

NOTICE

The information in this document reflects the current state of the product. Every effort has been made to verify the accuracy of this information; however, it is subject to change. Preliminary Notes are released in this form to provide the development community with essential information in order to work on compatible third-party products.

Table of Contents

List of Figures

Preface

The shared environment is usually thought to mean several workstations connected to a file server. A file server allows users on a network to share data, applications, and disk storage over the network. A multi-tasking operating system or software application such as Switcher™ can also be considered a shared environment. These software environments allow sharing applications as well as data sharing between applications. The increased use of AppleTalk networks and data sharing applications requires that applications for the Macintosh computer follow guidelines that will ensure application compatibility in the shared environment.

Application developers will find useful information in this document. Guidelines for application development using the shared environment will be discussed. All application developers benefit from the guidelines by having a framework that assists in developing applications that will operate properly in the shared environment. It is assumed that programmers developing applications are already familiar with the software environment they will use.

The contents include eight sections covering:

- Introduction to the Shared Environment
- File Servers
- Introduction to AppleShare
- AppleTalk Filing Protocol
- Opening Files
- Network Programming Guidelines
- Network Application Development
- Becoming Network Aware

Where to go for more information:

- *Inside Macintosh*, Volume 2, Chapter 10: The AppleTalk Manager
- *Inside Macintosh*, Volume 2, Chapter 6: The Device Manager
- *Inside AppleTalk*, Section XI: AppleTalk Session Protocol (ASP)
- AppleTalk Filing Protocol, Version 1.1
- *Inside Macintosh*, Volume 4, Chapter 19, "The File Manager"; Chapter 15, "The Standard File Package"
- *AppleShare User's Guide*
- *AppleShare Administrator's Guide*

Introduction to the Shared Environment

The shared environment provides the opportunity for application and data sharing among users attached to the network. Applications usefulness increases as data sharing between users increases. Networking further provides the ability to communicate and share data among users. Again, it should be emphasized that multi-tasking and applications such as Switcher share the same considerations and potential problems as found in a networking environment. If you are unfamiliar with a networking environment and want more information, refer to AppleShare Administrator's Guide or AppleShare User's Guide for a description of the AppleShare file server. Terminology and requirements for the AppleShare server environment are discussed there in depth.

Sharing can be done at a file and sub-file level. File level sharing can be divided into two categories, data file sharing and application file sharing. Data file sharing could be a project schedule that would be read by many users simultaneously but could be updated by only one user at a time. Simultaneous updates to the file must be prevented in order to protect the data in the file. A word processor, for example, might be an application shared as a read-only file among many users. A correctly written application, with a proper site license, would allow many users to use the same copy of the application at the same time.

Sub-file level sharing would be appropriate for applications such as data bases, spreadsheets, or any similar application. Several parts of the file could be updated by users simultaneously remembering that each part of the file can be updated by only one user at a time.

Certain file system operations normally taken for granted must be monitored to insure their successful completion. It can no longer be assumed that a computer is a single-user/single-machine situation. Availability of resources in a network or shared environment cannot be assumed. Appropriate error messages should be returned to the user to indicate the failure of an operation. Examples in a networking environment might be:

- a file read or write fails because the file has been removed, the file server has been shut down, or a break in the network has occurred
- creation of a file on the server fails due to an existing duplicate name
- a file cannot be opened for use because another user has already opened the file or the user does not have the proper access privileges.

Preflighting system operations becomes important in the shared environment. *Preflighting* is checking the availability of a resource before you attempt to use the resource. For example, if an application creates temporary files, the application should check to see if the names it gives to the temporary files already exist. If the name already exists, the application can then give the temporary files another name or warn the user of the impending problem. This example is especially true for computers attached to a network because file storage may not be local to the computer.

Network preflighting also involves checking resources prior to their use. Checking if a server volume is writeable before trying to write to the volume is an example of a preflight that would allow the return of an error message indicating the system operation could not be completed.

File Servers

A file server is a combination of a computer , special software, and one or more large-capacity disks attached to other computers via a network. In the file server context, the other computers attached to the network are known as workstations. The computer network allows communication between the file server and workstations. Users have easy access to programs, data, and disk storage provided by the file server.

In the case of private information, files can be protected by placing access restrictions on the folders that contain the files. These restrictions prevent access to files just as a locked file cabinet protects documents. Access to the information can be set up such that information in a folder can be shared only within a group. This would allow people in the group to share information, for example, about a project under development. When the project reaches completion, the access privileges could be changed to allow sharing the information with anyone that might need the information.

Introduction to AppleShare

The server application available from Apple Computer is AppleShare™. Explanation of the AppleShare file server environment should provide parallels for other shared environments.

Each hard disk attached to the server is called a file server volume. A selected server volume will appear on the workstation's desktop as an icon and can be used just like any Macintosh disk drive.

Access to the information contained in folders on the disk can be controlled by use of *access privileges*. These privileges allow a folder to be kept private, shared by a group of registered users, or shared with all users on the network. New users are registered, given passwords, and organized into groups. The user can belong to more than one group to provide better access to needed information. The information about the users and groups is stored in a data base on the server and is used to determine the access privileges the user or group has when they access an object on the server. Each folder has access privileges assigned for each of the three categories of users: owner, group and everyone.

In the AppleShare file server environment, access privileges control *who* has *what* kind of access to the contents of the folders contained on a volume. The access privileges are assigned on a folder by folder basis. This mechanism provides protection against unauthorized use of applications or access to data files in folders on file server volumes.

Access privileges function as follows:

- The owner of a folder specifies that folder's access privileges for the following user categories:

  Owner – user who currently holds ownership

  Group – any group established by the AppleShare administrator (AFP supports only one group designation per folder)

  Everyone – every user who has access to the file server (registered users and guests)

- Access privileges for a particular folder control the ability for each user category to:

  See folders - privilege to see other folders in the folder

  See files - privilege to see the icons and open documents or applications in that folder as well

  Make changes – create, modify, rename, or delete any file or folder contained in the particular folder (note: folder deletion requires other privileges as well).

An in depth discussion of access privileges can be found in the *AppleShare User's Guide*.

AppleTalk Filing Protocol

Applications access files on Macintosh-initialized volumes by making calls to the File Manager. Calls requesting services from a local volume are handled by the native file system. Calls that refer to files on a server are routed through an external file system translator and are converted into AFP requests to the file server. The server provides the requested services to the workstation. The interaction with the network was implemented in a way that provides transparent access to the mounted server volumes and files. The AFP was carefully designed to allow its extension in a very general way. This was done to enable future support of additional types of workstation file systems.

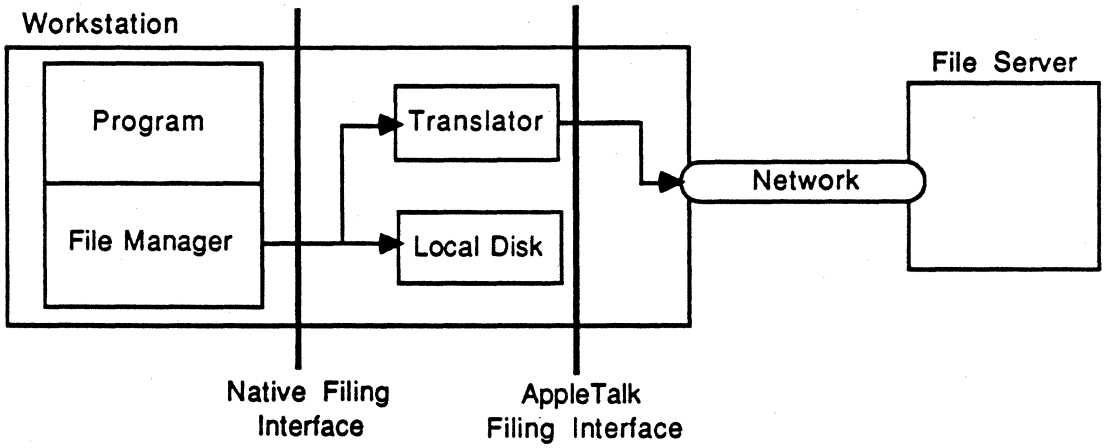
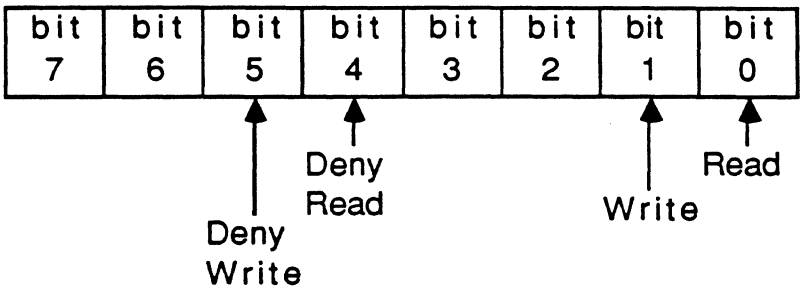


Figure 1. The Basic File Access Model

Permissions

The File Manager was originally designed with three file-access modes: read/write, read-only, and read/write if possible, otherwise, read-only. File servers introduced a situation where more than one user might simultaneously access a file. The original file-access rule, known as "single writer and/or multiple readers", was found to be inadequate.

AppleShare provides methods of specifying file access permissions that will prevent simultaneous writes to a file if the permissions are properly used. These new modes are known as *deny modes* and are used in two new file-opening calls: `OpenDeny` and `OpenRFDeny`. The setting of bits in the permission byte adds extra options to allow denial of read or write access to other users. This gives the caller the ability to *deny* access to other users and to specify the access required by the caller. Refer to Appendix A for a description of the new HFS calls used in supporting shared environments. The format of the new permission byte (used in the field `ioPermssn`) is as follows:



Bit:	Mask:	Meaning:
0	$01	Set if read permission requested
1	$02	Set if write permission requested
4	$10	Set to deny other readers to the file
5	$20	Set to deny other writers to the file

All other bits must be clear and `ioFLVersNum` must be zero. The permissions value for a given access mode can be determined by adding the masks for the desired bits together. For example, read/write/deny-writers would be: $01 + $02 + $20 = $23.

It is important to remember that these deny mode permissions take into account existing access paths to a file and other attempts to open new access paths to the file. This means that if you attempt to open a file with read/deny-writers permission (normal read-only access), the call will succeed only if no write access paths currently exist to the file and no access paths were opened with deny read. Additionally, attempts to open the file for write will fail until the read/deny-writers path is closed.

Another point to remember is that the *access modes* and *deny modes* are cumulative. The modes combine to form the access currently available for a file. Each successful open of a file combines its deny modes with previous deny modes. For instance, if the first open sets a deny mode of deny read and a second open sets a deny write, the file has a current deny mode of deny read/write. Access modes are also cumulative and combine to form the current access mode for a file.

Translation of the "Original" Permissions

AppleShare uses the new permissions exclusively. So existing applications will work, the external file system used by AppleShare (on each workstation) translates the original permissions into the new permissions. To prevent applications from damaging files, the basic rule of file access for AppleShare volumes has been changed to "single writer OR multiple readers, but not both". Now two applications cannot have access to the same file unless both open the file read-only. This eliminates the problem of a reader reading a file when the file is in an inconsistent state.

Note: This change in the basic rule currently applies only to AppleShare volumes when the "original" permissions are used. In the future, Apple system software may incorporate this change for local volumes. Be aware that if your application expects to get more than one read-write path to a file at the same time, it will fail.

Here is how the original permissions translate to the new permissions:

Original Permissions:	New Permissions:
`fsRdWrPerm` (read/write)	read/write/deny-readers/deny-writers, *or* read/deny-writers
`fsRdPerm` (read-only)	read/deny-writers
`fsWrPerm` (write-only)	read/write/deny-readers/deny-writers, *or* read/deny-writers
fsCurPerm (read/write if possible, otherwise, read-only)	read/write/deny-readers/deny-writers, *or* read/deny-writers

The *or* in the translation of `fsRdWrPerm`, `fsWrPerm`, and `fsCurPerm` means that if the call cannot be completed successfully with the first set of permissions, it is automatically retried by the translator using the second set of permissions. The deny portions of the translation are important for enforcing the updated basic rule of file access. If a read or write access path to a file being opened already exists with `fsCurPerm`, the first set of permissions will fail. The second set of permissions will succeed only if there is no existing write path to the file. Note that `fsRdWrPerm` and `fsWrPerm` are also retried read-only. Inside Macintosh states that `fsRdWrPerm` is granted even if the volume is locked. An error will not be returned until a `PBWrite`, `SetEOF`, or `PBAllocate` call is made.

Opening Files

There are several ways to open, use, and protect files using the new permissions. The following examples will show ways to share files and protect the files using the new *access/deny* permissions.

Browsing Only Access: read/deny-writers

Opening a file with browse only access allows multiple readers with no writers because the file can only be opened "read-only". These permissions would be used with commonly used files such as help files, dictionaries, or any file that is read by many readers but never modified by the readers. It might be appropriate to add a "Browse Only" checkbox to SFGetFile dialog,with a user prompt during a document open from the Finder, so the file could be explicitly opened as read-only.

Exclusive Access: read/write/deny-readers/deny-writers

Files opened with exclusive access allow only one user to read or write the file. Exclusive access to the file is provided and the other calls will succeed only if there is no existing path to the file. Any additional access path to the file will be denied until the current path is closed. Most "single-user" applications will probably use this access mode. Most existing applications use this mode by default by using `fsCurPerm` permissions.

On AppleShare, the call will fail if this method is used to open a path to a file in a folder to which you do not have both See Files and Make Changes privileges. A user could then, with an appropriate message, be offered a Browse Only copy of the file. If the user refuses, then close the Browse Only path.

Single Writers or Multiple Readers Access: read/write/deny-write/deny-read for writers, read/deny-write for readers

Opening files with single writers or multiple readers access allows only one user read/write access to a file. A read/write request for the file is returned as read-only if the file is currently in use. If the file is currently being modified, a message should be issued to a user requesting read access indicating the file is being written, access will be granted after the write is complete. This method might be the most easily implemented by existing applications that want to share data.

Single Writers and Multiple Readers Access: read/write/deny-write for writers, read for readers

Opening files with single writers and multiple readers access allows only one user read/write access to a file. If a write access path is open to the file, a request to open the file for write will be denied, but a request to open the file for read will be granted. This type of application would be more difficult to implement. The application is responsible for control of the file while it is being written to prevent other users from reading the file while it is being modified. During modification, the writer should Range Lock the part of the file being written. Also, applications should return a message to the user telling them that the file is being modified and is currently not available.

Shared Access: read/write (deny none)

The shared access method of opening files supports full multi-user access to its files. All users are allowed read and write access to any file concurrently. Range locking would have to be used by the application to prevent other users from accessing files while they are being modified. Error messages for the user become extremely important in this shared environment to make the user aware of what is happening. An example would be an error message letting a user know that a file is currently in use and they will have to wait to access the file.

Network Programming Guidelines

This section contains some additional guidelines for areas that could cause compatibility problems in a network environment.

Use the new HFS system calls.

The new calls would allow the application to determine if the file is stored on a server volume. The privileges could then be checked and feedback sent to the user if necessary. It is important in the shared environment to provide information to the user about normal situations in this environment that do not normally occur in a "single machine/single user" environment. Refer to Appendix A for a description of the new Hierarchical File System calls used in supporting the shared environments.

Try opening files with the new HFS OpenDeny and OpenRFDeny calls.

It is recommended to structure your code such that you try the new open calls first. Then check to see if paramErr is returned.This would indicate that the file does not reside on a server volume. If so, make the equivalent old style open call. Attempts to make the new calls specifying a local (non-AppleShare) volume will return a paramErr indicating that the local file system does not know how to handle the call.

An application should not write to itself (either data or resource forks).

Applications should not be designed to save information by writing into their own file. When information specific to one user (set-up parameters etc.) is saved in the the application's own file and that application is "shared" by two or more users, information owned by the first user may be overwritten by the second user, and so on.

Multi-user applications should not use the Resource Manager to structure their data in a resource fork.

The Macintosh Resource Manager assumes that when it reads the resource map into memory (during OpenResFile), it will be the only one modifying that file. If two write access paths existed to a resource fork, neither would have any way of notifying the other that the file had changed (and in fact, no way to re-read the map). So, only "sole writer" or "multiple-reader" access will work. (fsShared permission could cause inconsistent files or other problems, without returning an error.)

This means that if your application uses resource files for document storage, you cannot share data (for multi-user access); if you want to create a multi-user, or multi-launch version of your application you must find another way to store your data.

An application should not close a file while in the process of making changes to its contents.

When the application is written so that it opens a file, reads the file's contents into memory, and then closes the file, the application has checked out a copy of the file. After the file is closed, another user can open the file, read the contents of the file into memory, and then close it. Two copies of the file are now *checked out* to two different users.

Each user, after changing his checked out copy of the file, decides to save the changes to the original file. User one opens the file and writes the changes back into the original and closes the file. Then, user two opens the file and writes the changes back into the original and closes the file. The second user's write operation wipes out the first. This is a significant problem because neither user is aware of what has happened and neither has a way of finding out. The best method to prevent a problem is to keep the file open while in use. This will prevent other users from obtaining an access path and modifying the file while it currently open.

An application that intends to share data should use AFP open permission with access deny modes.

Each access path to a file has *open permission* information indicating whether data can be written to it or not through that path. When you open a file, you request an access path with permission to read from it or write to it or both. If the open permission assigned to an already existing access path doesn't allow I/O as requested by your call, a result indicating the error is returned.

The AppleShare workstation translator has been modified to prevent inadvertent file damage when used correctly. Applications should be designed with the new permissions and shared environment in mind to prevent compatibility problems.

An application should inform the user what access was granted to the document during the open process.

Shared environment applications should respond appropriately to errors returned by the file system. A more precise error reporting mechanism is used to communicate between the file server and an application program running in a workstation. Currently, most applications are not prepared to respond to this error reporting correctly. This has serious consequences since the user is not usually informed about the file access granted during the open process.

An application must be intelligent about how it manages temporary files.

Many programs that create and open temporary files give these files fixed names. If such an application is shared by many users, it is possible that a second user will launch the same application and the program will attempt to create a new temporary file and give the same name as that used by the first user. This situation is clearly not expected by most applications and can have serious consequences including crashing the application.

One solution is not to create any temporary files on disk, holding all information in memory. Another solution is to save temporary files in the system folder of the user's boot volume (startup disk) which is usually available for the system file writing. This solution is not perfect, however, since a person's boot volume may be a diskette with extremely limited space. A third solution is to generate unique names for temporary files.

Developers need to be aware that switch launching to the file server volume is not allowed because there are no workstation-accessable system files located on file server volumes.

Use the Scrap Manager to access the Scrapbook.

Don't implement your own scrap mechanism. Use the Rom Scrap Manager so that resources in the scrap can be shared among applications.

Do not directly examine or manipulate system data structures, such as file control blocks (FCB) or volume control blocks (VCB), in memory.

Use file manager calls to access FCB and VCB information.

When the application directly examines the list of data structures related to volumes that are currently mounted without using the appropriate calls to the File Manager, it is possible that these structures will not accurately reflect the structure of the data on file server volumes.

To give the file system the opportunity to update information, use GetVolInfo to determine volume information and GetFCBInfo to determine open file information.

The Allocate function is not supported by AppleShare.

Instead, use SetEOF to extend a file by setting the logical end-of-file.

Program segmentation swapping should be kept to a minimum.

The effect of program segmentation swapping is exaggerated when the application is launched from the file server, because segments are dynamically swapped in over the network. This will reduce the performance of the file server.

Use Byte Range Locking if your application will allow multiple users to concurrently read and write the same file.

Byte Range Locking

The LockRng call locks a range of bytes in an open file opened with shared read/write permission (mode 4). Call LockRng before writing to the file to prevent another user from reading from or writing to the locked range while you are making your changes.

When using byte-range locking:

- You can lock and unlock ranges within a file at any time while you have it open.
- You can keep other users from reading or writing a range.
- All range locks set by you are removed automatically when you close the file.
- You cannot read from or write to a range that's been locked by another access path with the LockRng call.

How To Use Byte Range Locking

On a file opened with a shared read/write permission, LockRng uses the same parameter block (HParamBlockRec) as both the Read and Write calls; by calling it immediately before Read or Write, you can use the information present in the parameter block for the Read or Write call.

Note: The ioPosOffset field is modified by the Read and Write calls and therefore must be set up again before making an UnLockRng call.

When finished using the range, be sure to call UnlockRng to free up that portion of the file for other users.

When calling LockRng, the ioPosMode field of HParamBlockRec specifies the position mode; bits 0 and 1 indicate how to position the start of the range, the same as the FileManager.

Updating a File

When updating a particular record and that update affects other records within the file, first determine the range of bytes affected by the updated information. Then call LockRng to lock out any other user from accessing this range of data. If the lock request succeeds, the required changes to the data can be made. Then release the lock and make the data available to other users again. If the lock fails, several retries should be done. After several unsuccessful retries,an error message could be issued to indicate that the file is busy and try again later.

Without this lock on the data, another user can read the range of data that you are in the process of manipulating, causing the data to appear inconsistent. For example, when implementing a data base or spreadsheet application with the intent of making files available for reading and writing by a group of users, you can use byte-range locking to preserve the integrity of the data within the files.

Appending Data to a File

Lock a range including the logical end-of-file and including the last possible addressable byte of the file ($7FFFFFFF-Hex) and then write to that range. This actually locks a range where data does not exist. Practically speaking, locking the entire unused addressable range of a file prevents another user from appending data until you unlock it.

Truncating a File

To truncate a file, lock the entire file, truncate the data, and then unlock the file. This will prevent another user from using a portion of the file while you are in the process of truncating it.

Network Application Development

AFP file servers will enable new kinds of applications to be created that will need additional sharing guidelines in addition to the general guidelines listed above. Some of these applications will be ones that can be put on a file server and launched and used by several people at the same time. Others will allow several users to update the same file at the same time. In order for these types of functions to happen properly in a shared environment, the applications and their associated document files must be managed correctly. Figure 2 shows four network application program categories.

The network application development categories are defined by the following key terms:

Single-user (private data) applications allow only one user at a time to make changes to a file.

Multi-user (shared data) applications allow two or more users to concurrently make changes to the same file.

Single-launch applications allow only one user at a time to launch and use a single copy of the application.

Multi-launch applications allow two or users at a time to launch and use a single copy of the application.

When *single-user* and *multi-user* are seen as describing data file sharing modes and *single-* and *multi-launch* describe the application launching characteristic of the applications, four categories of network applications emerge. These four categories shown in Figure 2 each have a combination of these two basic characteristics.

Network Application Categories

Application Sharing Mode	File Sharing Mode: Single-User (Private)	File Sharing Mode: Multi-User (Shared)
Single-Launch	Category 1: Single-Launch Single-User The application allows: -only one user at a time to launch and use a single copy of the application. -only one user at a time to make changes to a file.	Category 2: Single-Launch Multi-User The application allows: -only one user at a time to launch and use a single copy of the application. -two or more users may concurrently make changes to the same file.
Multi-Launch	Category 3: Multi-Launch Single-User The application allows: -two or more users to concurrently launch and use a single copy of the application. -only one user at a time to make changes to a file.	Category 4: Multi-Launch Multi-User The application allows: -two or more users to concurrently launch and use a single copy of the application. -two or more users may concurrently make changes to the same file.

Figure 2. Network Application Development Model

Single-user/Single-launch Application

Some applications will fall into this category for these reasons:

- Making an application multi-launch is a philosophical/business issue (site licensing, etc.) and some developers may not wish to develop multi-launch software.
- Many applications are not well suited to multi-user (shared data) versions. Where a database application is a likely candidate for a multi-user (shared data) version, a word processor may not be a likely candidate.

Applications that may be developed for this category include but are not limited to:

Word processingSpreadsheets
Databases
Accounting
Page Layout
Programming Languages
Project Management
Terminal emulation
Bit-oriented graphics
Object-oriented drawing
Games
Utilities

The Single-launch Application

The single-launch application allows only one user at a time to launch and use a single copy of the application.

The Single-user Application

The single-user application allows only one user to have write access to a document at a time.

The application could keep the document open while it is in use and by using the access deny modes which deny access to subsequent users of the document write access. The application might allow multiple-readers, but only if the application were capable of coordinating updates to the file with read-only users.

Multi-user/Single-launch Application

Applications in this category must adhere to the single-lauch guidelines listed above.

Applications that may be developed for this category include but are not limited to:

Spreadsheets
Databases
Page Layout
Project Management
Object-oriented drawing
Mail
Personal Calendars
Games

The Multi-user Application

The multi-user application allows two or more users to have write access to a document at a time. This type of application also correctly locks records while they are being modified.

Allowing and coordinating multiple writers to a single document can be accomplished by keeping the document open while it is in use and by using an open mode in the file system that specifically allows subsequent users of the document write access.

The multi-user application also has an update mechanism so that all users of a document receive updates when a record is changed.

The application must also use byte range locking (available as a standard HFS call) to permit only one writer in a byte range at a time.

Single-user/Multi-launch Application

This will be a new class of application at least partially enabled by file servers. Many users will desire it as it has distinct advantages in version control and in perceived economies for the customer (site licenses).

Applications that may be developed for this category include but are not limited to:

Word processing
Spreadsheets
Databases
Accounting
Page Layout
Programming Languages
Project Management
Terminal emulation
Bit-oriented graphics
Object-oriented drawing
Games
Utilities

This category of application must provide single-user data handling in addition to the multi-launch capabilities described below.

The Multi-launch Application

The multi-launch application allows two or more users to concurrently use one copy of an application.

Making an application multi-launch is more complex than making it single launch. The first step is setting the multi-launch or shared bit in the application's finder information. Use ResEdit or FEdit to set the shared bit.

The multi-launch application may or may not limit the total number of concurrent users of a given copy of the application.

Limiting the number of concurrent users requires that the application implement some method to count the users as they launch and quit the application.

Counting can get a little complex, for example, counting temporary files works but the temporary files may not all be in the same place and may in fact be in the user's boot volumes. Counting temporary files would also require some cleanup method to check whether or not the temporary files in existence were really in use or merely the remnants of a user crash.

One method to make things easier for the programmer is to require that a multi-launch application be able to create temporary files in the folder containing the application. You would, of course, have to document this so users would know that the application could not be launched from a read-only folder.

Multi-user/Multi-launch Application

This category of application combines both multi-user and multi-launch techniques to provide the most complex of network smart applications.

Applications that may developed for this category include but are not limited to:

Spreadsheets
Databases
Page Layout
Project Management
Object-oriented drawing
Mail
Personal Calendars
Games

Becoming Network Aware

Applications written under the assumption that it is being used in a "single user/single machine" environment may encounter problems when that application is used in the server environment. Applications now should use the new HFS calls that support the shared environment. The new calls are defined and explained in Appendix A. The new permissions will allow applications to operate properly in the shared environment. Applications should also handle temporary file names in a manner that will prevent duplicate file name problems.

Three general application types will be discussed. Examples of considerations necessary to make the examples "network aware" will be included. The first example will be an application that is "memory based". A data file is read into memory, worked with, and written back to the original file. The second example, "the switch", is also memory based but differs in the filing method. The third example is an application that is disk based, meaning the application operates primarily out of disk files.

Note: These are suggestions for applications that do not allow multiple writers to make changes to documents at the same time. These suggestions will allow the applications to work in a shared environment in which files are shared serially but not concurrently.

Example 1: Memory Based Applications

A memory based application opens the data file, reads it into memory, then closes the data file. The data is manipulated by the application. The data is then saved. The save operation consist of opening the data file and writing the data to a disk file and closing the file. Care should be exercised not to close the file while it is in use. Allowing another user write access to the file while it is currently being modified could easily damage the file. This type of problem is known as the "checkout" problem.

Considerations for a Shared Environment

Workstation A opens a file. Workstation B opens the same file while workstation A has the file open. Workstation A now saves the file. Workstation B now saves the file destroying the changes made by workstation A.

Suggested Modifications

1. The 'open' operation should leave the file open until the user is through with the file and has issued a 'close' to the file.
2. The file could be opened explicitly for read/write. If the file is busy, report that the file is in use and offer to give a read-only copy of the file if possible. The application should RangeLock the file when changes are being made to prevent a reader or browser from reading an inconsistent file.

Example 2: The "Switch"

This is not a recommended way to handle files. These suggestions will minimize potential problems if your application works this way.

The "switch" type application is also memory based. The application opens the data file, reads the file into memory, then closes the file. The 'save' operation consists of creating a new file, copying the Finder information from the old file, writing the updated data to the file, closing the file, deleting the old data file, and renaming the new data to the old data file name. This allows the old data file to be protected until the new data file is safely written.

Considerations for a Shared Environment

Workstation A opens a file. Now workstation B has opened the same file concurrently. Workstation A modifies the file while workstation B modifies the file. Workstation A saves the file and workstation saves the file. Workstation A deletes the original file. Workstation B attempts to delete the original file but receives an error because workstation A has already deleted the original file. Workstation A renames the file. Workstation B attempts to rename the file but receives an error because workstation A has already renamed the file.

Suggested Modifications

1. Copy the Finder information during the 'open' operation in case the original file gets deleted before the 'save' operation.
2. Check for errors deleting the source, particularly fileNotFound. This error would mean that someone else has modified the file. A dupFnErr from the rename would also mean that someone else had modified the file. The user could then be given the opportunity to save the file elsewhere or under a different name.

Example 3: Disk Based Applications

Disk based applications open the data file and leave the file open for the duration of the modification. A temporary file could be created to hold the changes, the changes could be written immediately, or the changes could be held for the short term in memory. A "save" operation consists of writing the changes to the data file and flushing the volume.

Considerations for a Shared Environment

The application should either exclude other readers and writers or be able to deal with concurrent operations through some type of locking mechanism.

Suggested Modifications

The file should be opened explicitly for read/write. If the file is busy, report that someone else has the file open.

Appendix A

Macintosh HFS File System Calls For Shared Environments

Network Systems Development

This appendix describes the interface to the new Hierarchial File System (HFS) calls used in supporting shared environments (e.g. network-based file servers). These calls are not documented in the File Manager chapter of Inside Macintosh Volume IV. Though the calls are not necessarily specific to AppleShare, most of this appendix keeps the implementation of AppleShare in mind when describing examples.

All of these calls are HFSDispatch ($A260) calls with an index value passed in register D0. This document only describes the low level assembly language interface. Higher level Pascal-like glue routines are unimplemented at this time. Please refer to Inside Macintosh for information on how to make HFSDispatch calls.

How HFS Calls Get Installed

For AppleShare startup volumes, these calls get installed by an INIT resource patch contained within the AppleShare file. Currently, this means that only startup volumes with the AppleShare file located in its System Folder will support these new HFS calls. Future versions of HFS may contain these calls in the ROM or a patch could be inserted to add these calls to the existing HFS.

This resource (Type INIT; ID=32; "ASFSInit") is executed by the INIT 31 resource in the System 3.3 (or later) file. The resource patch installs itself above the stack and all HFSDispatch calls are sent to it. Since this patch currently only handles external volumes, calls to local volumes will return with an error; however the AppleShare external file system code will get all calls made to AppleShare volumes. The file manager may be changed in the future to also handle these new calls.

Does Your HFS Support The New Calls?

The simplest way to determine if your HFS supports these new calls is to just make the GetVolParms call to a mounted volume. If a 'paramErr' error is returned in D0 and you have set the correct parameters, then the volume does not support these new calls. Making successive GetVolParms calls to each mounted volume is a good way to tell if any of the volumes support these calls. Once you find a volume that returns 'noErr' to the call, examine the information to see if that volume supports various functions (like access privileges, CopyFile, etc) that you may need.

Error Reporting

Whenever possible, error codes returned by these new HFS calls map directly into existing Macintosh error equates. For various reasons though, some error codes cannot translate into an existing error equate. Because of this reason, new error equates have been defined for these error codes. These are detailed below:

VolGoneErr	-124	Connection to the server volume has been disconnected, however the VCB is still around and marked offline
AccessDenied	-5000	The operation has failed because the user does not have the correct access to the file/folder
DenyConflict	-5006	The operation has failed because the permission or deny mode conflicts with the mode in which the fork has already been opened.
NoMoreLocks	-5015	The byte range locking has failed because the server cannot lock any additional ranges
RangeNotLocked	-5020	User has attempted to unlock a range that was not locked by this user
RangeOverlap	-5021	User attempted to lock some or all of a range that is already locked

The AppleTalk AFP protocol returns errors in the range of -5000 to -5030. Since it is possible, though unlikely, to receive error codes in this range, it would be wise to handle these undocumented error codes in a generic fashion. If you require it, the complete list of these error codes can be found in the AppleTalk AFP Protocol specification document.

Call Data Structures

Described below are some of the new data structures used by these calls. Specific information about the placement and setting of parameters is described later in the appropriate call section.

For GetLogInInfo, ioObjType contains the log in method where the following values are recognized:

0	guest user
1	registered user - clear text password
2	registered user - scrambled password
3-127	RESERVED by Apple for future use
128-255	User defined values

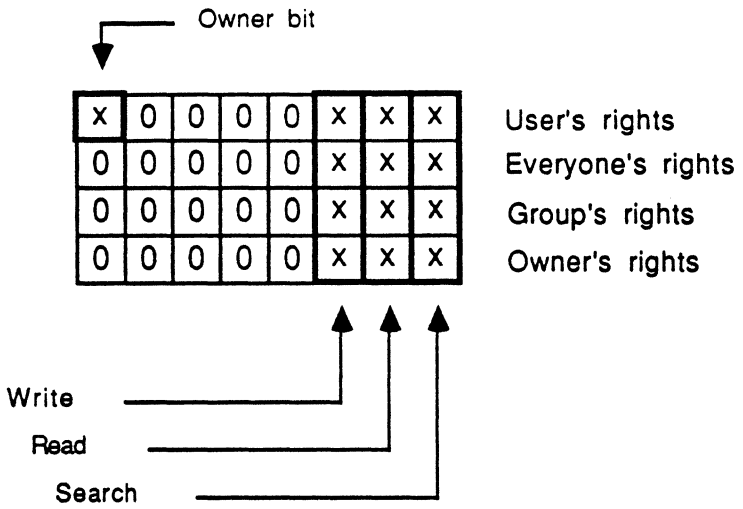
For MapName and MapID, ioObjType contains a mapping code. The MapID call recognizes these codes:

1 map owner ID to owner name
2 map group ID to group name

and MapName recognizes these codes:

3 map owner name to owner ID
4 map group name to group ID

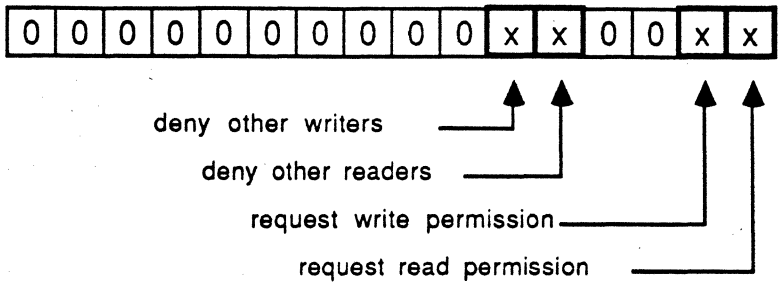
For GetDirAccess and SetDirAccess, ioACAccess is a LongInt which contains access rights information in the format '*uueeggoo*', where *uu* = user's rights, *ee* = everyone's rights, *gg* = group's rights, and *oo* = owner's rights. Unused bits should always be set or returned cleared. A pictorial representation is shown below (high order bit on the left):



The User's rights information is the logical 'OR' of Everyone's rights, Group's rights, and Owner's rights. It is only returned from the GetDirAccess call; it is never passed by the SetDirAccess call. Likewise, the Owner bit is only returned in the GetDirAccess call. To change a folder's owner, you must change the Owner ID field of the SetDirAccess call.

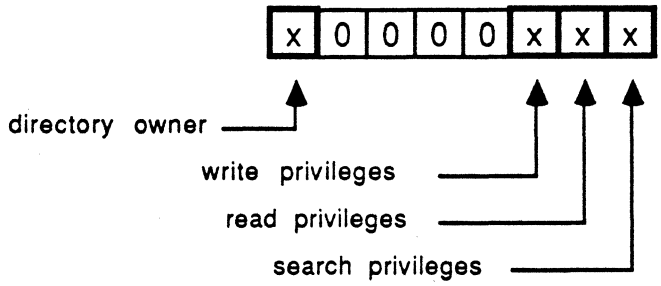
AppleShare 1.0 and 1.1 uses the Write bit to represent 'Make Changes' privileges. The Read bit is used for 'See Files' privileges and the Search bit is used for 'See Folders' privileges.

For OpenDeny and OpenRFDeny, ioDenyModes contain a word of permissions information. This is pictured below (high order bit on the left):



Bit		
	15-6	RESERVED; this should be zeroed
	5	If set, deny other writers to this file
	4	If set, deny other readers to this file
	3-2	RESERVED; this should be zeroed
	1	If set, requesting write permission
	0	If set, requesting read permission

For GetCatInfo, ioACUser (a new byte field) returns the user's access rights information for a directory whose volume supports access controls in the following format:



Bit		
	7	If set, user is not the owner of the directory If clear, user is the owner of the directory
	6-3	RESERVED; this is returned zeroed
	2	If set, user does not have Write privileges to the directory If clear, user has Write privileges to the directory
	1	If set, user does not have Read privileges to the directory If clear, user has Read privileges to the directory
	0	If set, user does not have Search privileges to the directory If clear, user has Search privileges to the directory

Shared Volume HFS File System Calls

GetVolParms

Trap: $A260; D0 = $30

Parameter Block:

```
->   12   long   ioCompletion    ; optional completion routine ptr
<-   16   word   ioResult        ; error result code
->   18   long   ioFileName      ; volume name specifier
->   22   word   ioVRefNum       ; volume refNum
<-   32   long   ioBuffer        ; ptr to vol parms data
->   36   long   ioReqCount      ; size of buffer area
<-   40   long   ioActCount      ; length of vol parms data
```

The GetVolParms call is used to return volume level information. ioVRefNum or ioFileName contain the volume identifier information. ioReqCount and ioBuffer contain the size and location of the buffer in which to place the volume parameters. The actual size of the information is returned in ioActCount.

The format of the buffer is described below. Version 01 of the buffer is shown below along with offsets into the buffer and their equates:

offset	0	vMVersion	word	version number (currently 01)
	2	vMAttrib	long	attributes (detailed below)
	6	vMLocalHand	long	handle used to keep information necessary for shared volumes
	10	vMServerAdr	long	AppleTalk server address (zero if not supported)

On creation of the VCB (right after mounting), vMLocalHand will be a handle to a 2 byte block of memory. The Finder uses this for its local window list storage, allocating and deallocating memory as needed. It is disposed of when the volume is unmounted. For AppleTalk server volumes, vMServerAdr contains the AppleTalk internet address of the server. This can be used to tell which volumes are for which server.

vMAttrib contains attributes information (32 flag bits) about the volume. These bits and their equates are defined as follows:

bit	31	bLimitFCBs	If set, Finder limits the number of FCBs used during copies to 8 (instead of 16)
	30	bLocalWList	If set, Finder uses the returned shared volume handle for its local window list
	29	bNoMiniFndr	If set, Mini Finder menu item is disabled
	28	bNoVNEdit	If set, volume name cannot be edited
	27	bNoLclSync	If set, volume's modification date is not set by any Finder action
	26	bTrshOffLine	If set, anytime volume goes offline, it is zoomed to the Trash and unmounted
	25	bNoSwitchTo	If set, Finder will not switch launch to any application on the volume

24-21		RESERVED - server volumes should return these bits set, all other disks should return these bits cleared
20	bNoDeskItems	If set, no items may be places on the Finder desktop
19	bNoBootBlks	If set, no boot blocks on this volume - not a startup volume. SetStartup menu item will be disabled; boot blocks will not be copied during copy operations
18	bAccessCntl	If set, volume supports AppleTalk AFP access con trol interfaces. The calls GetLoginInfo, GetDirAccess, SetDirAccess, MapID, and MapName are supported. Special folder icons are used. Access Privileges menu item is enabled for disk and folder items. The privileges field of GetCatInfo calls are assumed to be valid.
17	bNoSysDir	If set, volume doesn't support a system directory; no switch launch to this volume
16	bExtFSVol	If set, this volume is an external file system volume. Disk init package will not be called. Erase Disk menu item is disabled.
15	bHasOpenDeny	If set, supports _OpenDeny and _OpenRFDeny calls. For copy operations, source files are opened with enable read/deny write and destination files are opened enable write/deny read and write.
14	bHasCopyFile	If set, _CopyFile call supported. _CopyFile is used in copy and duplicate operations if both source and destination volumes have same server address.
13	bHasMoveRename	If set, _MoveRename call supported. (Finder 5.4 does not use this call)
12	bHasNewDesk	If set, all of the new desktop calls are supported (e.g. OpenDB, AddIcon, AddComment, etc).
11-0		RESERVED - these bits should be returned cleared

GetLogInInfo

Trap: $A260; D0 = $31

Parameter Block:

```
->   12   long   ioCompletion     ; optional completion routine ptr
<-   16   word   ioResult         ; error result code
->   22   word   ioVRefNum        ; volume refNum
<-   26   word   ioObjType        ; log in method
<-   28   long   ioObjNamePtr     ; ptr to log in name buffer
```

GetLogInInfo returns the method used for log in and the user name specified at log in time for the volume. The log in user name is returned as a Pascal string in ioObjNamePtr. The maximum size of the user name is 31 characters. The log in method type is returned in ioObjType.

GetDirAccess

Trap: $A260; D0 = $32

Parameter Block:

```
->   12   long   ioCompletion    ; optional completion routine ptr
<-   16   word   ioResult        ; error result code
->   18   long   ioFileName      ; directory name
->   22   word   ioVRefNum       ; volume refNum
<-   36   long   ioACOwnerID     ; owner ID
<-   40   long   ioACGroupID     ; group ID
<-   44   long   ioACAccess      ; access rights
->   48   long   ioDirID         ; directory ID
```

GetDirAccess returns access control information for the folder pointed to by the ioDirID/ioFIleName pair. ioACOwnerID will return the ID for the folder's owner. ioACGroupID will return the ID for the folder's primary group. The access rights are returned in ioACAccess.

A 'fnfErr' is returned if the pathname does not point to a valid directory. An 'AccessDenied' error is returned if you do not have the correct access rights to examine this directory.

SetDirAccess

Trap: $A260; D0 = $33

Parameter Block:

```
->   12   long   ioCompletion    ; optional completion routine ptr
<-   16   word   ioResult        ; error result code
->   18   long   ioFileName      ; pathname identifier
->   22   word   ioVRefNum       ; volume refNum
->   36   long   ioACOwnerID     ; owner ID
->   40   long   ioACGroupID     ; group ID
->   44   long   ioACAccess      ; access rights
->   48   long   ioDirID         ; directory ID
```

SetDirAccess allows you to change the access rights to a folder pointed to by the ioFileName/ioDirID pair. ioACOwnerID contains the new owner ID. ioACGroupID contains the group ID. ioACAccess contains the folder's access rights. You cannot set the owner bit or the user's rights of the directory. To change the owner or group, you should set the ioACOwnerID or ioACGroupID field with the appropriate ID of the new owner/group. You must be the owner of the directory to change the owner or group ID.

A 'fnfErr' is returned if the pathname does not point to a valid directory. An 'AccessDenied' error is returned if you do not have the correct access rights to modify the parameters for this directory. A 'paramErr' is returned if you try to set the owner bit or user's rights bits.

MapID

Trap: $A260; D0 = $34

Parameter Block:

```
->    12    long   ioCompletion     ; optional completion routine ptr
<-    16    word   ioResult         ; error result code
->    18    long   ioFileName       ; volume identifier (may be NIL)
->    22    word   ioVRefNum        ; volume refNum
->    26    word   ioObjType        ; function code
<-    28    long   ioObjNamePtr     ; ptr to returned creator/group name
->    32    long   ioObjID          ; creator/group ID
```

MapID returns the name of a user or group give its unique ID. ioObjID contains the ID to be mapped. The value zero for ioObjID is special cased and will always return a NIL name. AppleShare uses this to signify '<Any User>'. ioObjType is the mapping function code; it's 1 if you're mapping an owner ID to owner name or 2 if you're mapping a group ID to a group name. The name is returned as a Pascal string in ioObjNamePtr. The maximum size of the name is 31 characters.

A 'fnfErr' is returned if an unrecognizable owner or group ID is passed.

MapName

Trap: $A260; D0 = $35

Parameter Block:

```
->    12    long   ioCompletion     ; optional completion routine ptr
<-    16    word   ioResult         ; error result code
->    18    long   ioFileName       ; volume identifier (may be NIL)
->    22    word   ioVRefNum        ; volume refNum
->    28    long   ioObjNamePtr     ; owner or group name
->    26    word   ioObjType        ; function code
<-    32    long   ioObjID          ; creator/group ID
```

MapName returns the unique user ID or group ID given its name. The name is passed as a string in ioObjNamePtr. If a NIL name is passed, the ID returned will always be zero. The maximum size of the name is 31 characters. ioObjType is the mapping function code; it's 3 if you're mapping an owner name to owner ID or 4 if you're mapping a group name to a group ID. ioObjID will contain the mapped ID.

A 'fnfErr' is returned if an unrecognizable owner or group name is passed.

CopyFile

Trap: $A260; D0 = $36

Parameter Block:

```
->    12    long   ioCompletion      ; optional completion routine ptr
<-    16    word   ioResult          ; error result code
->    18    long   ioFileName        ; ptr to source pathname
->    22    word   ioVRefNum         ; source vol identifier
->    24    word   ioDstVRefNum      ; destination vol identifier
->    28    long   ioNewName         ; ptr to destination pathname
->    32    long   ioCopyName        ; ptr to optional name (may be NIL)
->    36    long   ioNewDirID        ; destination directory ID
->    48    long   ioDirID           ; source directory ID
```

CopyFile duplicates a file on the volume and optionally renames it. It is an optional call for AppleShare file servers. You should examine the returned flag information in the GetVolParms call to see if this volume supports CopyFile.

For AppleShare file servers, the source and destination pathnames must indicate the same file server, however it may point to a different volume for that file server. A useful way to tell if two file server volumes are on the same file server is to make the GetVolParms call and compare the server addresses returned. The server will open source files with enable read/deny write and destination files with enable write/deny read and write.

ioVRefNum contains a source volume identifier. The source pathname is determined by the ioFileName/ioDirID pair. ioDstVRefNum contains a destination volume identifier. AppleShare 1.0 required that it be an actual volume reference number, however on future versions it can be a WDRefNum. The destination pathname is determined by the ioNewName/ioNewDirID pair. ioCopyName may contain an optional string used in renaming the file. If it is non-NIL then the file copy will be renamed to the specified name in ioCopyName.

A 'fnfErr' is returned if the source pathname does not point to an existing file or the destination pathname does not point to an existing directory. An 'AccessDenied' error is returned if the user does not have the right to read the source or write to the destination. A 'dupFnErr' is returned if the destination already exists. A 'DenyConflict' error is returned if either the source or destination file could not be opened under the access modes described above.

MoveRename

Trap: $A260; D0 = $37

Parameter Block:

```
->    12    long   ioCompletion    ; optional completion routine ptr
<-    16    word   ioResult        ; error result code
->    18    long   ioFileName      ; ptr to source pathname
->    22    word   ioVRefNum       ; source vol identifier
->    28    long   ioNewName       ; ptr to destination pathname
->    32    long   ioBuffer        ; ptr to optional name (may be NIL)
->    36    long   ioNewDirID      ; destination directory ID
->    48    long   ioDirID         ; source directory ID
```

MoveRename allows you to move (not copy) an item and optionally rename it. The source and destination pathnames must point to the same file server volume.

ioVRefNum contains a source volume identifier. The source pathname is specified by the ioFileName/ioDirID pair. The destination pathname is specified by the ioNewName/ioNewDirID pair. ioBuffer may contain an optional string used in renaming the item. If it is non-NIL then the moved object will be renamed to the specified name in ioBuffer.

A 'fnfErr' is returned if the source pathname does not point to an existing object. An 'AccessDenied' error is returned if the user does not have the right to move the object. A 'dupFnErr' is returned if the destination already exists. A 'badMovErr' is returned if an attempt is made to move a directory into one of its descendent directories.

OpenDeny

Trap: $A260; D0 = $38

Parameter Block:

```
->    12    long   ioCompletion     ; optional completion routine ptr
<-    16    word   ioResult         ; error result code
->    18    long   ioFileName       ; ptr to pathname
->    22    word   ioVRefNum        ; vol identifier
<-    24    word   ioRefNum         ; file refNum
->    26    word   ioDenyModes      ; access rights data
->    48    long   ioDirID          ; directory ID
```

OpenDeny opens a file's data fork under specific access rights. It creates an access path to the file having the name pointed to by ioFileName/ioDirID. The path reference number is returned in ioRefNum.

ioDenyModes contains a word of access rights information. The format for these access rights is:

bits	15-6	RESERVED - should be cleared
	5	If set, other writers are denied access
	4	If set, other readers are denied access
	3-2	RESERVED - should be cleared
	1	If set, write permission requested
	0	If set, read permission requested

A 'fnfErr' is returned if the input specification does not point to an existing file. A 'permErr' is returned if the file is already open and you cannot open under the deny modes that you have specified. An 'opWrErr' is returned if you have asked for write permission and the file is already opened by you for write. The already opened path reference number is returned in ioRefNum. An 'AccessDenied' error is returned if you do not have the right to access the file.

OpenRFDeny

Trap: $A260; D0 = $39

Parameter Block:

```
->    12    long   ioCompletion    ; optional completion routine ptr
<-    16    word   ioResult        ; error result code
->    18    long   ioFileName      ; ptr to pathname
->    22    word   ioVRefNum       ; vol identifier
<-    24    word   ioRefNum        ; file refNum
->    26    word   ioDenyModes     ; access rights data
->    48    long   ioDirID         ; directory ID
```

OpenRFDeny opens a file's resource fork under specific access rights. It creates an access path to the file having the name pointed to by ioFileName/ioDirID. The path reference number is returned in ioRefNum. The format of the access rights data contained in ioDenyModes is described under the OpenDeny call.

A 'fnfErr' is returned if the input specification does not point to an existing file. A 'permErr' is returned if the file is already open and you cannot open under the deny modes that you have specified. An 'opWrErr' is returned if you have asked for write permission and the file is already opened by you for write. The already opened path reference number is returned in ioRefNum. An 'AccessDenied' error is returned if you do not have the right to access the file.

Modified Existing HFS Calls

GetCatInfo

Trap: $A260; D0 = $09 (_GetCatInfo)

Parameter Block (new fields only):

```
<-    31    byte    ioACUser        ; access rights for directory only
```

GetCatInfo returns information about the file and directories in a file catalog. Please refer to Inside Macintosh Volume IV for the exact format of the parameter block.

For server volume directories, in addition to the normal return parameters the ioACUser field returns the user's access rights in the following format:

Bit	7	if set, user is not the owner of the directory if clear, user is the owner of the directory
	6-3	RESERVED; this is returned zeroed
	2	If set, user does not have Make Changes privileges to the directory If clear, user has Make Changes privileges to the directory
	1	If set, user does not have See Files privileges to the directory If clear, user has See Files privileges to the directory
	0	If set, user does not have See Folders privileges to the directory If clear, user has See Folders privileges to the directory

For example, if ioACUser returns zero for a given server volume directory, you know that the user is the owner of the directory and has complete privileges to it.