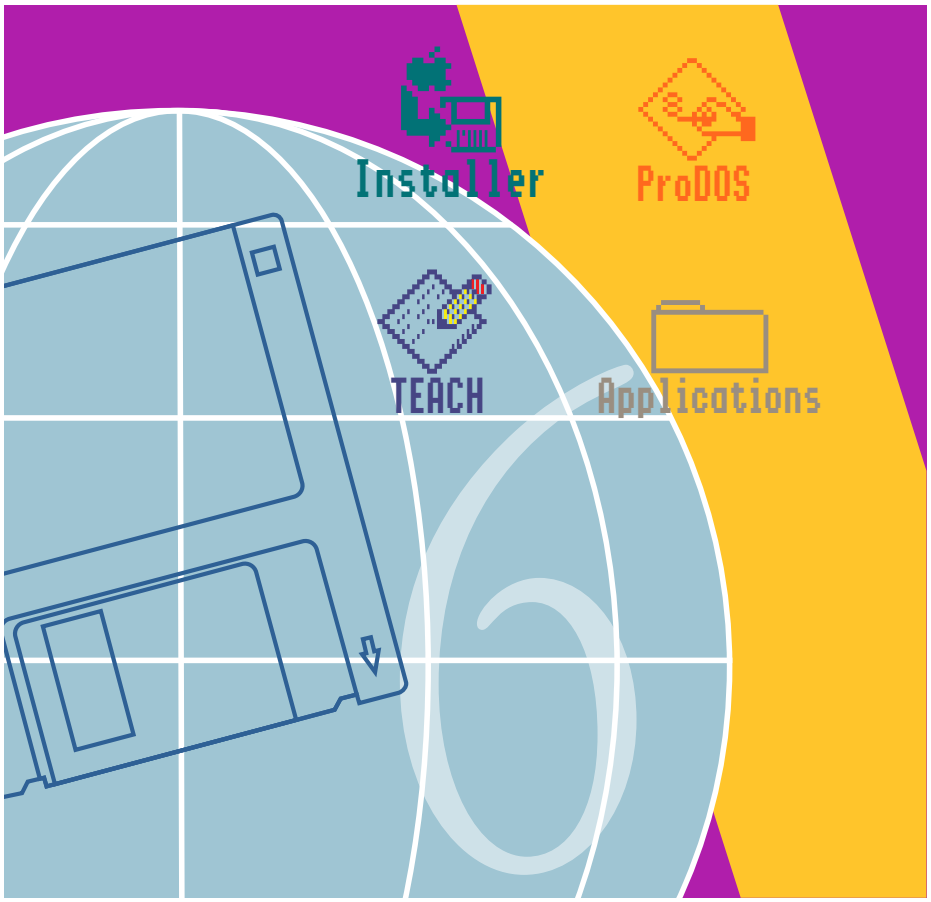
GS/OS® AppleShare File System Translator External ERS

Version 0.26CD



System 6 IIGS

Apple® IIGS® System Software 6.0 — Release Notes
Golden Master 3 Release — March 5, 1992

Engineering Reference Specifications (ERSs) for 6.0.
Version 0.26CD by Mark Day
PDF by khaibitgfx@outlook.com
July 8, 2019

20525 Mariani Avenue
Cupertino, CA 95014-6299
(408) 996-1010

khaibitgfx@outlook.com

Contents

Overview

This document describes the implementation of the AppleShare file System Translator for GS/OS®. It assumes a familiarity with GS/OS® and AppleTalk. Refer to the AppleShare IIGS Programmer's Guide for information about AppleTalk protocols. Please note that AppleShare for GS/OS® encompasses not only the AppleShare FST, but also the AppleTalk protocol stack, drivers, network booting (if desired), switching between the GS/OS® FST and the ProDOS 8 PFI (since AppleShare will be accessible from both ProDOS 8 and GS/OS®), and enhancements to the finder to make it network aware.

The AppleShare FST is the implementation of AppleShare for GS/OS®. It is meant to supersede AppleShare IIGS, the implementation of AppleShare for ProDOS 16. Since ProDOS 16 makes calls to ProDOS 8 to get its work done, AppleShare IIGS patches the ProDOS 8 MLI to intercept calls bound for the network. In this way, both ProDOS 8 and ProDOS 16 can use network volumes. GS/OS® is completely separate from ProDOS 8. The ProDOS 8 MLI will still be patched to intercept network calls while ProDOS 8 is running. When GS/OS® is running, GS/OS® will make calls directly to the AppleTalk routines via the AppleShare FST, instead of calling ProDOS 8 to make the AppleTalk calls. This will increase the speed of GS/OS® programs using files on the network (compared to ProDOS 16).

The AppleShare FST will only work with file servers supporting AFP version 2.0 or greater.

Compatibility

An important consideration for the AppleShare file System Translator is backwards compatibility with ProDOS 16 and ProDOS 8 implementations of AppleShare. All documented calls that were added to the ProDOS 8 MLI to support AppleShare will still be usable from ProDOS 8. The RamDispatch vector at $E11014 will continue to support full native mode calls from either ProDOS 8 or GS/OS. In addition, device-specific calls will be included to allow uniform access to the AppleTalk protocols and file system features such as Byte Range Lock.

The class 0 Open call will work as the Open call for AppleShare for ProDOS16 did. The class 1 Open call is more restrictive in its setting of deny modes which is safer for opening files. Please use class 1 Open whenever possible, and try to use the requested access parameter when possible (e.g.: only ask for read if that is all you will do with the file).

File not found and path not found errors will be reported correctly (when a file is not found, the FST will check for the existence of the parent and issue a path not found if the parent does not exist). This differs from ProDOS16 which reported path not found for both path not found and file not found error conditions.

Pathname Syntax

There are two kinds of syntactic restrictions on pathname syntax:

- Those imposed by GS/OS
- Those imposed by the FST (because of naming restrictions in AFP)

GS/OS may impose a maximum length on pathnames. The AppleShare FST does not.

The span of a pathname is the maximum number of characters in a filename (i.e. between pathname separators, including volume names). GS/OS imposes no restriction on maximum span. The AppleShare FST restricts the maximum span to be less than 32 characters. While AFP volume names are less than 28 characters, this part of the syntax is not checked. Volume names with a length of 28-31 will return a volume not found error.

GS/OS allows "/" or ":" to be a separator. The first "/" or ":" in the pathname is taken to be the separator. A ":" can never be used in a filename. A "/" cannot be used in a filename if the separator is "/". The AppleShare FST disallows a null byte in a pathname. All other characters are permitted.

◆ **Note** The high bit of a character is significant. Characters with values greater than or equal to 128 are considered extended ASCII and typically display as special symbols on Macintosh and IBM systems. ◆

Numbers as the first filename in a partial pathname are assumed by GS/OS to be prefix designators. Since numbers are valid filenames in AFP, a prefix designator should always be used explicitly with partial pathnames beginning with a number.

For example, “0:555:Hello” refers to a file “Hello” in a folder “555” relative to prefix 0; “555:Hello” will give an invalid path syntax error since GS/OS assumes that “555:” is a prefix designator for prefix 555, which is invalid.

Equivalence of Macintosh and GS/OS file Types

AppleShare file servers supporting AFP version 2.0 or greater maintain both Macintosh filetype and creator as well as ProDOS filetype and auxtype. Since the filetype information for the two operating systems are distinct, a workstation can set one kind of filetype for Macintosh and another type for ProDOS.

The AppleShare FST will use the ProDOS filetype and auxtype fields if either is non-zero. The AppleShare file Server version 2.0 uses a convention also used by Apple file Exchange and the MAX cross-development tools.

ProDOS files are distinguished by a Macintosh creator of “pdos”. The ProDOS filetype SYS (=$FF) has a Macintosh filetype of “PSYS”. The ProDOS filetype S16 (=$B3) has a Macintosh filetype of “PS16”. The ProDOS unknown filetype (=$00) has Macintosh filetype “BINA”. ProDOS text files (TXT = $04) with auxtype of $0000 (i.e. normal ASCII text, no records) has Macintosh filetype “TEXT”. These special cases allow Macintosh to display unique icons for these filetypes.

Macintosh files with creator “pdos” and a filetype of the form “XY “ (two hex digits followed by two spaces) will get ProDOS filetype $XY and auxtype $0000. Macintosh files with creator “pdos” and a filetype of the form $70uvwxyz ($70 is a lower-case “p”) have ProDOS filetype $uv and auxtype $wxyz (note the order of the bytes on the Macintosh they are stored high-low instead of low-high).

APW source files (ProDOS filetype $B0) are given Macintosh filetype “TEXT” so that they can be edited more easily.

The conversion rules are summarized in the following tables. If more than one rule applies, the one closest to the top of the table will be used.

AppleShare 2.0: ProDOS to Macintosh conversion

ProDos filetype	Auxtype	Macintosh Creator	filetype
$00	$0000	“pdos”	“BINA”
$B0 (SRC)	(any)	“pdos"	"TEXT"
$04 (TXT)	$0000	“pdos”	“TEXT”
$FF (SYS)	(any)	“pdos”	“PSYS”
$B3 (S16)	(any)	“pdos”	“PS16”
$uv	$wxyz	“pdos”	“p” $uv $wx $yz

AppleShare 2.0: Macintosh to ProDOS conversion

Macintosh Creator	filetype	ProDos filetype	Auxtype
(any)	"BINA"	$00	$0000
(any)	"TEXT"	$04 (TXT)	$0000
"pdos"	"PSYS"	$FF (SYS)	$0000
"pdos"	"PS16"	$B3 (S16)	$0000
"pdos"	"XY▲▲" *	$XY	$0000
"pdos"	"p" $uv $wx $yz	$uv	$wxyz
(any)	(any)	$00	$0000

* Where X,Y are hex digits (i.e. "0"-"9" or "A"-"F"), and ▲ is a space.

As of System Software 6.0d24, the AppleShare FST does some file type conversion in addition to that done by the file server. These new conversions are performed solely by the AppleShare FST and override the normal conversions described above. One of the conversions applies to GS/OS-aware applications so that their auxtypes may be preserved (since they contain important flags). The other conversions are for files that contain information that both an Apple IIGS and a Macintosh can access.

When setting the filetype/auxtype (such as in a SetfileInfo or Create call), and no option_list with finder Info is supplied, the Macintosh file type and creator are set as shown in the table below. finder Info present in an option_list always overrides these conversions. If the ProDOS filetype and auxtype do not match an entry in the table, no conversion is performed and the Macintosh type and creator are not explicitly changed (although the server's default filetype conversions still apply).

System 6.0: ProDOS to Macintosh conversion

ProDOS filetype	Auxtype	Macintosh Creator	filetype
$B3	$DBxy	"pdos"	"p" $B3 $DB $xy
$D7	$0000	"pdos"	"MIDI"
$D8	$0000	"pdos"	"AIFF"
$D8	$0001	"pdos"	"AIFC"

When getting the filetype/auxtype (such as in a GetfileInfo, Open, or GetDirEntry call), if the server returns an Apple II filetype and auxtype both equal to zero, they will be derived from the Macintosh type and creator as given in the following table. The Macintosh creator is ignored (any value is acceptable).

If the Macintosh type does not match an entry in the table below, no conversion is done and the Apple II filetype and auxtype are both returned as zeros.

System 6.0: Macintosh to ProDOS conversion

Macintosh filetype	ProDOS filetype	Auxtype
"MIDI"	$D7	$0000
"AIFF"	$D8	$0000
"AIFC"	$D8	$0001

finder Info

The option_list used by the AppleShare FST contains 32 bytes denoted as finder Info. The contents of the finder Info are not modified or used in any way by the FST. AppleShare, the file system, and the FST simply provide room for 32 bytes per file/folder. The Macintosh finder is solely responsible for the definition and interpretation of the contents of these bytes.

If you plan to use the information stored here, it is up to you to make sure the values make sense (and to do something appropriate if they don't make sense). Beware that the values are likely to be zero when an object is created with something other than the Macintosh finder, and may not have sensible values until you specifically change them via the Macintosh finder. Even worse, different versions of the Macintosh finder may store different values in the same fields.

For example, System 7.0's finder seems to set the window rectangle for new folders to a rectangle of zero area, but with non-zero coordinates. Apparently, this tells the finder the upper left corner of the window, but lets the window dynamically size to fit the number of items inside. I'm not sure about this; it is purely speculation on my part based on some behavior I've seen.

For your convenience, the description of the finder Info given in Inside Macintosh IV is copied below.

The finder Info for a file looks like this:

4 chars	Macintosh filetype	
4 chars	Macintosh creator	
word	finder flags	
	Bit	Meaning
	0	Set if file/folder is on the desktop (finder 5.0 and later)
	1	bFOwnAppl (used internally)
	2	reserved (currently unused)
	3	reserved (currently unused)
	4	bFNever (never SwitchLaunch) (not implemented)
	5	bFAlways (always SwitchLaunch)
	6	Set if file is a shareable application
	7	reserved (used by System)
	8	Inited (seen by finder)
	9	Changed (used internally by finder)
	10	Busy (copied from file System busy bit)
	11	NoCopy (not used in 5.0 and later, formerly called BOZO)
	12	System (set if file is a system file)
	13	HasBundle
	14	Invisible
	15	Locked
point	file icon's location in window	
	word	vertical
	word	horizontal
word	Window # file belongs to	
	-3 = file is in Trash window	
	-2 = file is on desktop	
	0 = file is in disk window	
word	Icon ID	
8 bytes	(unused)	
word	Comment ID	
long	Directory ID of home folder (for Put Away)	

For files, the finder Info looks like this:

rectangle	Folder's window rectangle	
	word	top
	word	left
	word	bottom
	word	right
word	finder flags (same as for a file??)	
point	Folder icon's location in window	
	word	vertical
	word	horizontal
word	Folder's view (??)	
point	Window's scroll position	
	word	vertical
	word	horizontal
long	Directory ID chain of open folders (??)	
word	(unused)	
word	Comment ID	
long	Directory ID of home folder (for Put Away)	

◆ **Note** The bytes in the finder Info are in high-low order (the same as a 68000, opposite that of a 65816). The finder Info is really the concatenation of the finfo and FXInfo structures in HFS on the Macintosh. This is particularly important for the multi-byte fields containing flags and integers -- they are in 68000 format. ◆

System Calls

This section describes differences of parameters between the AppleShare FST and the ProDOS FST. Please see the System Call ERS for more detailed information about these calls. Any calls not documented here behave as specified in the System Call ERS.

CREATE ($01)

The ProDOS filetype and auxtype will be set to the values given in the call; by default, the Macintosh creator will be set to "pdos" and the Macintosh filetype will be derived according to the rules above. All files will be created as extended files (i.e. have both a data and a resource fork) since there is no way to distinguish between a fork of length 0 and a fork that does not exist.

In a class 1 call, the EOF and resource_EOF fields are ignored. This is because the definition of the call states that the forks' EOFs will be set to 0, and it is impossible with AFP to allocate space in a fork past its EOF.

Only the low byte of the filetype and low word of the auxtype will be used. If the high byte of the filetype or high word of the auxtype is non-zero, an invalid parameter error will be returned.

When a directory is created, its access privileges, owner, and group will be set based upon the settings of the last SetDefPrivileges call. This lets you override the server's default of exclusive access to the owner only.

◆ **Note** If the default owner or group name is not valid for the server on which the folder is created, that setting will retain its default value (i.e. the user that created the folder or their primary group) and no error will be returned. ◆

SET_FILE_INFO ($05)

The ProDOS filetype and auxtype will be set to the values given in the call; by default, the Macintosh creator will be set to "pdos" and the Macintosh filetype will be derived according to the rules above. The option_list data is the same as for the GET_FILE_INFO call, except that only the finder Info is used (the other fields cannot be set); any data past the finder Info field is ignored.

If the file_sys_id field is not the same as AppleShare's file system ID ($0D) or the HFS file system ID (6), then the option_list is ignored. All FSTs will return their file system ID in the first word of the option_list and will ignore setting of the option_list info if they do not understand that file system's option_list format. This allows applications to always get and set the option_list as part of the copying process even when copying from one file system to another. The AppleShare FST and HFS FST have the same format in their option_list.

GET_FILE_INFO ($06)

Folders with no see files and no see folders access will have the read bit in their access word cleared; files, and folders with see files or see folders, have their read bit set. If the file's resource fork is not empty, the storage_type will be returned as $05 (extended), otherwise it will be returned as $01/$02/$03 (seedling, sapling, or tree) depending on the data fork's length.

The option_list's data is structured as follows:

- word file_Sys_ID ($0D for AppleShare)
- 32 bytes finder Info
- long Parent Directory ID
- 4 bytes Access rights (same format as Get-/SetPrivileges)

See Appendix B for a diagram of the option_list structure. The access rights field for directories is in the same format as used in the GetPrivileges and SetPrivileges calls. For files, the field is set to all zeros.

◆ **Note** This field was included to allow applications like the finder to determine what access a user has to a folder without having to do a separate GetPrivileges call. ◆

OPEN ($10)

The access, filetype, auxtype, and option_list parameters are as described in the SET_fiLE_INFO call. If request_access is $0000 (as permitted), an attempt will be made to open the file as read/write, deny read/write. If this fails, an attempt will be made to open the file as read-only, deny write. If this fails, an attempt will be made to open the file as write-only, deny read/write. If this also fails, an access denied error ($4E) will be returned. If the class is 0, an attempt will be made to open the file as read/write deny write. If this fails an attempt will be made to open the file as read-only deny nothing. If this fails, an attempt will be made to open the file as write-only deny write. If this also fails, an access denied error ($4E) will be returned. This behavior is the same as for ProDOS16 and was done for compatibility with ProDOS16.

◆ **Note** Using class 0 Open allows files to be opened by multiple users and does not fully prevent one user from changing data that another user is reading, but it does allow multiple users to read a file without changing existing code. Class 1 Open prevents one user from writing data that another user is reading, but does not allow multiple users to read a file without explicitly asking for read-only access. Putting a file in a folder with no make changes access will cause both class 0 Open and class 1 Open with request_access = 0 to open the file for read-only and will allow multiple users to read the file (and not allow the file to be written to). If request_access is $0001 (read-only), the file will be opened as read-only, deny write. If it is $0003 (read/write), the file will be opened as read/write, deny read/write. If it is $0002 (write-only), the file will be opened as write-only, deny read/write. If the file cannot be opened with the requested mode, an access denied error will be returned. ◆

If you want to open a file with permissions different than above, you should use the FST specific command "Special Open Fork". That call is essentially the same as the open command, but it lets you control all of the permission bits yourself.

The System Loader should be modified to load files by opening them Read-only, Deny Write (request_access as $0001).

By default, buffering will be turned on for files or directories opened with this call. The buffer will not be filled until the first Read or Get_Dir_Entry call is made (so that buffering may be turned off after the open but before the first read). The size of the buffer for files is 512 bytes; for directories it is 2048 bytes.

Folders with neither see files nor see folders access rights cannot be opened (since the only valid operation on an open folder is GET_DIR_ENTRY). The returned error code is $4E (access denied).

With a class 1 call, all of the parameters after the resource number are file information. Think of this as a combined GET_FILE_INFO and OPEN call (and in fact, that is how it behaves). In particular, the access word returned is not an indication of the access rights you have when the file is opened; it is really a "best case" access to the file.

The actual access you get when opening the file is controlled by several things:

- The access word
- Access privileges to ancestor and parent folders
- Access restrictions ("deny modes") imposed by other users who have the file open.

◆ Note Using a class 1 open with request_access = 0, is usually not a good idea since you don't know what access you really got to the file (until you try) because the FST will try several combinations as described above. ◆

If your application can deal with several different kinds of access to the file, it is best to try those different access modes individually until you get one you can handle. For example, if you can handle either read-write or read-only access but prefer read-write, try opening the file with request_access = 3 (read-write). If this fails, try opening with request_access = 1 (read-only). This way you will know exactly what access you have to the file. Rememer, too that if you use class 1 open for read-write, nobody else will be able to open the file and multiple users won't be able to run your application at the same time.

If you use a class 1 call with pCount > 4 (i.e. you are asking for file info to be returned), and you don't have privileges to see the object you are opening (if the object is in a drop box, for example), the call will return with an error $4E (access denied), since you don't have access to get the file info you requested.

READ ($12)

The READ call will not be supported for directories. ProDOS directories will not be synthesized. One should use the Get_Dir_Entry call to enumerate directories. A read on a directory will return error $4E (access denied).

If part of the range to be read is locked by another workstation, a $4E error will be returned and the transfer count will be set to indicate the number of bytes transfered before the locked range was encountered.

Regardless of the value in the cache priority field, data will not be put in the system cache. By default, the FST maintains a block buffer containing the 512 bytes of the block containing the current mark. This block buffer can be controlled on a per-file basis by the FST specific call “Buffer Control”.

If buffering is disabled and newline mode has been enabled with more than one newline character, the read will be completed one byte at a time. This is done because the server’s newline mechanism provides for only one newline character. Beware that this mode of reading a file imposes tremendous amounts of overhead and should be avoided if at all possible.

WRITE ($13)

Regardless of the value in the cache priority field, data will not be put in the system cache. By default the FST maintains a block buffer containing the 512 bytes of the block containing the current mark. This block buffer can be controlled on a per-file basis by the FST specific call “Buffer Control”.

Writes to directories are not allowed. They will return error $4E (access denied).

CLOSE ($14)

The file will always be closed (i.e. the FCR will be removed), even if there is an error. This is because any error an application gets may not be correctable by the application or the user (eg. the data to be flushed before the close is locked by another workstation, or a connection has been lost with the server). It seems better to simply indicate the error and clean up the system as much as possible (i.e. remove the FCR, and reduce the VCR’s open file count).

SET_EOF ($18)

If a fork is extended (make longer), the additional bytes will be allocated but might not all be zero.

In a class 1 call, if the base indicates that the EOF should be set to EOF - displacement, the server’s current EOF will be determined and the EOF will be set relative to that; this could be different than the workstations assumption of the EOF if another workstation has modified the fork’s EOF.

This could also delete data that another workstation has written between the times when the current EOF was determined and the new EOF set. This call will force any buffered data to be written to the server. The EOF will be set after this data is written.

GET_EOF ($19)

The fork's EOF will be determined from the server; this may not match the workstation's assumption of the EOF if another workstation has modified the fork's EOF. Note that another workstation could change the EOF after completion of this call, making the results inacurate.

This call will force any buffered data to be written to the server. The EOF will be determined after this data is written. This should avoid the problem of the returned EOF being less than the current mark.

GET_DIR_ENTRY ($1C)

Get_Dir_Entry is not supported for files. It will return the error $4E (access denied).

Folders enumerated by GET_DIR_ENTRY that have neither see files nor see folders will have the read bit in their access word cleared. files, and folders with see files or see folders, will have the read bit set.

The access, filetype, auxtype, and option_list parameters are as the GET_fiLE_INFO call. The FST will internally maintain the directory entry number (entry_num) to allow forward and backward scanning of the directory. By default, several entries will be buffered for better performance (this can be disabled by using the FST Specific call "Buffer Control"). An end of directory error ($61) will be returned when an entry is requested that does not exist in the buffer (or buffering is disabled for the directory), and that entry cannot be read from the server.

Since AppleShare is a shared file system, entry_num may change for a file, even while the directory is being scanned because other users could add or delete files in the directory. Also, if the base and displacement fields are both zero, the total number of entries will be returned.

◆ Note More or fewer entries may actually be returned if the directory is enumerated since other machines can create and delete files while you are enumerating the directory. The best way to enumerate a directory is to simply open the directory and make successive Get_Dir_Entry calls with base and displacement both set to $0001. When you get an error $61 (end of directory), you are finished enumerating. You should remove duplicate entries from your list. ◆

READ_BLOCK ($22)

This call will return an error $88 (network error) for AppleShare devices, in order to be compatible with System Disk 3.2. Remember, the preferred method for identifying a network volume is by doing a Volume call and seeing that the file_sys_id = $0D.

WRITE_BLOCK ($23)

This is an invalid operation for an AppleShare device. This call always return an error. The current error code is $4E (access denied). This is different from the $88 returned under 3.2, and may change in the future.

FORMAT ($24)

This is an invalid operation for an AppleShare device. This call always return an error. The current error code is $2B (write protected). This is different from the $88 returned under 3.2, and may change in the future.

ERASE_DISK ($25)

This is an invalid operation for an AppleShare device. This call always returns an error. The current error code is $2B (write protected). This is different from the $88 returned under 3.2, and may change in the future.

GET_BOOT_VOL ($28)

If GS/OS is booted over AppleTalk, this command will return the name of the user volume on the server the user logged in to during booting. All system files should be present on this volume just like any other boot volume.

GET_FST_INFO ($2B)

The file_sys_id will be returned as $0D (AppleShare). The attribute parameter will be returned as $0000 (System Call Manager should not uppercase pathnames, do not clear high bits of pathname, this is a block FST, formatting not supported). The block_size parameter will be returned as 512; this value is only useful in determining the number of bytes used, free, and total on a volume (since these values are given in blocks).

FST_SPECIFIC ($33)

The FST_SPECIFIC call is used to make the Buffer Control, Byte Range Lock and Special Open Fork calls. The FST number must be $0D (AppleShare). A command of $0000 is invalid. Commands $000E through $FFFF are reserved.

If the command number is out of range, error $53 (invalid parameter) will be returned. Error $52 (unknown volume type) will be returned if a refnum for a file opened by another FST is used. Error $52 will also be returned if a pathname uses a device name for a device other than an AFP (AppleShare) driver. Error $45 (volume not found) will be returned if a pathname specifies a volume name that does not match any mounted AppleShare volume (even if a volume by that name exists for a different file system).

Buffer Control

Command $0001 is the Buffer Control command. It is followed by a word specifying the reference number of a file/directory whose buffering is to be enabled/disabled. The next word is optional. It specifies the buffer disable flags; if the high bit is set, then buffering is disabled for that file/directory. The default value of the buffer disable parameter is $0000 (turn on buffering). A file reference number of 0 is invalid.

For folders, the buffer size is 2048 bytes. When buffering is off, each Get_Dir_Entry will immediately cause an enumerate of one entry from the server. When a Get_Dir_Entry call is made with buffering on, the requested entry will be returned from the buffer if possible. Otherwise, the buffer will be filled with as many entries from the server as possible, including the requested entry; then the requested entry will be returned. The buffer is not pre-filled when the folder is opened. The number of entries kept in the buffer is variable and depends on the size of the long and short names of the files/folders.

For files, the buffer size is 512 bytes (the same as the block size reported by the FST). When buffering is off, every Read and Write call transfers data from/to the user's data buffer directly to/from the server. When buffering is on, and a Read or Write of 512 bytes or more is made, any unwritten data in the buffer is written and the Read/Write is made from/to the user's data buffer directly to/from the server.

When buffering is on and a Read or Write of less than 512 bytes is made, the block (512 bytes, with a starting offset that is a multiple of 512 bytes) containing the first byte to be read/written is read into the FCR's buffer; if the block was already in the buffer, no read is done; if a different block is in the buffer, any unwritten data is written and the new block is read into the buffer. The read/write then proceeds to the end of the buffer. If the read/write extends past the end of the buffer, any unwritten data is written and the next block is read into the buffer. The read/write then completes by reading/writing from/to the buffer.

Unbuffered reads with 0 or 1 newline characters are handled directly by the server (i.e. the read to the server requests the same number of bytes as the user requested). Unbuffered reads with 2 or more newline characters turn into reads of one character at a time from the server (until a newline is encountered or all bytes have been read or end of file reached); please note that this takes a LONG time, and you are probably better off not using 2 or more newline characters with buffering off.

Buffered reads with 1 or more newline characters become reads of 512 bytes at a time, on 512 byte boundaries (as if it were a read of less than 512 bytes). Each block is read into the FCR's buffer and then the bytes are copied to the user's data buffer one at a time (while being compared against all the newline characters). Buffered reads with no newline characters are as described above.

Byte Range Lock

Command $0002 is the Byte Range Lock command. It is followed by five required parameters (so the PCount field should be 7, 2 for FST # and Command, 5 for the parameters of Byte Range Lock). The first parameter is a word containing the reference number of the file to lock. The second parameter is the Lock Flag. If bit 15 is

set, the range will be locked; if clear, it will be unlocked. If bit 14 is set, the offset is relative to the end of the file; if clear, the offset is relative to the start of the file. All other bits are reserved and should be set to 0.

The next parameter is a long word containing the offset into the file (may be negative if relative to the end of the file). The next parameter is the length of the range to be locked. The last parameter is the actual start of the locked range (relative to the beginning of the file) as returned by the server.

Possible errors are:

- $4D (position out of range -- user already has some or all of range already locked or unlocking a range not locked by that user)
- $4E (access denied -- some or all of range is locked by another user)
- $43 (invalid reference number)
- $53 (invalid parameter)

Special Open Fork

Command $0003 is the Special Open Fork command. It is followed by three required parameters and one optional parameter (so the PCount field should be 5 or 6: 2 for the FST# and Command, and 3 or 4 for the parameters of Special Open Fork). The first parameter is the reference number (ref_num) returned by GS/OS to the access path. Use this ref_num the same as you would a ref_num returned by an OPEN call. The second parameter is a pointer to a class 1 string representing the pathname of the file to be opened. The third parameter is the access mode giving the read/write permissions desired and to be denied to others as described below. The forth, and optional, parameter is the resource number: a value of $0000 will cause the data fork to be opened, a value of $0001 will cause the resource fork to be opened; a value of $0000 is assumed if the parameter is not given.

The access word is arranged as follows (if the bit is set, the condition is asserted):

- Bit 0 Request Read Access
- Bit 1 Request Write Access
- Bit 2,3 Reserved
- Bit 4 Deny Read to others
- Bit 5 Deny Write to others
- Bits 6..15 Reserved

◆ Note This parameter has the same meaning as in the ProDOS 8 Special Open Fork command. Possible errors: same as for OPEN command. A deny mode conflict will result in an access denied error. ◆

GetPrivileges

Command $0004 is the GetPrivileges command. It is followed by four parameters, the first two of which are required (so the minimum PCount is 4 and the maximum is 6). The first parameter is a pointer to a class 1 pathname of a directory whose access privileges are to be set or retrieved. The second parameter is a long where access rights for the directory will be returned. The third parameter is a pointer to a GS/OS output buffer where the owner's name will be stored. The fourth parameter is a poiner to a GS/OS output buffer where the group name will be stored.

The access rights field consists of four bytes:

- One each for user summary
- World access, group access, and owner access

For each of these bytes, bit 0 is search access (see folders), bit 1 is read access (see files), and bit 2 is write access (make changes). The user summary byte reflects the access that the current user has for that directory; if bit 7 is set, the current user is the owner of the directory.

If the folder is owned by the guest user (usually displayed as "<Any User>"), the owner name will be returned as a null string. If the folder has no group associated with it, the group name will be returned as a null string.

Possible errors include: $4B (bad storage type) if the pathname specifies a file instead of a folder.

SetPrivileges

Command $0005 is the SetPrivileges command. Its parameter list is the same as for the GetPrivileges command except that the access rights, owner name, and group name fields are input instead of output (since the values are being set, not retrieved). The owner name and group name point to structures similar to a GS/OS output buffer where the first word (normally a total buffer length) is ignored, the next word is the string length, and the rest of the buffer is the string itself. This structure definition allows you to do a GetPrivileges call, modify the data, and do a SetPrivileges call using the same owner name and group name pointers (the same way you can share the option_list parameter for Get_file_Info and Set_file_Info).

Setting the owner name to the null string assigns the folder to the guest user (usually known as "<Any User>"). The string "<Any User>" is not a valid user name (unless you have a registered user by that name). Setting the group name to the null string causes no group to be associated with the folder (and therefore the group's access rights are ignored).

Possible errors include:

- $4B (bad storage type) if the pathname specifies a file instead of a folder
- $4E (access denied) if the user is not the current owner of the folder
- $7E (unknown user) if the user name given is not the name of a registered user
- $7F (unknown group) if the group name given is not the name of a group

User Info

Command $0006 is the User Info command. This command will return the user name and primary group name of a user. It has two required parameters and one optional parameter. The first parameter is the device number of a volume on the server whose user info is to be returned. The second parameter is a pointer to a GS/OS output buffer where the user name is returned. The third parameter (optional) is a pointer to a GS/OS output buffer where the user's primary group name is returned.

If the user is logged on as a guest, the user name will be returned as a null string. If the user has no primary group, then it will be returned as a null string.

Copy file

Command $0007 is the Copy file command. This command will cause a file on a server to be copied by the server. The copy may be between different volumes as long as both volumes are on the same server. This call has two required parameters. The first is a pointer to a class 1 string containing the source file's name. The second is a poiner to a class 1 string containing the destination file's name.

Possible errors include:

- $53 (invalid parameter) if either volume is not a server volume or if the volumes are not on the same server
- $4A (version error) if the server does not support this call

GetUserPath

Command $0008 is the GetUserPath command. It returns a pointer to a class 1 string containing the pathname of the user's folder on the user volume, using colons as separators and without a trailing colon. If there is no user volume mounted, or the user name could not be determined for some reason, a data unavailable error is returned ($60). This path is constructed on each call (unlike the fiUserPrefix call). The string's contents will not change until the next call to GetUserPath.

◆ Warning Do not modify the string. The string is suitable for use as a parameter to a SetPrefix call. ◆

As of System Software 6.0d39, GetUserPath checks for the presence of the user's folder. It builds the path internally and then verifies that the folder actually exists and is indeed a folder. If the folder does not exist (or the path exists but is a file), the data unavailable error ($60) is returned.

◆ Note This check does not guarantee you actually have either read or write access to the folder, just that it exists. ◆

This above change means that the behavior of the "@" prefix has changed (since it uses the GetUserPath call). The "@" prefix will now be set to the user's folder only if it actually exists. If it does not exist, the "@" prefix will revert to the folder containing the application (the same as if the application was launched from a non-AppleShare volume, or if there was some other error determining the user path).

As a matter of clarification, the returned string is of the form

:UserVolume:Users:UserName

where *UserVolume* is the name of a volume that is marked as a user volume, and *UserName* is the name of the registered user used when logging on to that server or the string "<Any User>" if they logged on as a guest. If more than one volume is marked as a user volume, the one with the lowest device number is used.

OpenDesktop

Command $0009 is the OpenDesktop command. It takes a volume/path name and returns a desktop refnum (DTRefnum). A desktop refnum must be supplied for all other desktop database calls (currently, only for getting/setting file comments).

CloseDesktop

Command $000A is the CloseDesktop command. It takes a desktop refnum and volume/path name and frees all resources allocated when that refnum was opened.

GetComment

Command $000B is the GetComment command. It takes a DTRefnum and a pathname and returns a string (the comment associated with that file/folder). If no comment has been stored for that file/folder, then a null string will be returned for the comment.

SetComment

Command $000C is the SetComment command. It takes a DTRefnum, a pathname, and a string. If the string is non-null, then the comment for that pathname will be set to the given string. If the string is null, then the comment for that pathname will be removed.

◆ Note If the comment string is longer than 199 characters, it will be truncated to 199 characters without an error. ◆

GetSrvrName

Command $000D is the GetSrvrName command. It takes a pathname and returns the server name and zone name for that volume. If either of the server name or zone name buffer pointers are null ($0000 0000), that string will not be returned. If the server name or zone name are unknown, they will be returned as null strings

GetDefPrivileges

Command $000E is the GetDefPrivileges command. It returns the default access privileges, owner, and group of folders created with the Create command. The format of the parameter list is the same as the GetPrivileges command except that the Pathname field is now used for flags: bits 0,1, and 2 are set if the access rights, owner name, or group name (respectively) will be set with the Create command. The minimum PCount is 3 so that you can determine which fields will be set, but without determining what values they get set to. If you supply a null pointer for either the owner name or group result buffer poiners, that string will not be returned.

SetDefPrivileges

Command $000F is the SetDefPrivileges command. It allows you to set the default access privileges of folders created with the Create command. The format of the parameter list is the same as the SetPrivileges command except that the Pathname field is used for flags: bits 0, 1, and 2 are should be set if you want to set the access rights, owner name, or group name (respectively). The minimum PCount is 3. If the group name is not supplied, then the empty string (i.e. no group) will be used. If the owner name is not supplied, the empty string (i.e. guest) will be used. If null pointers are supplied for the owner name or group name, empty strings will be used (i.e. guest and no group). If the access rights are not supplied, but its bit is set in the flags, you will get an invalid parameter error.

The owner and group names must be less than 32 characters. If the owner name is too long, error $7E (bad user name) will be returned. If the group name is too long, error $7F (bad group name) will be returned.

The owner and group names are GS/OS strings (i.e. leading length word), not the pseudo-result buffer used by SetPrivileges. If you have code that currently uses SetPrivileges and you want it to use SetDefPrivileges, you could just point to two bytes into the pseudo-result buffer (at the actual length word).

General Implementation

When handling file system calls, the FST will itself create and send AFP packets to the server as opposed to trying to make the calls through PFI.

The only syntax checking that will be performed on pathnames is that the span (maximum length of a filename component) is less than or equal to 32 characters (the max for AFP); GS/OS itself will enforce the restriction against colons and nulls in a pathname component.

Normally, pathnames sent to the server will be relative to the root of the volume (i.e. the ancestor ID will be 2 = the volume directory). When a pathname is too long to fit in a packet, the FST will break it up into packet-size chunks by taking as many components from the start of the path as possible, finding its DirID, and repeat as needed until a DirID and partial path is obtained for accessing the file. This will cause more network traffic, but allow for long pathnames.

The FST is responsible for maintaining a file Control Record (FCR) for each open file. It will store at least the following: FCR refnum, pointer to the pathname, ID of owning FST, VCR ID of volume file is on, file level, pointer to newline list, newline count, newline mask, access byte.

The session/volume level information will be obtained from AppleShare device drivers. The .AFPn drivers maintain the relationship between an AppleShare volume and a Device Information Block (DIB). The FST maintains the Volume Control Record (VCR) for AppleShare volumes that are mounted.

If interrupts are disabled when the FST has to make an AppleTalk call (i.e. an SPCommand or SPWrite), an I/O Error (error code $27) will be returned instead of making the call. In most cases, this error will be propagated back to the user. Note that some calls may not require an AppleTalk call to be made (such as GetMark) and will complete correctly with interrupts disabled; some calls (such as Read and Write with small request counts, or GetDirEntry) may or may not complete with interrupts disabled (depending on the current mark, any data that is buffered, etc.).

◆ Note It is strongly encouraged that file system calls should not be made with interrupts disabled! ◆

Appendix A: FST_SPECIFIC Calls

Buffer Control

word Pcount (mininum = 3)
word FST# = $D
word Command = 1
word Reference #
word Buffer Disable flags (default = $0000)

Bit 15 set =		Disable buffering (every read/write goes to server,every GetDirEntry translated into a single FPEnumerate).
Bits 0-14		Reserved

Byte Range Lock

word PCount = 7
word FST# = $D
word Command = 2
word Reference #
word Lock Flag

Bit 15	set	Lock range
	clear	Unlock range
Bit 14	set	Offset relative to EOF
	clear	Offset relative to start of file

long Offset in file
long Length of Range
long Start of Range (returned)

For the Lock Flag, the following constants can be combined:
Lock_Range = $8000
Relative_to_EOF = $4000

Special Open Fork

word Pcount (minimum = 5)
word FST# = $D
word Command = 3
word Reference # (returned)
long Pointer to class 1 pathname
word Access mode

Bit 0	Request Read Access

	Bit 1	Request Write Access
	Bits 2,3	Reserved
	Bit 4	Deny Read to others
	Bit 5	Deny Write to others
	Bits 6..15	Reserved

word Resource number (default =$0000)

Get Privileges

word PCount (min = 4)
word FST# = $D
word Command = 4
long Pointer to class 1 pathname
long Access Rights (returned)

byte	User Summary		
		Bit 0	See Folders allowed
		Bit 1	See files allowed
		Bit 2	Make Changes allowed
		Bits 3..6	Reserved
		Bit 7	Owner (set if you are folder owner)
byte	World		
		Bit 0	See Folders
		Bit 1	See files
		Bit 2	Make Changes
		Bits 3..7	Reserved
byte	Group		
		Bit 0	See Folders
		Bit 1	See files
		Bit 2	Make Changes
		Bits 3..7	Reserved
byte	Owner		
		Bit 0	See Folders
		Bit 1	See files
		Bit 2	Make Changes
		Bits 3..7	Reserved

long Pointer to GS/OS output buffer for Owner Name
long Pointer to GS/OS output buffer for Group Name

Set Privileges

word PCount (min = 4)
word FST# = $D
word Command = 5
long Pointer to class 1 pathname
long Access Rights

byte	Reserved		
byte	World		
		Bit 0	See Folders
		Bit 1	See files
		Bit 2	Make Changes
		Bits 3..7	Reserved
byte	Group		
		Bit 0	See Folders
		Bit 1	See files
		Bit 2	Make Changes
		Bits 3..7	Reserved
byte	Owner		
		Bit 0	See Folders
		Bit 1	See files
		Bit 2	Make Changes
		Bits 3..7	Reserved

long Pointer to buffer where Owner Name is stored (same format as a GS/OS output buffer, but the buffer length word is ignored).
long Pointer to buffer where Group Name is stored (same format as a GS/OS output buffer, but the buffer length word is ignored).

User Info

word PCount (min = 4)
word FST# = $D
word Command = 6
word Device number (of any volume on the desired server)
long Pointer to GS/OS output buffer for User Name
long Pointer to GS/OS output buffer for Primary Group Name

Copy File

word PCount (min = 4)
word FST# = $D
word Command = 7
long Pointer to class 1 string of source pathname
long Pointer to class 1 string of destination pathname

GetUserPath

word PCount (min = 3)
word FST# = $D
word Command = 8
long Pointer to class 1 string containing prefix (returned)

OpenDesktop

word PCount (min = 4)
word FST# = $D
word Command = 9
word Desktop refnum (returned)
long Pointer to class 1 string of path/volume name

CloseDesktop

word PCount (min = 4)
word FST# = $D
word Command = $A
word Desktop refnum
long Pointer to class 1 string of path/volume name

GetComment

word PCount (min = 5)
word FST# = $D
word Command = $B
word Desktop refnum
long Pointer to class 1 string of pathname
long Pointer to class 1 output buffer for comment

SetComment

word PCount (min = 4)
word FST# = $D
word Command = $C
word Desktop refnum
long Pointer to class 1 string of pathname
long Pointer to class 1 string of comment (default = null string)

GetSrvrName

word PCount (min = 4)
word FST# = $D
word Command = $D
long Pointer to class 1 pathname
long Pointer to class 1 output buffer for server name
long Pointer to class 1 output buffer for zone name

GetDefPrivileges

word PCount (min = 3)
word FST# = $D
word Command = $E
long Flags
- Bit 0 Set access rights
- Bit 1 Set owner name
- Bit 2 Set group name

long Access Rights (returned)
- byte Reserved
- byte World
  - Bit 0 See Folders
  - Bit 1 See files
  - Bit 2 Make Changes
  - Bits 3..7 Reserved
- byte Group
  - Bit 0 See Folders
  - Bit 1 See files
  - Bit 2 Make Changes
  - Bits 3..7 Reserved
- byte Owner
  - Bit 0 See Folders
  - Bit 1 See files

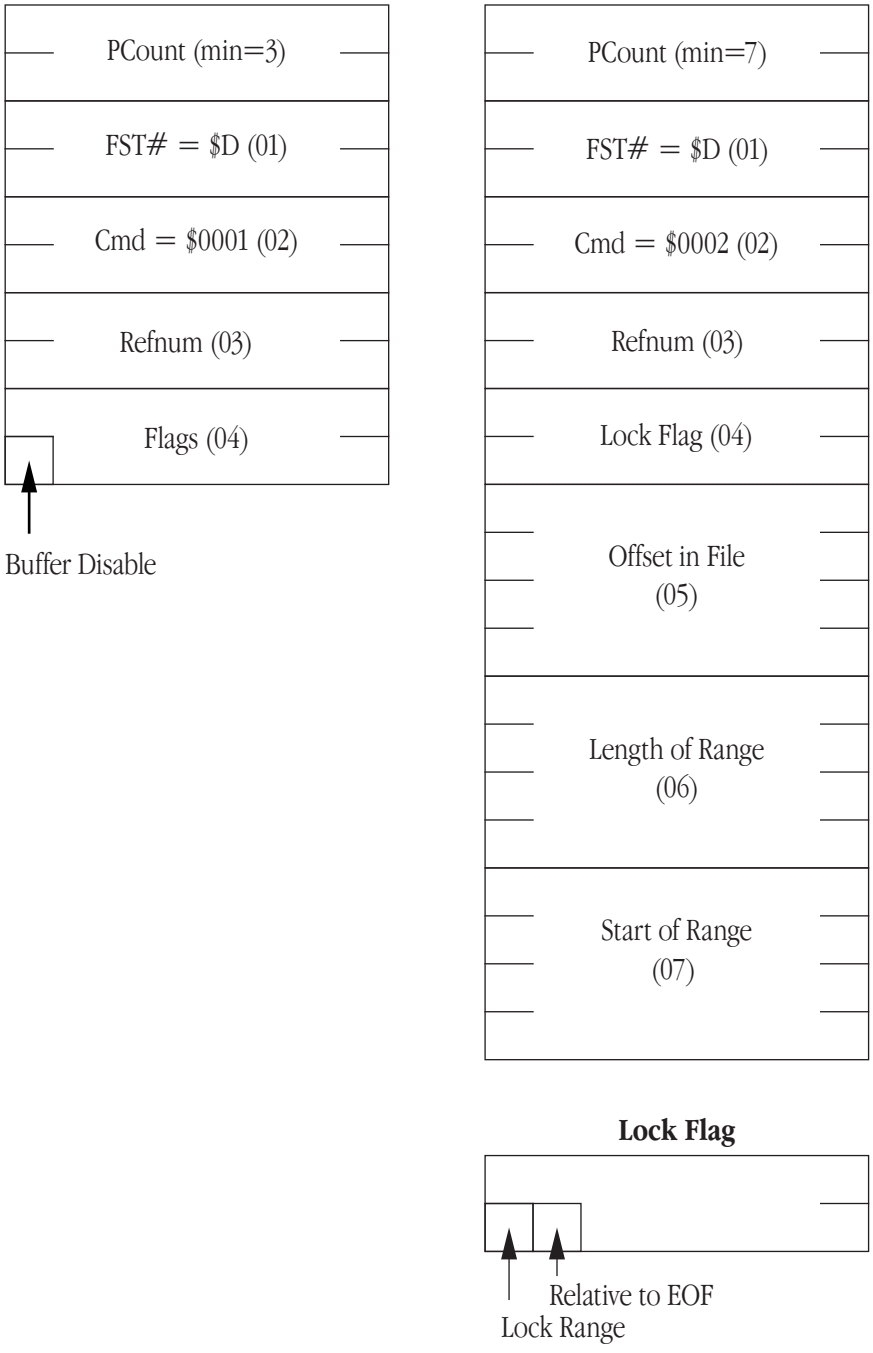
		Bit 2	Make Changes
		Bits 3..7	Reserved
long	Pointer to GS/OS output buffer for Owner Name		
long	Pointer to GS/OS output buffer for Group Name		

SetDefPrivileges

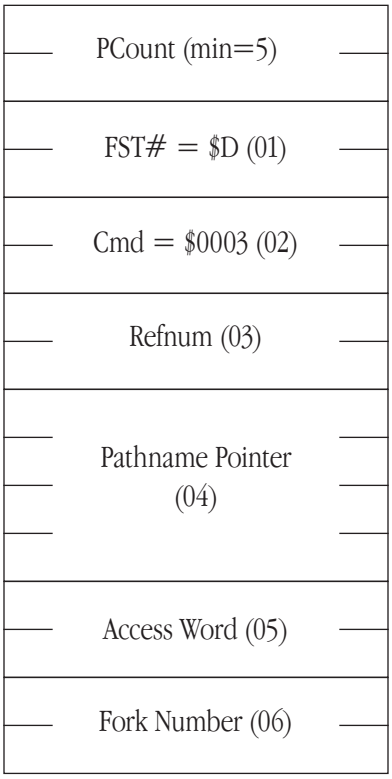
word	PCount (min = 3)		
word	FST# = $D		
word	Command = $F		
long	Flags		
	Bit 0	Set access rights	
	Bit 1	Set owner name	
	Bit 2	Set group name	
long	Access Rights		
	byte	Reserved	
	byte	World	
		Bit 0	See Folders
		Bit 1	See files
		Bit 2	Make Changes
		Bits 3..7	Reserved
	byte	Group	
		Bit 0	See Folders
		Bit 1	See files
		Bit 2	Make Changes
		Bits 3..7	Reserved
	byte	Owner	
		Bit 0	See Folders
		Bit 1	See files
		Bit 2	Make Changes
		Bits 3..7	Reserved
long	Pointer to GS/OS string containing Owner Name.		
long	Pointer to GS/OS string containing Group Name.		

Appendix B – Diagrams

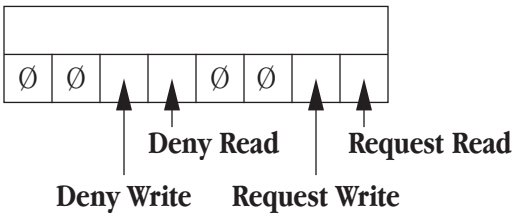
Buffer Control & Byte Range Lock



Special Open Fork



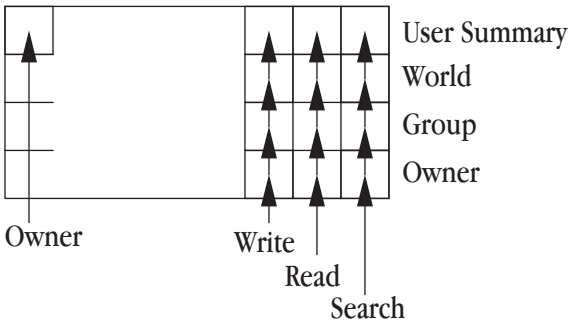
Access Word



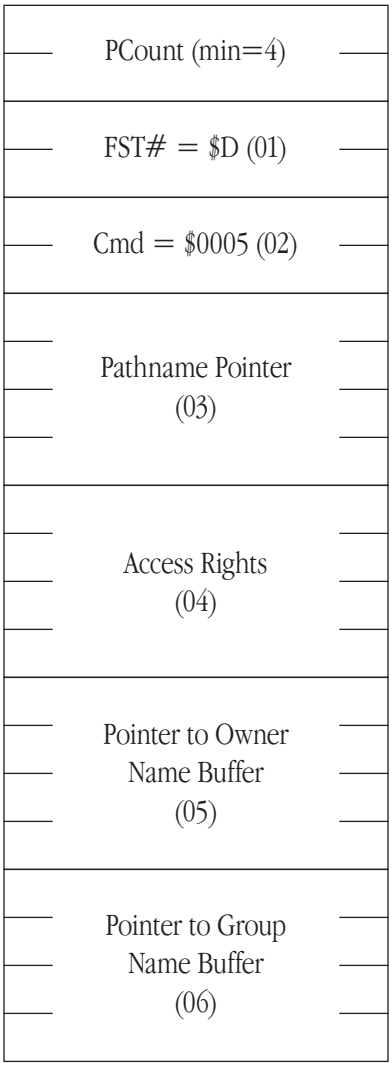
Get Privileges

PCount (min=4)
FST# = $D (01)
Cmd = $0004 (02)
Pathname Pointer (03)
Access Rights (04)
Pointer to Owner Name Buffer (05)
Pointer to Group Name Buffer (06)

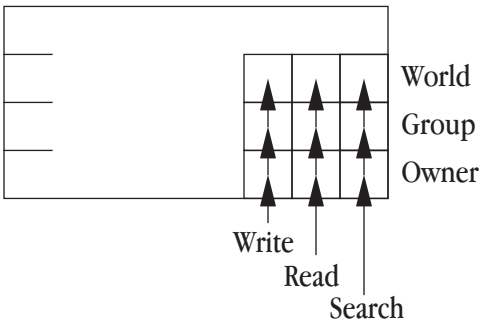
Access Rights



Set Privileges



Access Rights

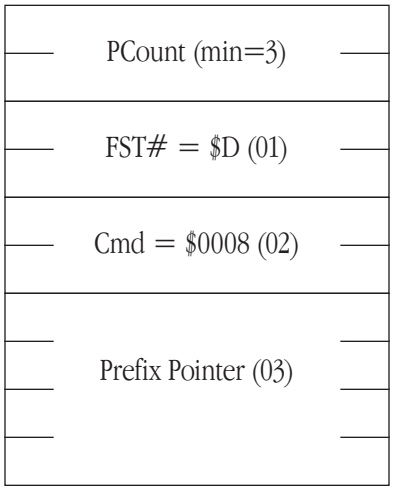


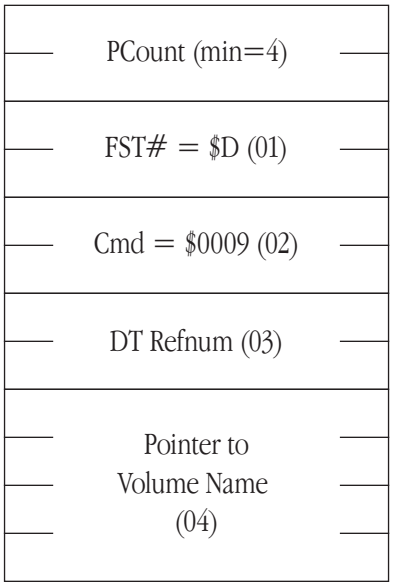
User Info & Copy File

PCount (min=4)
FST# = $D (01)
Cmd = $0006 (02)
Device Number (03)
Pointer to User Name Buffer (04)
Pointer to Pimary Group Name Buffer (05)

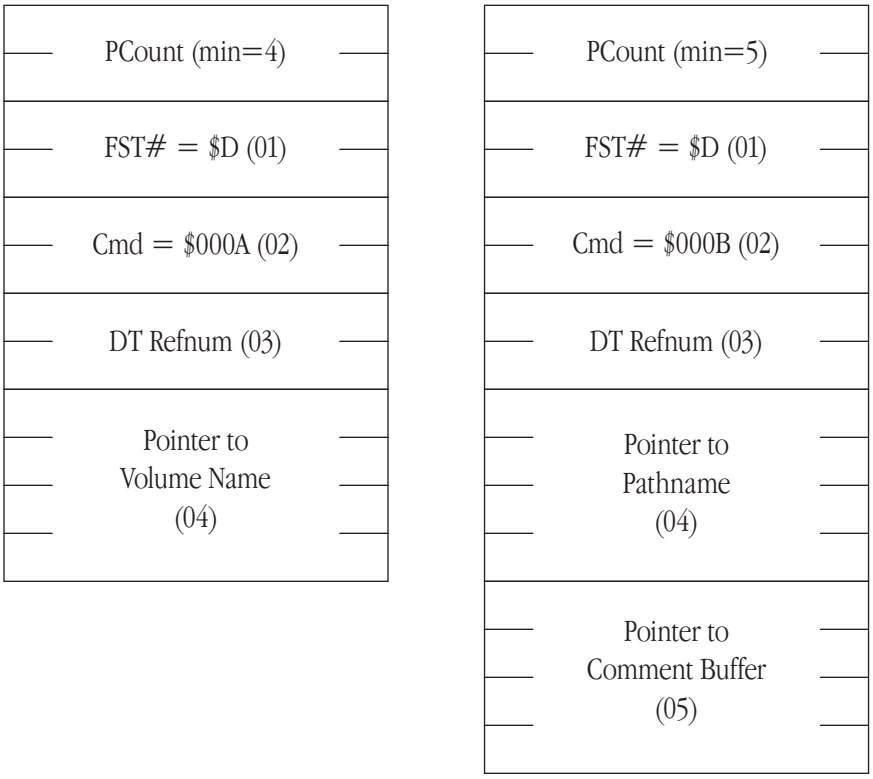
PCount (min=4)
FST# = $D (01)
Cmd = $0007 (02)
Source Pathname Pointer (03)
Destination Pathname Pointer (04)

GetUserPath & OpenDesktop





CloseDesktop & GetComment

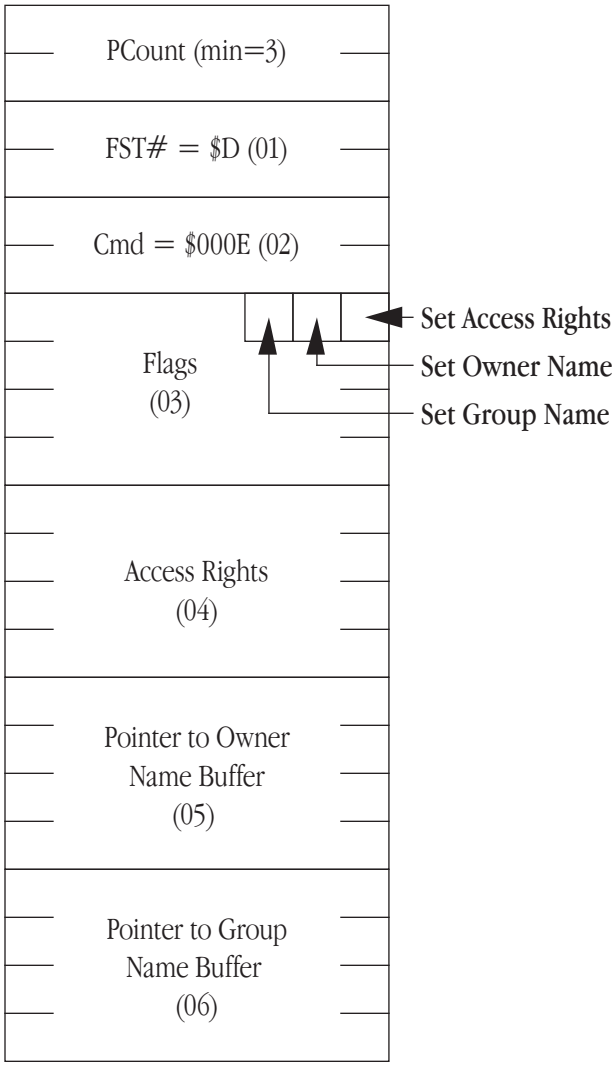


SetComment & GetSrvrName

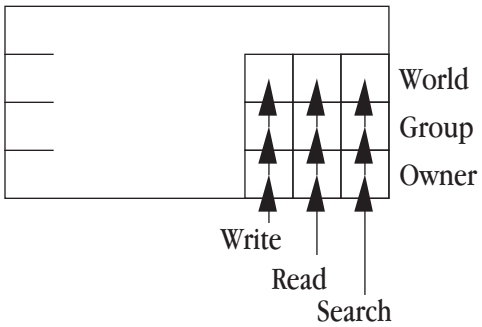
PCount (min=4)
FST# = $D (01)
Cmd = $000C (02)
DT Refnum (03)
Pointer to Pathname (04)
Pointer to Comment (05)

PCount (min=4)
FST# = $D (01)
Cmd = $000D (02)
Pointer to Pathname (03)
Pointer to Server Name Buffer (04)
Pointer to Zone Name Buffer (05)

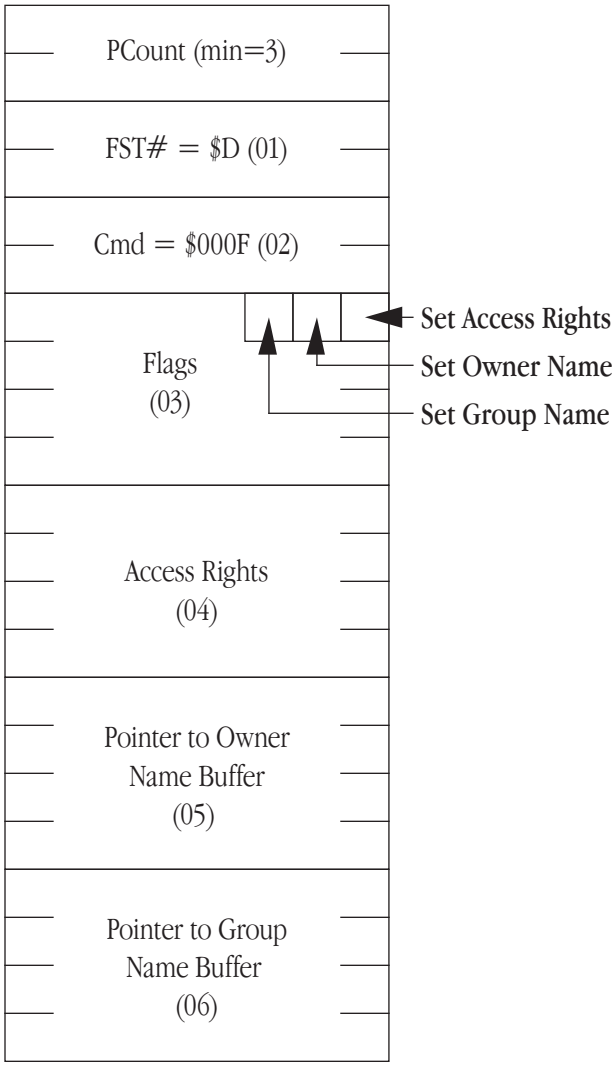
GetDefPrivileges



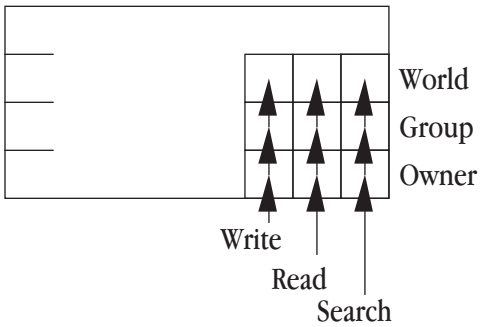
Access Rights



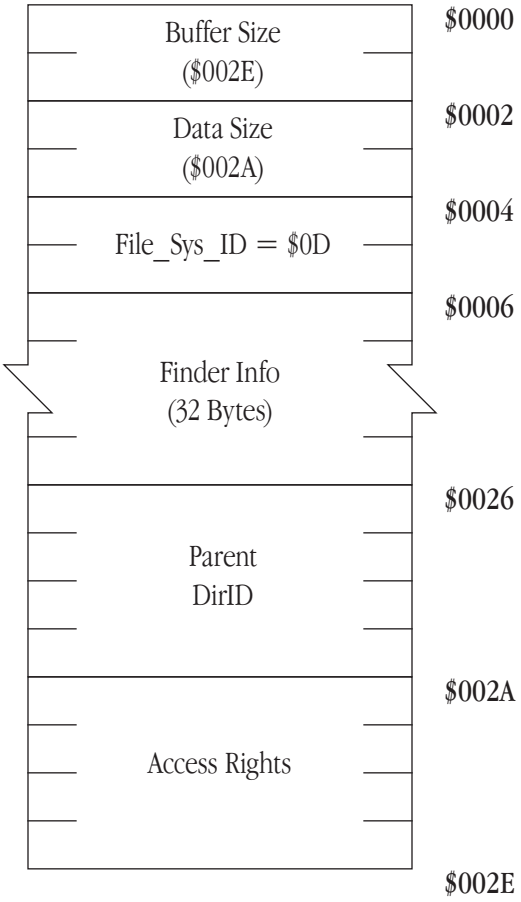
SetDefPrivileges



Access Rights



Option_List



◆ **Note** The bytes in the finder Info are in high-low order (the same as a 68000, opposite that of a 65816). The finder Info is really the concatenation of the finfo and FXInfo structures in HFS on the Macintosh. This is particularly important for the multi-byte fields containing flags and integers -- they are in 68000 format. ◆