PATHWORKS File System

Personal Computer Systems Group

Abstract

Written by: Michael Evans

Issued by: Michael Evans

Reviewed by:

Issue date: August 28, 1992

Revision/Update Information: Version #0.1

digital™

Table of Contents

1 PREFACE

2 REVISION HISTORY

Michael Evans	21-Apr-1992	Initial creation
Michael Evans	3-Aug-1992	Revise structure of document and add design details

3 INTRODUCTION

This specification describes the PATHWORKS File System for Hydra Servers. The specification describes the functional and structural components of the file system. External data structures and prominent internal data structures are defined. The specification provides a functional definition of library interfaces such that additional libraries may be developed and added to the file system.

The specification does not describe the packaging issues or functional delivery issues. Where appropriate, considerations for these issues are noted.

4 TERMS AND SPECIFICATION SYNTAX

Throughout this document certain terms are assumed to be familiar to the reader. Knowledge of the structural components of a file system are assumed. Structural components of the VMS ODS level 2 file system, MSDOS FAT file system and Macintosh HFS file system are provided as a reference to provide context for the functions described in this specification. Readers with detailed knowledge of these file systems will find this information somewhat tedious.

The following terms are used throughout the document and are presented here to eliminate confusion which may arise due to conflicting definitions. These defintions are not intended to be absolute nor are they likely to be precise. However, this is what is meant when then following terms are used.

Attributes	General file attributes including file characteristics (hidden, directory, etc), file times (creation, modification, backup, etc), file size, parent directory, file ID, etc.
Stream	Identification of file data associated with a particular identifier. Macintosh files have two such identifiers, "data" and "resource". All other files have only one such identifier.
Fork	Synonymous with *stream* and may be used interchangeably. Most prominet use is wihtin the context of describing the Macintosh file system
Directory	Structure containing files or other directories
Folder	Synonymous with *directory* and may be used interchangeably. Most prominent use is within the context of describing the Macintosh file system.
File	Data container. This term is used to describe an addressable entity. It generally does not include directories (which are considered part of the address).
Volume	Structure containing directories and files. Represents a collection of directories and files made accessible to the *client*.
Share	Synonymous with volume in the context of NOS structures. Represents the top level directory of a set of files which are made available to *clients*.
Client	Remote operating system or user. Generally used to refer to the orginator of file system requests.
Server	Layer of software above the file system which ececutes operations on behalf of a *client*. May also be used to indicate the entire system which executes operations on behalf of the client. Where confusion may arise the two uses will be noted as *server system* or *server software*.
NOS	Network Operating Sysytem. Generally used to indicate operations or attributes associated with a supported client type, NOS security, NOS file system, NOS user, etc.
Host	System on which server software executes. Refers to characteristics of this system, host security, host file system, host user, etc.
Namespace	Identifies the semantics and syntax of a file specification. Example namepsaces are MSDOS, Macintosh, Unix, VMS, etc.

Path	A file location specification. This may be presented in one of many namespaces.
File System	Defines the semantics for the storage and retrieval of files.
Meta Data	Information about the name, location and attributes of a file.
File Service	A set of routines which define the service offered to clients of one particular type, i.e. MSDOS file service or Macintosh file service.

The following syntax is used throughout the document.

[]	Denotes optional quantities
{}n	Denotes n or more of the term enclosed in quotes

5 REFERENCES

This specification was written using the following reference material:

Inside Macintosh, Volume IV

Inside AppleTalk

VMS File System

DOS 3.1 User Guide

Guide to porting LMU

PATHWORKS for OS2 Administrator's guide

LanManager Programming Handbook

The top level interface is designed to be a super-set of the FSI interface for Microsoft's LanManager for Unix. Routine call semantics are preserved where applicable. New routines have been added to provide a complete set of functions required to support file service access requirements.

6 DESIGN REQUIREMENTS

The following list summarizes the requirements which have influenced this design. Careful consideration has been paid to insure these requirements are met. The design is tightly coupled to these requirements such that a change in requirements could have great impact on the application of the design. It is possible that a change in requirements could cause major redesign efforts.

1. All information relating to a file must be tightly coupled to that file.

   For VMS, this requirement translates into a requirement to store all applicable file meta data in application ACEs attached to the file.

2. Top level interface must support LMU server software without major redesign

   This requirement is driven by an agressive schedule and limited resources.

3. The file system must perform at par or better than existing PATHWORKS products.

   This is a loosely defined metric which needs careful evaluation. Every effort has been made to optimize access given the features of the host file system. There is a tradeoff between file system performance and file system integrity.

4. The file system must operate in a distributed fashion across a VAXCluster.

   This requirement has particular significance to the various cache designs in the file system. All file caches must be distributed. Writeable caches must have data distributed and read only caches must have consistency distributed.

5. NOS security models and host security models are completely independent.

   There will no attempt to provide NOS security in terms of host security. The various security models involved are sufficiently different such that a mapped security model would not yield suitable results.

6. NOS file attributes are completely independent.

   Attributes set from one NOS will not affect attributes of another NOS, even if there is an obvious mapping between them. Currently there are a number of such file attributes, hidden, read-only, creation time and modification time. (While both MSDOS and Macintosh specify a SYSTEM attribute it is not clear a DOS system file is also a Macintosh system file. It is not clear such a file could even be shared between NOS types.)

NOTE

> A file may be readonly to one NOS and writeable to another. File modification times will not be visible between NOS types. This means that a file modified by one NOS may not be seen as modified by another.There is obvious detrimental behaviour as a result of this requirement.
>
> However, backup times represent attributes that although common between NOS types may not be suitable for sharing. Consider a backup utility running on both a Macintosh and a PC client. If backup times were shared neither client would have a complete backup of the file set. It may or may not be adviseable to set up such a backup scenario but the results certainly would not be what was intended.
>
> Restoration of client backups will necessarily destroy the file meta data (and potentially file data) associated with other NOS types. It is therefore not suggested that backups be done thru clients except in single NOS applications.

The host backup facilities should be exclusively used in multiple NOS environments.

7. Data presented to the cache interface must be in stream format.

   This requirement simplifies the data paths in the server. The file system must provide all record deblocking prior to file data being entered in the cache. This requirement PRECLUDES random access to non stream format files as there is no way to map stream offsets to record offsets. Sequential access will be allowed.

   This requirement also precludes output in non-stream format of cached files as there is no mechanism to guarantee the order of writes from the cache. There is no direct mapping between stream offsets and record offsets. Record offsets can only be calculated in sequential write order. Sequential writes to non stream files will be allowed.

NOTE

There is a pending request to allow the server system to create non-stream files. If this request is to be honoured it will require files be written without the assistance of the cache. This will likely be accomplished with the atomic copy function. (This function has the option of bypassing the cache).

There is no current mechanism for specifying the record attributes to attach to a file. All files created on behalf of a client will be created as stream files. We do not have the option of creating a VAR file format for what appears to be an ASCII file (see above description of cache requirements).

7 OVERVIEW

The file system interface for PATHWORKS has been abstracted to provide both NOS independence and PATHWORKS platform independence. PFS (PATHWORKS File System) provides this abstraction.

PFS provides a single interface to multiple host file systems. Functions are provided to access files, create files, delete files, rename files and provide access to information about files. Files may be accessed by the semantics of the supported client using the file syntax of the supported client. PFS provides all translation functions necessary to map the client access to a host file system access.

PFS is a superset of the Microsoft LMU FSI interface. This choice has been made due to a large body of existing code which uses this interface. While non of Microsoft's software is used, in whole or in part, in the implementation of PFS, it is necessary to credit the origin of the interface. An algorithmic view of the implementation of the LMU FSI is provided in Appendix D.

7.1 Functional overview

PFS provides functions to create files, delete files, rename files, store and retrieve file attributes and store and retrieve data associated with the file. PFS provides file access in the semantics and syntax of both MSDOS and Macintosh clients. This access includes file names, path names, data format and file attributes.

PFS operates in one of three namespaces (VMS, MSDOS or Macintosh), all of which are tightly coupled. This means there is a strict relationship between names in various namespaces. While this relationship presents obvious limitations, it allows file system functions be be significantly simplified.

PFS provides completely disjoint sets of file attributes to be associated with files.

PFS will provide functions for storing and retrieving NOS security information, provided the underlying file system is willing to accept the requests. This means that servers need to be prepared to store security data elsewhere if the underlying file system rejects the request (NET.ACC, USERMODE.LMX [is this the correct ACL file for LMX ??], etc).

PFS will provide host security checking provided that the server identifies the host user and all host user access privileges and rights. If this information is not provided PFS will allow access without regard to host security.

PFS supports all native file organizations for read access. Write access will be limited to stream format only. PFS will reject write access requests to non stream files, i.e. the file will not be opened in the hope that writes will not actually occur.

7.2 Summary of functions

The following table summarizes the functions provided by PFS.

Directory access functions

PFS_chdir	- Change default directory using PATHID structure
PFS_diridfcn	- Convert a directory ID to path name
PFS_diridini	- Initialize directory ID handling
PFS_getcwd	- Get the current default directory
PFS_getdents	- Get directory entries in "struct dirent" format
PFS_mkdir	- Create a directory
PFS_rmdir	- Delete a directory

File access functions

PFS_close	- Close a file
PFS_copyfile	- Atomic file copy
PFS_create	- Create a file
PFS_delete	- Move file to purge area or delete it (check attributes)
PFS_open	- Open a file for read and or write access
PFS_purge	- Delete a file
PFS_rename	- Rename file
PFS_truncate	- Trucnate the file
PFS_unmap	- Clean up a memory mapped file

Datapath functions

PFS_fsync	- Flush all written data associated with a file
PFS_lock	- Lock a byte range in a file
PFS_lseek	- Set the current file offset for read/write functions
PFS_read	- Read data from file
PFS_readdesc *	- Read data by reference
PFS_releasedesc *	- Release data descriptors
PFS_unlock	- Unlock byte range
PFS_write	- Write data to file
PFS_writedesc *	- Write data by reference

File attributes functions

PFS_chmod	- Change file protection
PFS_chown	- Change file owner
PFS_getattr	- Get file attributes
PFS_getextattr	- Get extended attributes (not supported)
PFS_getcomment	- Get comment associated with file
PFS_filesize	- Get file size in bytes
PFS_setattr	- Set file attributes
PFS_setcomment	- Set file comment
PFS_setextattr	- Set extended attributes
PFS_stat	- Get file attributes in "struct stat" format
PFS_utime	- Set file access and modification times

Path functions

PFS_dentpathid *	- Convert "struct dirent" format to pathid
PFS_didpathid *	- Trnaslate a default directory plus NOS path into a host path
PFS_getpathid	- Translate a NOS path into a host path
PFS_fullpath	- Get translated host path
PFS_mapname	- Translate a NOS filename to a host filename (name only)
PFS_shortpath	- Get translated path beyond current default directory
PFS_treetop	- Set start of NOS path in translated path

General support functions

PFS_init	- Initialize file system
PFS_mpxclose	- Close host file associated with file descriptor
PFS_needfds	- UNIX only
PFS_needinodes	- UNIX only
PFS_setlognores	- Set routine to call when resources are exhausted
PFS_setnotifympx	- Set routine to call when file multiplexing occurs
PFS_statvfs	- UNIX only
PFS_sync	- Flush all written data associated with all files

Security functions

PFS_access	- Check host access to a file
PFS_faccess *	- Check host access to an open file
PFS_getsecurity *	- Retrieve NOS security data
PFS_getuser *	- Return PFS_USER structure for specified host user
PFS_putsecurity *	- Store NOS security data

7.3 File service components

A file service may be defined by four major components; namespace, attributes, security and data paths. These components allow a service to offer files stored in its native file system to a client using another file system.To implement a file service a mapping between components is necessary.

There are number of components which comprise a file system. Some of these components are more visible to a file service than others. Some file system components will define file service components while others may only have incidental effects. The following table shows how file system components are mapped to file service components.

File System Component	File Service Component
File name syntax	Namespace
Directory structure	Namespace
File allocation	Data paths
File attributes	Attributes
Security	Security
File meta data	Attributes, Security, Namespacd
Quotas	Data paths

The overall effectiveness of a file service will depend on how complete the mapping between components can be defined. In general, the more robust the host file system, the more effective the file service will be.

The mapping of file service components to the VMS file system is defined in section 9.

7.3.1 Namespace

Namspace defines the file name syntax, path syntax and path semantics. A given file service may need to access files by name, by ID or possibly other mechanisms. These mechanisms need to be mapped to the supporting file system.

File name syntax varies amoung file services. Filenames range in length from 11 characters to 255 characters and consist of character sets ranging from alphanumeric to virtually unlimited. This range presents a challenge to a file service which must either be able to map names or limit the range to a more manageable set. Any limits imposed will be visible to the client

Path syntax and semantics also vary amoung file services. Path lengths may range from 1 member to virtually unlimited. File systems may impose limits on the number of members in a path and this will be visible to a client. Path semantics may be biased, relative or absolute.

Absolute path semantics specify each member of the path using a name appropriate to the file service. This name may be a character set name or an ID.

Relative path semantics specify members relative to prior members. There may be "special" names assigned to "parent" members (i.e. VMS path [-]) or parent members may be specified by the absense of a named member (i.e. Macintosh <null>).

Biased path semantics present a base path and a reative path. The base path may be a named path or an ID (i.e. VMS rooted path [member.][member] or Macintosh ID plus named path).

NOTE

The VMS file system is among the most restrictive with respect to namepsace. The filename character set is the most restrictive as is the effective path member length. VMS does allow filenames greater in length than both MSDOS and Macintosh but this is largely negated by character set limitations.

The following table sumarizes the namespace characteristics of various clients and the VMS file system.

Characteristic	MSDOS	Macintosh	VMS
Filename length	8 with 3 ch extension	31	39 with 39 ch extension
Character set	Any greater than <space> and less than <del> with the exception of % and *	Any except <null> and :	Alphanumeric or $, _, -
Path length	Unlimited depth [Character limit ??]	Unlimited depth [Character limit ??]	8 member depth* 255 character length
Path semantics	Absolute	Absolute, biased by ID, or relative	Absolute, biased by name, or relative
Access by:	Name or 16 bit ID**	Name or 32 bit ID***	Name or 48 bit ID
File limits	[FAT limit ??]	2**32	2**24 per volume 2**32 per bound set
Directory limits	2**16	2**32	2**24

* VMS provides a mechanism to "fix" a bias (concealed logical name) such that unlimited path lengths may be accessed via 8 member relative paths. This mechanism is not supported across all applications, most notibly BACKUP. The VMS path length limit applies to applications which use RMS only as there is no limit to the depth of a path processed directly via QIO.

** MSDOS provides "fixed" directory offsets and functions may reference files via this fixed offset. This in effect becomes a 16 bit ID which may reference the file within the context of a path.

*** Macintosh assigns a 32 bit ID to each file and directory created on a volume. This number is unique and will not be reused when a file or directory is deleted. The number bears to special relationship to the file and may be swapped between files (a numeric rename function). This number is used to establish links between files declared as "alias".

PFS makes no assumptions about the mapping functions associated with a file system. The mapping of client namepsace to file system namespace is entirely defined by the file system library. PFS uses file system library functions to map client names and IDs. PFS provides interface routines to translate biased paths into absolute paths.

7.3.2 Attributes

File attributes are maintained by the file service to provide information to clients about the files to which the attributes apply. Clients have their own set of attributes which they use for various purposes as does the host file system. Mapping these attributes to file system attributes is generally not complete.

File service attributes include information about when a file was created, last modified, last accessed or backed up. File characteristics such as whether the file is a directory, visible, archived, copy protected and so on are maintained by the file service. File data formats are set when the file is created or modified and are made availabe to the file service.

The following table shows the various atributes associated with the MSDOS, Macintosh and VMS file systems.

Attribute	MSDOS	Macintosh	VMS
Create time	Modified time	Create time	Create time
Modified time	Modified time	Modified time	Revised time
Access time	Modified time	Modified time	Revised time
Backup time	N/A	Backup time	Backup time
Directory	Directory	Implied	FCH$V_DIRECTORY
Archive	Archive	N/A	N/A
Visible	Hidden	Invisible	N/A
System	System	System	N/A
Backup Needed	N/A	Backup Needed	Backup time
Rename Inhibit	N/A	Rename Inhibit	Write access to directory
Delete Inhibit	N/A	Delete Inhibit	Delete access to file
Multiple User	N/A	Multi User	N/A
Write Inhibit	Read Only	Write Inhibit	Write access to file
Copy Protect	N/A	Copy Protect	N/A
Volume ID	Volume	N/A	N/A
Finder Information	N/A	Finder Info	N/A

As can be seen from the above table, mapping client attributes to file system attributes will not provide a sufficeint mapping. It is clear some form of storage and retrieval of attributes must be provided by the underlying file system. The complex challenge to a file system is how to reflect the client attributes in terms of file system characteristics in a "least surprise" fashion.

PFS makes no assumptions about the mapping between client attributes and host file system attributes. The underlying file system may be as complete or incomplete as necessary. PFS will pass the limitations on to the server which will necessarily make the limitations visible to the client.

PFS will honor the following attributes:

Read Only	PFS will not allow writes to the file
Delete Inhibit	PFS will not allow the file to be deleted
Rename Inhibit	PFS will not allow the file to be renamed
Copy Protect	PFS will not allow the atomic copy to be used on the file. However, there is no mechanism to prevent an application from copying the file by direct open and read.
Directory	PFS will not allow direct access to the file

The remaining attributes are stored and retrieved to support server functions. It is up to the server to apply these attributes to file service functions.

7.3.3 Security

Client security models vary greatly. There is so much disparity between sercurity models that any attempted mapping would compromise all models. Given this, client security must be implemented independently of the native file system security model. However, native file system access may still be restricted by the underlying native security model. This dual model provides for client security models to be implemented at the expense of additional system management to establish the relationship between client users and host system users.

PFS provides access to secuirty data stored by servers but does not interpret the data in any way. The services are provided to associate client security data with the objects to which they apply.

PFS will pass host user identification information to the file system library. The file system library may choose to use this information as necessary. It is the responsibility of the server to establish the relationship between the client user and the host user.

All file systems are expected to keep track of which files they create on behalf of a server. It is necessary for a file system to be able to distinguish between files it has created and those which were created outside the server such that a hybrid security model may be implemented. This model will check host security only if the file was not created by the server.

PFS will pass the current security mode to the file system library. It is the responsibility of the server to determine the security mode for the path. File system libraries are expected to be capable of dealing with the following three security modes:

Always	Check native security on all accesses
Foreign	Check native security only if the file was not created on behalf of the server.
Never	Ignore all native security

These file system security modes correspond to the sever security modes HOST, CREATOR and NOS.

7.3.4 Data Paths

File services generally deal with stream file formats but there is no assumption about the data formats of the underlying file systems. For this reason there may be a conversion required between native file formats and file service formats.

PFS provides for this conversion by allowing a file system to "claim" a data path. This claim function is more restrictive than the path claim function in that the claim is applied to the path file system only. This partitioning allows PFS to obtain the path owner, get the file characteristics and then ask the file system to claim the data path give the characterisitcs. This parititoning allows file systems to claim paths without necessariliy obtaining file characterisitics (which may only be imporatant if the file is actually to be accessed).

All file data which passes thru the data cache must be converted to stream format. It is the responisbility of the underlying file system to supply this conversion. The file system will be supplied context infomation to support this translation. The information is maintained completly by the file system, i.e. it is in no way interpretted outside the file system.

It is possible that a file system may implement a different set of functions to deal with files of various organization and record format. These functions will be established by the FSLIB_claim_datapath() function. This routine is called at file open time.

NOTE

There are a number of implemenation options around datapaths. These options are briefly described below. For the purposes of this specification option 2 listed below will be assumed.

1. Implement one set of datapath functions and dispatch the appropriate routines based on record format.

   This option allows one set of vectors to be referenced and reduces the data storage required for the PFS_FID stucture. The tradeoff here is that record formats may need to be checked on every access and an additional call is placed in the data path.

2. Implement a unique set of function vectors for every combination of record formats supported.

   This option allows vectors to be referenced at the expense one copy per file format. This is most likely the best compromise as file systems which only support one file format need to do nothing special. File systems which support multiple record formats need to create one set of function vectors for each supported record format.

3. Copy the function vectors to the PFS_FID structure and allow the FSLIB_claim_datapth() function to modify the copy.

   This option would support the most number of combinations in the simplest fashion as only a few vectors need to be modified.

7.4 Structural overview

PFS is partitioned into two major components, PATHWORKS File Interface and File System Library (FSLIB).

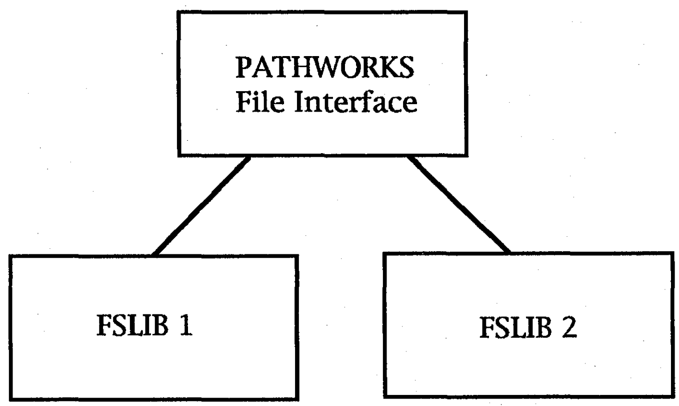


Figure 1. File system top level structure

7.4.1 Top level interface

The top level interface provides argument checking and dispatch functions to the appropriate FSLIB. The FSLIB is selected during PFS_getpathid() by calling each FSLIB's FSLIB_claim() routine. FSLIB_cliam() will determine if this file system owns this path and if so, will supply a set of vectors to handle all remaining FSLIB functions.

All FSLIBs are expected to handle ALL functions, even if the action is simply to return success or failure. The top level routines DO NOT check the validity of a vector prior to dispatch.

7.4.2 File System Library Interface (FSLIB)

A File System Library (FSLIB) is a collection of routines which implement the file system functions necessary to support PFS functions. There is no formal definition of file system such that a clear set of rules may be established on what is and what is not a file system. Suffce it to say that if a set of routines is prepared to handle file system functions, PFS will be prepared to use them.

An FSLIB may support "variant" file systems within it. Each variant is treated as an independent file system and only shares the FSLIB_init routine with its other variants. This mechanism allows multiple collections of routines to be grouped within a file system library. It should be stressed that each variant is treated separately and must be capable of identifying the paths on which it will operate without regard to its other variants.

The FSLIB functions are roughly parallel to the PFS functions, i.e. PFS does very little except find the appropriate FSLIB for a given path and dispatch FSLIB functions to handle PFS functions.

PFS requires that a given path resolve to at least one FSLIB. There is no implied hierarchy in a set of FSLIBs nor may any one library expect it being asked to handle a path given anopther has rejected the path. Each FSLIB must be capable of identifying paths which belong to it, independent of decisions made by another FSLIB. This is crucial given there is no implied order in sequencing path ownership functions.

7.4.2.1 FSLIB Path Claim

Each path on which PFS must operate must be claimed by at least one file system. There is a possibility that multiple file systems may handle a given path and if so, the first to claim it will be given the opportunity to service it. Once a path is claimed the FSLIB will be responsible for handling all subsequent operations on that path.

The FSLIB is given a "pseudo" file system path defining the "root" of the path. In most cases this path is sufficient to determine ownership. Path ownership may be a function of volume ACP or it may be a function of path component format (i.e. container files). The format of the "pseudo" path is given below.

device:{[directory_spec{.@container_file}]}

where device is a physical device name, directory_spec is a VMS hierarchial directory spec (of the form [directory{.directory}]) and @container_file is the name of a foreign file system container file.

NOTE

Foreign container files are currently limited to support of MSDOS FAT file system. However it is concievable that additional foreign file system container files nay be added. If this is the case additional work will be required to identify them without resorting to opening the file and scanning the format.

The FSLIB is also given the client path and client namespace identifier. This information may be used to select a variant file system within an FSLIB.

7.4.2.2 FSLIB Initialization

Each FSLIB is called at its initialization entry point during PFS initialization. The library should set up all data structures required to handle subsequent function requests. This routine will be called only once at system startup time.

The FSLIB is required to initialize a PFS_LIB_ENT structure at this time. The structure contains the name of the FSLIB, its characteristics and its function dispatch vectors.

PFS will locate the initialization routine by UNIVERSAL SYMBOL name. This routine must be globally defined in the library and MUST be of the form XXX_init, where XXX is the name of the FSLIB. This is the ONLY symbol in the FSLIB which is referenced by name. All other functions will be referenced by entries in the function dispatch vector.

The initialization routine will be called multiple times allowing the library to establish variant function dispatch vectors.This feature allows libraries to implement separate functions for handling various client anomolies if necessary. A library is not required to handle various clients in any particular fashion and variants are strictly optional.

7.4.3 Data cache interface

PFS vectors all file read and write requests thru the data cache with the exception of atomic file copy operations. The data cache is given a set of read and write file functions which it will use to fill the cache and perform writebacks.

The file read and write functions are established when the path in which the file resides is claimed. These functions are given to the data cache manager when the file is opened. It is important to note that file structure will affect the read and write functions and this information must be known when the path is claimed.

The interface to the cache is strictly read by reference. A list of data buffer descriptors is passed between data cache requests as well as being passed to file read and write functions. PFS will make these descriptors lists available to servers theu PFS_readdesc and PFS_writedesc functions.

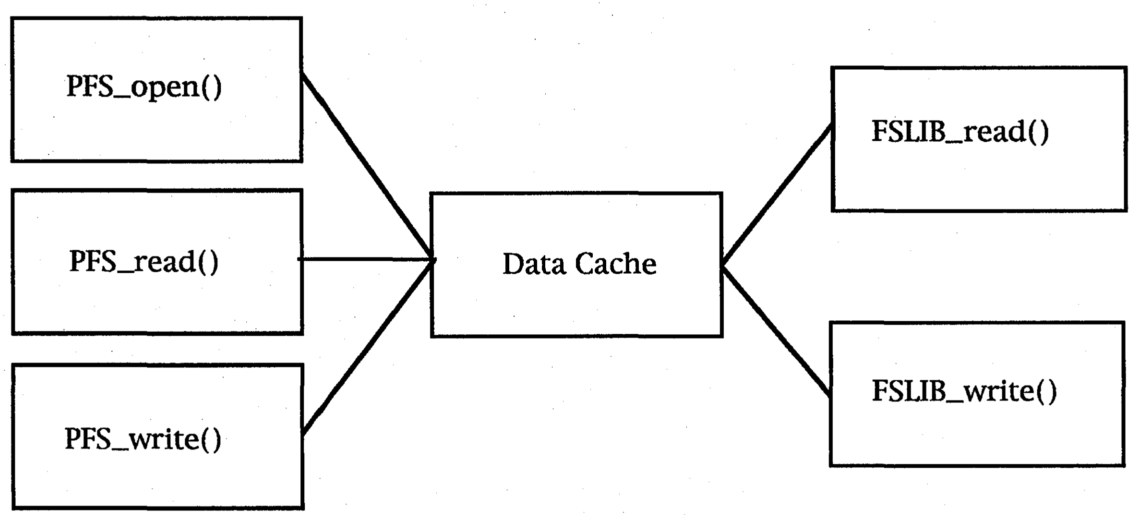


Figure 2. Data cache top level and FSLIB interface

7.4.5 Open file cache

PFS maintains a cache of most recently opened files. The primary purpose of this cache is to keep information which will allow an MSDOS batch file ot be quickly reopenned. The overall effectiveness of this cache is questionable as the server must translate the filename prior to finding it in the cache. This operation represents the vast majority of time spent in opening a file. However, file close is eliminated and this does represent a substantial savings.

While the primary purpose is specific the cache has application in other areas. The cache may be used to permanently open a file effectively "installing" it. Sharing options and file locking will affect the overall effectiveness of this application.

PFS will periodically close files in the open file cache as specified in a configuration parameter. This will simply represent a "delayed" close of the file. Files may have different close intervals. PFS will provide a function to set the close interval on an open file. This mechanism may be used by system management software to open and specify an infinite close interval, thereby "installing" an image.

NOTE

The assignment of tags to use in the location of entries is an open issue. PFS must be able to generate a global unique identifier for the file without actually opening it. As stated above, this represents the vast majority of work required to open a file and may potentially result in accessing a file on one channel merely to close it and use the already open channel.

Client filenames may not be used as a global tag without resolution of relative file paths. This is particularly true in the Macintosh namespace. This operation may result in accessing the file to resolve namespace translations.

It is clear more work needs to be done in the architecture of this cache with respect to the structure of PFS.

7.5 Data structures

PFS defines a number of data structures which are used to pass information between PFS and servers. These structures maintain "cached" information to eliminate redundant file system functions. This mechanism needs careful review when applied to a distributed file system. Certainly the possibility exists that this information could change without the accessor's knowledge resulting in use of stale data. In many cases this does not present a significant problem as there is sufficient ambiguity in the order of operations in a distributed file system. There is no interlock mechanism for modification of file meta data. To eliminate possible read-modify-write scenarios all data structres which hold modified data also hold a mask indicating which data is modified. This mask may be used by file system libraries to limit writeback modifications.

7.5.1 PFS_PATHID

The PFS_PATHID structure is used to hold mapped path information. This structure is returned by PFS_getpathid() and is used as a file specification for all PFS access functions. Servers must obtain a PFS_PATHID structure for a given path prior to any file service functions which are expected to deal with this path.

The structure has the following fields:

funcptrs	Pointer to the file system library dispatch vector in the PFS_LIB_ENT structure for the file system. This pointer is used to locate all file system functions.
fullpath	Resolved native file specification. This buffer holds the name of the host file or path which maps to the specified client path.
shortpath	Pointer to the start of the fullpath which needs further resolution. This filed is not currently used or suported by PFS. This field is used to optimize Unix access functions.
endtreetop	Pointer to the start of the fullpath which maps to client path, i.e. the point beyond the share directory or volume directory. PFS does not use this field. It is present for server use only. It is not suggested that new server software use this field.
fsflags	Pointer to file system characteristic flags in the PFS_LIB_ENT structure. This pointer is used by PFS to determine if a file system supports various features.

status	PFS_STAT structure holding file characteristics, location information and various other file system specific information. The information in this structure is known to and used by PFS. It is maintained by the file system library.
diridptr	Pointer to next directory ID to be assigned on directory creates. This field is not used by PFS and is not supported. This field represents a shared partitioning of assignment of directory IDs between the file system and the server. This partiton does not exist between PFS and associated servers.
client_name	This is a holding buffer for file system lihrary mapping functions. This buffer is used to pass information between PFS and file system libraries.
namespace	Identifies the namepsace in which this path is operating.
cp	Claim parameter. FSLIB_claim functions are allowed to return a longword parameter. This parameter is stored here and is made accessible to file system library functions thru this offset. No assumptions are made about the contents of this longword.

7.5.2 PFS_FID

The PFS_FID structure represents an open file. This structure is returned by PFS_open() and PFS_create(). This file identifier is necessary for all data path operations in PFS and is used for some of the file attributes functions as well. Any file opened by PFS will have an associated PFS_FID structure.

The PFS_FID structure contains the following fields:

forw	Forward link pointer. This field is not currently used by PFS.
back	Back link pointer. This field is not currently used by PFS.
funcptrs	Pointer to file system library dispatch vector. This pointer may be a pointer to the PFS_LIB_ENT vector or may be a specific vector based on file format.
status	PFS_STAT structure containing information about the file.
fd	Open file descriptor. This longword is reserved for use by file system libraries. The contents of this longword are not interpretted by PFS.
fdinfo	Information about the open file descriptor. This longword is reserved for use by file system libraries as they deem necessary. No assumptions are made about its contents by PFS.
stream	Data stream identifier. May be PFS_PRIMARY or PFS_RESOURCE representing the data or resource streams of a file.
offset	Current stream offset in file.
refcnt	Number of servers which are referencing this shared file.
nompx	Counter to inhibit multiplex closing of this file. This field is not used by PFS.

oflag	Copy of open mode passed to PFS_open() or PFS_create(). This field is used to support file multiplexing and as such is not used by PFS.
flags.mandlock	File has manditory locking set.
flags.dirty	File has been modified.
mapaddr	File memory map address. This feature is not supported by PFS.
maplen	Length of memory map. See above.
fsflags	Pointer to file system library flags in PFS_LIB_ENT structure for the file system which claimed this file.
endtreetop	This field is not used by PFS.
namespace	Namespace in which this file was opened.
cp	Claim parameter. This parameter is copied from the associated PFS_PATHID structure and is made available to file system libraries thru this offset.
fullpath	Copy of the PFS_PATHID fullpath buffer.

7.5.3 PFS_ATTR

The PFS_ATTR structure is used to store and retrieve file service attributes. The attributes structure has a mask associated with it which specifies which fields are to be read and written. This "bit set" model solves the problem of shared file access with "cached" data in the structure.

This structure has been modified to better support multiple file systems and platform independence. Additional fields have been added to represent attributes associated with supported clients. Where possible the structure has been modified in an upward compatible manner. Time field format changes are inevitably not upward compatible.

The fields of the PFS_ATTR structure are described below:

mask	Bit mask indicating the validity of each field in the structure. This mask specifies which fields are to be modified on a get operation and which fields are to be written on a set function.
dirid	Directory ID associated with the parent directory of the file being referenced. If no file is referenced (path only) this field is the same as the fileid field). This field carries the 32 bit Macintosh directory ID. This field may or may not have significance for other file services.
btime	Backup time in namespace format.
ctime	Create time in backup time format.
finder_info	32 bytes of information associated with the Macintosh Finder. This filed has no meaning for non Macintosh clients.
attr_bits.archive	File has been archived.
attr_bits.hidden	File is not visible to directory list operations.
attr_bits.issystem	File is a system file.
attr_bits.no_rename	File can not be renamed. See PFS_rename().

attr_bits.no_delete	File can not be deleted. See PFS_delete().
attr_bits.no_copy	File can be copied with atomic copy function. See PFS_copyfile().
attr_bits.read_only	File can not be written. See PFS_open().
attr_bits.mac_appl	File is a Macintosh application.
attr_bits.multi_user	File can be open by multiple readers. This bit only has significance if the mac_appl bit is also set.
attr_bits.no_purge	File can not be purged. See PFS_purge().
attr_bits.exec_only	File can only be open for execute access. It is not clear how this bit can be honoured but it is here just in case.
fileid	File ID. This field caries the 32 bit Macintosh file ID. It may or may not be applicable to other file services.
mtime	File modification time in namespace format.
file_count	Number of files contained in a directory. This count may be the total number of files or just the number of non directory files. The total number of files in a directory will always be represented as file_count + directory_count.
directory_count	Number of directories contained in a directory. File systems may report this value as 0 and return all files in file_count. This field is defined to support Macintosh security concerns.

7.5.4 PFS_STAT

This structure is a collection of Unix file attributes and has very little application to non Unix systems. It is highly questionable whther this information should be exported to servers. However, there is application within the file system. Location information may be saved such that additional file access may be eliminated when multiple references are made to a file.

NOTE

This structure has been modified to support the VMS ODS-2 file system. This is clearly a file system issue and should be defined elsewhere. It is likely more appropriate to keep this information in an opaque data structure managed by the file system library. To do so would require a maximum size be established for the structure such that PFS may continue to manage the allocation of the structres which contain this structure. This issue may be addressed in future developments of PFS.

The fields of the PFS_STAT structure are as follows:

mask	Mask indicating the validity of members of this structure
stat	Contains a "struct stat" structure defining various low level file attributes.
gen	File generation number. This field is maintained by the file system library and may or may not be supported.
stream	Data stream associated with this file. This field may have significance in file systems which implement serparate files for each data stream

supported. This field is maintained by the file system library and may or may not be supported.

p_ino	Parent INODE. This is a Unix concept and is only supported by Unix file system libraries.
p_gen	Parent generation number. This field is maintained by the file system library and may or may not be supported.
count	Number of files contained in a directory. This field is intended for export to server to support the Macintosh offspring count. This field is more appropriate in the PFS_ATTR strcture and has been defined there. This filed may or may not be maintained in parallel with the PFS_ATTR field.
dir_cnt	Directory offspring. This field is more appropriate in the PFS_ATTR structure and has been moved there. This field may or may not be maintained in parallel with the PFS_ATTR field.
file_cnt	File count. See above disclaimer.
attr	PFS_ATTR structure.
did	ODS-2 directory ID. This field is maintained by the ODS2 file system library. It is not intended for export.
fid	ODS-2 file ID. This field is maintained by the ODS2 file system library. It is not intended for export.
dev	ODS-2 volume name. This field is maintained by the ODS2 file system library. It is no intended for export.

As can be seen from the above descriptions, this structure is of little value outside PFS with the exception of the PFS_ATTR structure. The structure should be redefined to attempt to merge members which are relavent to a particular file system. This may be done in future developments of PFS.

7.5.5 PFS_NAMEID

The PFS_NAMEID structure is used by PFS_parse() to store information about components of a pathname. The structure has fields defined for components of various namespaces. This structure only deals with named paths and does not carry any translation information. It is used to provide common server parse function support.

The fields of the PFS_NAMEID structure are as follows:

dev	Pointer to device name in path buffer. If no device is present or if the namespace does not support device names the field will be NULL.
devlen	Length of the device string. The length includes the trailing device delimiter.
dir	Pointer to start of directory component. This will generally be the start of the path for namespaces which do not support devices. The pointer includes the leading directory delimiter.
dirlen	Length of the dirctory string. The length includes the trailing directory delimiter.
filename	Pointer to start of filename component.

filenamelen	Length of the filename string.
ext	Pointer to start of extension component. The pointer includes the leading extension deliminter.
extlen	Length of the extension string.
parent	Pointer to start of the path.
parentlen	Length of the parent string.
path	Buffer containing the full path string. This buffer is included in the structure such that the structure may be passed without requiring translation of pointers.

7.5.6 PFS_CWD

The PFS_CWD structure holds the data associated with the current working directory.

The fields of the PFS_CWD structure are as follows:

path	Buffer containing the full path string.
directoryID	Host directory ID (if needed).
funcptrs	Function pointers for file system in which default exists.

7.5.7 PFS_USER

The PFS_USER structure holds the host user identification, privileges and rights for the mapped host user. This structure is used to represent the client for various security related functions.

The fields of the PFS_USER structure are as follows:

uid	Host user identification in uid_t format.
privs	Quadword privilege mask.
rightlen	Length of rights list
rights	Pointer to rights list. This list consists of a set of two longword pairs. The first longword is the right identifier and the second is the rights attributes.

7.5.6 PFS_LIB_ENT

The PFS_LIB_ENT structure holds the data associated with a file system library.

The fields of the PFS_LIB_ENT structure are as follows:

fstype	Pointer to the name of the library.
funcptrs	Pointer to library dispatch vectors.
pfsflags.unixfs	Indicates unix file system.
pfsflags.resource	File system support resource forks.

pfsflags.extattrs	File system support extended attributes.
pfsflags.cscreate	File system supports case sensitive file names.
pfsflags.mappedfs	File system is mapped to another file system.
pfsflags.security	File system supports security data.
pfsflags.statmask	PFS_STAT elements supported.
pfsflags.attrmask	PFS_ATTR elements supported.

7.5.9 PFS_IDENT

The PFS_IDENT structure holds the data associated with a print file. While this information is presented in VMS format it is expected that this structure will be modified to suit other platforms.

The fields of the PFS_IDENT structure are as follows:

count	Length of device name. Limit of 15 bytes.
device	Device name string. This string is not counted and is limited to 15 bytes.
fid	6 byte file identification.
did	6 byte directory identification.

7.5.10 Stat structure

The stat structure is a Unix concept which is ported to various platforms for compatibility. The members of the structure may not have the same format nor the same implied function. The effectiveness of this structure outside of PFS is questionable.

NOTE

> Currently the device name and file INODE are used to identify a file. The Unix device identifier is 32 bits in length and assumed to uniquely identify a device. VMS has no such concept. Currently the device member is defined as a 16 character array. This is a good application for a nameservice.
>
> The INODE is also 32 bits and uniquely identifies a file within the Unix file system. Again VMS has no concept of a homogenous file system and assigns 48 bit file IDs relative to volumes. This means that to uniquely identify a file on VMS requires 176 bits. Is is not clear what the implications for other file systems may be.

The fields of the stat structure are as follows:

st_dev	Longword device identifier. This is currently defined as a pointer to a 16 byte structure. Is is not clear where that structure would be located.
st_ino	32 bit file identifier. This field is maintained by the file system library and is NOT unique across devices.
st_mode	Unix file format. The only bit of particular significance is the S_IFDIR bit which indicates the file is a directory. PFS uses this bit to determine if a file is a directory and assumes all file systems will set it accordingly. (It is not sufficient to use the PFS_ATTR directory bit as not all file services maintain this attribute).

st_nlink	Unix. This field may be set by file system libraries but is otherwise unused.
st_uid	File owner ID. This longword contains the host file owner. This field has no significance outside of PFS. PFS does not use this field for security checks but it may be used by file system libraries.
st_gid	File owner group ID. This field contains the host file owner group ID. This field has no significance outside PFS. PFS does not use this field for security checks but it may be used by file system libraries.
st_rdev	Unix. This field may be set by file system libraries but is otherwise unused.
st_size	File size. This field holds the file size in bytes. It is important to note that this is the native file size. This may include record format overhead and is likely to be of no significance outsize of PFS.
st_atime	Access time in Unix time format. This is the host file system access time. While there may be a relationship between this time and the client access time this is not necessarily true.
st_mtime	Modification time in Unix format. This is the host file system modification time. See above disclaimer.
st_ctime	Creation time in Unix format. This is the host file system creation time. See above disclaimer.
st_fab_rfm	ODS-2 record format. This field is used by the ODS2 file system library and has no significance outside the library.
st_fab_rat	ODS-2 record attributes. This field is used by the ODS2 file system library and has no significance outside the library.
st_fab_fsz	ODS-2 fixed size. This field is the length of the fixed portion of a VFC file foramt. This field is used by the ODS2 file system library and has no significance outside the library.
st_fab_mrs	ODS-2 maximum record size. This field is the length of the largest possible record in the file. This field is used by the ODS2 file system library and has no significance outside the library.

As can be seen from the above descriptions, this structure is for internal use only. It is described here only because it is contained within data structures which pass across the interface. (and because there is a body of server code which references fields within the structure). No access to this structure can be allowed outside PFS as the fields vary amoung file systems.

7.5.11 Dirent structure

The dirent structure is used by PFS_getdents() and contains information about files contained in a directory. This data structure is inteded for server consumption (with the exception noted below) and should be defined as a native PFS data structure. This is likely for future PFS developements.

The data structure is always allocated in longword quantities within the PFS_getdents() buffer. The record length member is used to calculate the offset to the next structure within the buffer.

The fields of the direct structure are as follows:

d_ino	Opaque quantity. This field contans file location informaton used by PFS_dentpathid() to improve directory search performance.
d_reclen	Length of the entire record, rounded to the next longword.
d_namelen	Filename length. This field is the byte count of the filename.
d_name	Filename buffer. This field contains the actual filename.

8 ROUTINE DESCRIPTIONS

The following section describes the routines available to servers. The server should include the file PFS.H in each module which uses these functions.

8.1 PFS interface

The top level PFS interface consists of a set of routines to perform file functions. Many of these routines are merely jacket routines for the underlying file system. PFS determines the file system to which a file belongs during PFS_getpathid(). All functions from this point on are dispatched thru the file system library vector returned by the FSLIB_claim() function.

Routines added to support the PATHWORKS server partioning are denoted by (*). These routines should be ported to all platform implementations even if they are not necessary. Where appropriate porting suggestions are given.

8.1.1 PFS_access *
8.1.2 PFS_faccess *

Description:

PFS_access checks the specified path for the specified access. This is a host security check only. NOS security must be checked separately. Note that PFS does not execute in the context of the host user. It is threfore possible to open a file and then check the file permissions. For VMS this may yeild some performance improvement for files to which the user has access. There will be a performance degradation for files to which the user has no access. It may be worth optimizing the success path and for such, PFS_faccess is provided.

NOTE

> This is the only PFS function which verifies host access to a file. The server should call this function when it needs to verify a specific access.
>
> Alternately, each function which needs to perform a security check can be modified to accept the PFS_USER structure.

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_access (PFS_PATHID *pathid, PFS_USER *user, PFS_SECURITY_MODE mode, int perms)

PFS_RETVAL PFS_faccess (PFS_FID *fp, PFS_USER *user, PFS_SECURITY_MODE mode, int perms)

Arguments:

pathid	Resolved pathid for path to check. This arrument is returned from PFS_getpathid().
fp	Open file pointer. The argument is returned from PFS_open().
user	PFS_USER structure from PFS_getuser().
mode	Volume security mode. The following values are defined:

PFS_NOS_SECURITY	Access functions will always succeed. This mode ignores host security. Security checks are limited to server NOS security checks.
PFS_HOST_SECURITY	Host security will be checked, regarless of file owner. This check is in addition to server NOS security checks.
PFS_CREATOR_SECURITY	Host security will be checked on files not created on behalf of a NOS client. This check is in addition to the server NOS security check.

perms — Unix style permission code. The following bits are defined:

000	File exists
001	Execute access
002	Write/Delete access
004	Read access

Translations taken from "Programming in VAX-11 C".

Return values:

PFS_SUCCESS - Access is allowed
PFS_FAILURE - No access or invalid path

8.1.3 PFS_chdir
8.1.4 PFS_fchdir

Description:

PFS_chdir sets the current working directory. Note that the process structure of PFS is such that all threads executing in the process will see this default. It is threfore required that the server save and restore this default on thread switch. It is furtehr required that all thread switching performed by PFS be routed thru the server to allow these tasks to be completed.

Synopsis:

```
#include <pfs.h>

PFS_RETVAL PFS_chdir (PFS_PATHID *pathid)

PFS_RETVAL PFS_fchdir (PFS_FID *fp)
```

Arguments:

pathid	Resolved pathid pointing to a directory.
fp	Open file pointer for a directory. Only PFS_open() will return this pointer.

Return values:

PFS_SUCCESS - Default set
PFS_FAILURE - Invalid pathid or fp

8.1.5 PFS_chmod
8.1.6 PFS_fchmod

Description:

PFS_chmod changes the file protection of a host file. Note that there is no access checking with respect to the client for which this function is being executed. It is the responsibility of the server to determine if the client has the requisite privileges to affect the change.

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_chmod (PFS_PATHID *pathid, mode_t mode)

PFS_RETVAL PFS_fchmod (PFS_FID *fp, mode_t mode)

Arguments:

pathid	Resolved pathid returned from PFS_getpathid().
fp	Open file pointer.
mode	New file protection. The following bits are defined:

0400	Owner : Read
0200	Owner : Write
0100	Owner : Execute
0040	Group : Read
0020	Group : Write
0010	Group : Execute
0004	World : Read
0002	World : Write
0001	World : Execute

System is always given the same protection as Owner. Write privilege implies Delete. Translations taken from "Programming in VAX-11 C".

Return values:

PFS_SUCCESS - Protetion changed
PFS_FAILURE - Invalid path

8.1.7 PFS_chown
8.1.8 PFS_fchown

Description:

PFS_chown changes the host owner of a file. The NOS owner is not affected. It is expected that this function be used in conjunction with PFS_setsecurity() to affect an owner change consistent with both NOS and host file systems. Note that there is no access checking with respect to the client for which this function is being executed. It is the responsibility of the server to determine if the client has the requisite privileges to affect the change.

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_chown (PFS_PATHID *pathid, uid_t uid, gid_t gid)

PFS_RETVAL PFS_fchown (PFS_FID *fp, uid_t uid, gid_t gid)

Arguments:

pathid	Resolved pathid from PFS_getpathid()
fp	Open file pointer
uid	User identification code
gid	Group identification code

Return values:

PFS_SUCCESS - Owner changed
PFS_FAILURE - Invalid path

8.1.9 PFS_close

Description:

PFS_close will close an open file. This function will flush any modified buffers, remove all locks associated with the file and update the volume modification time, if required. This call should be made when a file is actually to be closed (i.e. after open file cache expiration time).

NOTE

> It is possible that the PFS_FID may be shared among threads of the same process. If this is the case a reference count will be decremented and the actual file close will only occur when the count reaches zero.

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_close (PFS_FID *fp)

Arguments:

fp	Open file pointer

Return values:

PFS_SUCCESS - File closed
PFS_FAILURE - Invalid file pointer

8.1.10 PFS_closeandpurge *

Description:

PFS_closeandpurge will close an open file and then purge it. This function should be used when a temporary file is to be deleted.

NOTE

> It is possible that the PFS_FID may be shared among threads of the same process. If this is the case a reference count will be decremented and the actual file close will only occur when the count reaches zero.

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_closeandpurge (PFS_FID *fp)

Arguments:

fp	Open file pointer

Return values:

PFS_SUCCESS - File closed and deleted
PFS_FAILURE - Invalid file pointer

8.1.11 PFS_copyfile

Description:

PFS_copyfile copies a file from one source to a destination. The source and destination may be in different file system libraries. The function is subject to source file system attribute PFS_ATTR.attr_bits.no_copy.

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_copyfile (PFS_PATHID *source, PFS_PATHID *dest,
PFS_SUPORT_STREAMS dostream, PFS_COPY_ACTION action)

Arguments:

source	Resolved pathid structure for source file. The file must not have the PFS_ATTR no_copy bit set.
dest	Resolved pathid structure for destination file. The file must not reside in a read only file system.
dostream	Action to be taken if the destination file system does not support all streams of the source file.

PFS_NONE_OK	Copy the supported streams only. The remaining streams are lost.
PFS_MUST_RESOURCE	If the destination does not support resource streams and the source file has a resource stream then fail.
PFS_MUST_EXTATTRS	If the destination does not support extended atrtributes and the source file has extended attrbiutes then fail.
PFS_MUST_SECURITY	If the destination does not support security data and the source has security data then fail.
PFS_MUST_ALL	If the destination does not support either resource streams nor extended

		attributes streams and the source file has either then fail.
action	Action to be taken if the destination file already exists.	
	PFS_TRUNCATE	Truncate all destination streams.
	PFS_APPEND	Append PFS_PRIMARY data stream. Truncate the resource stream. Leave extended attributes stream unchanged. The source extended attributes are lost.

Return values:

PFS_SUCCESS - File copied
PFS_FAILURE - Invalid path, conflicting file systems or copy protect

8.1.12 PFS_create

Description:

PFS_create creates a new file (PFS_PRIMARY stream only) or truncates an existing file. The file is left open after the function executes.

Synopsis:

```
#include <pfs.h>

PFS_RETVAL PFS_create (PFS_PATHID *pathid, mode_t mode, uid_t uid, gid_t gid,
                       PFS_CREATE_TYPE type, PFS_FID **fp)
```

Arguments:

pathid	Resolved pathid from PFS_getpathid()	
mode	File protection. Set PFS_chmod() for a description.	
uid	Host file owner user ID.	
gid	Host file owner group ID.	
type	Type of file to create. The field has one of the following values:	
	PFS_CREATEIT	If the file already esists then truncate it.
	PFS_MAKETMP	Create a temporary file. Pathid points to the directory in which to create the file. The filename is generated.
	PFS_MAKNEW	If the file exists PFS_create() fails.
fp	Pointer to return file pointer. The file pointer is allocated by PFS_create() and must be returned on PFS_close().	

Return values:

PFS_SUCCESS - File created
PFS_FAILURE - Invalid path or file exists and PFS_MAKENEW specified

8.1.13 PFS_delete

Description:

PFS_detete deletes a file. If the PFS_ATTR bit no_purge is set the file is moved to a holding area. [Where ??]. If no_purge is not set then the file is actually deleted.

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_delete (PFS_PATHID *pathid)

Arguments:

pathid	Resolved pathid structure from PFS_getpathid().

Return values:

8.1.14 PFS_dentpathid *

Description:

PFS_dentpathid converts a directory "struct dirent" to a PFS_PATHID structure. This function is used to improve performance of directory search functions. If a file system does not support this function or if sufficient information is not in the struct dirent then the file system should return failure. PFS_dentpathid() will then call PFS_getpathid() using the filename from the "struct dirent". It is assumed that PFS_chdir() has been called to set the default directory to that being searched prior to this call. While PFS_dentpathid() does not use the default directory, fallbacks to PFS_getpathid() will.

Synopsis:

#include <pfs.h>
#include <dirent.h>

PFS_RETVAL PFS_dentpathid (PFS_FID *fp, struct dirent *dirent, PFS_PATHID *pathid)

Arguments:

fp	Open file pointer to directory to be searched
dirent	Struct dirent from PFS_getdents().
pathid	Resulting PFS_PATHID structure for file.

Return values:

PFS_EXISTS - Path exists as specified.
PFS_NOEXIST - Path does not exist but the parent path does (i.e. a new file specification).
PFS_FAILED - Neither parent nor path exists.

8.1.15 PFS_didpathid *

Description:

PFS_didpathid will accept a default directory structure instead of the root string as in PFS_getpathid(). The remaining function is identical to PFS_getpathid().

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_didpathid (PFS_CWD *dirid, char *path, PFS_NAMESPACE namespace,
PFS_PATHID *pathid)

Arguments:

dirid	PFS_CWD structure as returned by PFS_diridfunc. The path member of the structure is not used. The directoryID member is used as the root directory.
path	NOS path name.
namespace	NOS path name space identifier. This argument specifies the namespace in which the path resides.
pathid	Resolved pathid structure.

Return values:

PFS_EXISTS - Path exists as specified.
PFS_NOEXIST - Path does not exist but the parent path does (i.e. a new file specification).
PFS_FAILED - Neither parent nor path exists.

8.1.16 PFS_diridfunc

Description:

PFS_diridfunc supports translation of directory IDs. The function will open a set of IDs, close a set of IDs or translate the IDs into native file system structures. The back translation to path string is somewhat expensive on VMS and is not required for lookups. The interface has been changed to return a structure of the same form as used by PFS_cwd(). This structure will only carry the full VMS directory ID and may be used as input to the function PFS_didpathid().

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_diridfunc (PFS_DIRID_CMD cmd, char *root, unsigned long dirid, PFS_CWD
*dirptr)

Arguments:

cmd	Directory ID command. The command is one of the following:	
	PFS_DIRID_OPEN	Open a new set of directory IDs. The root argument carries the volume name to be opened.
	PFS_DIRID_GET	Translate the given directory ID to a PFS_CWD structure. The root argumenet is not used.

	PFS_DIRID_CLOSE	Close a set of directory IDs. The root argument carries the name of the volume to close.

root	Volume root directory. This field is used for PFS_DIRID_OPEN functions only.
dirid	Directory ID to translate
dirptr	Return directory ID structure. This structure may contain the path name as a string or the native directory ID or both.

Return values:

PFS_SUCCESS - Directory ID translated
PFS_FAILURE - Invalid directory ID or no directory ID set open.

8.1.17 PFS_diridinit

Description:

PFS_diridinit initializes the generation of directory IDs for file systems which do not direcltly support directory IDs.

Synopsis:

#include <pfs.h>

void PFS_diridinit (PFS_DIRIDS_MATTER dodorods, unsigned long *diridptr)

Arguments:

dodirids — Flag to indicate whether to generate directory IDs. The flag has the following values:

PFS_DIRIDS	Generate directory IDs for PFS_mkdir(). PFS_getattr() will return the generated ID.
PFS_NODIRIDS	Do not generate directory IDs. The file system will handle the function directly.

diridptr — Pointer into shared memory for the next unique directory ID.

Return values:

None

8.1.18 PFS_filesize
8.1.19 PFS_ffilesize

Description:

PFS_ffilesize returns the size of an open file. The function may be required to read the file to determine its size. If so, the filesize will be saved in the ACE associated with the file.

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_filesize (PFS_PATHID *pathid, PFS_DATA_STREAM stream, off_t *size)

PFS_RETVAL PFS_ffilesize (PFS_FID *fp, off_t *size)

Arguments:

pathid	Resolved pathid structure as returned by PFS_getpathid().
stream	Data stream to obtain size of.
fp	Open file pointer
size	Pointer to return file size longword

Return values:

PFS_SUCCESS - Return size is valid
PFS_FAILURE - Invalid file pointer

8.1.20 PFS_fsync

Description:

PFS_fsync flushes all modified data associated with a file. This includes modified header data.

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_fsync (PFS_FID *fp)

Arguments:

fp	Open file pointer

Return values:

PFS_SUCCESS - File flushed
PFS_FAILURE - Invalid file pointer

8.1.21 PFS_fullpath

Description:

PFS_fullpath returns the full file specification for an open file.

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_fullpath (PFS_FID *fp, char *pathbuf, int buflen)

Arguments:

fp	Open file pointer
pathbuf	Buffer for return file specification

buflen	Length of return buffer

Return values:

PFS_SUCCESS - Path written to buffer
PFS_FAILURE - Invalid file pointer to buffer too small

8.1.22 PFS_getattr
8.1.23 PFS_fgetattr

Description:

PFS_getattr will return the file attributes structure. The attributes structure is described in section 7.5.

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_getattr (PFS_PATHID *pathid, unsigned long mask, PFS_ATTR *attrp)

PFS_RETVAL PFS_fgetattr (PFS_FID *fp, unsigned long mask, PFS_ATTR *attrp)

Arguments:

pathid	Resolved pathid structure from PFS_getpathid()
fp	Open file pointer
mask	Mask of elements requested. There is one bit in the mask for each field in the PFS_ATTR structure. This mask has the exact same format as that in the PFS_ATTR structure.
attrp	Pointer to return attributes structure.

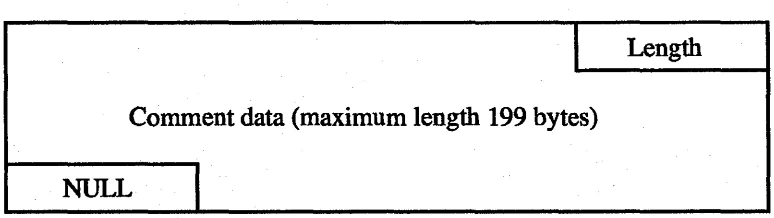
Return values:

PFS_SUCCESS - Attributes updated
PFS_FAILURE - Invalid parameters

8.1.24 PFS_getcomment

Description:

PFS_getcomment will return the comment record associated with a file. The comment is limited to 199 bytes. The first byte of the comment buffer contains the length of the comment string. The string is NULL terminated.



Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_getcomment (PFS_PATHID *pathid, char *comment, int buflen)

Arguments:

pathid	Resolved pathid from PFS_getpathid().
comment	Return buffer for comment.
buflen	Length of return buffer.

Return values:

PFS_SUCCESS - Coment returned
PFS_FAILURE - Invalid parameters

8.1.25 PFS_getcwd

Description:

PFS_getcwd returns the current working directory.

Synopsis:

```
#include <pfs.h>
```

PFS_RETVAL PFS_getcwd (PFS_CWD *cwd)

Arguments:

cwd	Pointer to working directory structure

Return values:

PFS_SUCCESS - Directory returned
PFS_FAILURE - No directory set

8.1.26 PFS_getdents

Description:

PFS_getdents returns directory entries in a struct dirent buffer. The buffer is written with as many full directory entries as will fit (or as many as are in the directory). The struct dirent is defined in section 7.5.

The function should be called until the bytesread paramater indicates zero bytes written to the output buffer. The function will not fail when the end of the directory is reached.

Synopsis:

```
#include <pfs.h>
#include <dirent.h>
```

PFS_RETVAL PFS_getdents (PFS_FID *fp, struct dirent *direntp, unsigned int nbytes, off_t *offset, PFS_NAMESPACE namespace, unsigned int bytesread)

Arguments:

fp	Open file pointer for the directory to be enumerated.

direntp	Buffer to receive directory entries.
nbytes	Size of the buffer.
offset	Pointer to receive context longword for resuming directory enumeration. This longword must not be modifed bewteen calls to PFS_getdents.
namespace	Namespace in which to return directory entries.
bytesread	Return count of how many bytes were written to the output buffer.

Return values:

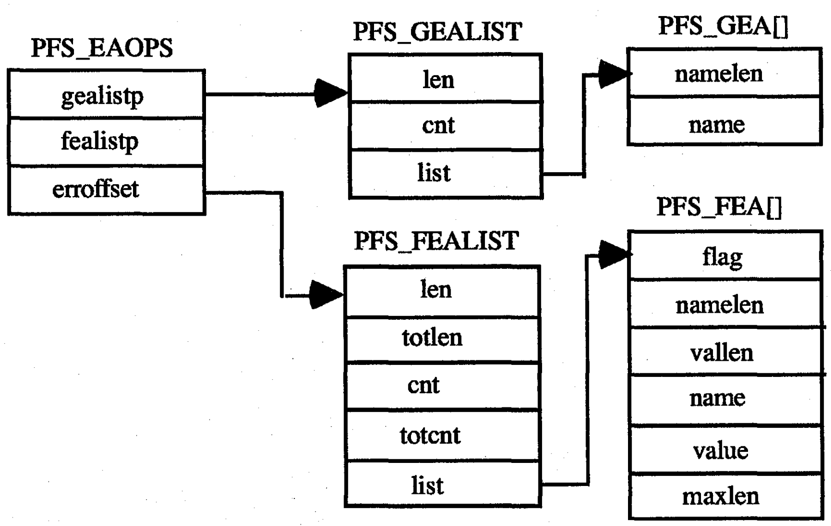
PFS_SUCCESS - Buffer written (including no entries)
PFS_FALURE - Invalid parameters

8.1.27 PFS_getextattr
8.1.28 PFS_fgetextattr

Description:

PFS_getextattr will return the extended attributes associated with a file. The interface uses a number of structures to carry requested attributes and return attributes found on the file.

The arrangement of structures is shown below:



Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_getextattr (PFS_PATHID *pathid, PFS_EAOPS *eaopsp)

PFS_RETVAL PFS_fgetextattr (PFS_FID *fp, PFS_EAOPS *eaopsp)

Arguments:

pathid	Resolved pathid from PFS_getpathid().
fp	Open file pointer

eaopsp	Pointer to return extended attributes structure. [This structure is not yet defined as there is still question as to whether we support extended attributes].

Return values:

PFS_SUCCESS - Extended attributes written to buffer
PFS_FAILURE - Invalid parameters

8.1.29 PFS_getpathid

Description:

PFS_getpathid resolves a NOS file path given the top of the directory tree, the NOS path and NOS type. PFS_getpathid locates the file system which handles this path and sets the file system library dispatch vectors for future reference.

PFS_getpathid calls the following library functions:

FSLIB_claim	File system is asked to claim the path. If a file system claims a path then it will be responsible for all future requests for that path.
FSLIB_convert	Convert filename to native file specification
FSLIB_lookup	Locate the file
FSLIB_stat	Return file location, type, size, etc.

Synopsis:

```
#include <pfs.h>

PFS_RETVAL PFS_getpathid (char *root, char *path, PFS_NAMESPACE namespace,
                          PFS_PATHID *pathid)
```

Arguments:

root	Top of directory tree or directory which corresponds to the volume root.
path	NOS file path.
namespace	Namespace in which path resides. The following values are defined:

PFS_DOSNAME	DOS filename format
PFS_MACNAME	Macintosh filename format
PFS_VMSNAME	VMS filename format
PFS_UNIXNAME	Unix name format
PFS_NATIVENAME	Native file system format

pathid	Return resolved pathid structure

Return values:

PFS_EXISTS - Path exists as specified.
PFS_NOEXIST - Path does not exist but the parent path does (i.e. a new file specification).
PFS_FAILED - Neither parent nor path exists.

8.1.30 PFS_getprintident *

Description:

PFS_getprintident returns file identification information in the PFS_IDENT structure. This structure is used primarity by the print subsystem to identify a file.

The PFS_IDENT structure is defined in section 7.5.

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_getprintident (PFS_PATHID *pathid, PFS_IDENT *identp)

Arguments:

pathid	Resolved pathid structure as returned by PFS_getpathid().
identp	Pointer to return identification structure.

Returns:

PFS_SUCCESS - Print identification returned
PFS_FAILURE - Insufficient information

8.1.31 PFS_getsecurity *
8.1.32 PFS_fgetsecurity *

Description:

PFS_getsecurity returns stored NOS security data for a given object. The function does not interpret the data.

Lan Manager security data is stored as a set of named objects. The name may be any string and the data is variable length. PFS_getsecurity will retrieve named objects if requested. The semantics are identical to those for PFS_getextattr(). Lan Manager security data is accessed by specifying PFS_LMXSECURE as the securspace argument.

Macintosh security data is stored in the Macintosh ACE. The format of the Macintosh security data returned is identical to that stored in the Macintosh ACE. Macintosh security data is accessed by specifying PFS_MACSECURE as the securspace argument.

As with PFS_getextattr(), it is the responsibility of the caller to deallocate the PFS_FEALIST structure.

Note that the PFS_GEA structure is used to request named security data. This structure is initialized by the caller. The PFS_FEA structure holds one record per requested named security data, even if the data does not exist. The vallen field will be set to zero in the event that the data does not exist. In this manner there is a one-to-one correspondence between input array offsets and output array offsets. The caller must release the PFS_EAOPS pointer fealistp. All pointers within the PFS_FEA and PFS_FEALIST are contained in this buffer.

NOTE

PFS does not understand the hierarchy of security data. PFS will store and retrieve security data as it applies to file objects only. It is the server's responsibility to locate security information stored elsewhere in the path which may apply to the current request. This partition may cause additional path processing and file access to locate inherited security data. The server will make repeated calls to PFS to obtain pathid structures and security data for each member of the path for which access information is not stored.

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_getsecurity (PFS_PATHID *pathid, PFS_SECURSPACE securspace, void *securp)

PFS_RETVAL PFS_fgetsecurity (PFS_FID *fp, PFS_SECURSPACE securspace, void *securp)

Arguments:

pathid	Resolved pathid structure returned by PFS_getpathid().	
fp	Open file pointer	
securspace	Security data space to be modified. The following values are defined:	
	PFS_LMXSECURE	Lan Manager security space.
	PFS_MACSECURE	Macintosh security space.
securp	Pointer to security data access structure. For PFS_LMXSECURE the structure is identical to the extended attributes structure. For PFS_MACSECURE the data starts at offset "ownerID" and is returned exactly as specified in the Macintosh ACE. The data is 11 bytes in length.	

Return values:

PFS_SUCCESS - Returned attributes
PFS_FAILURE - Invalid parameters

8.1.33 PFS_getuser *

Description:

PFS_getuser returns information about the specified host user in a PFS_USER structure. This structure is used for access checking functions PFS_acess() and PFS_faccess().

The function reads information from the system User Authorization File (UAF) and returns the user identification, rights and privileges.

It is the callers responsibility to release the memory associated with the structure. The rights list must be released as well as the structure itself.

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_getuser (char *username, PFS_USER **user)

Arguments:

username	Host username for which information is desired
user	Return user structure defining user rights and privileges.

Return values:

PFS_SUCCESS - Obtained user info
PFS_FAILURE - No such user

8.1.34 PFS_init

Description:

PFS_init initializes the file system interface, loads and calls all file system libraries. Addon libraries are located in the directory pointed to by PWRK$ADDON_LIBRARY: and must named PWRK$name_FSLIB.EXE, where name is the file system name. The file system library must have a universal symbol of the form name_init. For example, the FAT file system would be found as PWRK$FAT_FSLIB.EXE and would have a universal symbol FAT_init. The universal symbol is the entry point to the library and is responsible for initializing the PFS_LIB_ENT structure, including setting up the library vectors.

PFS_init must be called prior to any PFS file access.

Synopsis:

#include <pfs.h>

void PFS_init (void)

Arguments:

None

Return values:

None

8.1.35 PFS_lock

Description:

PFS_lock establishes a byte range lock on the file in the underlying file system. This function is provided for establishing byte range locks in the file system itself. Btye range locks are handled withing PATHWORKS by the PATHWORKS Lock Manager and may not involve the file system.

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_lock (PFS_FID *fp, short type, off_t offset, short whence, off_t length,
PFS_WAIT_LOCK dowait, off_t *start)

Arguments:

fp	Open file pointer	
type	Type of lock to set. The following values are defined:	
	F_RDLCK	Read lock (shared).
	F_WRLCK	Write lock (exclusive).
offset	Position to start lock.	

whence	Position from which to measure offset. The following values are defined:	
	SEEK_SET	Offset is relative to the start of the file. The offset should be a positive number.
	SEEK_END	Offset is relative to the end of the file. The offset should be a negative number.
length	Length of range locked. If zero is specified as a length the remainder of the file is locked from the position defined by offset and whence.	
dowait	The following values are defined:	
	PFS_WAIT	Wait for release of existing lock prior to resuming execution.
	PFS_NOWAIT	If any portion of the range is locked, PFS_lock fails.
start	Return position relative to the start of the file where the locked range starts. The pointer may be NULL in which case it is ignored.	

Return values:

PFS_SUCCESS - Lock set
PFS_FAILURE - Invalid parameters or lock conflict

8.1.36 PFS_lseek

Description:

PFS_lseek positions the current file pointer to the position specified.

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_lseek (PFS_FID *fp, off_t offset, int whence)

Arguments:

fp	Open file pointer	
offset	Position to set file pointer to, relative to whence argument.	
whence	Position from which to measure offset. The following values are defined:	
	SEEK_SET	Offset is measured from the start of the file.
	SEEK_END	Offset is measured from the end of the file.

Return values:

PFS_SUCCESS - File position changed
PFS_FAILURE - Invalid parameters

8.1.37 PFS_mapname
8.1.38 PFS_fmapname

Description:

PFS_mapname will translate the last member of a given path to the namespace specified. Note that this function may require file system I/O to complete the translation. This function should not be used to pretranslate filenames prior to access. PFS_getpathid() provides this translation.

It is not clear wht this function is intended for but it should be avoided in performance critical applications.

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_mapname (PFS_PATHID *pathid, PFS_NAMESPACE namespace, char *namebuf, int buflen)

PFS_RETVAL PFS_fmapname (PFS_FID *fp, PFS_NAMESPACE namespace, char *namebuf, int buflen)

Arguments:

pathid	Resolved pathid structure from PFS_getpathid()
fp	Open file pointer
namespace	Namespace in which translated name is to be returned
namebuf	Buffer in which to return name
buflen	Length of return buffer

Return values:

PFS_SUCCESS - Name translated
PFS_FAILURE - Invalid parameters or buffer too small

8.1.39 PFS_mkdir

Description:

PFS_mkdir creates a directory. The host owner is set to that specified as welll as the access permissions.

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_mkdir (PFS_PATHID *pathid, mode_t mode, uid_t uid, gid_t gid, unsigned long *dirid)

Arguments:

pathid	Resolved pathid for the directory to be created.
mode	Access permissions to be applied to the directory. See PFS_chmod() for a description of this parameter.
uid	Host user identification for the directory owner.
gid	Host group identification for the directory owner.
dirid	Value is incremented and used as the directory ID unless NULL. If NULL the file system will generate its own internal IDs.

Return values:

PFS_SUCCESS - Directory created
PFS_FAILURE - Invalid parameters

8.1.40 PFS_mpxclose

Description:

PFS_mpxclose closes the file system file. The PFS file pointers are maintained as if the file was still open. This function allows freeing file descriptors for reuse. If a file which has been multiplex closed is referenced it will be reopened.

[File multiplexing needs to be reviewed.]

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_mpxclose (PFS_FID *fp)

Arguments:

fp	Open file pointer

Return values:

PFS_SUCCESS - File is close in the underlying file system
PFS_FAILURE - Invalid parameters

8.1.41 PFS_needfds

Description:

PFS_needfds will multiplex close open files such that the requested number of file descriptors are available for use.

[File multiplexing needs to be reviewed]

Synopsis:

#include <pfs.h>

void PFS_needfds (int count)

Arguments:

count	Required number of file descriptors

Return values:

None

8.1.42 PFS_needinodes

Description:

PFS_needinodes will multiplex close a number of files to attempt to free inodes in the Unix file system. This call has no effect on VMS systems.

Synopsis:

#include <pfs.h>

void PFS_needinodes (int timescalled, PFS_OPS *funcptrs)

Arguments:

timescalled	Number of times the function has been called attempting to get inodes released. The function will increase the number of files closed on each successive call which increments this counter.
funcptrs	File system function pointers [which file system ??]

Return values:

None

8.1.43 PFS_open

Description:

PFS_open will open a file stream. The function has the ability memory map the file under Unix. For VMS this option is ignored.

Synopsis:

#include <pfs.h>

```
PFS_RETVAL PFS_open (PFS_PATHID *pathid, PFS_STAT *statbufp, int oflag,
                     PFS_DATA_STREAM stream, PFS_MEM_MAP dommap,
                     PFS_FID **fp)
```

Arguments:

pathid	Resolved pathid as returned by PFS_getpathid()
statbufp	Status buffer pointer. If the server has the status buffer for a non primary stream and wishes to open the stream it can save a PFS_stat() call by specifying the buffer.
oflag	Open mode flags. The following bits are defined:

00000	O_RDONLY	Open the file for read access only.
00001	O_WRONLY	Open the file for write access only.
00002	O_RDWR	Open file for both read and write access.

00010	O_APPEND	Open file for append access.
01000	O_CREAT	Create stream if it does not exist
02000	O_TRUNC	Truncate stream if it exists

stream — Specifies the stream to open (or create). Note that the primary stream must already exist.

dommap — Memory map the file. This option has significance under Unix only.

fp — Return open file pointer.

Return values:

PFS_SUCCESS - File open
PFS_FAILURE - Invalid parameters

8.1.44 PFS_parse *

Description:

PFS_parse returns a structure defining the components of a file path in a specified namespace. This function is intended to remove file system namespace manipulation assumptions from the server. The server should use this function to process components of a path specification. The PFS_NAMEID structure is defined in section 7.5.

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_parse (char *path, PFS_NAMESPACE namespace, PFS_NAMEID *nameid)

Arguments:

path — File specification

namespace — Namespace in which path exists

nameid — Return structure defining the components of the path.

Return values:

8.1.45 PFS_purge

Description:

PFS_purge deletes a file. The PFS_ATTR no_purge attribute is ignored and the file is deleted.

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_purge (PFS_PATHID *pathid)

Arguments:

pathid — Resolved pathid as returned by PFS_getpathid().

Return values:

PFS_SUCCESS - File deleted
PFS_FAILURE - Invalid parameters

8.1.46 PFS_read

Description:

PFS_read will read data from an open file stream.

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_read (PFS_FID *fp, void *buffer, unsigned int nbytes, off_t offset,
unsigned int *bytesread)

Arguments:

fp	Open file pointer
buffer	Buffer for return data
nbytes	Size of return buffer
offset	Position relative to start of the stream from which to read data.
bytesread	Return count of bytes actually read from the stream.

Return values:

PFS_SUCCESS - Bytes read (including none)
PFS_FAILURE - Invalid parameters

8.1.47 PFS_readdesc *

Description:

PFS_readesc will read data from an open stream and return a set of mapping pointers describing the data. The data itself remains in the data cache.

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_readdesc (PFS_FID *fp, unsigned int nbytes, off_t offset, PFS_DESC **desc)

Arguments:

fp	Open file pointer.
nbytes	Number of bytes to read.
offset	Offset releative to the start of the stream from which to read.
desc	Pointer to receive data description. [Need specification of work element for data cache].

Return values:

PFS_SUCCESS - Data read
PFS_FAILURE - Invalid parameters

8.1.48 PFS_releasedesc *

Description:

PFS_releasedesc will release the data associated with the descriptors previously returned by PFS_readdesc().

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_releasedesc (PFS_DESC *desc)

Arguments:

desc	Descriptor pointer returned by PFS_readdesc().

Return values:

PFS_SUCCESS - Data released
PFS_FAILURE - Invalid parameter

8.1.49 PFS_rename

Description:

PFS_rename will rename a file or directory. The files MUST both exist in the same file system. The function accepts two pathid structures describing the files to be renamed. These structures will have previously resolved namespace considerations. However, BOTH files must exist in the same namespace.

There are cross namespace implications in renaming a file. The VMS, DOS and Macintosh names are all releated and therefore a change to one will result in a change to all. The mapping is as follows:

VMS	DOS	Macintosh
If changed	VMS name	VMS name
DOS name	If changed	DOS name
Short name	Short name	If changed

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_rename (PFS_PATHID *oldpathid, PFS_PATHID *newpathid)

Arguments:

oldpathid	Resolved pathid structure for origninal file name.
newpathid	Resolved pathid structure for new file name.

Return values:

PFS_SUCCESS - File renamed
PFS_FAILURE - Invalid parameters, conflicting file systems or conflicting namespace

8.1.50 PFS_rmdir

Description:

PFS_rmdir will delete a directory. The directory must be empty.

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_rmdir (PFS_PATHID *pathid)

Arguments:

pathid	Resolved pathid structure as returned by PFS_getpathid().

Return values:

PFS_SUCCESS - Directory deleted
PFS_FAILURE - Invalid parameter, directory not empty

8.1.51 PFS_setattr
8.1.52 PFS_fsetattr

Description:

PFS_setattr sets NOS file atributes. Each member of the PFS_ATTR structure has a corresponding mask bit. Only the attributes indicated by the mask are affected. In this manner, conncurrent update may be handled without additional synchronization. Simultaneous updates to the same fields without external synchronization will not yeild predictable results as the order of individual field updates can not be guaranteed. However, simultaneous updates to different fields will yeild the expected results.

A file must be writeable in order to modify the attributes. If a file is open when the attributes are modified it must be open for write access.

Note that the mask of elements to modify is contained in the attributes structure, not specified separately as in PFS_getattr().

The PFS_ATTR structure is defined in section 7.5.

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_setattr (PFS_PATHID *pathid, PFS_ATTR *attrp)

PFS_RETVAL PFS_fsetattr (PFS_FID *fp, PFS_ATTR *attrp)

Arguments:

pathid	Resolved pathid structure as returned by PFS_getpathid()
fp	Open file pointer. File must be open for write access.
attrp	Pointer to PFS_ATTR structure containing attributes to modify

Return values:

PFS_SUCCESS - Attributes modified
PFS_FAILURE - Invalid paramters or file not writeable

8.1.53 PFS_setcomment

Description:

PFS_setcomment will associate a text string with a file. The comment format is as defined for PFS_getcomment().

If a comment is already associated with the file it is replaced.

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_setcomment (PFS_PATHID *pathid, char *comment)

Arguments:

pathid	Resolved pathid as returned by PFS_getpathid()
comment	Comment block as defined in PFS_getcomment().

Return values:

PFS_SUCCESS - Comment written
PFS_FAILURE - Invalid paramters or file not writeable

8.1.54 PFS_setextattr
8.1.55 PFS_fsetextattr

Description:

PFS_setextattr sets the extended attributes of a file. The extended attrributes are described in PFS_getaextattr(). The PFS_GEALIST member is ignored on set operations.

The PFS_FEALIST points to an array of extended atrributes blocks, PFS_FEA. Each array member describes one attribute to add, delete or modify.

If the attributes does not exist it is added. If the attribute already exists it is replaced. If the value length field of the attribute is zero, the attribute is deleted.

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_setextattr (PFS_PATHID *pathid, PFS_EAOPS *eaopsp)

PFS_RETVAL PFS_setextattr (PFS_FID *fp, PFS_EAOPS *eaopsp)

Arguments:

pathid	Resolved pathid structure as returned by PFS_getpathid()
fp	Open file pointer.

eaopsp — Pointer to extended attributes structure. The gealist member of the structure is ignored. The structure is defined in PFS_getextattr().

Return values:

PFS_SCCESS - Attributes modified
PFS_FAILURE - Invalid parameters or file not writeable

8.1.56 PFS_setlognores

Description:

PFS_setlognores accepts a function pointer to be called when resources are exhausted.

Synopsis:

#include <pfs.h>

void PFS_setlognores (void (*func) ())

Arguments:

func — Address of routine entry mask. This routine will be called when PFS exhausts resources. Memory, disk space, IO channels, etc will be reported. Note that no arguments are passed to the called function.

Return values:

None

8.1.57 PFS_setnotifympx

Description:

PFS_setnotifympx accpets a function pointer to be called when file multiplexing occurs.

Synopsis:

#include <pfs.h>

void PFS_setnotifympx (void (*func) ())

Arguments:

func — Address of routine entry mask. This routine will be called when PFS multiplex closes a file. Note that no arguments are passed to the called function.

Return values:

None

8.1.58 PFS_setsecurity *
8.1.59 PFS_fsetsecurity *

Description:

PFS_setsecurity associates NOS security data with a file object. The security data is not interpretted.

For PFS_LMXSECURE the interface is identical to that of PFS_setextattr().

For PFS_MACSECURE the interface accepts a pointer to the "ownerID" member of the Macintosh ACE and writes 11 bytes of data to the file's Macintosh ACE.

If PFS_setsecurity() returns PFS_NOTSUPPORTED the server must be prepared to find alternate storage means for the security data. Not all underlying file systems support association of security data with files.

Synopsis:

#include <pfs.h>

```
PFS_RETVAL PFS_setsecurity (PFS_PATHID *pathid, PFS_SECURSPACE securspace,
                            void *securp)
```

```
PFS_RETVAL PFS_fsetsecurity (PFS_FID *fp, PFS_SECURSPACE securspace, void *securp)
```

Arguments:

pathid	Resolved pathid structure as returned by PFS_getpathid()
fp	Open file pointer
securspace	Security data space to be modified. The following values are defined: PFS_LMXSECURE — Lan Manager security space. PFS_MACSECURE — Macintosh security space.
securp	For PFS_LMXSECURE, a pointer to extended attributes structure containing named security data. The semantics are identical to those of PFS_setextattr(). For PFS_MACSECURE a pointer to the "ownerID" member of the Macintosh ACE.

Return values:

PFS_SUCCESS - Data associated
PFS_FAILURE - Invalid parameters or file not writeable
PFS_NOTSUPPORTED - Security data not supported

8.1.60 PFS_shortpath

Description:

PFS_shortpath sets the shortpath member of the pathid structure relative to the current working directory, if working directories are supported. Not all file systems support this and for those which dont, this function is a NOP.

Synopsis:

#include <pfs.h>

```
void PFS_shortpath (PFS_PATHID *pathid)
```

Arguments:

pathid	Resolved pathid structure as returned by PFS_getpathid()

Return values:

None

8.1.61 PFS_stat
8.1.62 PFS_fstat

Description:

PFS_stat will obtain file location information and file structure information for the file containing the given stream. The data structures returned are indended to be opaque. This call is provided for potential performance optimizations in the server. It is expected that the return PFS_STAT structure is to be given back to PFS at some later time, potentially saving multiple stat calls in PFS.

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_stat (PFS_PATHID *pathid, PFS_DATA_STREAM stream,
unsigned long mask, PFS_STAT *statbufp)

PFS_RETVAL PFS_fstat (PFS_FID *fp, unsigned long mask, PFS_STAT *statbufp)

Arguments:

pathid	Resolved pathid structure as returned by PFS_getpathid().
fp	Open file pointer
stream	File stream for which information is to be returned.
mask	Mask of elements to be returned.
statbufp	Pointer to return PFS_STAT structure.

Return values:

PFS_SUCCESS - Information obtained
PFS_FAILURE - Invalid parameters

8.1.63 PFS_statvfs
8.1.64 PFS_fstatvfs

Description:

PFS_statvfs provides information about a mounted file system under Unix only. This call is a NOP on VMS systems.

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_statvfs (PFS_PATHID *pathid, statvfs_t *fsbufp)

PFS_RETVAL PFS_fstatvfs (PFS_FID *fp, statvfs_t *fsbufp)

Arguments:

pathid	Resolved pathid as returned by PFS_getpathid()
fp	Open file pointer
fsbufp	Return file system information block. This structure is defined for Unix systems only.

Return values:

PFS_SUCCESS - Information obtained
PFS_FAILURE - Invalid parameters

8.1.65 PFS_sync

Description:

PFS_sync will flush all modified data. All file headers are written out to disk as well as all file data.

Synopsis:

#include <pfs.h>

void PFS_sync (void)

Arguments:

None

Return values:

None

8.1.66 PFS_treetop

Description:

PFS_treetop sets the treetop member of the pathid structure to that specified. This field is not used by PFS and is provided for server use.

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_treetop (PFS_PATHID *pathid, char *treetop)

Arguments:

pathid	Resolved pathid structure as returned by PFS_getpathid()
treetop	Pointer to be copied to treetop member of pathid structure. PFS makes no assumptions about the contents of this pointer.

Return values:

PFS_SUCCESS - Treetop written
PFS_FAILURE - Invalid parameters

8.1.67 PFS_ftruncate

Description:

PFS_truncate will truncate a file at a given offset. This call yeilds the same result as PFS_write() with a zero buffer length.

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_ftruncate (PFS_FID *fp, off_t size)

Arguments:

fp	Open file pointer
size	Position at which to truncate the file.

Return values:

PFS_SUCCESS - File truncated
PFS_FAILURE - Invalid parameters or file not writeable

8.1.68 PFS_unlock

Description:

PFS_unlock releases a file system byte range lock established by PFS_lock().

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_unlock (PFS_FID *fp, off_t offset, short whence, off_t length)

Arguments:

fp	Open file pointer	
offset	Position at which lock starts	
whence	Position from which to measure offset. The following values are defined:	
	SEEK_SET	Offset is measured from the start of the file. Offset should be a positive number.
	SEEK_END	Offset is measured from the end of the file. Offset should be a negative number.
length	Length of range lock. A value of zero indicates range from offset to the end of the file.	

Return values:

PFS_SUCCESS - Range lock removed
PFS_FAILURE - Invalid parameters or no range locked

8.1.69 PFS_unmap

Description:

PFS_unmap cleans up a memory mapped file. It does not close the file.

Synopsis:

#include <pfs.h>

void PFS_unmap (PFS_FID *fp)

Arguments:

fp	Open file pointer which was memory mapped by PFS_open().

Return values:

None

8.1.70 PFS_utime
8.1.71 PFS_futime

Description:

PFS_utime sets the file modification time and file access time. The time is specified PLATFORM DEPENDENT, i.e. it will be in Unix time format for Unix systems and VMS format for VMS systems. This function does not affect the NOS times associated with a file. Use PFS_setattr() for modification of NOS times.

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_utime (PFS_PATHID *pathid, void *timebufp)

PFS_RETVAL PFS_utime (PFS_FID *fp, void *timebufp)

Arguments:

pathid	Resolved pathid structure as returned by PFS_getpathid().
fp	Open file pointer
timebufp	Platform dependent time format. For Unix this is a pointer toa longword and for VMS this is a pointer to a quadword time buffer.

Return values:

PFS_SUCCESS - Time modified
PFS_FAILURE - Invalid parameters or file not writeable

8.1.72 PFS_write

Description:

PFS_write will write data to a file. The file must be open for write access.

If nbytes is zero the file is TRUNCATED at the current offset.

[Is this really necessary given the PFS_truncate function ?? Sure makes a mess of the write code!]

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_write (PFS_FID *fp, void *buffer, unsigned int nbytes, off_t offset,
unsigned int byteswritten)

Arguments:

fp	Open file pointer
buffer	Data buffer to write
nbytes	Size of buffer to write. If this argument is zero the file will be truncated at the position specified by offset.
offset	Position relative to the start of the file at which data is to be written.
byteswritten	Return count of bytes actually written.

Return values:

PFS_SUCCESS - File written
PFS_FAILURE - Invalid parameters or file not open for write.

8.1.73 PFS_writedesc *

Description:

PFS_writedesc will write data to a file by descriptor reference. The descriptor format is defined in PFS_readdesc(). The server is responsible for creating the descriptors and releasing the storage associated with them.

Synopsis:

#include <pfs.h>

PFS_RETVAL PFS_writedesc (PFS_FID *fp, off_t offset, PFS_DESC *desc)

Arguments:

fp	Open file pointer
offset	Position relative to start of file at which data is to be written.
desc	Descriptior list pointer.

Return values:

PFS_SUCCESS - File written
PFS_FAILURE - Invalid parameters or file not open for write access.

8.2 File System Library (FSLIB) interface

8.2.1 FSLIB_access *

8.2.2 FSLIB_faccess *

Description:

See PFS_access().

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_access (PFS_PATHID *pathid, PFS_USER *user, PFS_SECURITY_MODE mode, int perms)

PFS_RETVAL FSLIB_faccess (PFS_FID *fp, PFS_USER *user, PFS_SECURITY_MODE mode, int perms)

Arguments:

See PFS_access().

Return values:

PFS_SUCCESS - Access is allowed
PFS_FAILURE - No access or invalid path

8.2.3 FSLIB_chdir
8.2.4 FSLIB_fchdir

Description:

See PFS_chdir().

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_chdir (PFS_PATHID *pathid)

PFS_RETVAL FSLIB_fchdir (PFS_FID *fp)

Arguments:

See PFS_chdir().

Return values:

PFS_SUCCESS - Default set
PFS_FAILURE - Invalid pathid or fp

8.2.5 FSLIB_chmod
8.2.6 FSLIB_fchmod

Description:

See PFS_chmod().

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_chmod (PFS_PATHID *pathid, mode_t mode)

PFS_RETVAL FSLIB_fchmod (PFS_FID *fp, mode_t mode)

Arguments:

See PFS_chmod().

Return values:

PFS_SUCCESS - Protetion changed
PFS_FAILURE - Invalid path

8.2.7 FSLIB_chown
8.2.8 FSLIB_fchown

Description:

See PFS_chown().

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_chown (PFS_PATHID *pathid, uid_t uid, gid_t gid)

PFS_RETVAL FSLIB_fchown (PFS_FID *fp, uid_t uid, gid_t gid)

Arguments:

See PFS_chown().

Return values:

PFS_SUCCESS - Owner changed
PFS_FAILURE - Invalid path

8.2.9 FSLIB_claim

Description:

FSLIB_claim determines if the given path is in the current file system. The function may store one longword in the pathid structure cp member for future use.

Synopsis:

#include <pfs.h>

PFS_FSTATUS FSLIB_claim (PFS_PATHID *pathid, char *path, PFS_NAMESPACE namespace)

Arguments:

pathid	Partially resolved pathid structure. This structure will contain the root name in the fullpath member. The file system should use this and possibly the client path name to determine if it owns the path.
path	Client path.
namespace	Namespace in which path resides.

Return values:

PFS_EXISTS - File exists and is claimed by this library
PFS_NOEXIST - File does not exist but the parent path does and is claimed by this library.
PFS_UNCLAIMED_EXISTS - File exists but this library does not support it
PFS_UNCLAIMED_NOEXIST - File does not exist but the parent does. It is not supported by this library.
PFS_FAILED - Neither the file nor the parent exist in this library.

8.2.10 FSLIB_close

Description:

See PFS_close().

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_close (PFS_FID *fp)

Arguments:

See PFS_close().

Return values:

PFS_SUCCESS - File closed
PFS_FAILURE - Invalid file pointer

8.2.11 FSLIB_convert *

Description:

FSLIB_convert translates a filename from NOS format to native format. This function is used by PFS_getpathid() to resolve filenames prior to file lookup. If the file system does not require name translation this function should simply return PFS_SUCCESS.

The root directory is assumed to be located in the fullpath member of the pathid structure.

The resultant filename is returned in fullpath member of the pathid structure.

It is possible that this function will be merged with FSLIB_mapname.

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_convert (PFS_PATHID *pathid, char *path, PFS_NAMESPACE namespace)

Arguments:

pathid	Resolved pathid structure as returned by PFS_getpathid()
path	Client file path
namespace	Namespace in which path resides.

Return values:

PFS_SUCCESS - Name translated
PFS_FAILURE - Invalid parameters

8.2.12 FSLIB_create *

Description:

See PFS_create().

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_create (PFS_PATHID *pathid, mode_t mode, uid_t uid, gid_t gid,
PFS_CREATE_TYPE type, PFS_FID **fp)

Arguments:

See PFS_create().

Return values:

PFS_SUCCESS - File created
PFS_FAILURE - Invalid path or file exists and PFS_MAKENEW specified

8.2.13 FSLIB_dentpathid *

Description:

See PFS_dentpathid().

Synopsis:

#include <pfs.h>
#include <dirent.h>

PFS_FSTATUS FSLIB_dentpathid (PFS_FID *fp, struct dirent *dirent, PFS_PATHID *pathid)

Arguments:

See PFS_dentathid().

Return values:

PFS_EXISTS - Path exists as specified.
PFS_NOEXIST - Path does not exist but the parent path does (i.e. a new file specification).
PFS_FAILED - Neither parent nor path exists.

8.2.14 FSLIB_didpathid *

Description:

See PFS_didpathid().

Synopsis:

#include <pfs.h>

PFS_FSTATUS FSLIB_didpathid (PFS_CWD *dirid, char *path, PFS_NAMESPACE namespace, PFS_PATHID *pathid)

Arguments:

See PFS_didpathid().

Return values:

PFS_EXISTS - Path exists as specified.
PFS_NOEXIST - Path does not exist but the parent path does (i.e. a new file specification).
PFS_FAILED - Neither parent nor path exists.

8.2.15 FSLIB_diridfunc

Description:

See PFS_diridfunc().

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_diridfunc (PFS_DIRID_CMD cmd, char *root, unsigned long dirid, PFS_CWD *dirptr)

Arguments:

See PFS_diridfunc().

Return values:

PFS_SUCCESS - Directory ID translated
PFS_FAILURE - Invalid directory ID or no directory ID set open.

8.2.16 FSLIB_diridinit

Description:

See PFS_diridinit().

Synopsis:

#include <pfs.h>

void FSLIB_diridinit (PFS_DIRIDS_MATTER dodorods, unsigned long *diridptr)

Arguments:

See PFS_diridinit().

Return values:

None

8.2.17 FSLIB_filesize
8.2.18 FSLIB_ffilesize

Description:

See PFS_ffilesize().

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_filesize (PFS_PAHID *pathid, PFS_DATA_STREAM stream, off_t *size)

PFS_RETVAL FSLIB_ffilesize (PFS_FID *fp, off_t *size)

Arguments:

See PFS_ffilesize().

Return values:

PFS_SUCCESS - Return size is valid
PFS_FAILURE - Invalid file pointer

8.2.19 FSLIB_fsync

Description:

See PFS_fsync().

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_fsync (PFS_FID *fp)

Arguments:

See PFS_fsync().

Return values:

PFS_SUCCESS - File flushed
PFS_FAILURE - Invalid file pointer

8.2.20 FSLIB_getattr
8.2.21 FSLIB_fgetattr

Description:

See PFS_getattr().

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_getattr (PFS_PATHID *pathid, unsigned long mask, PFS_ATTR *attrp)

PFS_RETVAL FSLIB_fgetattr (PFS_FID *fp, unsigned long mask, PFS_ATTR *attrp)

Arguments:

See PFS_getattr().

Return values:

PFS_SUCCESS - Attributes updated
PFS_FAILURE - Invalid parameters

8.2.22 FSLIB_getcomment

Description:

See PFS_getcomment().

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_getcomment (PFS_PATHID *pathid, char *comment, int buflen)

Arguments:

See PFS_getcomment().

Return values:

PFS_SUCCESS - Coment returned
PFS_FAILURE - Invalid parameters

8.2.23 FSLIB_getdents

Description:

See PFS_getdents().

Synopsis:

#include <pfs.h>
#include <dirent.h>

PFS_RETVAL FSLIB_getdents (PFS_FID *fp, struct dirent *direntp, unsigned int nbytes, off_t *offset, PFS_NAMESPACE namespace, unsigned int bytesread)

Arguments:

See PFS_getdents().

Return values:

PFS_SUCCESS - Buffer written (including no entries)
PFS_FALURE - Invalid parameters

8.2.24 FSLIB_getextattr *
8.2.25 FSLIB_fgetextattr *

Description:

See PFS_getextattr().

This is an optional function. If the library does not support extended attributes the corresponding bit in the PFS_LIB_ENT structure should be clear and this routine need not be present.

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_getextattr (PFS_PATHID *pathid, PFS_EAOPS *eaopsp)

PFS_RETVAL FSLIB_fgetextattr (PFS_FID *fp, PFS_EAOPS *eaopsp)

Arguments:

See PFS_getextattr().

Return values:

PFS_SUCCESS - Extended attributes written to buffer
PFS_FAILURE - Invalid parameters

8.2.26 FSLIB_getprintident *

Description:

FSLIB_getprintident returns file identification information in the PFS_IDENT structure. This structure is used primarity by the print subsystem to identify a file.

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_getprintident (PFS_PATHID *pathid, PFS_IDENT *identp)

Arguments:

pathid	Resolved pathid structure as returned by PFS_getpathid().
identp	Pointer to return identification structure.

Returns:

PFS_SUCCESS - Print identification returned
PFS_FAILURE - Insufficient information

8.2.27 FSLIB_getsecurity *
8.2.28 FSLIB_fgetsecurity *

Description:

See PFS_getsecurity().

This is an optional function. If the library does not support security data the corresponding bit in the PFS_LIB_ENT structure should be clear and this routine need not be present.

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_getsecurity (PFS_PATHID *pathid, PFS_SECURSPACE securspace,
void *securp)

PFS_RETVAL FSLIB_fgetsecurity (PFS_FID *fp, PFS_SECURSPACE securspace, void *securp)

Arguments:

See PFS_getsecurity().

Return values:

PFS_SUCCESS - Returned attributes
PFS_FAILURE - Invalid parameters

8.2.29 FSLIB_init

Description:

FSLIB_init initializes the file system library. The routine is called once, when the file system is loaded. This routine should be given a universal name of the form <name>_init where <name> is the name of the file system if the library is to be dynamically loaded.

See PFS_init() for a description of dynamic library loading.

Synopsis:

#include <pfs.h>

PFS_FSTATUS FSLIB_init (PFS_LIB_ENT *libentp, FILE *log_file)

Arguments:

libentp	Pointer to next library entry slot.
log_file	File pointer for debug use. This may be a temporary debug aid for product qualification.

Return values:

None

8.2.30 FSLIB_lock

Description:

See PFS_lock().

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_lock (PFS_FID *fp, short type, off_t offset, short whence, off_t length,
PFS_WAIT_LOCK dowait, off_t *start)

Arguments:

See PFS_lock().

Return values:

PFS_SUCCESS - Lock set
PFS_FAILURE - Invalid parameters or lock conflict

8.2.31 FSLIB_lookup *

Description:

FSLIB_lookup will locate the file in the current file system.

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_lookup (PFS_PATHID *pathid, char *client_name)

Arguments:

pathid	Resolved pathid structure as returned by PFS_getpathid().
client_name	Specified a client name to translate by lookup. This is used for the Macintosh filename translation algorithm.

Return values:

PFS_SUCCESS - File located
PFS_FAILURE - File not found

8.2.32 FSLIB_lseek

Description:

See PFS_lseek().

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_lseek (PFS_FID *fp, off_t offset, int whence)

Arguments:

See PFS_lseek().

Return values:

PFS_SUCCESS - File position changed
PFS_FAILURE - Invalid parameters

8.2.33 FSLIB_mapname
8.2.34 FSLIB_fmapname

Description:

See PFS_mapname().

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_mapname (PFS_PATHID *pathid, PFS_NAMESPACE namespace, char *namebuf, int buflen)

PFS_RETVAL FSLIB_fmapname (PFS_FID *fp, PFS_NAMESPACE namespace, char *namebuf, int buflen)

Arguments:

See PFS_mapname().

Return values:

PFS_SUCCESS - Name translated
PFS_FAILURE - Invalid parameters or buffer too small

8.2.35 FSLIB_mkdir

Description:

See PFS_mkdir().

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_mkdir (PFS_PATHID *pathid, mode_t mode, uid_t uid, gid_t gid, unsigned long *dirid)

Arguments:

See PFS_mkdir().

Return values:

PFS_SUCCESS - Directory created
PFS_FAILURE - Invalid parameters

8.2.36 FSLIB_mpxclose

Description:

See PFS_mpxclose().

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_mpxclose (PFS_FID *fp)

Arguments:

See PFS_mpxclose().

Return values:

PFS_SUCCESS - File is close in the underlying file system
PFS_FAILURE - Invalid parameters

8.2.37 FSLIB_mpxopen

Description:

FSLIB_mpxopen will multiplex open a file previously multiplex closed. The open mode is stored in the PFS_FID structure and will be used to reopen the file in the same mode as previously open. Note that a file with locks can not be multiplex closed. This eliminates the need to attempt to reestablish lcoks within the file.

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_mpxopen (PFS_FID *fp)

Arguments:

fp	Previously open file pointer. The file will be reopened in the same mode as originally open.

Return values:

PFS_SUCCESS - File open
PFS_FAILURE - Invalid parameters

8.2.38 FSLIB_open

Description:

See PFS_open().

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_open (PFS_PATHID *pathid, PFS_STAT *statbufp, int oflag,
PFS_DATA_STREAM stream, PFS_MEM_MAP dommap,
PFS_FID **fp)

Arguments:

See PFS_open().

Return values:

PFS_SUCCESS - File open
PFS_FAILURE - Invalid parameters

8.2.39 FSLIB_purge
8.2.40 FSLIB_fpurge

Description:

See PFS_purge().

Note that PFS_fpurge is provided to support PFS_closeandpurge. The file pointer points to a closed file but the file pointer is still valid. Information in the file pointer should be used to guarantee the exact file opened will be deleted (care must be taken if the file is to be deleted by name as multiple versions and search paths may cause the name to be ambiguous). The file system library should provide some means to uniquely identify a file and store this information in the pathid/fid structure.

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_purge (PFS_PATHID *pathid)

PFS_RETVAL FSLIB_fpurge (PFS_FID *fp)

Arguments:

See PFS_purge().

Return values:

PFS_SUCCESS - File deleted
PFS_FAILURE - Invalid parameters

8.2.41 FSLIB_read

Description:

See PFS_read().

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_read (PFS_FID *fp, void *buffer, unsigned int nbytes, off_t offset, unsigned int *bytesread)

Arguments:

See PFS_read().

fReturn values:

PFS_SUCCESS - Bytes read (including none)
PFS_FAILURE - Invalid parameters

8.2.42 FSLIB_readdesc *

Description:

See PFS_readesc().

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_readdesc (PFS_FID *fp, unsigned int nbytes, off_t offset, PFS_DESC **desc)

Arguments:

See PFS_readdesc().

Return values:

PFS_SUCCESS - Data read
PFS_FAILURE - Invalid parameters

8.2.43 FSLIB_rename

Description:

See PFS_rename().

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_rename (PFS_PATHID *oldpathid, PFS_PATHID *newpathid)

Arguments:

See PFS_rename().

Return values:

PFS_SUCCESS - File renamed
PFS_FAILURE - Invalid parameters, conflicting file systems or conflicting namespace

8.2.44 FSLIB_rmdir

Description:

See PFS_rmdir().

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_rmdir (PFS_PATHID *pathid)

Arguments:

See PFS_rmdir().

Return values:

PFS_SUCCESS - Directory deleted
PFS_FAILURE - Invalid parameter, directory not empty

8.2.45 FSLIB_setattr
8.2.46 FSLIB_fsetattr

Description:

See PFS_setattr().

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_setattr (PFS_PATHID *pathid, PFS_ATTR *attrp)

PFS_RETVAL FSLIB_fsetattr (PFS_FID *fp, PFS_ATTR *attrp)

Arguments:

See PFS_setattr().

Return values:

PFS_SUCCESS - Attributes modified
PFS_FAILURE - Invalid paramters or file not writeable

8.2.47 FSLIB_setcomment

Description:

See PFS_setcomment().

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_setcomment (PFS_PATHID *pathid, char *comment)

Arguments:

See PFS_setcomment().

Return values:

PFS_SUCCESS - Comment written
PFS_FAILURE - Invalid paramters or file not writeable

8.2.48 FSLIB_setextattr *
8.2.49 FSLIB_fsetextattr *

Description:

See PFS_setextattr().

For a description of optional functions see FSLIB_getextattr().

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_setextattr (PFS_PATHID *pathid, PFS_EAOPS *eaopsp)

PFS_RETVAL FSLIB_setextattr (PFS_FID *fp, PFS_EAOPS *eaopsp)

Arguments:

See PFS_setextattr().

Return values:

PFS_SCCESS - Attributes modified
PFS_FAILURE - Invalid parameters or file not writeable

8.2.50 FSLIB_setsecurity *
8.2.51 FSLIB_fsetsecurity *

Description:

See PFS_setsecurity().

For a description of option functions see FSLIB_getsecurity().

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_setsecurity (PFS_PATHID *pathid, PFS_SECURSPACE securspace,
void *securp)

PFS_RETVAL FSLIB_fsetsecurity (PFS_FID *fp, PFS_SECURSPACE securspace, void *securp)

Arguments:

See PFS_setsecurity().

Return values:

PFS_SUCCESS - Data associated
PFS_FAILURE - Invalid parameters or file not writeable
PFS_NOTSUPPORTED - Security data not supported

8.2.52 FSLIB_stat
8.2.53 FSLIB_fstat

Description:

See PFS_stat().

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_stat (PFS_PATHID *pathid, PFS_DATA_STREAM stream,
unsigned long mask, PFS_STAT *statbufp)

PFS_RETVAL FSLIB_fstat (PFS_FID *fp, unsigned long mask, PFS_STAT *statbufp)

Arguments:

See PFS_stat().

Return values:

PFS_SUCCESS - Information obtained
PFS_FAILURE - Invalid parameters

8.2.54 FSLIB_statvfs
8.2.55 FSLIB_fstatvfs

Description:

See PFS_statvfs().

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_statvfs (PFS_PATHID *pathid, statvfs_t *fsbufp)

PFS_RETVAL FSLIB_fstatvfs (PFS_FID *fp, statvfs_t *fsbufp)

Arguments:

See PFS_statvfs().

Return values:

PFS_SUCCESS - Information obtained
PFS_FAILURE - Invalid parameters

8.2.56 FSLIB_sync

Description:

See PFS_sync().

Synopsis:

#include <pfs.h>

void FSLIB_sync (void)

Arguments:

None

Return values:

None

8.2.57 FSLIB_ftruncate

Description:

See PFS_ftruncate().

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_ftruncate (PFS_FID *fp, off_t size)

Arguments:

See PFS_ftruncate().

Return values:

PFS_SUCCESS - File truncated
PFS_FAILURE - Invalid parameters or file not writeable

8.2.58 FSLIB_unlock

Description:

See PFS_unlock().

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_unlock (PFS_FID *fp, off_t offset, short whence, off_t length)

Arguments:

See PFS_unlock().

Return values:

PFS_SUCCESS - Range lock removed
PFS_FAILURE - Invalid parameters or no range locked

8.2.59 FSLIB_unmap

Description:

See PFS_unmap().

Synopsis:

#include <pfs.h>

void FSLIB_unmap (PFS_FID *fp)

Arguments:

See PFS_unmap().

Return values:

None

8.2.60 FSLIB_utime
8.2.61 FSLIB_futime

Description:

See PFS_utime().

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_utime (PFS_PATHID *pathid, void *timebufp)

PFS_RETVAL FSLIB_utime (PFS_FID *fp, void *timebufp)

Arguments:

See PFS_utime().

Return values:

PFS_SUCCESS - Time modified
PFS_FAILURE - Invalid parameters or file not writeable

8.2.62 FSLIB_write

Description:

See PFS_write().

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_write (PFS_FID *fp, void *buffer, unsigned int nbytes, off_t offset,
unsigned int byteswritten)

Arguments:

See PFS_write().

Return values:

PFS_SUCCESS - File written
PFS_FAILURE - Invalid parameters or file not open for write.

8.2.63 FSLIB_writedesc *

Description:

See PFS_writedesc().

Synopsis:

#include <pfs.h>

PFS_RETVAL FSLIB_writedesc (PFS_FID *fp, off_t offset, PFS_DESC *desc)

Arguments:

See PFS_writedesc().

Return values:

PFS_SUCCESS - File written
PFS_FAILURE - Invalid parameters or file not open for write access.

9 STANDARD LIBRARIES

PFS provides two standard file system libraries for VMS, ODS2 DOS library and ODS2 MAC library. These two libraries are very tightly coupled. The partion is provided for functional separation only. The underlying file storage is mapped on the ODS2 file system for both libraries.

9.1 ODS2 DOS library

The ODS2 DOS library is responsible for all DOS related file access in the ODS2 file system. The library provides name translation, attribute storage, data storage and Lan Manager security data storage.

9.1.1 Namespace

The ODS2 DOS library uses the following algorithm for mapping DOS filenames to VMS filenames:

```
Convert lowercase to uppercase
[What do we do about underscore in filename ??]
FOR each VMS illeagal character DO
    Insert two underscores (__)
    Convert character to hex ASCII code
IF extention EQL "DIR" insert two underscores (__)

For example, name@.dir would be converted to NAME__40.__DIR
```

The ODS2 DOS library uses the following algorithm to map VMS filenames to DOS filenames:

```
Convert uppercase to lowercase
FOR each double underscore DO
    Remove underscore
    Convert hex ASCII code to character
IF extention EQL "__dir" remove double underscore

For example, NAME__40.__DIR would be converted to name@.dir
```

9.1.2 Attributes

The ODS2 DOS library stores DOS file attributes in a VMS applicaton ACE. The format of this ACE is shown below.

ace$w_flags		ace$b_type	ace$b_size	0
ace$w_facility_flags		ace$w_facility		4
pfs$b_flags	pfs$b_version	pfs$b_type	pfs$b_size	8
dos$l_file_size		dos$r_attributes		12
dos$q_modify_time		dos$l_file_size (cont)		16
dos$q_modify_time (cont)				20
dos$l_create		dos$q_modify_time (cont)		24
dos$l_lastaccess		dos$l_create (cont)		28
dos$l_lastmod		dos$l_lastaccess (cont)		32
		dos$l_lastmod (cont)		36

The ACE contains DOS file times in DOS format. The file size is calculated by reading the file and counting record lenghts (VAR file format). This count is marked with the file modification time such that it need not be recalculated unless the file has been modified.

The DOS$L_LASTACCESS field holds the time at which the file was last accessed, regardless of modification. This is an expensive field to maintain and may not be supported. This field would need to be set for read-only files (such as application images) each time the file was opened or closed.

9.1.3 Security

The DOS file system has no security data associated with it. However, DOS clients are supported by Lan Manager which defines its own security model. The ODS2 DOS library will support storage and retrieval of security information for Lan Manager.

NOTE

> The file system library has no concept of the security model, association of security data nor security data hierarchy. This may force the server to querry all members of a file path in search of security data (Lan Manager security inheritance). This access may nullify the advantages of using path caches to optimize file access as the path may need to be processed member by member to obtain security data.
>
> Alternately, the file system library may understand the security data hierarchy and obtain security data above the requested object. The path cache may perform better in this environment.

9.1.4 Datapaths

The ODS2 DOS library supports native DOS file formats (stream) as well as native VMS file formats (VAR, FIXED, INDEXED, etc). The library will use RMS for complex file organization (INDEXED). The library will use QIO for simple file organization (VAR, VFC, FIXED, STREAM XX).

The ODS2 DOS library will only create files in STREAM format. If a particular file format is desired on the host an external file conversion utility must be used.

9.1.5 FID cache

To facilitate ACE lookups and security data processing the ODS2 DOS library maintains recently accessed file headers in a cluster-wide distributed cache. This cache is used to obtain the DOS ACE on directory search functions as well as file access.

The FID cache is invalidated when a file is deleted or header data is modified. The local node cache invalidates the specific entry involved. Cluster-wide invalidation is limited to the hash chain which contains the entry in order to limit the amount of distributed locks.

9.1.6 Directory cache

The ODS2 DOS library maintains a cache of directory entries. This cache presents directories as fixed length structures to support the Search SMB. It is currently not clear whether this cache will be necessary given the Lan Manager directory structures.

9.1.7 Path cache

The ODS2 DOS library maintains a cache of recently translated paths. This cache is used to improve performance of file lookups. This cache contains path name to directory ID (DID) translations. Host security data for the directory is also stored here to imporve host security checking. It should be noted that VMS (RMS in particular) only stipulates security checking on the last member of a path. Was this not the case the path cache would not be useful for security data.

9.2 ODS2 MAC library

The ODS2 MAC library supports Macintosh clients. The library provides filename mapping, attribute storage, security data storage and Macintosh multifork file formats. The library maps all functions to the VMS ODS2 file system.

9.2.1 Namespace

The ODS2 MAC library stores Macintosh filenames in an application ACE associated with the file. This arrangement provides consistency between Macintosh filenames and Macintosh files. However, VMS provides very little to support filename lookups by anything other than VMS directories. It is necessary to establich external structures to provide this translation. The performance of the Macintosh file system is extremely dependent on the efficiency of these external structures.

NOTE

> The design of the ODS2 MAC library creates all external structures in memory, on demand. There are no external files to buffer the building of these in memory structures. While this design provides a high degree of consistency there are huge differences in perfomance for functions which hit the in memory structures and those for which the structures must be built. The overall performance will depend of the availability of memory to hold these structures and the frequency of invalidates.

In order to limit the length of directory searches for a Macintosh file there is a strict relationship between the Macintosh filename and the VMS filename (and hence the DOS filename). To allow an unrestricted filename relationship with no external structure file support would result in unacceptable performance for moderate to large directories. Preliminary performance measurements show that header lookups take between 5-15ms (VAX3100 M48). This would mean up to 30 seconds to locate a file in a directory of 2000 files (actual measurement).

File creates are done by converting the Macintosh name to a VMS name and creating the file. The file create will either a) fail due to existing file of the same name (and version) b) succeed with a higher version warning or c) succeed. Case a) and b) are failures. Macintosh filename creates are mapped to VMS names using the following algorithm:

```
Split filename and extension at first '.'
Remove all DOS illegal characters
Truncate to 8.3 format
UNTIL unique DO
```

```
    Convert DOS name to VMS name
    IF file exists replace last DOS character with digit 0-9
    IF replacement at '9' then filename conflict
END

For example, "Macintosh Filename" would be converted to MACINTOS., MACINTO1.,
MACINTO2., etc.
```

Macintosh filename lookups are performed by converting the Macintosh filename to a VMS wildcarded pattern and then searching for this filename. For each match the Macintosh ACE is examined for a matching filename. Macintosh filename lookups are converted to VMS names using the following algorithm:

```
Split filename and extension at first '.'
Remove all DOS illegal characters
Truncate to 8.3 format
Remove last DOS character
Convert DOS name to VMS name
Append VMS wilcard to filename and extension (*)
Search directory and match Macintosh filename against name stored in ACE

For example, "Macintosh Filename" would be converted to MACINTO*.*.
```

NOTE

It is possible that this algorithm may be modified for Macintosh-only installations. The conversion to DOS legal format could be replaced by a conversion to VMS legal format. This change may result in more meaningful filenames to Macintosh/host users.

9.2.2 Attributes

The ODS2 MAC library stores Macintosh attributes in an application ACE. This ACE shares the same physical VMS ACE as the DOS ACE. The header is shown here for completeness but is actually the same ACE header as for the DOS ACE. The PFS header is present on the Macintosh ACE, i.e. the data starting at byte 8.

ace$w_flags		ace$b_type	ace$b_size	0
ace$w_facility_flags		ace$w_facility		4
pfs$b_flags	pfs$b_version	pfs$b_type	pfs$b_size	8
longName				12
shortName				44
parentID				56
fileID				60
finderInfo		attributes		64
finderInfo (cont)				68
offspringCount		finderInfo (cont)		96
createDateTime				100
modifyDateTime				104
backupDateTime				108
ownerID				112
groupID				116
proDOSInfo	worldRights	groupRights	ownerRights	120
proDOSInfo (cont)				124
dataForkLength			proDOSInfo (c)	128
resForkLength			dataForkLen (c)	132
			resForkLen (c)	136

The times stored in the ACE are in Macintosh format. The fileID and parentID fields are modified VMS FID format. The sequence field and relative volume number fields are shortened to map to 32 bits. This requires a direct index file lookup to translate the file number back to the VMS FID. Again, this design choice has been made to eliminate the need for external files describing the Macintosh environment.

The library does not maintain the file creation times, modification times or backup times in the Macintosh ACE. This is the responsibility of the server.

9.2.3 Security

The ODS2 MAC library stores Macintosh security data in the appropriates fields of the Macintosh ACE (described above). The data is not interpretted.

Macintosh has a concept of "giving folders away". To Macinsosh, this is as simple as changing the owner of a directory. As file protection in Macintosh is inherited there is no further modification required. VMS has no such concept. It may be required for the server to modify each file in the directory to be "given away". This is likely a time consuming task. PFS supports this function by providing functions to change the host owner of a file. The server must use a combination of directory enumerates and change owner functions to complete the host mapping of security changes associated with the new owner.

9.2.4 Datapaths

The ODS2 MAC library supports the native VMS file formats are Macintosh data fork only files. The resource length for these files will always be returned as zero. The library supports a special file format to handle two forks per file. This file format is described in Appendix E.

The library will create files in STREAM format UNLESS a resource fork is created prior to the data fork. In this case the Macintosh file format is used on create. If a resource fork is added to a STREAM file it will be converted to a Macintosh format file (by remapping block 1 of the file). Non-stream files can not have a resource fork added to them with out conversion to stream format first. The library does not support this.

9.2.5 Name cache

The ODS2 MAC library maintains a cache of recently translated Macintosh filenames. This cache can only be used for file lookups as a miss in the cache is never sufficient to declare a file does not exist. (However, a hit would indicate the file already exists). The cache is tightly coupled to the ODS2 DOS FID cache and invalidates are done in the exact same manner.

Appendix A - VMS ODS level 2 file system

This appendix contains information about the VMS ODS level 2 file system. This information is presented to provide context for evaluating the mapping between NOS file systems and ODS2. The information herein is meant to be complete. Only the attributes and semantics relavent to NOS file systems are presented.

A.1 Directory structure

ODS2 uses hierarchial directory structure. Directory files appear as contiguous variable length files with variable record format. Records can not cross record boundaries.

Directory entries are packed within disk blocks. The records are arranged alphabetically. The records will be shuffled when a new entry is added to maintain alphabetic order.

Directory entries contain filenames, version numbers and file identificaton information. File identification information (FID) is associated with a specific version of each file. The individual version to FID mapping records are stored in order following the filename record.

Directory nesting depth is not limited by ODS2. However, RMS limits the depth of a directory specification to 8 levels. Unlimited depth may be processed by RMS with the use of concealed logical names. The names specify at least the part of the directory tree which would exceed 8 levels.

VMS BACKUP is limited to 8 levels in a directory tree. There is no warning that additional directory levels are being skipped.

A.2 File structure

ODS2 associates file attributes and generic file meta data with files in the file header. The file header is stored in the index file and is accessed by the file identification number (FID). The FID is a 48 bit structure which contains the relative volume number (volume number of bound volume set), sequence number (used to identify the instance of the file number) and file number. The file number is the index into the index file.

ODS2 provides direct file access by FID. The FID is sufficient to uniquely identify a file on a volume. No additional directory information is required.

The file header maps the virtual blocks in the file to logical blocks on the disk. Only one set of file retrieval pointers are maintained. This set coresponds to the data portion of the file.

The file header contains ACE information. This information specifies aditional security information, RMS attributes, or application specific information. PFS makes extensive use of the application ACE capabilities of ODS2 to associate NOS data with a file.

A.2.1 Access Control Lists (ACL)

ODS2 provides storage of information in the file header. Multiple file headers will be used if this information will not fit in the primary header. Additional fil headers are allocated from the index file and chained to the primary header. (This reduces the number of file which can be stored on the volume).

Access control lists are lists of Access Control Elements (ACE). While the term implies the information associated with the list controls access to the file, generic information may be stored in ACEs.

VMS XQP (extended QIO processor) provides access to information strored in the file header. ACEs may be added, deleted or modified. The XQP accepts multiple ACE manipulation functions per invocation. However, not all combination of functions will result in predictable results. The XQP maintains context information which is used to address a specific ACE. There is no direct address capability, however, a specific ACE may be located and the current pointer set to it. Once an ACE is located it may be modified or deleted. It is possible to locate multiple ACEs and modify them in a single invocation but FIND functions must be interspersed within the function list to maintain correct ACE positioning. Multiple new ACEs may be added without too much trouble.

A.3 File attributes

ODS2 maintains a set of file attributes. These attributes specify the file organization, record format, access times, etc. While this information may be common with some of the NOS file attributes, a complete mapping is not possible.

A.3.1 File creation time

The time at which a file is created is stored in the file header. This time is set when an XQP create function is executed.

A.3.2 File revision time

The file revision time is modified when a write function or modify function is executed. Initially the revision time is set to the creation time.

A.3.3 File backup time

The VMS BACKUP utility sets the time at which a file is backed up. This field may be used to determine if a file has been modified since the last backup.

A.3.4 File expiration time

ODS2 allows a specification of time at which file may not be accessed.

A.3.5 File organization

ODS2 itsefk does not provide any support for file organization. Files are simply collections of logical blocks. However, VMS RMS does provide various file organizations and in order to process a file the organization must be known and understood. RMS provides three file organizations:

SEQUENTIAL	Records are stored sequentially. Access is allowed either sequentially or by record address. Records may be variable length or fixed.
RELATIVE	Records are stored in fixed length blocks addressable via block number. The records within the blocks may be of any size. The record blocks may or may not be related to disk blocks.
INDEXED	Records are chained to an index key. Multiple indicies may be used. Records may be variable length.

A.3.6 Record structure

ODS2 itself does not provide any record structure. Files are accessible via 512 bytes blocks. However, VMS RMS does provide record format and in order to process a file this record format needs to be known and understood. Neyther ODS2 nor RMS provide any means for determining how much data is actually stored in a file. The only information available is where the current end of file mark is. RMS provides the following record formats:

VARIABLE	Records are prefixed by a count. The count is one word in length and counts the actual number of bytes in the record. All records are aligned on a word boundary (so there may be an extra byte in the actual record as stored in the file). Most VMS text files are stored in this format.

VFC	Variable with fixed control. The record contains a fixed number of bytes followed by a variable format record. File created with DCL OPEN, DCL WRITE and DCL CLOSE will be of this format.
FIXED	Fixed length records. The records may be of any size although 512 is most common. VMS images will be of this format.
STREAM	No record prefix. Records are terminated with <CR>, <LF> or <FF>, <VT> or <CR><LF>.
STREAM_CR	No record prefix. Records are terminated with <CR>.
STREAM_LF	No record prefix. Records are terminated with <LF>.
UNDEFINED	No record format.

A.3.7 Record attributes

ODS2 does not define any record attributes. As with record formats, the record attributes must be known and understood to process the file. VMS RMS defines the following record attributes:

BLK	Records may not cross block boundaries. Blank space may be found at the end of disk blocks.
CR	<CR><LF> to be prefixed to record when displayed on carraige control device. (Not applicable to file service).
FTN	Fortran carraige control. (Not applicable to file service).
PRN	Print carraige control. (Not applicable to file service).

A.4 File allocation

ODS2 allocates file in groups of disk blocks called clusters. The cluster size is determined when the disk is initialized.

Files may be allocated contiguous meaning all logical blocks of the file are contiguous.

A.4.1 File header

The file header contains all information stored about a file. The file attributes, ACEs, retrieval pointers and linkage to extension headers is stored in the file header.

File headers reside in the index file. Prior to processing a file, the index file must be read to obtain at least the primary header. This read is in addition to filename processing information located in the directory file.

A.4.2 Index file

ODS2 volumes contain an index file, INDEXF.SYS. This file is present on each volume, including each volume in a bound volume set. The index file is used to store file headers. There is a set of fixed length file headers which may be chained to store information about a file. The index file is addressed with the file identification number from the FID or DID.

A.4.3 Bitmap file

ODS2 contains a bitmap file which marks disk clusters either in use or free. The bitmap file is rebuilt if the volume is improperly dismounted. The bitmap is rebuilt by reading the index file and processing each file's retrieval pointers. In this manner, multiple linkages to disk blocks can be eliminated.

A.4.4 Quota file

ODS2 provides disk usage quotas for specific users. Any file allocations are subtracted from the user's quota. The user will be prevented from allocating more blocks than specified in the quota and overdraft limits provided by the quota file.

A.5 Security model

ODS2 security is provided in two levels, user identification (UIC) and ACLs. Four classes of users are defined:

OWNER	User whose UIC matches the file owner
GROUP	Users whose group portion of their UIC matches the group portion of the file owner UIC.
SYSTEM	Users in the system group [1,].
WORLD	Any user who does not fall in one of the above.

For each class there are four access modes:

READ	Users may read file or perform wildcard directory lookups.
WRITE	Users may write file or change its attributes.
EXECUTE	Users may execute file or perform specific directory lookups.
DELETE	USers may delete a file.

ACLs provide the same basic access with one additional access:

CONTROL	??

ACLs differ from UIC protection in that they are checked against a user's rights identifiers, not the user's UIC. In this manner groups of users may be granted or denied access independent of UIC group.

ACLs are applied after the UIC check is made. Therefore a user may be given access based on UIC even if the user is denied access based on the ACL.

Appendix B - MSDOS FAT file system

This appendix provides information about the MSDOS FAT file system. This file system is used in MSDOS clients. While many functions the server needs to provide are outside the scope of the FAT file system, many are directly related to the structure. This appendix is provided to present the issues which relate to the FAT file system.

B.1 Directory structure

MSDOS FAT provides variable length directories consisting of fixed records. Each record corresponds to a file. All file information is stored in the directory entry with the exception of file allocation table entries (FAT) which are pointed to by the directory record.

The filename is limited to 8.3 format, described in section 7.3.

The directory is not arranged in any partcular order. The directory expands as files are added and does not shrink once files are deleted. A given file's directory entry will occupy exactly the same position in the directory as long as the file exists. Once a file is deleted, its slot in the directory is free for reuse.

NOTE

> The Search SMB is very much dependent on this directory structure. The success of a given server implementation is largely determined by the degree of consistency between the server's virtual directory stucture and that of MSDOS FAT. This structure can be seen in LanManager's implementation of the Search SMB.

B.2 File structure

MSDOS FAT files contain one data stream only. There are no extended attributes associated with the file. The file has no record structure.

B.3 File attributes

MSDOS FAT provides a set of five file attributes, described in section 7.3. There is only one file modification time stored with the file. This time is initially set to the file creation time.

B.3.1 Modification time

The MSDOS FAT file system saves the time at which a file is created or modified. Once a file is modified, the original file creation time is lost.

B.4 File Allocation

The MSDOS FAT file system allocates file blocks in groups called clusters. The size of a cluster is set at volume initialization time. For each cluster there is a 12 or 16 bit field in the File Allocation Table (FAT). 12 bit FATs are used for small volumes (less than 20740 blocks). 16 bit FATs are used for large volumes. The largest volume supported by FAT is approximately 10M bytes.

B.5 Security model

MSDOS FAT file system provides no native security. The security associated with an MSDOS client will be that of the server. If the MSDOS client is served by LanManager the security requirements will be those of LanManager. If the MSDOS client is supported by AFP the security model will be that of AFP.

Appendix C - Macintosh HFS file system

This appendix provides information about the Macintosh Hierarchial File System (HFS). This information is provided as a reference to the requirements placed on the server and hence on the file system.

C.1 Directory structure

Macintosh maintains a hierarchial directory structure. The directory information is stored in the catalog tree on each volume.

Filenames are limited to 32 characters and are described in section 7.3.

Each directory is assigned a unique 32 bit value which may be used to directly reference the directory. This information may be held by Macintosh applications including the Macintosh Finder.

A server file system must be capable of associating a 32 bit ID with each directory and provide direct access to the directory by this ID.

C.2 File structure

Macintosh maintains various file attributes associated with the file. The information is stored in the directory entry for the file. Macintosh assigns a unique 32 bit ID to each file. This ID may be stored in "alias" entries in V7 Macintosh file systems. Starting with V7, files must be accessible via this 32 bit ID. V7 clients can specify that a file ID is o be "swapped" between two files.

A server must be capable of assigning a unique 32 bit ID to each file and directly accessing the file by its ID. The server file system must either swap the file IDs on request or swap the data associated with each file ID.

Macintosh also supports two data streams per file. There are two sets of mapping pointers for each file.

A server file system must be capable of associating two data streams with each file.

C.2.1 Data fork

Macintosh files have a data stream associated with them. This data stream contains nromal file data and is accessible to all clients supported by a server.

C.2.2 Resource fork

Macintosh also associates a resource stream with the file. This stream is Macintosh specific and is assumed to have a specific format. The information stored in this stream is of little or no use to other client types. This stream may not be addressible to non Macintosh clients.

C.3 File attributes

Macintosh maintains a set file file attributes stored in the directory entry for the file. These attributes describe access to the file and file visibility. The Macintosh file attributes are described in section 7.3.

C.3.1 File creation time

Macintosh stores the time at which a file is created. This time will not be modified. All Macintosh times are signed 32 bit quantities designating the time before or after 00:00:00 January 1, 2000.

C.3.2 File modification time

Macintosh stores the time at which a file is modified. This time is initially set to the creation time.

C.3.3 File backup time

Macintosh also stores a file's backup time. This time may be set and used by external backup utilities.

C.4 File allocation

Macintosh allocates groups of disk blocks into allocation blocks. The size of the allocation block is set at volume intializatino time. Groups of allocation blocks are stored in records called extents. Each extent is an allocation block number followed by the count of blocks in the extent. The first set of extents (3 records) is stored in the directory entry for the file. If a file requires additional extents they are stored in the extents tree. The extents tree is arranged as a set of index nodes containing three extent descriptors. The index nodes are kept sorted by file ID and file allocation block number.

This arrangement is conceptually similar to ODS2s retrieval pointers.

C.5 Security model

The Macintosh file system provides no native security. The security associated with a Macintosh client will be that of the server. Currently Macintosh clients are only served by AFP.

Appendix D - FSI interface

This appendix was written from notes taken during the initial review of the LMU FSI implementation. The structure and function of the FSI is described. Various notes about the application of the design to the VMS file system are included.

D.1 General Architecture

The LMU File System Interface (FSI) is designed to be used with multiple back end file systems. The file system selection is based on the concept of "path" ownership. This design will allow multiple file systems to be used in a server environment. The basic assumption is that various file systems can be used anywhere in the UNIX file system space. The path is the client file name translated into the server file system's semantics.

The FSI routines use a function dispatch table to execute file system specific tasks associated with the path's file system. Most of these routines are mapped one-to-one with the FSI routines. This scheme partitions the file system into two levels, general file access (performed at the FSI layer) and file system specific access (performed thru dispatch table). Functions common to all file systems are performed at the FSI layer. Functions to perform file system specific tasks are collectively known as "libraries".

NOTE

> The FSI functions use UNIX features and assume that UNIX is under the file system (errno and UNIX error numbers). This assumption greatly reduces the overall effect of the library partition.
>
> The concept of file system libraries is further reduced by the use of "special libraries" which map FSLIB functions to special FSI functions, most notably the FSI_setvmtime(), FSI_update_dt() and FSI_checkvolume() which map to MACUTILS library FSLIB functions FSLIB_chdir(), FSLIB_fchdir() and FSLIB_access(), respectively. While this provides dynamic loadable support for MAC style access, it is certainly to be viewed as somewhat less than clean. This library mapping is handled by a special check in the INIT_ENTRY loadable library support. The MAC library is not otherwise mapped. We do not have sources for this library extension.
>
> It would probably be best if the FSI routines directly handled MAC extensions.

D.1.1 File Descriptor Multiplexing

The FSI implements "file descriptor multiplexing" to prevent client access failure due to server process file descriptor resource depletion.

The FSI makes calls available to perform multiplexing and to provide notification when automatic multiplexing is done.

NOTE

> This mechanism seems to be UNIX specific and may not be required with SVR4. It would appear that this service really belongs in the file system library level, not the FSI. The FSI should be a platform independent interface and file multiplexing may be a phenomenon peculiar to UNIX.

Multiplexing will occur when an FSLIB_open call fails and returns the UNIX errno ENFILE (file table overflow) or EMFILE (too many open files). ENFILE will result in a call to FSI_needinodes which uses an internal count to determine how many files to close based on the number of times the function is called (the more times it is called the greater the number of files it will try to close at once). There is a limit of calling the function 4 times (magic number declared in multiple routines, forceopen, opencreate, FSI_needinodes). EMFILE will result in a call to a local routine (close_a_file) which will close FSI_closecount files. This variable is set by INI package calls and is fixed for each call. Both mechanisms result in a call to FSLIB_mpxclose to actually close open files.

D.1.2 Volume Services

The FSI does not provide volume level services. It must assume that the server layer has some knowledge of volumes and path names to volume directories. The server layer must also maintain information about volume status.

D.1.3 Directory IDs

The FSI has functions to support MAC style directory IDs. The interface is somewhat primitive and does not appear to be fully implemented. The current implementation returns a UNIX pathname of <root>/n for pathnames to MAC directories, where <root> is the UNIX pathname to the volume root and n is the directory ID number. This would likely force the listed "folder" names to be the same as their directory ID. While this may provide file storage it most likely would not be viewed as acceptable. The FSI also does not enforce this name convention of directories it creates. I would assume it would be left to the server to assign a directory ID to the FPCreateDir and pass this ID as the directory name to FSI_mkdir. The original folder name would either be stored by the server in its own database or lost. I would further assume that the server would have to convert the directory name to its ID for return to the client on FPOpenDir. If the server stored the actual name it would need to convert directory names on FPEnumerate.

NOTE

> Certainly this mechanism needs to be changed. In the VMS environment, the file ID provides a unique 24 bit number (32 with RVN) which could be used as the directory ID. It is not clear whether direct access via this FID is provided.
>
> A similar mechanism in the UNIX/OSF space needs to be investigated.

D.1.4 Namespace

The FSI has some knowledge of namespace, however, it is not implemented. The FSI could attempt to "claim" a path in the client namespace and convert the name to UNIX. The FSI could also return directory entries in the client namespace. The FSI passes all namespace issues to the underlying file system. This may result in many duplicated functions. While it may be less efficient a more general architecture would specify the native namespace as used by the FSI and require all FSLIB routines to operate in this namespace, converting names as required. The FSI would then convert names to the client namespace as required.

D.1.5 Streams

The FSI supports two streams per file, the primary stream (data stream) or Macintosh resource stream. The implementation of streams is left to the library. The FSI will specify the stream on file open calls. The basic assumption is only one stream may be accessed per open file and the stream must be specified at open time. This allows implementation of separate streams in separate files or both streams in one file. If a file is to have a resource stream it must also have a primary stream. The primary stream is always created when a file is created, regardless of the stream specified.

D.1.6 Extended Attributes

The FSI implements support of OS/2 extended attributes. This support is provided at the FSI level, not in the libraries. The library open function allows three streams, unlike the FSI open function. The FSI level uses library read and write functions to access the data in the extended attribute stream and interprets the data directly. The entire file containing the stream is locked for the duration of attribute access.

Extended attributes have an ASCII name and non specified data value associated with them. They are referenced by name and may be added, deleted or modified. Access to the attributes is provided thru two parallel structures, one specifying the name of the attribute to return and the other containing the return attribute (get functions) or attribute and value to add, modify or delete (set functions).

Attributes are stored in the stream as an array of attribute structures followed by the attribute name followed by the attribute value. The structure contains a field which indicates the total size of the structure plus name and value length. This size can be used to calculate the offset of the next attribute. Deleted attributes are marked in the EA header and the space may be used to store a new attribute, if it fits. If no slots are found the stream will be extended to hold the new attributes. Attributes are not sorted. Deleted attributes are marked by a NULL name pointer. The header will hold 10 deleted attribute pointers. The rest must be found by searching the attribute list.

NOTE

The support of extended attributes is in rough shape. The attribute stream is read into a static buffer and specified attributes are copied to the return structure as needed. This design relies on non-preemtive scheduling and single process access. A lock would be required to synchronize extended attribute functions. There are file access calls which use separate static buffers (one for the EA header and one for the data). These file read functions can not cause a process switch. While the current LMU tasker may provide for this it is not clear this is desired in a general I/O environment. If the tasker is changed to allow process switch while read data is fetched from disk this code could break and allow a second process to overwrite the EA header or data buffers.

Certainly the support of extended attributes needs to be moved to file system libraries.

Extended attributes are stored in a file with the suffix ".r". It would appear this file can be opened directly, however, directory enumerates specifically suppress listing the files. It is not clear what would happen if a user created a file with the ".r" suffix. The UFS library open function checks for a ".r" suffix and if present opens the file as is. If it is not the filename is appended with ".r". This would result in "FILENAME.EXT.R". While UNIX may allow this filename, VMS would not. I would guess that if a user created a ".r" file, it would be suppressed on directory enumerates.

The code references the support of extended attributes as EAHACK. It is possible this support was added in a last minute fashion and will be reworked.

D.1.7 FSI Routine Classification

The FSI routines can be grouped into 5 major classes, directory access functions, file access functions, file attributes functions, path access functions and general support functions.

Directory access functions

FSI_chdir
FSI_diridfcn
FSI_diridini
FSI_getcwd
FSI_getdents
FSI_mkdir
FSI_rmdir

File access functions

FSI_access
FSI_close
FSI_copyfile
FSI_create
FSI_delete
FSI_fsync
FSI_lock
FSI_lseek
FSI_open
FSI_purge
FSI_read
FSI_rename
FSI_sync
FSI_truncate
FSI_unlock
FSI_utime
FSI_unmap
FSI_write

File attributes functions

FSI_chmod
FSI_chown
FSI_getattr
FSI_geteas
FSI_getcmnt
FSI_filesize
FSI_setattr
FSI_setcmnt
FSI_seteas

Path functions

FSI_getpathid
FSI_fullpath
FSI_shortpath
FSI_treetop

General support functions

FSI_init
FSI_mapname
FSI_mpxclose
FSI_needfds
FSI_needinodes
FSI_setlognores
FSI_setnotifympx
FSI_stat
FSI_statvfs

D.2 PATH ID (FSI_PATHID)

Path IDs are structures which describe the path to a file. The path has attributes associated with it which may be the parent directory attributes.

Many calls operate on a path ID which contains the following information:

Function pointers

This field contains a pointer to the function dispatch table for the file system which "owns" this path.

Full path name

This field contains a string of fixed length to hold the full file name in the syntax of the server file system.

Short path name

This field contains a pointer into the full path name buffer which points to the start of the path which needs to be resolved, i.e. the point past the current default directory.

End tree top

This field contains a pointer into the full path name buffer which points to the start of the path beyond the "tree top" (volume directory on the server file system). This field must be set by the server application.

FSI flags

This field points to the FSI flags of the file system which "owns" this path. The flags include the following information:

File system is real UNIX file system
Resource forks are supported
Extended attributes are supported
Case sensitive file names are supported
File system is mapped to another file system (alias)
Mask of FSI status elements supported
Mask of FSI attributes supported

Status

This field contains the FSI status structure. This structure contains the following information:

Mask of which elements are valid
UNIX stat() function structure
File generation number
Data stream identifier (resource, data, attributes)
Parent INODE structure (this must assume that UNIX file system is present)
Parent generation number
Count of entries in directory
Count of files in directory
Count of directories in entry
File attributes

Directory ID pointer

This field is a pointer to a cell which contains the current directory ID generation number to be used with MAC variable ID format AFP calls.

D.2.1 File ID (FSI_FID)

Many calls operate on a file ID, a structure which contains the following information:

Function pointers

This field contains a pointer to the function dispatch table for the file system which "owns" this path.

File status

This field contains the FSI status structure. This structure contains the following information:

Mask of which elements are valid
UNIX stat() function structure
File generation number
Data stream identifier (resource, data, attributes)
Parent INODE structure (this must assume that UNIX file
system is present)
Parent generation number
Count of entries in directory
Count of files in directory
Count of directories in entry
File attributes

File descriptor

This field contains the file system descriptor (UNIX). The descriptor may be marked as closed if multiplexing has occurred.

File descriptor information (low, high, closed)

This field indicates which type of file descriptor this file is associated with.

Data stream identifier

This field indicates which data stream is being accessed, primary stream (data stream), resources stream or attributes stream.

Current file offset

The current file offset is preserved in case the file is closed due to file descriptor multiplexing. The file will be reopened and positioned here when the file is next accessed.

Open file reference count

This field contains the count of file IDs which point to this file.

Multiplex control count

This field contains the count of how many times the file is currently ineligible for multiplexing. The file may be closed if the count is zero.

Open flags

The file open mode must be preserved in case the file descriptor is closed due to multiplexing. The file will be reopened using this mode when it is next referenced.

File flags

Locking flag and file written flag.

File mapping information

[Need to find out how this works.]

FSI flags

This field points to the FSI flags of the file system which "owns" this path. The flags include the following information:

File system is real UNIX file system
Resource forks are supported
Extended attributes are supported
Case sensitive file names are supported
File system is mapped to another file system (alias)
Mask of FSI status elements supported
Mask of FSI attributes supported

End tree top pointer

This field contains a pointer into the full path name buffer which points to the start of the path beyond the "tree top" (volume directory on the server file system). This field must be set by the server application.

Full path name

This field contains a string of fixed length to hold the full file name in the syntax of the server file system.

D.3 ROUTINE SUMMARY

D.3.1 FSI_access

Description:

This function will determine if a file may be accessed according to the mode specified.

Synopsis:

FSI_Access (FSI_PATHID *pathid, int perms)

Algorithm:

```
BEGIN
    Check perms argument for validity
    Dispatch FSLIB_ACCESS (pathid, perms)
END
```

D.3.2 FSI_chdir

Description:

This function will change the working directory. Modifies global variables FSI_curdir, FSI_curdirlen.

Synopsis:

FSI_chdir (FSI_PATHID *pathid)

Algorithm:

```
BEGIN
    IF Current directory <> pathid THEN BEGIN
            Dispatch FSLIB_chdir (pathid)
            Copy fullpath to FSI_curdir
    END
END
```

D.3.3 FSI_fchdir

Description:

This function will change the working directory. Modifies global variables FSI_curdir, FSI_curdirlen.

Synopsis:

FSI_fchdir (FSI_FID *fp)

Algorithm:

```
BEGIN
    IF Current directory <> fp THEN BEGIN
            IF fp multiplex closed THEN Dispatch FSLIB_mpxopen (fp)
            Dispatch FSLIB_fchdir (fp)
            Copy fullpath to FSI_curdir
    END
END
```

D.3.4 FSI_chmod

Description:

This routine will change the access permission to the specified file.

Synopsis:

FSI_chmod (FSI_PATHID *pathid, mode)

Algorithm:

```
BEGIN
    Dispatch FSLIB_chmod (pathid, mode)
END
```

D.3.5 FSI_fchmod

Description:

This routine will change the access permission to the specified file.

Synopsis:

FSI_fchmod (FSI_FID *fp, mode)

Algorithm:

```
BEGIN
    IF fp multiplex closed THEN Dispatch FSLIB_mpxopen (fp)
    Dispatch FSLIB_chmod (fp, mode)
END
```

D.3.6 FSI_chown

Description:

This routine will change the owner of the specified file

Synopsis:

FSI_chown (FSI_PATHID *pathid, user_id, group_id)

Algorithm:

```
BEGIN
    Dispatch FSLIB_chown (pathid, user_id, group_id)
END
```

D.3.7 FSI_fchown

Description:

This routine will change the owner of the specified file

Synopsis:

FSI_fchown (FSI_PATHID *pathid, user_id, group_id)

Algorithm:

```
BEGIN
    IF fp multiplex closed THEN Dispatch FSLIB_mpxopen (fp)
    Dispatch FSLIB_chown (pathid, user_id, group_id)
END
```

D.3.8 FSI_close

Description:

This function will close a file.

NOTE

The data fork and extended attributes fork are currently implemented in separate files. These files will not have equivalent modification times and may therefore not be backed up as a pair.

Synopsis:

FSI_close (FSI_FID *fp)

Algorithm:

```
BEGIN
    IF written THEN Set volume modify time
    IF last reference THEN BEGIN
            DEQUEUE fp
            Dispatch FSLIB_close
    END
END
```

D.3.9 FSI_copyfile

Description:

This function will copy a file. The resource fork is discarded (truncated) if the destination file system does not support it. The extended attributes fork is also discarded if the destination file system does not support it.

Synopsis:

FSI_copyfile (FSI_PATHID *src, FSI_PATHID *dest, dostream, action)

Algorithm:

```
BEGIN
    IF src does not exist THEN error
    IF read only fs OR no copy THEN error
    IF dostream resource AND dest not supported THEN error
    IF dostream attributes AND dest not supported THEN error
    Open src primary stream
    IF dest does not exist THEN BEGIN
            Create dest primary stream
            Copy file attributes from src
    END
    ELSE Open dest primary stream
    FOR all streams DO BEGIN
            Dispatch FSI_lock (src)
            Dispatch FSI_lock (dest)
            Dispatch FSLIB_ffilesize (src)
            FOR all bytes DO BEGIN
                    Dispatch FSLIB_read (src)
                    Dispatch FSLIB_write (dest)
            END
            Dispatch FSI_unlock (src)
            Dispatch FSI_unlock (dest)
            Dispatch FSI_close (src)
            Dispatch FSI_close (dest)
            Open next src stream
            Open next dest stream
```

```
    END
    IF FSLIB_copyfile THEN Dispatch FSLIB_copyfile
    IF MAC application THEN update desktop
    Set volume modify time
END
```

D.3.10 FSI_create

Description:

Create a new file or truncate an existing file.

NOTE

> There is an FSI_FID sharing mechanism which will return a pointer to a previously allocated FID if the file is already open. Note that FSI_create will truncate this file without explicit lock checking or synchronization with other readers and or writers. Mandatory locking appears to be defeated in this case.

FSLIB_open allocates the FSI_FID. It also initializes the following fields of the FSI_FID:

fullpath (copied from pathid)
fd (returned by UNIX)
fdinfo (fd low or high)
status (copied from passed statbufp)
flags (mandlock set from st_mode)
mapaddr (allocated)
maplen (st_size)

FSI_create initializes the following fields of the FSI_FID following a successful call to FSLIB_open (actually done in local routine opencreate):

refcnt (set to 1)
stream (FSI_PRIMARY)
oflag (O_RDWR)
funcptrs (copied from pathid)
fsflags (copied from pathid)
endtreetop (copied from pathid)
nompx (if file not regular or directory)

Synopsis:

FSI_create (FSI_PATHID *pathid, mode, uid, gid, FSI_CREATE_TYPE type, FSI_FID **fp)

Algorithm:

```
BEGIN
    IF type is FSI_MAKETMP THEN BEGIN
      [Need to supply description here]          END
    ELSE BEGIN
            IF path does not exist THEN get parent attributes
            ELSE get file attributes
            IF read only THEN error
            IF type is FSI_MAKENEW and file exists THEN error
            IF file already open THEN BEGIN
                    Dispatch FSLIB_truncate
```

```
                Use existing FID
            END
            ELSE BEGIN
                Dispatch FSLIB_open (pathid, &pathid->status, -
                    O_RDWR | O_CREAT | O_TRUNC, mode, FSI_PRIMARY, -
                    FSI_NOMAP, fp)
                IF no more file space THEN start multiplex close
                IF too many open files THEN close any files
                IF access denied OR image busy OR readonly fs -
                OR no memory OR no space OR no more processes -
                THEN error
                Initialize the remainder of the FID
                Add it to the FID list
                IF file does not exist THEN BEGIN
                    IF FSI_PRIMARY stream THEN -
                    Dispatch FSLIB_fchown
                    Dispatch FSLIB_fstat
                    IF FSI_PRIMARY THEN pathid->status = fp->status;
                END
            END
        END
    END
```

D.3.11 FSI_delete

Description:

Delete a file.

Synopsis:

FSI_delete (FSI_PATHID *pathid)

Algorithm:

```
BEGIN
    IF file does not exist THEN error
    IF readonly OR no delete THEN error
    IF no_purge THEN error
    Dispatch FSLIB_purge (pathid)
    IF MAC application THEN update desktop
    Set volume modify time
END
```

D.3.12 FSI_diridinit

Description:

This routine will initialize the handling of directory IDs.

Synopsis:

FSI_diridinit (dodirids, diridptr)

Algorithm:

BEGIN
FOR any non mapped file system DO
Dispatch FSLIB_diridinit (dodirids, diridptr)
END

D.3.13 FSI_diridfunc

Description:

This routine handles directory ID functions, FSI_DIRID_GET, FSI_DIRID_OPEN, FSI_DIRID_CLOSE. The DIRID_OPEN call indicates a volume has been mounted and a new set of directory IDs are to be used. DIRID_GET will convert a directory ID and UNIX pathname (representing the volume root) to a full unix pathname to the directory. DIRID_CLOSE indicates the volume has been closed.

Synopsis:

FSI_diridfunc (cmd, startpath, dirid, ptr, pathbuf, buflen)

Algorithm:

BEGIN
Verify startpath is a valid UNIX directory
Dispatch FSLIB_diridfunc
END

D.3.14 FSI_ffilesize

Description:

This function will return the number of bytes stored in the file.

Synopsis:

FSI_ffilesize (FSI_FID *fp, size)

Algorithm:

BEGIN
IF fp multiplex closed THEN Dispatch FSLIB_mpxopen (fp)
Dispatch FSLIB_ffilesize (fp, size)
END

D.3.15 FSI_fsync

Description:

This routine will flush any written buffers associated with the file.

Synopsis:

FSI_fsync (FSI_FID *fp)

Algorithm:

BEGIN
    IF fp multiplex closed THEN Dispatch FSLIB_mpxopen (fp)
    Dispatch FSLIB_fsync (fp, size)
END

D.3.16 FSI_fullpath

Description:

This function will return the full UNIX pathname of the file

Synopsis:

FSI_fullpath (FSI_FID *fp, pathbuf, pathlen)

Algorithm:

BEGIN
    Copy fullpath from fp to pathbuf
END

D.3.17 FSI_getattr

Description:

This function will return the requested file attributes and update the FSI_PATHID structure. Attributes which may be requested are as follows:

FSI_AT_DIRID - Directory ID
FSI_AT_BTIME - Backup time
FSI_AT_CREATE - Creation time
FSI_AT_F_INFO - Finder info
FSI_AT_ARCHIVE - File is archived
FSI_AT_HIDDEN - File is archived
FSI_AT_SYSTEM - File is a system file
FSI_AT_NOREN - File can not be renamed (can be copied)
FSI_AT_NODEL - File can not be deleted
FSI_AT_NOCOPY - File can not be copied (can be renamed)
FSI_AT_READONLY - File is read only
FSI_AT_NOPURGE - File is deleted on delete
FSI_AT_MACAPPL - File is Macintosh application
FSI_AT_MULTIUSER - File can be opened simultaneously
FSI_AT_EXECONLY - File is execute only
FSI_AT_INDEXED - Netware index file
FSI_AT_TRANS - Netware transation tracking
FSI_AT_RDAUDIT - Netware transaction tracking
FSI_AT_WRAUDIT - Netware transaction tracking

Synopsis:

FSI_getattr (FSI_PATHID *pathid, mask, FSI_ATTR *attrp)

Algorithm:

BEGIN
    Dispatch FSLIB_getattr (pathid, mask, attrp)
    Copy attrp to pathid

END

D.3.18 FSI_fgetattr

Description:

This routine will return the requested attributes and update the FSI_FID structure. Attributes and masks are the same as for FSI_getattr.

Synopsis:

FSI_fgetattr (FSI_FID *fp, mask, FSI_ATTR *attrp)

Algorithm:

```
BEGIN
    IF fp multiplex closed THEN Dispatch FSLIB_mpxopen (fp)
    Dispatch FSLIB_fgetattr (fp, mask, attrp)
    Copy attrp to fp
END
```

D.3.19 FSI_getcomment

Description:

This function will return a comment associated with a file. This is presumably present to support the MAC desktop database.

Synopsis:

FSI_getcomment (FSI_PATHID *pathid, commentbuf, commentlen)

Algorithm:

```
BEGIN
    Dispatch FSLIB_getcomment (pathid, commentbuf, commentlen)
END
```

D.3.20 FSI_getcmd

Description:

This function will return the current working directory.

Synopsis:

FSI_getcwd (cwdptr)

Algorithm:

```
BEGIN
    Set cwdptr to FSI_curdir
END
```

D.3.21 FSI_getdents

Description:

This function will return directory entry names in a specified format. The directory structure contains a longword ID, length word and text buffer. The name is returned in the namespace as specified below.

FSI_UNIXNAME - Use UNIX format
FSI_DOSNAME - Use DOS format
FSI_OS2NAME - Use OS/2 format
FSI_MACNAME - Use Macintosh format

The NBYTES parameter specifies how large the direntp buffer is and BYTESREAD specifies how much data was actually written to the buffer. OFFSET specifies where to start the read.

The dirent structures is defined as follows:

unsigned long d_ino; /* Unique identifier for file */
unsigned short d_reclen; /* Size of this record */
unsigned short d_namlen; /* Length of filename */
char d_name[MAXNAMLEN+1]; /* Buffer for filename */

Multiple entries may be packed in the buffer and may be found by using the buffer offset plus the d_reclen parameter.

Synopsis:

FSI_getdents (FSI_FID *fp, struct dirent *direntp, nbytes, offset, namespace, bytesread)

Algorithm:

BEGIN
IF fp multiplex closed THEN Dispatch FSLIB_mpxopen (fp)
Dispatch FSLIB_getdents (fp, direntp, nbytes, offset, namespace, bytesread)
END

D.3.22 FSI_getextattr

Description:

This function will return the extended attributes specified in the return buffer provided. The function will return the size of the return attributes in case the buffer is not large enough. It is the caller's responsibility to allocate a buffer large enough and call the function again. The function will fail if the buffer is not large enough.

Synopsis:

FSI_getextattr (FSI_PATHID *pathid, FSI_EAOPS *eaopsp)

Algorithm:

BEGIN
IF file does not exist THEN error
IF extended attributes not supported by fs THEN error
Dispatch FSLIB_stat for FSI_EXTATTRS stream
Open or create the stream
Dispatch FSI_lock for whole file containing stream
Dispatch FSLIB_ffilesize
IF no stream THEN BEGIN
IF return buffer large enough THEN-

```
            set return buffer to indicate no attributes
            Set size of return buffer
    END
    ELSE BEGIN
            Dispatch FSI_read for EA header
            Dispatch FSI_read for EA data
            Scan all attributes and calculate stream size
            Store stream size in return buffer
            IF no attributes requested THEN done
            FOR all requested attributes DO BEGIN
                    If requested attribute in list THEN -
                    append attribute and value to return buffer
                    ELSE append NULL attribute value to return buffer
                    Set size of return buffer
            END
    END
    Dispatch FSI_unlock
    Dispatch FSI_close
END
```

D.3.23 FSI_getpathid

Description:

This function will initialize an FSI_PATHID structure used for subsequent access to the path. The function will determine if the path exists and which file system it belongs to. The function uses a combination of UNIX stat() calls and FSLIB_claim calls to resolve the path. There is a provision for handling path translation from the client namespace but it is not yet implemented.

The function may return one of three return codes, FSI_EXISTS (path exists and pathid contains information about the path), FSI_NOEXIST (path does not exist but parent does and pathid contains information about the parent) or FSI_FAILURE (neither path nor parent exists and pathid does not contain any information).

The UNIX stat() function will succeed if the path exists in one of the UNIX file systems. If the path represents a "pseudo" file system, (i.e. a file system not mounted) then stat() may fail even though the FSI can access the path. In this case, FSLIB_claim would be responsible for identifying the path.

NOTE

> The FSI implementation is not complete within the LMU server. There are server functions which directly call UNIX I/O functions on resolved path names (chksvr4() called by chkuxpath() called by smbcreate() uses open() to check write access to a directory). This would seem to preclude support of "pseudo" file systems as generic entities. Pseudo file systems may find application in specific sections of the LMU server, for example, the implementation of the desktop database may use a pseudo file system.

The FSLIB_claim function may return one of 4 return codes, FSI_EXISTS (path exists and is in the file system supported by this library), FSI_NOEXIST (path does not exist but the parent does and is in the file system supported by this library), FSI_UNCLAIMED_EXISTS (path exists in UNIX file system but is not supported by this library) or FSI_UNCLAIMED_NOEXIST (path does not exist but parent exists in the UNIX file system but is not supported by this library.

NOTE

> The algorithm used by FSI_getpathid seems to resort to "forced claims" as the algorithm progresses. If a library returns an UNCLAIMED status it is "forced" to

accept the path if no one else did. It is not clear what the benefit of this "last chance" mechanism could be. Either the path is supported or it is not.

Synopsis:

FSI_getpathid (path, startcase, FSI_PATHID *pathid)

Algorithm:

```
BEGIN
    Clear pathid structure
    IF relative path THEN append to working directory
    Copy resolved path to fullpath
    Setup shortpath
    FOR FSI_PSEUDO_FS file systems DO BEGIN
            Dispatch FSLIB_claim
            IF claimed THEN done FSI_EXISTS or FSI_NOEXIST
    END
    UNIX stat() the path
    IF path exists THEN BEGIN
            IF NOT FILLPATH(path pathid) THEN error
            IF UNIX namespace THEN done FSI_EXISTS
            IF NOT case sensitive creates supported THEN BEGIN
                    Dispatch FSLIB_claim
                    IF claimed AND file exists THEN done FSI_EXISTS
                    ELSE error
            END
            ELSE done FSI_EXISTS
    END
    Get parent path
    UNIX stat() the path
    IF path exists THEN BEGIN
            IF path is not a directory THEN error
            IF NOT FILLPATH(parent pathid) THEN error
            IF UNIX namespace THEN done
            Dispatch FSLIB_claim
            IF exists THEN BEGIN
                    IF NOT FILLPATH(path pathid) THEN error
                    done FSI_EXISTS
            END
            ELSE done FSI_NOEXIST
    END
    IF UNIX namespace THEN error
    FOR ALL file systems DO BEGIN
            Dispatch FSLIB_claim
            IF file exists AND unclaimed THEN BEGIN
                    IF NOT FILLPATH(path pathid) THEN error
                    done FSI_EXISTS
            END
            IF file does not exist AND unclaimed THEN BEGIN
                    IF NOT FILLPATH(parent pathid) THEN error
                    done FSI_NOEXIST
            END
    END
    IF not claimed THEN error
END
```

D.3.24 FSI_lock

Description:

This function sets a byte range lock on a file. The file may be locked from the start of the file or the end of the file. The offset is the distance from the set point. The length argument specifies how many bytes to lock (NULL implies the remainder of the file). The routine will either fail if the lock is set or it will block until the lock is released (the block is the responsibility of the FSLIB). The offset from the start of the file to the lock point is returned, if requested.

NOTE

> There appears to be a bug in this routine in that the start argument is not updated unless the file is actually locked, i.e. if the file is mapped the start argument is not guaranteed to be correct.

Synopsis:

FSI_lock (FSI_FID *fp, type, offset, whence, length, dowait, start)

Algorithm:

```
BEGIN
    Check whence argument (SEEK_SET or SEEK_END)
    Check lock type (F_RDLCK or F_WRLCK)
    IF FSI_locksmatter AND file not memory mapped THEN BEGIN
            IF fp multiplex closed THEN Dispatch FSLIB_mpxopen (fp)
            IF offset is over UNIX limit THEN done (YIKES!!!)
            IF end of lock is over UNIX limit THEN -
            set end of lock to UNIX limit
            Dispatch FSLIB_lock (fp, type, offset, whence, length, dowait, start)
            IF NFS not running THEN ignore NFS errors
            ELSE IF error THEN error
            Disallow multiplex closing this file (increment reason)
    END
    ELSE set start argument to start offset of lock
END
```

D.3.25 FSI_lseek

Description:

This function will set the current file position to that specified.

NOTE

> This routine must be present for compatibility only as file position is not guaranteed across any FSI calls.

Synopsis:

FSI_lseek (FSI_FID *fp, offset, whence)

Algorithm:

```
BEGIN
    Check whence argument (SEEK_SET or SEEK_END)
    Check offset argument
    Dispatch FSLIB_lseek (fp, offset, whence)
```

END

D.3.26 FSI_mapname

Description:

This function will convert the last component of the path to the namespace requested.

Synopsis:

FSI_mapname (FSI_PATHID *pathid, namespace, namebuf, buflen)

Algorithm:

```
BEGIN
    Dispatch FSLIB_mapname (pathid, namespace, namebuf, buflen)
END
```

D.3.27 FSI_fmapname

Description:

This function will convert the last component of the path to the namespace requested.

NOTE

This function DOES NOT reopen a multiplex closed file. This may be an oversight, especially if the file system library expects to store converted names in the file itself.

Synopsis:

FSI_fmapname (FSI_FID *fp, namespace, namebuf, buflen)

Algorithm:

```
BEGIN
    Dispatch FSLIB_fmapname (fp, namespace, namebuf, buflen)
END
```

D.3.28 FSI_mkdir

Description:

This function will create a directory.

NOTE

The directory ID parameter is not supported in UFS and will be forced to 0.

Synopsis:

FSI_mkdir (FSI_PATHID *pathid, mode, uid, gid, dirid)

Algorithm:

BEGIN

```
    IF directory exists THEN error
    IF readonly THEN error
    Dispatch FSLIB_mkdir(pathid, mode, dirid)
    Dispatch FSLIB_chown(pathid, uid, gid)
    Set volume modify time
END
```

D.3.29 FSI_mpxclose

Description:

This function will multiplex close the specified file. It is presumed to be present to allow servers to determine the best candidates for multiplex closing without resorting to FSI forced multiplexing.

Synopsis:

FSI_mpxclose (FSI_FID *fp)

Algorithm:

```
BEGIN
    IF NOT fp multiplex closed THEN Dispatch FSLIB_mpxclose (fp)
END
```

D.3.30 FSI_needfds

Description:

This function specifies a number of UNIX file descriptors which must be available. If this number of file descriptors is not available then the function will multiplex close files until it is.

Synopsis:

FSI_needfds (count)

Algorithm:

```
BEGIN
    IF dup() a file descriptor THEN BEGIN
            close() the new file descriptor
            fcntl(F_DUPFD) as many file descriptors as needed
            close() them all
            IF not enough THEN multiplex close the balance
    END
END
```

D.3.31 FSI_needinodes

Description:

This function will multiplex close a number of files.

NOTE

This routine is primarily for internal FSI use.

Synopsis:

FSI_needinodes(timescalled, FSLIB_ptrs)

Algorithm:

```
BEGIN
    Get the number of files to close based on timescalled
    FOR all open multiplexable fp DO BEGIN
            Dispatch FSLIB_mpxclose (fp)
            IF notify on mpxclose THEN Dispatch FSI_notifympx
    END
END
```

D.3.32 FSI_open

Description:

This function will open a data stream for read or write access.

Synopsis:

FSI_open (FSI_PATHID *pathid, statbufp, oflag, stream, domap, fp)

Algorithm:

```
BEGIN
    IF NOT primary stream OR resource stream THEN error
    IF resource stream AND resource not supported THEN error
    IF primary stream AND NOT open read/write OR readonly OR -
    writeonly THEN error
    IF file does not exist THEN error
    IF read only AND open for write THEN error
    IF NOT primary stream THEN BEGIN
            Dispatch FSLIB_stat
            Get UID, GID and protection mode
    END
    ELSE use file default UID, GID and protection mode
    IF directory THEN open for read only
    IF file already open THEN BEGIN
            Dispatch FSLIB_truncate
            Use existing FID
    END
    ELSE BEGIN
            Dispatch FSLIB_open
            IF no more file space THEN start multiplex close
            IF too many open files THEN close any files
            IF access denied OR image busy OR readonly file system -
            OR no memory OR no disk space OR no more processes -
            THEN error
            Initialize the remainder of the FID
            Add it to the FID list
            IF file does not exist THEN BEGIN
                    IF FSI_PRIMARY stream THEN -
                    Dispatch FSLIB_fchown (fp, uid, gid)
                    Dispatch FSLIB_fstat (fp, FSI_ST_USTAT, &fp->status)
                    IF FSI_PRIMARY THEN pathid->status = fp->status;
            END
    END
END
```

D.3.33 FSI_purge

Description:

This function will delete a file.

Synopsis:

FSI_purge (FSI_PATHID *pathid)

Algorithm:

```
BEGIN
    IF file does not exist THEN error
    IF read only fs OR no delete THEN error
    Dispatch FSLIB_purge
    IF MAC application THEN update desktop
    Set volume modify time
END
```

D.3.34 FSI_read

Description:

This function will read data from the file.

Synopsis:

FSI_read (FSI_FID *fp, buffer, nbytes, offset, bytesread)

Algorithm:

```
BEGIN
    IF fp multiplex closed THEN Dispatch FSLIB_mpxopen (fp)
    IF NOT mandlock AND NOT mapped THEN lock record
    Dispatch FSLIB_read
    IF record locked THEN unlock record
END
```

D.3.35 FSI_rename

Description:

This function will rename a file.

Synopsis:

FSI_rename (FSI_PATHID *old, FSI_PATHID *new)

Algorithm:

```
BEGIN
    IF both paths not in same fs THEN error
    IF old file does not exist THEN error
    IF read only fs OR no delete THEN error
```

Dispatch FSLIB_rename
IF MAC application THEN update desktop
Set volume modify time
END

D.3.36 FSI_rmdir

Description:

This function will delete a directory.

NOTE

This function makes a call to FSI_checkvolume to determine if the directory can be deleted on a MAC volume. There is no information on this function call.

Synopsis:

FSI_rmdir (FSI_PATHID *pathid)

Algorithm:

BEGIN
IF read only fs OR no delete THEN error
IF current directory THEN set current to root
Dispatch FSLIB_rmdir
Set volume modify time
END

D.3.37 FSI_setattr

Description:

This function will set the file's attributes. These attributes are those which the FSI operates on.

Synopsis:

FSI_setattr (FSI_PATHID *pathid, attrip)

Algorithm:

BEGIN
IF attributes not supported by fs THEN error
Dispatch FSLIB_setattr
Set volume modify time
END

D.3.38 FSI_fsetattr

Description:

This function will set the file's attributes. These attributes are those which the FSI operates on.

Synopsis:

FSI_fsetattr (FSI_FID *fp, attrip)

Algorithm:

```
BEGIN
    IF attributes not supported by fs THEN error
    IF fp multiplex closed THEN Dispatch FSLIB_mpxopen (fp)
    Dispatch FSLIB_setattr
    Set volume modify time
END
```

D.3.39 FSI_setcomment

Description:

This function will associate a string of up to 199 characters with a file name.

Synopsis:

FSI_setcomment (FSI_PATHID *pathid, string)

Algorithm:

```
BEGIN
    IF comment too long THEN truncate to 199
    Dispatch FSLIB_setcomment
    Set volume modification time
END
```

D.3.40 FSI_setextattr

Description:

This function will add, delete or modify extended attributes associated with a file. The GEA list member of the EAOPS structure is ignored for this function. The FEA list contains a list of attributes and their values. If the attribute does not exist it will be added. If the attribute exists it will be modified unless the value pointer is NULL, in which case the attribute will be deleted. Attributes not specified in the FEA list will remain unchanged.

Synopsis:

FSI_setextattr (FSI_PATHID *pathid, FSI_EAOPS *eaopsp)

Algorithm:

```
BEGIN
    IF file does not exist THEN error
    IF extended attributes not supported by fs THEN error
    IF no FEA list THEN done
    Dispatch FSLIB_stat for FSI_EXTATTRS stream
    Open or create the stream
    Dispatch FSI_lock for whole file containing stream
    Dispatch FSLIB_ffilesize
    IF no stream THEN BEGIN
            Write out attributes in FEA list
            Done
    END
    ELSE BEGIN
```

```
            Dispatch FSI_read for EA header
            Dispatch FSI_read for EA data
            FOR all FEA in list DO BEGIN
                IF attribute found THEN BEGIN
                    IF value fits THEN modify existing entry
                    ELSE BEGIN
                        Delete attribute
                        IF empty space THEN add attribute
                        ELSE increase size of stream buffer
                    END
                END
                ELSE BEGIN
                    IF empty space THEN add attribute
                    ELSE increase size of stream buffer
                END
            END
        END
        Dispatch FSI_write for EA data
        Get clean stream buffer for remaining FEA
        FOR all FEA which did not fit DO add attribute
        Dispatch FSI_write for EA data
        Dispatch FSI_write for EA header
        Dispatch FSI_unlock
        Dispatch FSI_close
    END
```

D.3.41 FSI_setlognores

Description:

This function specifies a routine for the FSI to call when resources have been exhausted.

Synopsis:

FSI_setlognores (routine)

Algorithm:

```
BEGIN
    Set FSI_LogNoResource to routine
END
```

D.3.42 FSI_setnotifympx

Description:

This function specifies a routine for the FSI to call when multiplexing begins.

Synopsis:

FSI_setnotifympx (routine)

Algorithm:

```
BEGIN
    Set FSI_notifympx to routine
END
```

D.3.43 FSI_shortpath

Description:

This function will set the short path element of the FSI_PATID structure to the point past the current directory, if it is part of the full path. If not the short path is set to the full path.

Synopsis:

FSI_shortpath (FSI_PATHID *pathid)

Algorithm:

```
BEGIN
    IF current directory in fullpath THEN -
            set short path past current directory
    ELSE set short path to full path
END
```

D.3.44 FSI_stat

Description:

This function will return file statistics in the UNIX stat format.

Synopsis:

FSI_stat (FSI_PATHID *pathid, stream, mask, statbufp)

Algorithm:

```
BEGIN
    IF mask not supported by fs THEN error
    IF stream NOT (FSI_PRIMARY OR FSI_RESOURCE) THEN error
    IF stream is FSI_RESOURCE AND not supported by fs THEN error
    Dispatch FSLIB_stat
    IF stream is FSI_PRIMARY THEN copy statbufp to pathid status
END
```

D.3.45 FSI_fstat

Description:

This function will return file statistics in the UNIX stat format.

Synopsis:

FSI_fstat (FSI_FID *fp, stream, mask, statbufp)

Algorithm:

```
BEGIN
    IF mask not supported by fs THEN error
    IF stream NOT (FSI_PRIMARY OR FSI_RESOURCE) THEN error
    IF stream is FSI_RESOURCE AND not supported by fs THEN error
    IF fp multiplex closed THEN Dispatch FSLIB_mpxopen (fp)
```

Dispatch FSLIB_stat
IF stream is FSI_PRIMARY THEN copy statbufp to fp status
END

D.3.46 FSI_statvfs

Description:

This function will return file information in a UNIX statvfs structure.

Synopsis:

FSI_statvfs (FSI_PATHID *pathid, statvfsbufp)

Algorithm:

BEGIN
Dispatch FSLIB_statvfs
END

D.3.47 FSI_fstatvfs

Description:

This function will return file information in a UNIX statvfs structure.

Synopsis:

FSI_fstatvfs (FSI_FID *fp, statvfsbufp)

Algorithm:

BEGIN
IF fp multiplex closed THEN Dispatch FSLIB_mpxopen (fp)
Dispatch FSLIB_fstatvfs
END

D.3.48 FSI_sync

Description:

This function will flush all file system information from memory to disk.

Synopsis:

FSI_sync ()

Algorithm:

BEGIN
FOR all non mapped fs DO Dispatch FSLIB_sync
END

D.3.49 FSI_ftruncate

Description:

This function will truncate a file.

Synopsis:

FSI_ftruncate (FSI_FID *fp, offset)

Algorithm:

```
BEGIN
    IF fp multiplex closed THEN Dispatch FSLIB_mpxopen (fp)
    Dispatch FSLIB_ftruncate
END
```

D.3.50 FSI_tretop

Description:

This function sets the treetop pointer in the pathid structure to the point past the "root directory" for a share point or mounted volume.

Synopsis:

FSI_treetop (FSI_PATHID *pathid, treetop)

Algorithm:

```
BEGIN
    IF treetop in full path THEN set treetop
    ELSE error
END
```

D.3.51 FSI_unlock

Description:

This function will unlock a range of bytes in the file.

NOTE

> The function will multiplex open a file and then try to release a lock. This should be guaranteed to fail as UNIX releases locks when a file is closed. Perhaps it is best to either leave the file closed and return success (the lock was actually released) or reopen the file and return success without unlocking anything.

Synopsis:

FSI_unlock (FSI_FID *fp, offset, whence, length)

Algorithm:

```
BEGIN
    IF whence NOT (SEEK_SET OR SEEK_END) THEN error
    IF FSI_locksmatter AND file not memory mapped THEN BEGIN
    IF fp multiplex closed THEN Dispatch FSLIB_mpxopen (fp)
    IF offset is over UNIX limit THEN done (YIKES!!!)
    IF end of lock is over UNIX limit THEN -
```

set end of lock to UNIX limit
Dispatch FSLIB_unlock
Allow multiplex closing this file (decrement reason)
END
END

D.3.52 FSI_unmap

Description:

This function will clean up a memory mapped file.

Synopsis:

FSI_unmap (FSI_FID *fp)

Algorithm:

BEGIN
Dispatch FSLIB_unmap
END

D.3.53 FSI_utime

Description:

This function will set the modification time of a file.

Synopsis:

FSI_utime (FSI_PATHID *pathid, timebufp)

Algorithm:

BEGIN
Dispatch FSLIB_utime
END

D.3.54 FSI_futime

Description:

This function will set the modification time of a file.

Synopsis:

FSI_futime (FSI_FID *fp, timebufp)

Algorithm:

BEGIN
Convert fp to pathid
Dispatch FSLIB_utime
END

D.3.55 FSI_write

Description:

This function will write bytes to a file.

Synopsis:

FSI_write (FSI_FID *fp, buffer, nbytes, offset, byteswritten)

Algorithm:

```
BEGIN
    IF fp multiplex closed THEN Dispatch FSLIB_mpxopen (fp)
    IF NOT mandlock AND NOT mapped THEN lock record
    Dispatch FSLIB_write
    Set dirty flag
    IF record locked THEN unlock record
END
```

Appendix E - RMS Extent for Macintosh file format

The Macintosh file system provides two data contexts per file rather than the usual single context. This concept makes it difficult to represent a Macintosh file in a file system which supports only single contexts. VMS Files 11 is such a file system.

There appear to be two approaches to this problem. The first represents the Macintosh file as two separate files. It is the responsibility of the software which provides access to the Macintosh file to associate the two separate files. This approach has a number of advantages, most noteworthy is the ability of the native file system to access either of the Macintosh contexts directly. However, there is a serious disadvantage in that the two separate files may be modified, moved or deleted such that the association is no longer valid. The second approach represents both data contexts in the same file. This approach solves the asociation of data contexts, however, Files 11 can not access either data context directly. It is necessary to have a translator between the native file system and the user such that the appropriate data context may be accessed. RMS provides such a mechanism in what is called an RMS extent. The capabilities of the extent are somewhat limited but do provide the basics for the type of translation necessary to access each data context. The only caveat is that the user MUST use RMS to access the file. If not, the internal file format is visible.

This paper describes a trial development of an RMS extent which provides access to the Macintosh DATA fork (one of the data contexts in the Macintosh file).

E.1 File Format

The trial development file format borrows from the mapping concepts of FIles 11 in that there is a set of retrieval pointers which map the virtual blocks of each data context to virtual blocks within the file. This mapping is exactly analogous to the virtual block to logical block mapping provided by Files 11.

The file consists of a header followed by data blocks or mapping blocks. To limit the overhead the first set of mapping pointers is contained in the header. As a further simplification all data blocks are aligned on a block boundary which provides for a simple revectoring of virtual blocks within the data context (alias data stream) to the virtual blocks within the file.

Data blocks are allocated as needed and mapped to one of two streams, the DATA stream or the RESOURCE stream. Each stream has its own set of mapping pointers and the file is limited to two streams for the purpose of this trial development. The extensions to multiple streams are straightforward, however, the file format assumes a fixed number of streams.

Mapping blocks are allocated as the streams become fragmented and there is no longer sufficient space in the file header to map blocks. The format of the mapping blocks is identical to that of the mapping blocks in the header. This concept is not implemented in the trial development. The streams may not be arbitrarily fragmented. The implementation chooses 16 mapping pointers in the header block (completely arbitrary choice, although the upper limit would be about 60 per stream). This limits the fragmentation to 16 individual segments. The tests performed were on contiguous streams mapped by a single pointer.

E.1.1 File Semantics

RMS provides a *File Semantic* feature which identifies the internal structure of a file which requires an RMS extent for access. RMS scans a file semantic tag structure to locate the routines which will provide access to the file. This tag structure is loaded at system startup time by the initialization routines of the extent image.

The trial development uses the file sematic tag "MACFILE". This tag is represented as the ASCII translation of "MACFILE" in the RMS *stored semantics*. When RMS finds a file with this tag, it will dispatch the routines specified by the extent image.

RMS will bypass the routines if the user supplies an *access sematics* tag which matches the *stored semantics*. It is assumed that the caller want to directly access the internal file format in this case. This access mode is not currently used by would be useful to a utility which could modify the Macintosh file internals (for building such a file directly from the host system for example).

E.1.2 Header

The header consists of three sections; allocation, stream descriptors and mapping blocks. The allocation information is simply the next virtual block for allocation (although this information could be obtained from internal RMS structures). The stream descriptors consist of three longwords; the end of stream VBN, the end of stream byte within the VBN and the offset to the first set of mapping pointers. The mapping pointers are one logword each and consist of four types; NULL, MAP, FREE and OFFSET.

E.1.2.1 Allocation

The allocation section simply consists of the next virtual block to be allocated. This section must be expanded to include allocation data for the mapping blocks

E.1.2.2 Stream Descriptors

There are currently two stream descriptors; one for the DATA fork and one for the RESOURCE fork. Each descriptor consists of three longwords to contain the end of stream VBN, the end of stream byte and the offset to the mapping pointers. If the number of streams per file is to be increased new descriptors would need to be added.

E.1.2.3 Mapping Pointers

The mapping pointers consist of one longword each and map streams, free blocks or mapping blocks. The pointer format is variable based on a two bit field in the upper two bits of the longword. Currently VBNs are limited to 16 bits which sets the upper limit to 32MByte file which can be represented. This number can be changed to accomodate the largest file expected or extended to map to the full Files 11 limit.

E.1.3 Data Stream Format

Currently the data stream is represented as a STREAM CR format with the assumption that it contain text. Clearly this needs to be extended. It is not clear how the data fork should be interpretted, there is no indication, short of scanning for non-printable characters which could be used to determine the data format. RMS based applications assume the file format is stored in the file and can be used to determine how to process the file. For the purposes of this tial development, we have assumed that the data should be returned as RMS VAR format with CR attributes and is stored as a stream of records terminated by <CR> (sample Macintosh text file is stored this way).

E.1.4 Resource Stream Format

The resource stream is simply represented as a stream of bytes. The caller will have as many bytes returned as will fit in the buffer. The file is still reported as RMS VAR format with CR attributes as this information needs to be returned before the stream access is done (RMS $CONNECT). While the support to read the resource fork is implemented it can not be accessed by RMS in this trial development.

E.2 Extension Structure

The RMS extent is structured as an initialzation routine, series of callout routines and a set of support routines. The basic approach was to use RMS facilities to read and write a block of the file and perform the rest of the record access in support routines. RMS supplies two such routines; RMS$GET_BUFFER and RMS$RELEASE_BUFFER. The routines access the RMS data cache and will perform file reads (if requested) or file writes (if requested). For the purpose of this trial development only those routines necessary to access a sequential text file have been implemented. Both sequential and random access to these records is provided.

The callout routines completely replace the normal RMS routines. This means the routines must update user fields of the RMS structures (FAB/RAB/XAB) and also move data to and from the user buffers pointed to in these structures.

E.2.1 Initialization

The extent initialization consists of a call to add the semantic tag to the RMS table and provide a set of callout entry points. RMS will call the routines declared by a non zero entry in the appropriate slot in the dispatch table. If an entry is zero, RMS will handle the function internally using the normal RMS access. RMS provides a standard "not implemented" callback routine which should be used if an RMS function is to be denied. IF the normal RMS handling for a function is sufficient, a zero should be placed in the dispatch table slot.

This is the only execution time function of the extent. It simply sets up the table, calls RMS to identify the sematics and exits. The remainder of the extent remains mapped and will be called in the context of an *RMS thread.*

E.2.2 RMS support routines

RMS provides a series of routines to provide the extent with access to internal data structures and file data. These routines are only partially implemented. The history of the RMS extent development is such that only the first phase of development was completed. This development was done to support the CDA architecture. Unfortunately, this developement only required read access to files and as such, no simple mechanisms for file writes and file extend operations are provided. These functions must be implemented by using the low level RMS calls and direct QIO access to the file (for file extends).

The trial development uses a small set of routines primarily for block level read and write access to the Macintosh file.

E.2.3 Data Structures

The extent uses a simple context block to store information needed across calls to process the file. This information includes the stream mapping data and various data buffers.

The extent caches mapping data while accessing the file. this data will be written back to the file when the stream is disconnected (either thru an RMS $DISCONNECT or an RMS $CLOSE).

E.2.3.1 CXT - Context Block

The context block is used to store the extent internal state information across RMS access calls. The information stored maps access to the file, identifies the current position in the stream and also stores pointers to RMS internal structures.

CXT$L_STMEOS - End of stream VBN

The stream EOS position is the last virtual block of the stream. This pointer is obtained from the Macintosh file header block (VBN 1 in the Macintosh file). This pointer will be updated for writes beyond end of file (if they cross into a new VBN). The field will be written back to the Macintosh file header when the stream is disconnected.

CXT$L_STMFFB - First free byte

The stream first free byte is the first free byte in the last block of the stream. Together, the two fields define the end of the stream. This position will be updated for writes beyond the end of stream. This field will be written back to the Macintosh file header when the stream is disconnected.

CXT$L_WINDOW - Mapping window

The mapping window contains a set of stream mapping pointers. The window is "turned" as the stream is accessed, if necessary. While the concept of turning is defined, it is not implemented for this trial development. A number of issues have been identified which will necessarily require modifications to the data structures to handle window turns.

As defined currently, a backward turn requires the window be turned back to the start of the stream and then turned forward to the requested VBN. A linked list of window pointers could be maintained to reduce this operation but there is a limit as to how many pointers may be maintained. While this implementation would reduce the need for complete turns, it would not eliminate it.

This area needs to be explored for the final development.

CXT$L_WINBASE - Base stream VBN of window

This field defines the starting stream VBN of the current mapping window. This field is used while translating stream VBN to file VBN.

CXT$L_BUF - General buffer

A 512 byte general bufer is allocated. It is not currently used but is envisioned to be necessary to handle window turns.

CXT$L_BUFFER - Data buffer

The current RMS data buffer's data pointer is stored in the context block, as well as the RMS data buffer pointer itself. The data buffer's data pointer is used as the base address for stream reads and writes.

CXT$L_BUFPTR

The current buffer pointer is stored to support sequential record access and also for record de-blocking. As the extent routines completely replace RMS routines, this de-blocking must be performed by the extent.

E.2.4 Global Routines

The trial development supports access routines necessary to test host based applications which deal with text files. These routines are generally limited to $GET, $PUT and $FIND access. The extensions to include $READ and $WRITE are straightforward. $EXTEND functions will be difficult to implement as the routines to actually extend a file are not provided by RMS. $TRUNCATE would simply convert the mapping pointers in the Macintosh file header to FREE pointers.

E.2.4.1 EXT_CONNECT - RMS $CONNECT callout

The $CONNECT callout allocates space for the context block and links it to the internal RMS IRAB structure. This internal structure pointer is passed on all access callouts.

This callout is only a supplement to the normal RMS $CONNECT handling.

E.2.4.2 EXT_GET - RMS $GET callout

The $GET callout translates stream VBNs to file VBNs, reads file blocks and de-blocks file records. File access by RFA or sequential access are supported. Read ahead access is not supported and is ignored.

This routine completely replaces the normal RMS $GET function.

E.2.4.3 EXT_PUT - RMS $PUT callout

The $PUT callout translates stream VBNs to file VBNs and merges records into the file block. File access by RFA or sequential access are supported. Write behind access is not supported and is ignored.

This routine completely replaces the normal RMS $PUT routine.

E.2.4.4 EXT_FIND - RMS $FIND callout

The $FIND callout simply sets the next VBN and BUFPTR fields for subsequent access. The specified file block is read in if not already in the data buffer.

This routine completely replaces the normal RMS $FIND routine.

E.2.4.5 EXT_DISCONNECT - RMS $DISCONNECT callout

The $DISCONNECT callout writes out the final data buffer (if present), updates the Macintosh file header and deallocates the context block and data buffers.

E.2.4.6 EXT_DISPLAY - RMS $DISPLAY callout

The $DISPLAY callout simply sets the FAB record format and attributes field to indicate a VAR CR file format. The XAB chain is also scanned to modify any XABFHC blocks present.

The $DISPLAY callout supplements the $DISPLAY function and the $OPEN function.

E.2.4.7 EXT_MUCK_XABFHC - callback for scan XAB

This routine is called if an XABFHC block is present in the XAB chained, linked to the FAB.

This routine is called as a callback for the RMS$SCAN_XAB_CHAIN routine.

E.3 Restrictions

There are a number of restrictions on the extent and on its use.

E.3.1 File writes

File writes are possible but a direct QIO call must be made to extend the file allocation. It is not clear what internal state must be modified after this is done. (It is assumed that IFB$L_HBK is sufficient).

E.3.2 Buffer usage

The buffer use must be completely understood. There are a number of issues around buffer use which are somewhat unclear. In particular, the IFB$L_AVALCL field, which indicates the number of buffers available, is not maintained by the buffer calls provided. The field is currently updated after calls to RMS$GET_BUFFER. Failure to do this will bugcheck RMS as no buffers are available.

E.3.3 File updates

Tests have been performed with EDT and EMACS to determine if the contents of the file may be modified an preserve the file structure. Unfortunately, both editors (and probably most others) create a new version of the file which does not propagate the file structure. The resource fork is lost. Note that this is precisely the same behaviour as would be seen if the file was edited from a DOS client.

It is clear that the PATHWORKS file system MUST support Macintosh files in both *native mode* file structure and *Macintosh format* file structure. (It is currently envisioned that Macintosh files would not be stored in the *Macintosh format* unless a resource fork was present. Note that converting a *native mode* file to a *Macintosh format* file involves simply moving the first block and adding the Macintosh file header with two mapping pointers. This would have to be done when a resource fork was added to a *native mode* file).

E.3.4 Printing Files

It has been rumoured that various print symbionts do not use RMS to access the records of a file. This needs to be looked into closely as this would mean *Macintosh format* file could not be printed without conversion. This may be an issue for the PATHWORKS Print Subsystem

E.4 Issues

The purpose of this trial development was to demonstrate the capabilities or the RMS extent as applied to Macintosh files, test simple host utilities and get some insight into the issues around the file format. These objectives have been accomplished. It now needs to be decided if we should continue with this two data context file concept or if we should address the problems of representing Macintosh files as two separate files.