8-1-1986

A Study of the Xerox XNS Filing Protocol as Implemented on Several Heterogenous Systems

Edward Flint

Rochester Institute of Technology

School of Computer Science and Technology

A Study of the Xerox XNS Filing Protocol as Implemented on Several Heterogenous Systems

August, 1986

Ed Flint

A thesis, submitted to the Faculty of the School of Computer Science and Technology, in partial fullfillment of the requirements for the degree of Master of Science in Computer Science.

Approved by

Leslie Jill Miller	8/7/86
John L. Ellis	8/26/86
James E. Heliotis	8/23/86

Table of Contents

Table of Contents . . 2

Abstract . . . 5

Key Words and Phrases . . 5

Computing Review Subject Codes . . 5

1. Introduction . . . 6

2. XNS Architecture . . . 8

2.1. ISO Model . . 8

2.2. Servers, Services and Clients . . 10

2.3. XNS Protocols and Standards 12

2.3.1. Physical/Data Link Protocols 12

2.3.2 Internet Transport Protocols 12

2.3.3. Courier (Remote Procedure Call Protocol) 13

2.3.4. Application Protocols . 14

3. Filing Protocol Overview . . . 17

3.1. Clients and Servers . 17

3.2. Procedures . . 18

3.3. Sessions . . . 19

3.4. Users and Authentication . . 20

3.5. Files: content and attributes . . 20

3.6. Handles . . 21

3.7 Controls . . 21

3.8. Scopes 22

3.9. Errors . . 22

4. Implementation Descriptions . . . 24

4.1. Attribute Acceptance - Services . . 25

4.1.1. Xerox 8037 File Service 25

4.1.2 VAX/VMS File Service 25

4.1.3. UNIX File Service 27

4.2. Attribute Usage - Clients 28

4.2.1. XDE FileTool . . 28

4.2.2 VAX/VMS Client . . 29

4.2.3. UNIX Client . . 29

4.3. Procedure Support - Services . . . 30

4.3.1. Xerox 8037 File Service . . 30

4.3.2 VAX/VMS File Service . 30
4.3.3. UNIX File Service . 32
4.4. Procedure Support - Clients 33
4.4.1. XDE FileTool 33
4.4.2 VAX/VMS Client . . 35
4.4.3. UNIX Client. 35
5. Implementation Incompatibilities . . . 37
5.1. Areas of Incompatibility 37
5.1.1. Non-support of Filing constructs by the native operating system 38
5.1.2. Alternate specifications in the Filing Protocol . 38
5.2 Resolving incompatibilities . 39
5.2 1. Non-support of Filing constructs by the native operating system 39
5.2.2 Alternate specifications in the Filing Protocol 42
6. FilingSubset 45
6.1. Overview . 45
6.1.1. Motivation . . 45
6.1 2. Requirements and Goals. 46
6.2 Definition 47
6.3. Attributes . . . 48
6.3.1. Mandatory Attributes 48
6.3.1.1. createdOn 48
6.3 1 2 dataSize. . 48
6.3.1.3. isDirectory . 48
6.3 1 4 modifiedOn . 49
6.3.1 5 pathname . 49
6.3.1.6. type . 50
6.3.2. Implied Attributes . 50
6.3.3. Optional Attributes 50
6.4. Remote Procedure Support. . . 50
6.4.1. Session Support. . . 51
6.4.2. Opening and Closing Files . . 51
6.4.3. Enumerating Files in Directories 52
6.4.3.1. Scopes . 52
6.4.3.2 Attribute Support 52
6.4.4. Storing Files . . 52
6.4.5 Retrieving Files . . 54

6.4.6. Deleting Files . . . 54
6.5. Remote Procedure Restrictions . . . 54
6.6. Remote Errors . . . 56
6.7. Procedures and Attributes 56
6.8. Courier Definition 60
References . . 67
Appendices
A. FilingSubset Implementor's Guide . . 69
B. Filing Protocol . . . 70

Abstract

The Xerox Network System is composed of heterogeneous processors connected across a variety of transmission media. A series of protocols is defined to describe the communication mechanisms between system elements. One of these protocols, the Filing Protocol, defines a general purpose file management system. Current implementations of the protocol, although derived from the Xerox specification, fall short of providing the interconnectivity between elements desired in a heterogeneous network system. The definition of an easily implemented protocol subset that provides the common file system functions of retrieval, storage, enumeration/location and deletion is derived from experiences with several implementations. This definition and an accompanying implementation document provide a mechanism to guide future implementations toward increased interconnectivity.

Keywords and Phrases
Network Protocols; Distributed File Systems

Computing Review Subject Codes

C.2.2 **[Computer-Communication Networks]**: Network Protocols *Protocol architecture*; C.2.4 **[Computer-Communication Networks]**: Distributed Systems *Distributed applications*; D.4.3 **[Operating Systems]**. File Systems Management - *Distributed file systems*;

1. Introduction

The Xerox Network Systems (XNS) architecture defines a series of protocols for use between systems in a distributed computing environment. These protocols provide a mechanism for communicating between a variety of machines across a variety of transmission media, and encompass the full range of the ISO/OSI reference model from the physical through application layers. Application protocols are defined for filing, printing, network object lookup and user authentication.

The Xerox Network Systems provide a general purpose file management system which is heirarchical in nature and supports a wide variety of functions, including file transfer, access control, file location and enumeration, random access, serialization and deserialization of directory heirarchies. The Filing Protocol represents a formal definiton of this file system as well as a guide for accessing the system.

As the network model has been refined, implementations of this model became more prevalent. It became clear that there are many operating systems and application processes which 1) have a specific need for certain portions, or just a subset, of the Filing functions and/or 2) do not have the requirements or resources to support the Filing Protocol in its entirety. For example, a subset intended for simple file transfer and enumeration has a range of uses within the areas of distributed printing, electronic publishing and interconnectivity of heterogeneous systems.

Currently several commercial implementations of the Filing Protocol exist for different system configurations. Although each of these implementations is based upon the XNS specification for Filing, in reality, each of these implementations falls short of the full implementation The result is a lack of interconnectivity because there is no formal definition of a standard Filing subset to guide coordination of the implementation schemes

This thesis is divided into several sections documenting the evolution of a subset of the Filing Protocol for use as a simple file transfer protocol. The thesis was motivated by my inital experiences involved with providing compatible file transfer functionality between three existing Filing implementations My work resulted in the formal definition and incorporation of the FilingSubset Protocol into the Xerox Filing Protocol and the development of the accompanying implementor's guide.

Section 2 is an overview of the XNS architecture and the relationship between the XNS protocol family and the ISO/OSI Reference Model. Section 3 describes the concepts and terms defined by the Filing Protocol as a foundation for understanding the more detailed discussions in later sections. Section 4 describes several existing Filing implementations and points out the incompatibilities evidenced between them. This description highlights the difficulty in providing interoperability between various implementations. Section 5 presents a specific solution to the problems from these implementations of the Filing Protocol. The choices made in this section are then formalized into a subset of the Filing Protocol that provides the intended file transfer functionality. This formal specification of the FilingSubset Protocol is presented in Section 6. Appendix A includes the FilingSubset Implementor's Guide, a detailed implementation strategy for the FilingSubset Protocol. This strategy describes the implementation of the protocol from general perspective and describes specific support required on the UNIX and VMS operating systems. A copy of verion 6 of the Xerox Filing Protocol is included in Appendix B for reference. This version contains the Xerox definition of the FilingSubset Protocol, which is an edited version of the specification presented in Section 6 of this thesis

2. XNS Architecture

The XNS architecture defines a family of protocols and standards to provide for the exchange and handling of information within a distributed network environment. This architecture is an outgrowth of the 3MB Ethernet used within Xerox from the early 1970's into the 1980's. This experimental network was developed at Xerox's Palo Alto Research Center (PARC) and has been the subject of numerous published papers.

The XNS protocols represent a refinement of the early research protocols. As the architecture graduated to the 10MB Ethernet and other transmission media, research continued into numerous application areas and products were developed to provide greater diversity and richer functionality.

This section presents an overview of the XNS architecture and its relationship to the ISO/OSI reference model. The intent is to provide the reader with some background on the structured approach of the XNS architecture and the inter-relationship between the various protocols that it defines. This inter-relationship is important in understanding the definition and use of the Filing Protocol later in this thesis.

2.1. ISO Model

In 1981 the International Organization for Standardization (ISO) defined a reference model [11] for Open Systems Interconnection (OSI) consisting of 7 layers of protocols as shown in Figure 2.1. Each of these layers offers distinct functions that depend upon the lower layers and in turn are relied upon by the higher layers. The layers included in this model are:

Layer 1: Physical	This layer performs the actual transmission of data over the physical communication medium
Layer 2: Data Link	This layer provides reliable transmission by organizing the data into frames and providing error detection and optionally correction

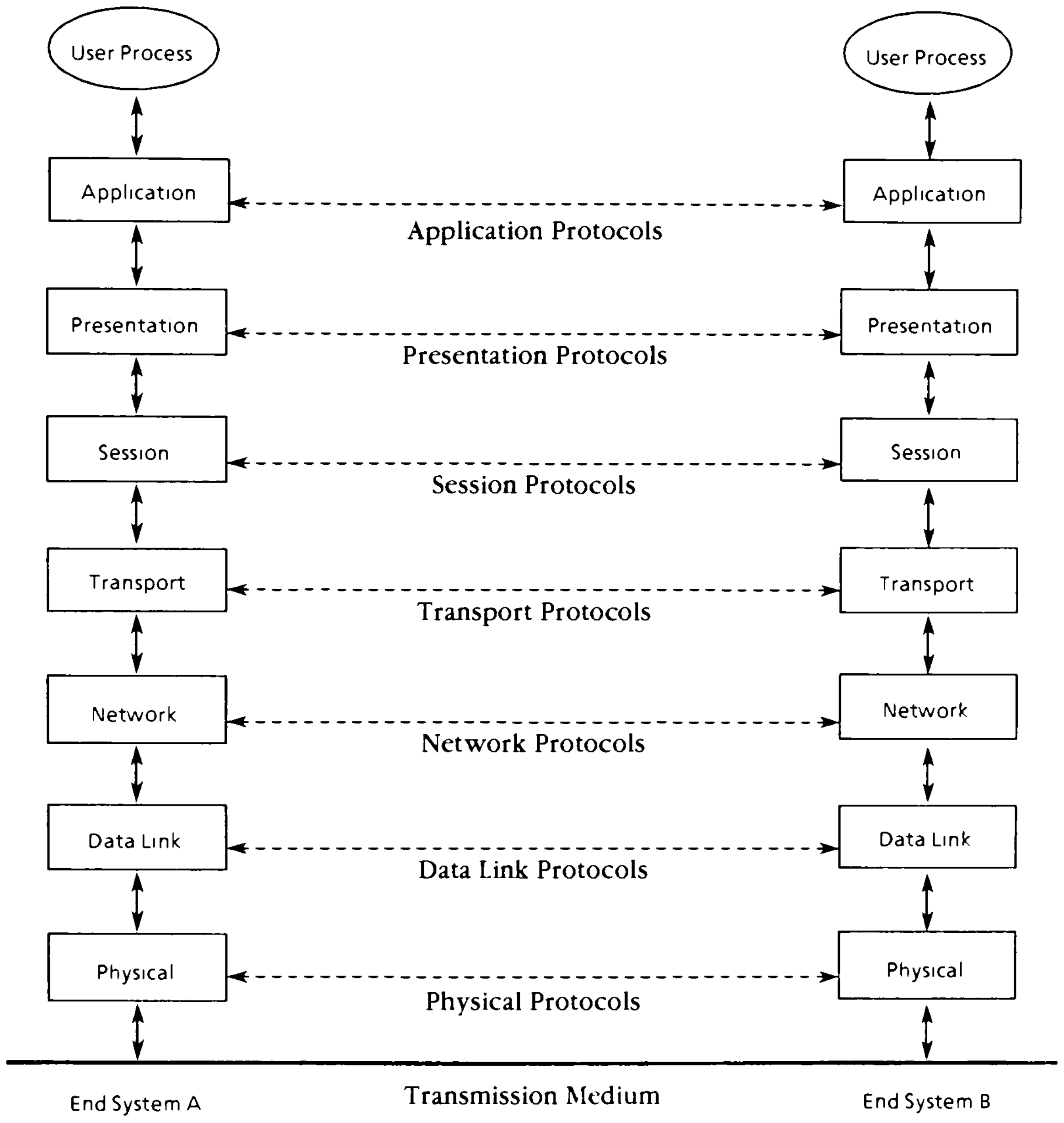

Figure 2 1 ISO/OSI Reference Model Protocols

Layer 3: Network	The responsibility of this layer is to organize higher level data into packets which are transmitted to the recipient To perform this, the network layer must provide for packet addressing and routing
Layer 4: Transport	This layer provides reliable end-to-end transmission independent of any intermediate nodes.
Layer 5: Session	The session layer provides for the establishment, management, synchronization and termination of a user level connection from source

	to destination. This connection exists independent of the underlying transport connections
Layer 6: Presentation	This layer translates user data objects into the form transmitted between systems. Common translations may include data compression, encryption and character code conversions.
Layer 7: Application	This layer performs applications specific to the environment in which the network is used.

The XNS architecture is organized into a series of protocol layers which closely resemble the ISO model. However, in several instances, a single XNS layer corresponds to multiple ISO layers. The XNS layers and their association to the ISO model is depicted in Figure 2.2.

2.2. Servers, Services and Clients

Two terms used quite frequently when describing the XNS architecture are those of *clients* and *servers*.

Any device attached to the network for the purposes of providing a service to network users is referred to as a *server*. The collection of software which accomplishes a specific task by following a defined protocol for interaction is commonly referred to as a *service*.

Several types of servers are evident in the XNS environment. A dedicated server performs a specific task and nothing else. A print server is an example of such a dedicated server. A server may also be a general purpose computer capable of providing several services simultaneously, such as naming, authentication and mail services. In addition, a user workstation may function as a network server on occasion, for example, a temporary file service.

The entity which makes requests of a service is called a *client*. Clients typically perform work on behalf of a user; however, services may in fact be clients of other services under certain conditions, such as when a file service contacts an authentication service to validate user credentials

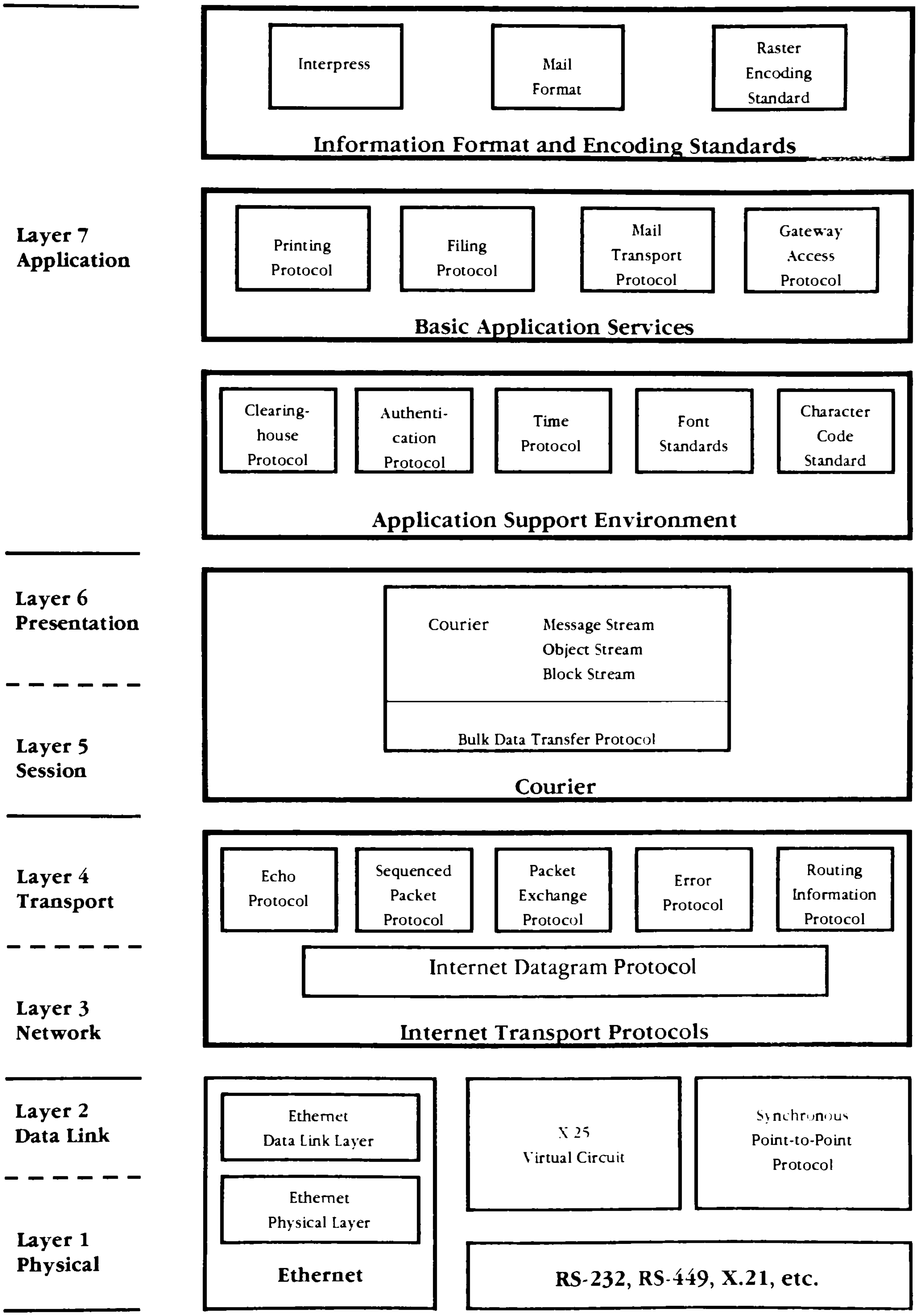

Figure 2.2 XNS Protocols

2.3. XNS Protocols and Standards

2.3.1. Physical/Data Link Protocols

The primary transmission medium used by XNS is the Ethernet [7]. The Ethernet specification describes both the physical and data link layers as defined by the ISO model. XNS also supports other standard physical interfaces (RS-232, RS-449 and X.21) and data link protocols (X.25 and synchronous point-to-point protocol).

2.3.2 Internet Transport Protocols

Typically, a network consists of many sub-network configurations which are connected in some manner. Each of these sub-networks may use a different physical transmission medium. However, an internet protocol provides the functions which allow these sub-networks to be addressed as a single uniform network. The XNS network, like other packet switching networks, routes each packet, or datagram, individually. This concept differs from that of a virtual circuit in that no prior setup is necessary between the respective source and destination nodes before data can be exchanged. A transport protocol must provide the mechanisms to insure that packets are delivered to the receiver in the same order in which they were sent, with no duplication or omission.

The XNS Internet Transport Protocols [9] provide these services through several layered protocols, each performing a distinct function.

The Internet Datagram Protocol defines the fundamental unit of data, an internet packet, which is passed within the internet, and also defines the means for these packets to be addressed, routed and delivered. Each packet contains addressing information (source and destination addresses), control information (checksum, length, packet type and transport control) and data (encoded higher level protocol data).

The Sequenced Packet Protocol (SPP) provides for the reliable delivery of packets from source to destination. It is this protocol that guarantees the delivery of packets in order with no duplication or ommission.

The Packet Exchange Protocol (PEP) provides a facility for efficient request-response oriented communication. This is typically used when a single internet packet can contain the response data and the reliability achieved through the use of the Sequenced Packet Protocol is not required.

The Error Protocol provides a standard mechanism for errors to be communicated.

The Routing Information Protocol defines the means by which network routing tables are maintained. Each network node must maintain a routing table which is used to route individual packets from one network to another. This protocol provides for the broadcast and maintenance of the information contained within the tables.

The Echo Protocol defines a simple means to verify the existence and correct operation of any network host.

2.3.3 Courier (Remote Procedure Call Protocol)

The Courier Protocol [6] defines the manner in which clients and servers interact within the distributed XNS environment. Courier defines a single request/reply, or transaction, mechanism upon which all higher level XNS protocols are based

Courier is based upon the remote procedure call model, where an active client invokes operations provided by a passive network server. A Courier call is analagous to a subroutine call where arguments are passed on the call and values may be conveyed on the return, as shown in Figure 2.3.

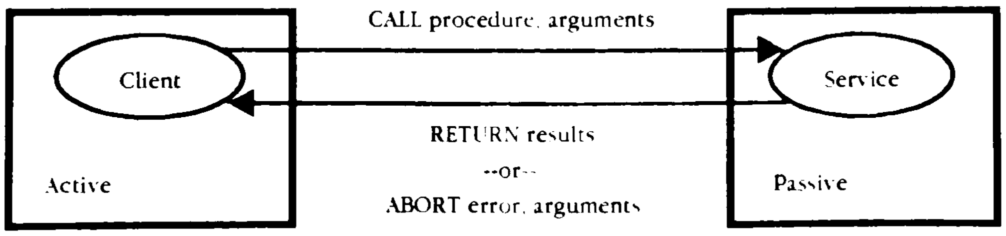

Figure 2.3 Courier model

Courier is defined as consisting of three layers: the block stream, the object stream and the message stream. The block stream encapsulates the binary data from higher layer packets for transmission by the

Sequenced Packet Protocol. The object stream imposes structure onto this binary data in the form of common data types (such as booleans, cardinals, strings, etc.). The message stream structures these data types into Courier procedure calls and replies.

The message and data types supported by Courier also form the definition for a language in which all XNS application level protocols are written. The formal definition for higher level protocols consists of transaction-oriented expressions written in this Courier language.

There may be instances where applications desire to send large amounts of data for which it does not make sense to pass the data as a procedure argument. For this reason, Courier also includes a Bulk Data Protocol [2] which defines the mechanism for transmitting simple streams of data between two Courier applications. For the purposes of transmission, this data may be viewed as a single Courier data object which is interpreted based upon the context of a recent Courier call. Bulk data transfers may take place in two forms: ***immediate***, where the initiator is either the sender or receiver of the data and ***third party***, where the initiator causes the data to be sent from a separate sender to another receiver

2.3.4 Application Protocols

The XNS application protocol layer consists of a set of protocols and standards that support and enhance those protocols. This layer is further subdivided into three distinct levels, the Application Support Environment, Basic Application Services and Information Format and Encoding Standards

The lowest level, the Application Support Environment, provides those services used by the majority of higher level applications. Specifically this level provides the following functions to present a secure and reliable network for the higher levels:

- location of resources and individuals within the network
- user authentication
- a common time base for the entire network
- a common character encoding format for files and Courier strings

- standardization for the use of fonts and font services

The Clearinghouse Protocol [4] defines the mechanisms for object naming and addressing within the network. The Clearinghouse service and associated database is decentralized and replicated throughout the network. This protocol defines the information stored by the service and the means whereby users can retrieve the information.

The Authentication Protocol [1] defines the methods used by clients and servers to identify each other in a secure and reliable manner. This protocol defines the credentials which a user provides to gain access to the network services and the procedures employed by clients and services to verify these credentials

The Time Protocol [14] defines a standard format for the representation of time within the network and the manner in which network clients receive the current time from a network time service.

The Character Code Standard [3] defines the encoding of character data within the network. This encoding provides character assignments for Ascii and ISO 646 characters as well as special characters from many different alphabets, mathematical symbols and graphics characters.

The second level, consisting of the Basic Application Protocols, defines the protocols needed by users of the network on a regular basis. These protocols provide the following common applications:

- a global network file system
- remote printing
- mail services
- interactive terminal services

The Printing Protocol [12] defines the manner in which print requests are communicated to print servers and the status of print requests is determined.

The Filing Protocol [8] defines a general purpose distributed file system and the mechanism for transfer of files within the network.

The Gateway Access Protocol supports terminal emulation and file exchange between non-XNS clients and XNS services as well as XNS clients and non-XNS services.

The Mail Protocol defines the format of mail messages and the means employed to send and receive these messages.

The third level, consisting of the Information and Encoding Standards, defines specific formats or languages for the encoding and decoding of files within the network. This level provides a uniform format for similar files in an effort to provide users with the ability to edit, print or transmit a file anywhere within the network, regardless of host hardware or software.

Interpress [10] defines a standard representation for documents which are to be printed It is this language that is interpreted by a print service prior to the actual printing. Interpress provides device independence for the creators of print files within the network.

The Raster Encoding Standard is the definition of a general purpose encoding for digital images This standard is used to represent stand-alone images to be viewed as well as images included within Interpress files.

3. Filing Protocol Overview

The Xerox Network Systems provide a general purpose file management system which is hierarchical in nature and supports a wide variety of functions, including file transfer, access control, file location and enumeration, random access, serialization and deserialization of directory heirarchies. The Filing Protocol represents a formal definiton of this file system as well as a guide for accessing the system.

The protocol being defined by this thesis evolved as a subset of the Filing Protocol because there were existing implementations of the Filing Protocol and compatability with these implementations could be preserved. Since the Filing Protocol contained the level of functionality desired for the protocol being defined, that functionality simply needed to be separated from the full protocol into the definition of the subset.

A brief description of the basic concepts of the Filing Protocol is presented here to provide a background for the remainder of the thesis. A detailed description of the protocol is not intended, since many of these details are not relevant to this thesis. Instead, the complete Xerox specification of the Filing Protocol is included, for reference, in Appendix B. The description of the protocol that follows defines the terms and concepts intrinsic to understanding the work presented later

3.1 Clients and Servers

The Xerox Filing Protocol is a formal specification of a general purpose file system. It defines the interaction that takes place between a *filing client*, an entity requesting work to be done on behalf of a user, and a *file service*, the entity accepting requests for work.

A client may have an explicit user interface, where specific user inputs control the client process actions, or it may be invoked during the execution of a process where there is no user interaction.

A file service may, but is not required to, exist on a separate processor or server. Multiple separate file services may in fact reside on a single server. A general time-sharing computer may provide a file service to allow some level of file access to users not resident on that machine

All files within a file service are organized in a heirarchical manner. Each service contains a single root directory which in turn contains various descendant subdirectories and files. All files residing directly in a directory are referred to as children of the directory file. The directory which contains a file is, in turn, that file's parent.

3.2 Procedures

The Filing Protocol defines a set of procedures for various levels of file access and transfer. The procedures defined in the Xerox specification are:

Session Management--

Logon	establish a session
Logoff	terminate a session
Continue	retain an open session during a period of inactivity

File Access--

Open	open a file
Close	close a file
Create	create a file with no content
Delete	delete a file

Access Control Management--

ChangeControls	modify the controls in effect for a previously opened file
GetControls	retrieve the controls in effect for a previously opened file
UnifyAccessLists	unify the access lists for a subtree

Attribute Management--

ChangeAttributes	modify the attributes for a file

GetAttributes	retrieve the attributes for a file

Remote File Management--

Copy	copy a file and its descendants to another directory
Move	move a file and its descendants to another directory

File Transfer--

Retrieve	retrieve the contents of a file
Store	create a file with contents
Replace	replace the contents of an existing file
Serialize	encode a file and its descendants into a stream of bytes
Deserialize	reconstruct a file and its descendants from a stream of bytes

Random Access--

ReplaceBytes	overwrite or append to the existing content of a file
RetrieveBytes	read a range of bytes from an existing file

File Enumeration--

Find	locate and open a file in a directory
List	return attributes about files in a directory

3.3 Sessions

The Filing Protocol is a session oriented protocol, in that an explicit user logon must be performed to establish a session, with a subsequent logoff terminating the session. All procedure calls issued by the client relative to the initial logon request take place in the context of that session.

A unique identifier called a session handle is used by both client and service processes to maintain the context of a user session. Upon successful validation of the user credentials, the service creates the session handle and returns it to the client. This handle is then used on all subsequent client calls within the same session until a logoff occurs. The logoff signals termination of the session and causes the client and service to discard the corresponding session handle

Sessions may vary greatly in their duration and amount of activity. A file service may terminate a session at any time when a remote procedure call is not in progress.

3.4 Users and Authentication

Each session requires that some user identification be presented to the service process The Filing Protocol provides for the use of both primary and secondary credentials. Primary credentials are in a form defined in the Authentication Protocol [1] and are validated against a network Authentication service. Secondary credentials are file service specific and may be required as needed by the underlying service operating system. The service performs the necessary validation of the credentials presented as a part of logging on to the service.

3.5 Files: content and attributes

The basic unit of operation within the Filing Protocol is that of a file A file is a logical grouping of data which is stored as a single unit.

Each file is viewed as either temporary or permanent. Temporary files do not reside in any directory and only exist for the duration of all sessions which have the file open. Permanent files reside in a specific directory and remain there until explicitly deleted

Each file consists of two distinct parts: content and attributes The content of the file is the data actually contained in the file as a sequence of eight-bit bytes This data is uninterpreted by the file service except in the case where a specific format is defined for the transfer of the data

Attributes are additional data items associated with a file's content that may be used to provide additional identification or description of a file. Attributes may either be interpreted, in which case they have a specific meaning to a file service and result in a defined behavior, or uninterpreted, in which case they are stored on the file service but interpreted only by the client

Each attribute is designated by an attribute type, many of which are defined by the Filing Protocol. Each of these defined attribute types must be interpreted by all file services implementing the Filing Protocol.

Attributes normally are obtained and modified by explicit client action; however, certain procedures do result in file service modification of attributes.

Interpreted attributes exist in several different categories:

identification related	fileID, isDirectory, isTemporary, name, pathname, type and version
content related	checksum, dataSize and storedSize
parent related	parentID and position
event related	createdBy, createdOn, modifiedBy, modifiedOn, readBy and readOn
directory related	childrenUniquelyNamed, numberOfChildren, ordering, subtreeSize and subtreeSizeLimit
access related	accessList and defaultAccessList

3.6 Handles

Filing clients issue requests to a file service to operate on files When a service creates a new file or opens an existing file on behalf of a client, a file handle is created and returned to the client. This handle is used by the client and service to identify the file within the context of a session. The handle is discarded once the file is closed or the session is terminated

The actual structure of a file handle is service specific and is only interpreted by the file service.

3.7 Controls

A client may request access to a file with certain access characteristics to be imposed by the service. These characteristics, called controls, are presented with the request to open the file and specify the intended interaction with a file by the client. Controls can be used by the service to determine the level of interaction with a file that can occur simultaneously by several clients. Since controls are relative to a given file handle, they can also be used to control access by the same client if a file is opened multiple times in a single session. Specifically, controls can designate:

lock	a type of lock (none, share or exclusive)
timeout	a timeout period to be used in waiting for a lock
access	a set of access permissions requested for a file

3.8 Scopes

When clients enumerate or locate files, they specify arguments, called scopes, to describe the selection criteria to use Scopes can specify the following:

count	the maximum number of files to present to the client
depth	the nesting level of descendants to consider during the search
direction	the direction of examination, either from beginning to end or end to beginning
filters	the conditions on attributes to be used for identification (condition is True or False, condition equal to a constant, or a logical combination of conditions)

3.9 Errors

Consistent with the Courier model, the Filing Protocol will return appropriate errors when a procedure call cannot be serviced correctly. Specifically, the following classes of errors may be returned

AccessError	the desired file access is not possible
ArgumentError	a specified argument (attribute, control or scope) type or value was invalid
AuthenticationError	the user could not be validated

ConnectionError	the bulk data connection could not be established
HandleError	the specified file handle was invalid
InsertionError	the specified file could not be inserted into the directory
RangeError	the specified random access byte range was invalid
ServiceError	the session could not be created or terminated
SessionError	the specified session handle was invalid
SpaceError	the specified storage for the file could not be allocated
TransferError	the bulk data transfer encountered a problem
UndefinedError	an implementation dependant problem occured

4. Implementation Descriptions

This section presents descriptions of several existing implementations based upon the Xerox Filing Protocol definition. These descriptions are intended to demonstrate the potential areas of incompatibility arising from the differences in each implementor's choice of a specific subset.

This section will discuss the following implementations : 1) the Xerox Network Systems 8037 File Server and Xerox Development Environment FileTool client, 2) Implementation A for VAX/VMS and 3) Implementation B for VAX/4.2BSD UNIX and System V UNIX. The anonymous identification of the latter two implementations results from the proprietary nature of the corresponding commercial products. They are included in the discussion because they are working examples of different implementations, each of which attempted to support useful functionality. They provide concrete examples of the various forms of incompatibilities which may be experienced.

The descriptions presented will compare the implementations with regard to two categories: attribute acceptance/usage and procedure support. The discussion of attribute acceptance and usage focuses on the acceptance and retention of Filing attributes by a file service and the usage of attributes by the clients. The discussion of procedure support describes where the implementations differ from the Filing Protocol in their support for Filing procedures. This description does not encompass the full detail of these implementations; rather, it is intended to provide a perspective on the alternatives available when implementing a subset of the protocol and the interoperability problems which are a result of these alternatives.

The problems pointed out in this section provide the motivation for this thesis. They forcibly demonstrate the need for a single well defined protocol, in order to achieve interoperability between heterogeneous implementations. Section 5 describes some of the specific choices made in defining this standard subset and Section 6 specifies the resulting Filing Protocol subset which provides the desired file transfer facility.

4.1 Attribute Acceptance - Services

Table 4.1 lists the attributes allowed on those procedures accepted by at least two of the various server implementations. From this table, it is evident that only a small subset of attributes is actually accepted and subsequently retained by each of the implementations and that the intersection of all three represents an even smaller subset.

4.1.1 Xerox 8037 File Service

The Xerox 8037 File Service supports the full Filing Protocol with respect to attribute support The List and GetAttributes procedures allow specification of any attributes and will return appropriate values for the attributes requested. The Open and Store procedures allow only those attributes specified as legal in the series of tables in Section 3.10 of the Filing Protocol [8]. Attribute specification is also supported for the Copy, Deserialize, Move and Replace attributes although they are not included in Table 4.1 since neither the VMS or UNIX implementations support these procedures

4.1.2 VAX/VMS File Service

The VAX/VMS service only supports those attributes which readily map into specific VMS file system constructs. In addition, the range of attributes supported is not consistent across each of the procedures implemented. Attributes that are not included in the table are not accepted and the corresponding procedure is rejected. However, not all attributes accepted on various procedures are. in turn, retained by the file service.

A List procedure is rejected if an attribute type other than createdBy, createdOn, dataSize, fileID, isDirectory, modifiedOn, name, readOn, type or version is specified. Appropriate values are returned for each of these attributes.

The Open procedure only accepts the fileID, name, parentID and version attributes. The Open will be rejected if the parentID value is not nullfileID

Procedure	Xerox 8037 File Service	VAX/VMS File Service	UNIX File Service
ChangeAttributes	accessList checksum childrenUniquelyNamed createdBy createdOn defaultAccessList name ordering position subtreeSizeLimit type version	procedure not supported	all attribute types
Create	accessList checksum childrenUniquelyNamed createdby createdOn dataSize defaultAccessList isDirectory isTemporary name ordering position subtreeSizeLimit type version	procedure not supported	all attribute types
GetAttributes (values returned for these attributes)	all attribute types	procedure not supported	dataSize isDirectory modifiedOn name type
List (values returned for these attributes)	all attribute types	createdBy createdOn dataSize fileID isDirectory modifiedOn name readOn type version	dataSize isDirectory modifiedOn name type

Table 4.1
Service acceptance of attribute types

Procedure	Xerox 8037 File Service	VAX/VMS File Service	UNIX File Service
Open	fileID name parentID pathname version	fileID name parentID version	all attribute types
Store	accessList checksum childrenUniquelyNamed createdby createdOn dataSize defaultAccessList isDirectory isTemporary name ordering position subtreeSizeLimit type version	createdBy createdOn dataSize isDirectory name type version	all attribute types

Table 4.1 (continued)
Service acceptance of attribute types

The Store procedure is rejected if the client specifies an attribute type other than createdBy, createdOn, dataSize, isDirectory, name, type or version. Only the attributes name, type and version are actually retained with the file; values for the other attributes are accepted but ignored.

The ChangeAttributes, Create and GetAttributes procedures are not supported by the VMS implementation.

4.1.3 UNIX File Service

The UNIX file service supports a different subset of Filing attributes. Again, only those attributes readily mapped into Unix file system constructs are supported and retained with the files. The file service will accept, but not retain, attributes which it does not support.

The ChangeAttributes procedure only allows the name attribute to be specified. If the name attribute is not specified, the procedure is rejected; however, specification of other attributes does not cause the procedure call to be rejected.

The Create, Open and Store procedures accept all attributes and do not reject the procedure call according to the legality of the attributes as specified in the Filing definition. Although specification of the isDirectory and name attributes are necessary for the service to process the Open procedure call, the procedure is not rejected if either of these attributes are missing. All attributes specified on the Create and Store procedures, with the exception of isDirectory and name are not retained

The dataSize, isDirectory, modifiedOn, name and type attributes are accepted by the GetAttributes and List procedures and appropriate values returned. Other attributes are accepted, but values are simply omitted from the list returned to the client.

4.2 Attribute Usage - Clients

Table 4.2 compares the use of attributes by clients on those procedures supported by one or more clients For each procedure, the client may use any combination of the attributes listed This table in conjunction with Table 4.1 is useful in identifying the incomptabilities between the various clients and services

4.2.1 XDE FileTool

The Xerox FileTool uses a subset of the defined Filing attributes depending upon the higher level user function being performed. For example, the Open procedure may specify the pathname or fileID attribute depending upon whether a previous GetAttributes was performed to determine the fileID attribute value. The List procedure only requests those attributes actually specified by an option in the user interface.

Usage of attributes by the FileTool client is legal as defined by the tables in Section 3.10 of the Filing Protocol [8].

Procedure	XDE FileTool	VAX/VMS Client	UNIX Client
ChangeAttributes	procedure not used	procedure not used	name
Create	procedure not used	procedure not used	name type
GetAttributes	pathname	procedure not used	dataSize modifiedOn name type
List	createdBy createdOn dataSize modifiedOn name pathname readOn type version	dataSize isDirectory name type	dataSize modifiedOn name type
Open	fileID name pathname	name	isDirectory name
Store	createdOn dataSize name type	dataSize isDirectory name type	dataSize modifiedOn name type

Table 4 2
Client usage of attribute types

4.2.2 VAX/VMS Client

The VAX/VMS client only uses those atributes necessary to identify files and maintain round-trip integrity of the data between Xerox and corresponding VMS services. Those attributes supported include dataSize, isDirectory, name and type

4.2.3 UNIX Client

The UNIX client also makes use of only a small set of attributes for the same reasons as the VMS client However, the modifiedOn attribute is also specified on several procedures even though it is illegal as defined by the Filing Protocol.

4.3 Procedure Support - Services

Table 4.3 depicts the level of support for Filing defined procedures by the various implementations. Each non-empty entry describes digressions from the Filing Protocol that are evidenced by these implementations.

4.3.1 Xerox 8037 File Service

The Xerox 8037 File Service implementation was the original implementation of the Filing Protocol and provides support for much of the protocol as defined. Some anomalies are present which are of specific importance when considered in conjunction with the other implementations.

The List procedure does not allow specification of the pathname attribute on a filter of type matches

The Create and Store procedures allow the isDirectory and type attributes to have conflicting values. The isDirectory attribute is the sole indicator of directory files on the file service; specification of a tDirectory type value without an accompanying isDirectory value of TRUE does not create a directory on the file service.

4.3.2 VAX/VMS File Service

The VAX/VMS implementation supports a limited set of procedures. The Logon, Logoff, Open, Close, Delete, Store, Retrieve, List and Continue routines are the only procedures supported. Of these, only the Logoff, Close, Delete and Continue procedures do not impose some restrictions upon their use.

The Logon procedure will only accept Authentication credentials of type simple. An appropriate rejection is issued if other credentials types are specified.

The Open procedure is restrictive in the type of attributes it allows. The acceptance of attributes is described in Section 4.1.1.

The Store procedure also imposes restrictions on the types of attributes accepted and subsequently retained as discussed in Section 4.1.1. Only controls of type exclusive lock are accepted; all others are

Procedure	Xerox File Service	VAX/VMS File Service	UNIX File Service
Logon	--	credentials (simple)	credentials (not in Xerox form)
Logoff	--	--	--
Open	--	attribute support controls (only exclusive lock)	attribute support all controls accepted but not implemented
Close	--	--	--
Create	--	not supported	attribute support all controls accepted but not implemented
Delete	--	--	--
GetControls	--	not supported	not supported
ChangeControls	--	not supported	not supported
GetAttributes	--	not supported	--
ChangeAttributes	--	not supported	--
Move	--	not supported	not supported
Store	--	controls (only exclusive lock) remove CR from tText records	separate procedure for tText storage all controls accepted but not implemented
Retrieve	--	added CR to tText records	separate procedure for tText retrieval
Serialize	--	not supported	not supported
DeSerialize	--	not supported	not supported
Find	--	not supported	not supported
List	--	scopes (count or filter of type matches on pathname or equal on version attribute)	implemented own wildcarding procedures scopes (not implemented)
Continue	--	--	--
UnifyAccessLists	--	not supported	not supported
RetrieveBytes	--	not supported	not supported
ReplaceBytes	--	not supported	not supported

Table 4.3
Service implementations - Non-support of procedures

rejected. A conversion of file content is performed on incoming files of type tText to preserve the editability of simple text files throughout the network. The service assumes that a carriage return is the record delimiter and subsequently strips these characters to form records when writing to VMS files.

The Retrieve procedure performs the inverse conversion of file content if the file being transferred is of type tText. Specifically, a carriage return character is inserted at the end of each record as it is entered into the bulk data stream.

The VMS service does not allow storage or retrieval of directory files.

The List procedure only allows specification of a few types of scopes: 1) count or 2) filter of type matches on the name and 3) filter of type equal on the version attribute. In addition, the count scope is not used when formatting the returned attribute list. The List procedure also imposes restrictions on acceptance of attribute types as described in Section 4.1.1.

4.3.3 UNIX File Service

The UNIX service implementation supports only the Logon, Logoff, Open, Close, Create, Delete, ChangeAttributes, GetAttributes, Store, Retrieve, List and Continue procedures. The Logoff, Close, Delete and Continue are the only procedures which impose no restrictions beyond the Filing Protocol definition

The Logon procedure accepts only credentials in the simple form; however, the encoding of the credentials data does not adhere to the Authentication Protocol, since the XNS model of authentication is not supported.

The Open and Create procedures impose restrictions on the types of attributes accepted, as described in Section 4.1.3. In addition, several non-Filing attributes are defined and are required by the service to identify files. These procedures also accept all controls types, even though no form of controls is provided by the service.

The Store procedure, like Open and Create, accepts all controls types without providing specific support for them. The preservation of editability of simple text files is also provided, but in a manner unlike that chosen by the VMS implementation. Specifically, an implementation-specific procedure is defined which is used to store files of type tText. The content of these files is actually maintained across the bulk data stream by defining a format for encoding the data into the bulk data stream.

The Retrieve procedure is identical to that defined in the Filing definition. However, a separate routine is implemented for retrieval of files of type tText. This procedure simply performs the inverse of the encoding performed by the analagous Store procedure.

The List procedure was totally incompatible with the Filing definition since the scope selection mechanism was not implemented. The client used an implementation-specific procedure to establish the context for wildcard searches on the service. The List procedure was then used to determine the next file matching the previously specified file specification and return the requested attributes.

4.4 Procedure Support - Clients

Table 4.4 describes those procedures used by the various client implementations and any restrictions on their use. Many of the restrictions described are complementary to restrictions enforced by the corresponding file service.

4.4.1 XDE FileTool

The Xerox client only uses the Logon, Logoff, Open, Close, Delete, GetAttributes, Store, Retrieve, List and Continue procedures. All procedures are implemented in accordance with the Filing definition. The Delete procedure does, however, make use of multiple transport connections specifying a single Filing session handle. Although this feature is allowed by the definition of Courier, it is not used by the other client implementations.

Controls specified on the Open and Store procedures are of type empty. The List specifies a filter of type matches on the name attribute.

Procedure	XDE FileTool	VAX/VMS Client	UNIX Client
Logon	--	credentials (simple)	credentials (not in Xerox form)
Logoff	--	--	--
Open	attribute use controls (empty)	attribute use controls (empty)	attribute use controls (empty)
Close	--	--	--
Create	not used	not used	attribute use
Delete	uses multiple transport connections for a single session	--	--
GetControls	not used	not used	not used
ChangeControls	not used	not used	not used
GetAttributes	--	not used	attribute use
ChangeAttributes	not used	not used	attribute use
Move	not used	not used	not used
Store	attribute use controls (empty)	attribute use controls (empty) add CR to tText records	attribute use controls (empty) separate procedure for tText storage
Retrieve	--	remove CR from tText records	separate procedure for tText retrieval
Serialize	not used	not used	not used
DeSerialize	not used	not used	not used
Find	not used	not used	not used
List	scopes (only filter of type matches on name attribute)	attribute use scopes (only filter of type matches on name attribute)	attribute use implemented own wildcarding procedures scopes (not implemented)
Continue	--	--	--
UnifyAccessLists	not used	not used	not used
RetrieveBytes	not used	not used	not used
ReplaceBytes	not used	not used	not used

Table 4 4
Client implementations - Non-support of procedures

4.4.2 VAX/VMS Client

The VMS client implements the same procedures as FileTool with the exception of the GetAttributes procedure. Attribute usage on those procedures is legal as defined by the protocol, although only a limited number of attributes are used, as described in Section 4.2.2.

Simple credentials are specified by the client on a Logon procedure.

Controls specified on the Open and Store are empty. The List procedure performs selection with a filter of type matches on the name attribute.

The Store and Retrieve procedures perform the same conversion of tText file content as the VMS service

4.4.3 UNIX Client

The UNIX client implements the ChangeAttributes and Create procedures in addition to the procedures supported by the XDE client. The subset of attributes implemented for use on those procedures is described in Section 4.2.3.

The Logon procedure uses credentials of type simple, but does not encode the data according to the Xerox Authentication Protocol.

Empty controls are specified on the Open, Create and Store procedures. The Create procedure is only used to create empty directory files. However, the type attribute is used to convey a type of tDirectory without an accompanying isDirectory attribute. This use is legal according to the Filing definition but is in conflict with the Xerox file service implementation

The client also uses two implementation-specific procedures for the storage and retrieval of type tText files. These procedures perform the same encoding/decoding of file content as their counterparts implemented in the UNIX file service.

Wildcard listing is also performed via an implementation-specific procedure which is used by the client to set up an initial search context. The List procedure is then used to retrieve the desired attributes about each file matching the wildcard criteria. The Filing selection mechanism of scopes is not implemented by the client.

5. Implementation Incompatibilities

A well defined protocol, such as the Filing Protocol, minimizes the opportunities for incompatibilities within implementations of the protocol. However, implementation of only a portion of the full protocol increases the possibility that the interoperability will be reduced. Section 4 describes specific points where the implementations discussed were not compatible because there was only a small intersection in the subsets of the protocol chosen for implementation. The resulting disparity in implementation strategies points out the need for a single subset which is supported across all implementations.

Maximum compatibility between implementations of a subset protocol is based on a well-specified set of procedures and attributes which form a required basis for all implementations to support. This definition must precisely specify the actions performed by each of the procedures and the handling of the defined attributes within each of those procedures. A richer set of functions may be implemented as incremental levels of support beyond the defined base level.

This section discusses the potential areas of incompatibility and selects specific solutions that are used to build the specification of the subset protocol in Section 6. Alternatives are selected based on the need for file transfer between heterogeneous systems.

5.1 Areas of incompatibility

The descriptions of the various Filing Protocol implementations in Section 4 indicate many areas of incompatibility between the implementations. The causes of the incompatability witnessed in these implementations can be divided into two categories:

- non-support of Filing constructs by the native operating system
- alternate specifications in the Filing Protocol

5.1.1 Non-support of Filing constructs by the native operating system

The XNS protocols were defined for use in a distributed system consisting of homogenous processors and operating systems. As heterogeneous processors and operating systems were introduced into this environment, the need to implement the protocol on top of existing file systems became critical because a great deal of host software was already based on the underlying file system.

Successful implementation of the Filing Protocol on an existing file system must integrate Filing constructs with the native operating/file system, thus enabling the protocol to act as a natural extension of the native system. In many cases, the mapping of defined Filing constructs to existing file systems is not easily accomplished. This is due to the inability of the native operating or file system to provide Filing functionality in the following areas:

- procedure support
- attribute retention and retrieval
- use of controls to insure file integrity
- syntax of file names
- deletion of directories and all descendants

5.1.2 Alternate specifications in the Filing Protocol

The Filing Protocol is the detailed definition of a complete file system. In many cases, the richness of the protocol allows alternative methods to achieve the same results. For implementations not supporting the full protocol, these alternatives lead to incompatible implementations.

The following examples point out several alternatives for specific actions which exist in the Filing Protocol:

- specification of Open attributes to identify a file

- opening a file by use of the pathname attribute on Open or successive opening of nested directories specifying the name attribute
- use of the isDirectory and type (tDirectory) attributes to designate a directory on a Store

5.2 Resolving Incompatibilities

Resolution of the above-stated incompatabilities is required to support interoperability between different implementations. This section proposes particular solutions to these problems. These choices are used as the basis for the subset protocol definition presented in Section 6.

The alternatives chosen are based upon the requirements of the protocol subset. The protocol is intended to provide:

- common file system functions of retrieval, storage, enumeration and deletion
- round-trip equality of file content
- compatibility with common host processing activities (i.e., text editing, backup/restore, etc.)
- preservation of attributes essential to the above functions

5.2.1 Non-support of Filing constructs by the native operating system constructs

The usefulness of a Filing Protocol implementation on a specific operating system requires successful integration of Filing constructs with the existing file system. Much of this integration depends upon the features provided by the host operating/file system. Each of the areas presented in Section 5.1.1 are examined and a specific alternative is presented, under the constraint of keeping the subset as small as possible.

Procedure support

The initial step in supporting a subset of the full protocol on an existing file system is to decide which procedures must be implemented. The functions of retrieval, storage, enumeration and deletion can be easily accomplished with the following set of procedures: Logon, Logoff, Open, Close, Delete, Store, Retrieve, List and Continue. The remaining Filing procedures introduce added functionality and complexity that is not essential to the common file exchange functions desired or simply duplicate actions that can be performed by one of the above procedures.

The GetAttributes and List procedures allow retrieval of the attributes associated with a given file or files, respectively. List allows the client to specify the selection criteria for the files to be listed, while the GetAttributes procedure is constrained to a single file which has previously been opened. The desire to support specification of wildcard file formats led to the choice of the List procedure in the set of mandatory procedures. The GetAttributes procedure is not considered essential since the specification of a single file on a List can be used to obtain the same results.

The Create and Store procedures allow the creation of empty and non-empty files, respectively. However, the Store procedure can also be used to create empty files by sending a bulk data stream containing no data or by specifying BulkData nullSource as the source stream. Since use of the Store is the sole procedure for conveying file content to a service, use of the Create is not essential for the creation of empty files. A given service must allow the use of a null bulk data stream on the Store procedure to effect creation of an empty file.

This set of implemented procedures provide the context for discussion of other incompatibilites. Problems evidenced on procedures which are not included in this list are irrelevant to the remaining discussion.

Attribute retention and retrieval

The support for retention and retrieval of Filing attributes is the largest area of discrepancy between the implementations discussed previously. Each implementation uses its own subset of attributes for the desired functions and assumes compatability only with similar implementations.

Comparing the facilities provided by different operating and file systems with the requirements of the subset protocol indicate a set of attributes which are commonly implementable and useful across all systems. These core Filing attributes convey the information which is most useful to other host utilities. The Filing attributes createdOn, dataSize, isDirectory, modifiedOn, pathname and type are designated as the set of attributes which must be supported by all implementations. This level of support implies that a service implementation must retain the value for each of these attributes when presented on a Store procedure and must also return the appropriate value when requested on a List Clients are also responsible for retaining these attributes when retrieving a file from a service.

The majority of the remaining Filing-defined attributes do not easily map to existing file system constructs on many popular file systems. In some cases (i.e , createdBy, fileID), legal Filing values cannot be maintained by the file system; in others (i.e., accessList, ordering, version), the functionality implied by the attribute is non-existant.

The definition of these mandatory attributes, in combination with the set of required procedures, provides the backdrop for the discussion of more specific problems as they relate to these procedures and attributes.

Use of controls to insure file integrity

Controls are used by the Filing Protocol to insure file integrity by controlling multi-client access to files. When dealing with existing file systems, this functionality may already exist, although in a potentially different form.

For those procedures designated as mandatory, it is reasonable to assume that the default access mechanisms provided by the resident operating system are sufficient. The ability to implement Filing-defined controls may require changes to the resident file system and therefore contradict a major goal of defining the subset protocol Therefore, no specific support for controls is required or assumed by a given service implementation

Syntax of file names

One of the most obvious inconsistencies across heterogeneous systems is that of file name conventions. The constraints imposed upon file names by various systems can sometimes lead to very complicated conversion algorithms. The definition of the Xerox pathname syntax in the Filing Protocol introduces yet another convention for implementations to consider.

When coupling the intended use of the implementations with the desire for ease of implementation, the obvious strategy is to allow specification of pathname values in the syntax of the intended service file system. This strategy implies that the filename syntax on two systems may be radically different, but in most cases, the user knows the conventions of the intended system and can specify an appropriate value. The specification of service-specific wildcard file names on a List is also allowed by this scheme. The need to define and implement any mapping algorithms is eliminated; however, individual implementations may still support alternate syntaxes (such as the Filing pathname syntax) as desired.

Deletion of directories and all descendants

Deletion of directories as specified in the Filing Protocol implies deletion of all descendant files for that directory. Support for this feature may differ from system to system depending upon the structure of the resident file system For this reason, deletion of directories is not required and a given service is allowed to return an appropriate error.

5.2.2 Alternate specifications in the Filing Protocol

The Filing Protocol allows the use of alternate mechanisms for accomplishing specific functions. The existence of these alternatives implies that more than one method can be used to effect a given result. This in turn fosters incompatibility when an implementation chooses to support one alternative without providing support for the others.

The resolution of this class of problem involves mandating support for one of the alternatives. All implementations are required to support the single alternative defined by the subset protocol. Support for other alternatives may exist at the discretion of individual implementations This provides a common ground for all implementations to perform the desired actions and thus maximizes interoperability with other implementations.

Specification of Open attributes to identify a file

The Filing Protocol allows a client to specify a given file on an Open procedure through the combination of several attribute types: fileID, name, parentID, pathname and version. Since the fileID, name, parentID and version types are not included in the set of mandatory attributes to be supported by each implementation, use of the pathname attribute becomes the required method of identifying files on an Open.

Use of the pathname attribute on an Open or successive opening of nested directories

For historical reasons, several implementations of the Filing Protocol have used the name attribute with an accompanying directory handle to identify a file on an Open. The name attribute specifies the name of the file as it exists relative to the directory associated with the supplied directory handle. Since the value for the name attribute is relative to the immediate parent, the client must successively issue an Open (and Close) for each directory identified as an ancestor of the desired file For instance, to open the file A/B/C/D (assuming the Xerox specified pathname syntax), the client would open A, open B, (close A), open C, (close B) and open D. Use of the pathname attribute allows the client to simply specify the absolute form, A/B/C/D, on the Open.

In Section 5.2.1.5 the syntax of the pathname attribute value was defined to be service-specific. This implies that a given client may not know a given service syntax and thus be unable to parse the pathname value and call the successive Opens Since pathname was previously defined to be a mandatory attribute and the form of the pathname value is specified relative to the root directory, use of the successive Opens is not encouraged A service implementation may require use of the pathname attribute on the Open with an accompanying directory handle value of nullHandle It is recognized that in certain cases, the inability of the client to know the appropriate service syntax may in fact cause the service to open a file different from that intended by the client. However, for the implementations examined by this thesis, the number of these cases is relatively small

Use of the isDirectory and type (tDirectory) attributes to designate a directory file

The Filing Protocol includes two mechanisms for specifying directory files. Although there appears to be a redundancy in this definition, actual implementations use only the isDirectory attribute to designate a directory file. A type value of tDirectory when not used in conjunction with an isDirectory value of True, will create a non-directory file of type tDirectory.

To prevent confusion on the part of clients, this ambiguity is resolved by dictating that the isDirectory and type (tDirectory) attributes are essentially the same Specification of one without the other will have the same effect; conflicting values when both attributes are specified will result in an error. This allows directory files to be created on a service by specifying either an isDirectory value of True, a type value of tDirectory, or both.

6. FilingSubset

The previous sections have provided background related to the the development of a subset of the Filing Protocol primarily for use as a file transfer protocol between heterogeneous systems. This section presents the formal specification of the FilingSubset, a subset protocol of the Filing Protocol that makes use of the choices identified in Section 5. The specification which follows was edited and subsequently adopted by Xerox as the definition of the FilingSubset Protocol included in version 6 of the Filing Protocol (Appendix B).

6.1 Overview

6.1.1 Motivation

The Filing Protocol represents the specification of a general purpose filing system which defines the interaction that takes place between client and server processes within the Xerox Network System. This protocol acts as the definition of the file system as well as a guide on how to use that system.

As the formal specification for a file system, the Filing Protocol provides mechanisms for file access, transfer and management. As the network model matured, the addition of new network processors and application processes introduced requirements for varying levels of filing service. The motivation to provide a consistent base level of service across all processors increased with the inability of new implementations to provide the full level of service stated in the Filing Protocol. The motivations for defining a consistent subset for implementation across a wide variety of processors stems from several areas including:

- the ability to allow file exchange without the support necessary for a full level of service
- the ability to provide XNS Filing access to the native file system resident on a 'foreign' host
- the ability to allow XNS Filing access to the files on a system whose primary purpose is not to provide a file service

- the inability of the native operating system on several 'foreign' hosts to support Filing features and constructs
- the expense of software development and resource utilization necessary to support the full Filing Protocol

6.1.2 Requirements and Goals

The definition of a subset of the Filing Protocol is guided by a set of requirements and goals. In general, the requirements are to provide a useful and compatible level of service within the context of the Filing Protocol. The requirements set forth for the definition of the FilingSubset are:

- provide the common file system functions of retrieval, storage, enumeration and deletion.
- facilitate compatibility by remaining a proper subset of the Filing Protocol.
- build upon the Xerox Network Systems Authentication model in addition to the native operating system model for authentication.
- retain round-trip equality of file content.

A set of goals is also defined which, although not required nor guaranteed, are important to the overall usefulness for elements implementing the subset. The following goals are desirable in the definition of the FilingSubset:

- round-trip preservation of attributes (the ability to store a file on a remote system and retrieve it at a later date with all attributes intact).
- the ability to perform common processing activities on a file regardless of on which system it currently resides (for example, text editing, data base listing and backup/restore).
- ease of implementation of both client and server code on a wide variety of systems.

6.2 Definition

The FilingSubset defines a guaranteed minimal level of service which is supported by both clients and servers implementing the Subset. This service is defined as a set of restrictions on the Filing Protocol definition. The mandatory restrictions define specific implementation alternatives specific to the FilingSubset.

The FilingSubset is a proper subset of the Filing Protocol. This guarantees that both actions and responses defined in the subset are identical to those in the Filing Protocol. The subset also guarantees that clients that implement the subset can interact with a service implementing the full Filing Protocol In addition, a client using the full protocol can interact with a subset service by restricting its use of procedures and arguments to those included in the subset definition.

In all cases, the procedures, arguments and errors defined in the subset are identical to those in the Filing Protocol. In providing a lower level of service, the subset does, however, restrict the choices available for argument and error values. The subset also allows recognizes that it may not be possible for a given service to support the semantics of certain operations. The subset dictates appropriate errors to be returned under those conditions.

Maximum interoperability is ensured when both client and server implementations support this minimum level and make no assumptions regarding the availability of a broader functionality. However, increased levels of functionality, up to full Filing, may be supported by individual implementations with the restriction that appropriate actions be taken in the event that the additional functionality is not supported by other implementations.

The complete Courier definition of the FilingSubset is presented in Section 6.8.

6.3 Attributes

In defining a minimum level of Filing service, the FilingSubset differentiates three levels of support for file attributes. These levels are *mandatory*, *implied* and *optional*.

6.3.1 Mandatory Attributes

Mandatory attributes represent the specific set of attributes whch must be interpreted by all Subset implementations. These attributes are guaranteed to be retained by any service implementing the FilingSubset and must also be accepted on specific procedure calls to the extent that they are legal arguments on that same procedure within the Filing Protocol.

The attributes createdOn, dataSize, isDirectory, modifiedOn, pathname, and type are defined by the Subset as mandatory and must be supported by all Subset implementations. In addition, an implementation must support the type values tAsciiText, tDirectory and tUnspecified. This support implies that a service implementation must accept these attributes on a Store procedure, if they are legal arguments, and must also return the appropriate non-null value when requested with a List procedure.

6.3.1.1 createdOn

The createdOn attribute is identical to that in Filing.

6.3.1.2 dataSize

The Filing Protocol states that the dataSize attribute contains the number of eight-bit bytes in the content of the file. The FilingSubset recognizes that it not always straightforward for specific implementations to determine the actual content size, so all Subset implementations should regard the value of the dataSize attribute as an estimate of the file size rather than the amount of valid data in the file.

6.6.1.3 isDirectory

The isDirectory attribute is identical to that in Filing

6.3.1.4 modifiedOn

The modifiedOn attribute is identical to that in Filing.

6.3.1.5 pathname

The FilingSubset requires all implementations to permit the use of the pathname attribute to identify a file. The value of the pathname attribute will always specify a remote file's access path in a form which is recognized by the service. A Subset client should make no assumptions as to the syntax of this attribute since it will vary from service to service.

Two types of pathname values are supported by Filing: 1) absolute, which is specified from the root file and 2) relative, which is specified relative to an accompanying directory Handle or parentID attribute. All FilingSubset service implementations are required to allow specification of the absolute syntax for the pathname attribute on an Open, List or Store procedure. However, when the absolute form is specified on one of these procedures, a FilingSubset service is permitted to reject the procedure if the directory Handle specified on the call is not the nullHandle. Likewise, the procedure may be rejected if the parentID attribute is not the nullfileID if it is specified on an Open or Store procedure in conjunction with the absolute pathname form.

FilingSubset services are not required to support a relative pathname syntax. A service may reject a procedure call with an AttributeValueError if the pathname value specified is in the relative form. Clients should recognize that support for this syntax may not be provided by a given service and there is no explicit mechanism to determine if such support does exist. Use of the relative pathname syntax by a client may result in either an undefined behavior or rejection from the service.

The value of the pathname attribute returned from a List procedure should always be an absolute pathname so that it can be used directly for subsequent operations.

6.3.1.6 type

The tAsciiText, tDirectory and tUnspecified values must be supported by all FilingSubset implementations.

FilingSubset services should also permit the specification of these types on the Open call. This usage of the type attribute indicates the client's intention to receive the file as the specified type regardless of the file type as stored on the service.

6.3.2 Implied Attributes

Those non-mandatory attributes which receive an default value when a file is created are designated *implied attributes*. In maintaining consistency with the Filing Protocol, the Subset requires that implementations must allow specification of these *implied* values for the accessList, childrenUniquelyNamed, defaultAccessList, isTemporary, ordering, subtreeSizeLimit and version attributes even though the attribute may not be fully supported by the implementation. A Store procedure must allow specification of these attributes insofar as the accompanying attribute value is in fact the service-specific supported value (see Section 6.7). Specification of an unsupported value, under these circumstances, must be rejected with an AttributeValueError if the attribute is not fully supported.

6.3.3 Optional Attributes

Optional attributes comprise the remaining attributes. Support for these attributes is optional. If an implementation provides support for any of these additional attributes, that support must be within the definition of the Filing Protocol.

6.4 Remote Procedure Support

The FilingSubset supports only those procedures which provide the essential functions required for file retrieval, storage, listing/enumeration and deletion. These procedures are Logon, Logoff, Continue, Open, Close, Retrieve, Store, List and Delete.

The FilingSubset also requires all implementations to permit file identification through the use of the pathname attribute.

This section describes the expected level of support by all Subset implementations for the remote procedures defined.

6.4.1 Session Support

The Logon, Logoff and Continue procedures are defined to be identical to the Filing Protocol.

6.4.2 Opening and Closing Files

The Subset Open procedure must permit use of the pathname attribute for file identification. Specification of the parentID, type and version attributes must be allowed in conjunction with the pathname attribute although the required set of allowable values for each of these attributes is limited (the values nullFileID for parentID, tAsciiText, tDirectory, tUnspecified for type and highestVersion andlowestVersion for version). A server implementation must not return an AttributeTypeError if the parentID, type or version attributes are specified on an Open; instead, an AttributeValueError may be returned if the value of the attribute is not one of the above.

The Open procedure may be rejected if controls is not the empty sequence or directory is not the nullHandle.

The Filing Protocol specifies that while a client has a file open, the file may not be deleted by other network clients. The Subset does not presume that a service implementation can prevent a previously opened file from being deleted by other users. regardless of whether they are general interactive users or other network clients. Subset clients should be prepared to deal with directories or files which cannot be accessed even after a valid handle is obtained (the error HandleError[problem: invalid] is returned)

The Close procedure is defined to be identical to the Filing Protocol.

6.4.3 Enumerating Files in Directories

The FilingSubset defines a minimal file enumeration capability, the List procedure, which is based on use of the pathname attribute.

6.4.3.1 Scopes

The FilingSubset requires a minimum level of support for scopes. Specifically, count and filter are the only scope types which are required to be supported. A Subset service must also permit the matches filter type which specifies the pathname attribute.

6.4.3.2 Attribute Support

The List procedure is required to return values for all attributes requested. Specifically, non-null values must be returned for the mandatory and implied attribute types accessList, childrenUniquelynamed, createdOn, dataSize, defaultAccessList, isDirectory, isTemporary, modifiedOn, ordering, pathname, subtreeSizeLimit, type and version. For the remainder of the Filing attributes, appropriate non-null values must be returned if that attribute is supported by the implementation; the null, or empty, sequence (Attribute: TYPE = RECORD [type: AttributeType, value: SEQUENCE 0 OF UNSPECIFIED]) must be returned for all unsupported attributes.

The BulkData.immediateSink and BulkData.nullSink transfer types must be supported by all implementations for returning the appropriate information from a List request.

6.4.4 Storing Files

The Subset Store procedure requires implementations to permit the use of the pathname attribute for file identification. The type andversion attributes must be allowed in conjunction with the pathname attribute; however, the required set of allowable values for each of these attributes may be quite small: tAsciiText, tDirectory, tUnspecified for type and highestVersion for version.

Treatment of the remaining attributes depends upon the level of support for those attributes within each service implementation. A service cannot reject a Store procedure with an AttributeTypeError if any of the mandatory attributes (except modifiedOn) are specified. An AttributeValueError may be returned if the accompanying value is determined to be invalid.

Likewise, a service can not reject a Store procedure with an AttributeTypeError if an optional attribute is specified. The procedure must not be rejected with an AttributeValueError if the accompanying value is the server-specific supported value as shown in the table in Section 6.7. The procedure must be rejected with an AttributeValueError if the accompanying value is, indeed, not supported by the service implementation.

All other attributes must be rejected with an AttributeTypeError if that type is not supported or an AttributeValueError if the specified value is invalid or unsupported.

The Store procedure may be rejected if controls is not the empty sequence or directory is not the nullHandle.

The BulkData.immediateSource and BulkData.nullSource transfer types must be supported by all implementations for sending the file content during a Store request

The Store procedure allows both directory and non-directory files to be created. Directory files may be created by specifying all of the following: 1) isDirectory attribute value of TRUE, 2) type attribute value of tDirectory and 3) a bulk data stream type of Bulkdata.immediateSource with no file content or the Bulkdata.nullSource bulk data stream type.

A FilingSubset service is permitted to reject the creation of directory files. if that action is not supported by the service. The error AccessError [problem: accessRightsInsufficient] should be reported if the service does not allow the creation of directory files. A service may optionally only allow creation of empty directory files. In this case, the error AttributeValueError [problem: unreasonable, type: isDirectory] should be returned if a client attempts to create a non-empty directory.

6.4.5 Retrieving Files

The FilingSubset Retrieve procedure is identical to the Filing Protocol.

The BulkData.immediateSink and BulkData.nullSink transfer types must be supported by all implementations for returning the file content from a Retrieve request.

6.4.6 Deleting Files

The FilingSubset Delete procedure is defined to be identical to the Filing Protocol for non-directory files.

The Filing Protocol states that if a Delete procedure is specified where the intended file is a directory, all descendants of that directory will also be deleted. The FilingSubset does not presume that all descendants of a directory can in fact be deleted when the directory itself is deleted. Server implementations that cannot guarantee this behavior should return the error AccessError [problem: accessRightsInsufficient]. Client implementations should be prepared to deal with this error condition when observed. Likewise, an error encountered during deletion of the directory tree may result in only partial deletion of the files within that directory tree.

6.5 Remote Procedure Restrictions

The FilingSubset defines a minimum capability that must be supported by all implementations This minimum level of support is imposed by defining those argument values which all Subset implementations must accept. Specific server implementations may accept a larger range of values; however, support for these values should not be assumed on the part of client implementations. To the extent that additional argument values are supported, this support must still be compatible with the full Filing Protocol. Indeed, a valid implementation must return an error if the expected functionality cannot be provided. FilingSubset procedures are allowed to return appropriate errors if the described conditions are observed for the following procedures and that implementation does not support the functionality implied by values other than those specified

This section summarizes the restrictions in a convenient itemized form. The intent is to provide a list of those conditions which may result in an error from a Subset service.

Open--

- directory specifies a handle other than nullHandle
- controls does not specify the empty sequence
- attributes does not contain the pathname attribute type
- attributes contains an attribute type other than parentID, pathname, type or version
- the parentID attribute specifies a handle other than nullFileID
- the type attribute specifies a value other than tAsciiText, tDirectory or tUnspecified
- the version attribute specifies a value other than highestVersion

Store--

- directory specifies a handle other than nullHandle
- controls does not specify the empty sequence
- content specifies a source other than type BulkData.immediateSource or BulkData.nullSource
- attributes does not contain the pathname attribute type
- attributes contains one of the attribute types fileID, modifiedBy, modifiedOn, name, numberOfChildren, parentID, readBy, readOn, storedSize or subtreeSize
- the type attribute specifies a value other than tAsciiText, tDirectory or tUnspecified
- the version attribute specifies a value other than highestVersion

Retrieve--

- content specifies a sink other than type BulkData.immediateSink or BulkData.nullSink

List--

- directory specifies a handle other than nullHandle
- scope includes a scope type other than filter or count
- filterType specifies a filter type other than matches
- a matches filter contains an attribute type other than pathname
- listing specifies a sink other than type BulkData.immediateSink or BulkData.nullSink
- types specifies an attribute type other than mandatory or implied attributes

6.6 Remote Errors

Each of the FilingSubset procedures report remote errors identical to the Filing Protocol. The FilingSubset guarantees the return of specific errors for certain conditions. Additional errors may be returned under the appropriate conditions if a specific service can detect the condition.

6.7 Procedures and attributes

The tables on the following pages describe the effects of FilingSubset procedures with respect to attributes. If a procedure never modifies interpreted attributes, no table is given. If an entry in the table is empty, the corresponding attribute is never changed. Otherwise, a brief indication of the change is given Where specification of an attribute will result in an error condition, the appropriate error is identified

In the case of List, the table specifies the required return values from a service. Although this procedure does not modify attributes, its behavior is defined by the FilingSubset. The Open table specifies the attribute values that must be allowed by a subset service on the Open. The Store table shows both the attribute types and values that must be allowed by a service as well as the values that must be retained by the service regardless of whether a value was specified for the attribute on the procedure.

List

Attribute	If Requested[1]
accessList	returned
checksum	returned
childrenUniquelyNamed	returned
createdBy	returned
createdOn	non-null value must be returned
dataSize	non-null value must be returned[2]
defaultAccessList	returned
fileID	returned
isDirectory	non-null value must be returned
isTemporary	returned
modifiedBy	returned
modifiedOn	non-null value must be returned
name	returned
numberOfChildren	returned
ordering	returned
parentID	returned
pathname	non-null value must be returned
position	returned
readBy	returned
readOn	returned
storedSize	returned
subtreeSize	returned
subtreeSizeLimit	returned
type	non-null value must be returned
uninterpreted	returned
version	returned

1 The FilingSubset states that all attributes which are supported by a server must return an appropriate value. If an attribute is not supported by a server, then Attribute : TYPE = RECORD [type: AttributeType, value: SEQUENCE 0 OF UNSPECIFIED] must be returned.

2 The value returned for the dataSize attribute should be viewed by the client as an approximation of the actual content size.

Open

Attribute	If a Parameter
accessList	illegal, AttributeTypeError
checksum	illegal, AttributeTypeError
childrenUniquelyNamed	illegal, AttributeTypeError
createdBy	illegal, AttributeTypeError
createdOn	illegal, AttributeTypeError
dataSize	illegal, AttributeTypeError
defaultAccessList	illegal, AttributeTypeError
fileID	optional[2]
isDirectory	illegal, AttributeTypeError
isTemporary	illegal, AttributeTypeError
modifiedBy	illegal, AttributeTypeError
modifiedOn	illegal, AttributeTypeError
name	optional[2]
numberOfChildren	illegal, AttributeTypeError
ordering	illegal, AttributeTypeError
parentID	optional [3]
pathname	file with this value is opened
position	illegal, AttributeTypeError
readBy	illegal, AttributeTypeError
readOn	illegal, AttributeTypeError
storedSize	illegal, AttributeTypeError
subtreeSize	illegal, AttributeTypeError
subtreeSizeLimit	illegal, AttributeTypeError
type	file with this value is opened[1]
uninterpreted	ignored
version	optional[3]

[1] The FilingSubset permits the use of type to indicate that the transferred form of the file is of the specified type. This may impose some form of alteration on the actual file content as the file is transferred. If a specified type is not supported by the implementation, the procedure should be rejected with an AttributeValueError.

[2] FilingSubset implementations are not required to support this attribute. If support is not provided, the service should return an AttributeTypeError.

[3] The Filing Protocol allows the combination of parentID, pathname and version. However, a given subset implementation is free to reject the Open with an AttributeValueError if either parentID is not equal to nullFileID or version is not highestVersion and the implementation does not support either attribute.

Store

Attribute	If a Parameter[1]	Supported Values	If not a Parameter[5]
accessList	set if value supported	[defaulted: TRUE]	set to [defaulted: TRUE]
checksum	set if type supported	unknownChecksum	set appropriately
childrenUniquelyNamed	set if value supported	TRUE/FALSE[4]	TRUE/FALSE[4]
createdBy	set if type supported	--	currently logged-in user
createdOn	set	--	current date and time
dataSize	initial allocation (hint)	--	number of bytes transferred
defaultAccessList	set if value supported	[defaulted: TRUE]	set to [defaulted: TRUE]
fileID	illegal, AttributeTypeError	--	system-assigned value
isDirectory	set	--	FALSE
isTemporary	set if value supported	FALSE	FALSE
modifiedBy	illegal, AttributeTypeError	--	currently logged-in user
modifiedOn	illegal, AttributeTypeError	--	current date and time
name	illegal, AttributeTypeError[2]	--	implementation dependant
numberOfChildren	illegal, AttributeTypeError	--	0
ordering	set if value supported	defaultOrdering	defaultOrdering
parentID	illegal, AttributeTypeError	--	fileID of resulting parent
pathname	set	--	consistent with ancestry
position	set if type supported	--	depends on parent's ordering
readBy	illegal, AttributeTypeError	--	""
readOn	illegal, AttributeTypeError	--	nullTime
storedSize	illegal, AttributeTypeError	--	set appropriately
subtreeSize	illegal, AttributeTypeError	--	set appropriately
subtreeSizeLimit	set if value supported	nullSubtreeSizeLimit	nullSubtreeSizeLimit
type	set if value supported[3]	tAsciiText, tDirectory, tUnspecified	tDirectory or tUnspecified
uninterpreted	set if type supported	--	null
version	set if value supported	highestVersion	next available

[1] The FilingSubset must treat attributes in one of four ways (1) An attribute marked *"illegal"* will be rejected with AttributeTypeError (2) An attribute marked *"set"* must not be rejected with AttributeTypeError and must normally accept non-null values In unusual cases it may reject a non-null value, such as a string which is too long (3) An attribute marked *"set if value supported"* must not reject with AttributeTypeError. It must not result in an AttributeValueError if the value is one of the supported values as listed above An AttributeValueError may be reported for other values which cannot be supported. (4) An attribute marked *"set if type supported"* must be rejected with AttributeTypeError or AttributeValueError if the implementation does not fully support the type or value respectively.

[2] The FilingSubset does not require support for this attribute If support is not provided, an AttributeTypeError should be reported

[3] The types tAsciiText, tDirectory and tUnspecified must be supported by all implementations

[4] The supported value for childrenUniquelyNamed is implementation specific depending upon whether version is supported If multiple versions are supported, childrenUniquelyNamed is TRUE

[5] These values are the default values if the attribute type is supported by the implementation

6.8 Courier Definition

The complete Courier definition of the FilingSubset follows. All Courier types are identical to those specified in the Filing Protocol.

```
FilingSubset: PROGRAM 1500 VERSION 1 =
BEGIN
	DEPENDS UPON
	  BulkData (0) VERSION 1,
	  Clearinghouse (2) VERSION 3,
	  Filing (10) VERSION 6,
	  Authentication (14) VERSION 3;

-- TYPES AND CONSTANTS --

-- Attributes--

AttributeSequence: TYPE = Filing.AttributeSequence;
AttributeTypeSequence: TYPE = Filing.AttributeTypeSequence;
allAttributeTypes: Handle = Filing.allAttributeTypes;

-- Controls --

ControlSequence: TYPE = Filing.ControlSequence;
ControlTypeSequence: TYPE = Filing.ControlTypeSequence;

-- Handles and Authentication --

Credentials: TYPE = Filing.Credentials;

SecondaryType: TYPE = Filing.SecondaryType;

Handle: TYPE = Filing.Handle;

nullHandle: TYPE = Filing.nullHandle;

Session: TYPE = Filing.Session;

Verifier: TYPE = Authentication.Verifier;

-- Scopes --

ScopeSequence: TYPE = Filing.ScopeSequence;

-- REMOTE PROCEDURES --

-- Logging On and Logging Off --
```

```
Logon: PROCEDURE [
        service: Clearinghouse.Name, credentials: Credentials, verifier: Verifier]
RETURNS [session: Session]
REPORTS [AuthenticationError, ServiceError, SessionError, UndefinedError] =
        Filing.Logon;

Logoff: PROCEDURE [session: Session]
REPORTS [AuthenticationError, ServiceError, SessionEror, UndefinedError] =
        Filing.Logoff;

Continue: PROCEDURE [session: Session]
REPORTS [AuthenticationError, SessionError, UndefinedError] =
        Filing.Continue;

-- Opening and Closing Files --

Open:PROCEDURE [attributes: AttributeSequence, directory: Handle,
        controls: ControlSequence, session: Session]
RETURNS [file: Handle]
REPORTS [AccessError, AttributeTypeError, AttributeValueError, AuthenticationError,
        ControlTypeError, ControlValueError, HandleError, SessionError, UndefinedError] =
        Filing.Open;

Close: PROCEDURE [file: Handle, session: Session]
REPORTS [AuthenticationError, HandleError, SessionError, UndefinedError] =
        Filing.Close;

-- Deleting Files --

Delete: PROCEDURE [file: Handle, session: Session]
REPORTS [AccessError, AuthenticationError, HandleError, SessionError, UndefinedError] =
        Filing.Delete;

-- Transferring Bulk Data (File Content) --

Store: PROCEDURE [directory: Handle, attributes: AttributeSequence,
        controls: ControlSequence, content: BulkData.Sink, session: Session]
RETURNS: [file, Handle]
REPORTS [AccessError, AttributeTypeError, AttributeValueError, AuthenticationError,
        ConnectionError, ControlTypeError, ControlValueError, HandleError, InsertionError,
        SessionError, SpaceError, TransferError, UndefinedError] = Filing.Store;

Retrieve: PROCEDURE [file: Handle, content: BulkData.Sink, session: Session]
REPORTS [AccessError, AuthenticationError, ConnectionError, HandleError, SessionError,
        TransferError, UndefinedError] = Filing.Retrieve;

-- Listing Files in a Directory --

List: PROCEDURE [directory: Handle, types: AttributeTypeSequence,
        scope: ScopeSequence, listing: BulkData.Sink, session: Session]
REPORTS [AccessError, AttributeTypeError, AuthenticationError, ConenctionError,
        HandleError, ScopeTypeError, ScopeValueError, SessionError, TransferError,
        UndefinedError] = Filing.List;
```

```
-- REMOTE ERRORS --

-- problem with an attribute type or value --

AttributeTypeError: ERROR [problem: ArgumentProblem, type: AttributeType] =
        Filing.AttributeTypeError;
AttributeValueError: ERROR [problem: ArgumentProblem, type: AttributeType] =
        Filing.AttributeValueError;

-- problem with an control type or value --

ControlTypeError: ERROR [problem: ArgumentProblem, type: ControlType] =
        Filing.ControlTypeError;
ControlValueError: ERROR [problem: ArgumentProblem, type: ControlType] =
        Filing.ControlValueError;

-- problem with an scope type or value --

ScopeTypeError: ERROR [problem: ArgumentProblem, type: ScopeType] =
        Filing.ScopeTypeError;
ScopeValueError: ERROR [problem: ArgumentProblem, type: ScopeType] =
        Filing.ScopeValueError;

ArgumentProblem: TYPE = Filing.ArgumentProblem;

-- problem in obtaining access to a file --

AccessError: ERROR [problem: AccessProblem] = Filing.AccessError;
AccessProblem: TYPE = Filing.AccessProblem;

-- problem with a credentials or verifier --

AuthenticationError: ERROR [problem: AuthenticationProblem] =
        Filing.AuthenticationError;

-- problem with a bulk data transfer --

ConnectionError: ERROR [problem: ConnectionProblem] = Filing.ConnectionError;
ConnectionProblem: TYPE = Filing.ConnectionProblem;

-- problem with a file handle --

HandleError: ERROR [problem: HandleProblem] = Filing.HandleError;
HandleProblem: TYPE = Filing.HandleProblem;

-- problem during insertion in directory (or changing attributes) --

InsertionError: ERROR [problem: InsertionProblem] = Filing.InsertionError;
InsertionProblem: TYPE = Filing.InsertionProblem;

-- problem during random access operation --

RangeError: ERROR [problem: ArgumentProblem] = Filing.RangeError;
```

```
-- problem during logon or logoff --

ServiceError: ERROR [problem: ServiceProblem] = Filing.ServiceError;
ServiceProblem: TYPE = Filing.ServiceProblem;

-- problem with a session --

SessionError: ERROR [problem: SessionProblem] = Filing.SessionError;
SessionProblem: TYPE = Filing.SessionProblem;


-- problem obtaining space for file content or attributes --

SpaceError: ERROR [problem: SpaceProblem] = Filing.SpaceError;
SpaceProblem: TYPE = Filing.SpaceProblem;

-- problem during bulk data transfer --


TransferError: ERROR [problem: TransferProblem] = Filing.TransferError;
TransferProblem: TYPE = Filing.TrasnferProblem;

-- some undefined (and implementation-dependent) problem occurred --


UndefinedError: ERROR [problem: UndefinedProblem] = Filing.UndefinedError;
UndefinedProblem: TYPE = Filing.UndefinedProblem;

-- INTERPRETED ATTRIBUTE DEFINITIONS--

accessList: AttributeType = Filing.accessList;
AccessList: TYPE = Filing.AccessList;

checksum: AttributeType = Filing.checksum;
Checksum: TYPE = Filing.Checksum;

childrenUniquelyNamed: AttributeType = Filing.childrenUniquelyNamed;
ChildrenUniquelyNamed: TYPE = Filing.ChildrenUniquelyNamed;

createdBy: AttributeType = Filing.createdBy;
CreatedBy: TYPE = Filing.CreatedBy;

createdOn: AttributeType = Filing.createdOn;
CreatedOn: TYPE = Filing.CreatedOn;

dataSize: AttributeType = Filing.dataSize;
DataSize: TYPE = Filing.DataSize;

defaultAccessList: AttributeType = Filing.defaultAccessList;
DefaultAccessList: TYPE = Filing.DefaultAccessList;

FileID: AttributeType = Filing.FileID;
FileID: TYPE = Filng.FileID;
```

```
isDirectory: AttributeType = Filing.isDirectory;
IsDirectory: TYPE = Filing.IsDirectory;

isTemporary: AttributeType = Filing.IsTemporary;
isTemporary: TYPE = filing.IsTemporary;

modifiedBy: AttributeType = Filing.modifiedBy;
ModifiedBy: TYPE = Filing.ModifiedBy;

modifiedOn: AttributeType = Filing.modifiedOn;
ModifiedOn: TYPE = Filing.ModifiedOn;

name: AttributeType = Filing.name;
Name: TYPE = Filing.Name;

numberOfChildren: AttributeType = Filing.numberOfChildren;
NumberOFChildren: TYPE = Filing.NumberOfChildren;

ordering: AttributeType = Filing.ordering;
Ordering: TYPE = Filing.Ordering;

pathname: AttributeType = Filing.pathname;
Pathname: TYPE = Filing.Pathname;

parentID: AttributeType = Filing.parentID;
ParentID: TYPE = Filing.ParentID;

position: AttributeType = Filing.position;
Position: TYPE = Filing.Position;

readBy: AttributeType = Filing.readBy;
ReadBy: TYPE = Filing.readBy;

readOn: AttributeType = Filing.readOn;
ReadOn: TYPE = Filing.ReadOn;

storedSize: AttributeType = Filing.storedSize;
StoredSize: TYPE = Filing.StoredSize;

subtreeSize: AttributeType = Filing.subtreeSize;
SubtreeSize: TYPE = Filing.SubtreeSize;

subtreeSizeLimit: AttributeType = Filing.subtreeSizeLimit;
SubtreeSizeLimit: TYPE = Filing.SubtreeSizeLimit;

type: AttributeType = Filing.type;
Type: TYPE = Filing.Type;

version: AttributeType = Filing.version;
Version: TYPE = Filing.Version;

-- BULK DATA FORMATS --
-- Attribute series Format, used in List --

StreamofAttributeSequence: TYPE = Filing.StreamOfAttributeSequence;

-- Line-oriented ASCII text file format, used in file interchange --

StreamOfAsciiText: TYPE Filing.StreamOfAsciiText;

END;
```

References

[1] Xerox Corporation. *Authentication Protocol*. Xerox System Integration Standard, Stamford, Connecticut, April 1984, XNSS 098404 (XSIS 098404).

[2] Xerox Corporation, *Bulk Data Transfer*, Xerox System Integration Standard, Stamford, Connecticut, April 1984, XNSS 038112 (XSIS 038112), Addendum 1a.

[3] Xerox Corporation *Character Code Standard*. Xerox System Integration Standard, Stamford, Connecticut, April 1984, XNSS 058404 (XSIS 058404).

[4] Xerox Corporation. *Clearinghouse Protocol*. Xerox System Integration Standard, Stamford, Connecticut, April 1984, XNSS 078404 (XSIS 078404).

[5] Xerox Corporation. *Clearinghouse Entry Formats* Xerox Network Systems Standard, Stamford, Connecticut, April 1984, XNSS 168404 (XSIS 168404).

[6] Xerox Corporation. *Courier: The Remote Procedure Call Protocol*. Xerox Network Systems Standard, Stamford, Connecticut, December 1981, XNSS 038112 (XSIS 038112).

[7] Digital Equipment Corporation, Intel Corporation, and Xerox Corporation *The Ethernet. A Local Area Network: Data Link Layer and Physical Layer Specifications Version 2.0*. September 1980.

[8] Xerox Corporation *Filing Protocol* Xerox Network Systems Standard, Stamford, Connecticut, May 1986, XNSS 108605 (XSIS 108605)

[9] Xerox Corporation. *Internet Transport Protocols*. Xerox Network Systems Standard, Stamford, Connecticut, December 1981, XNSS 028112 (XSIS 028112)

[10] Xerox Corporation. *Interpress Electronic Printing Standard, Version 3.0*. Xerox Network Systems Standard, Stamford, Connecticut, January 1986, XNSS 1048601 (XSIS 048601).

[11] International Organization for Standardization. *ISO Open Systems Interconnection - Basic Reference Model.* ISO/TC 97/SC 16 N 719, August 1981

[12] Xerox Corporation. *Printing Protocol* Xerox Network Systems Standard, Stamford, Connecticut, April 1984, XNSS 118404 (XSIS 118404).

[13] Xerox Corporation. Secondary Credentials Formats. Xerox Network Systems Standard, Stamford, Connecticut, May 1986, XNSS 258605 (XSIS 258605).

[14] Xerox Corporation. *Time Protocol.* Xerox Network Systems Standard, Stamford, Connecticut, October 1982, XNSS 088210 (XSIS 088210).

A. FilingSubset Implementor's Guide

During the course of development of the FilingSubset Protocol, it was recognized that there was a need for an implementation guide to accompany the protocol specification. This guide would provide a concise scheme for the implementation of the protocol to maximize interoperability. The FilingSubset Implementor's Guide was written as a response to this need. This document describes the implementation of the FilingSubset Protocol from both a client and service perspective. This description is presented at two levels: 1) independent of any underlying file system and 2) specific support for implementation in the UNIX and VMS operating systems.

A copy of the FilingSubset Implementor's Guide is included in the same form in which it is being distributed by Xerox.

FILINGSUBSET IMPLEMENTOR'S GUIDE

XEROX

Ed Flint

July 1986

Notice

All text and graphics prepared on the Xerox 8010 Information System.

TABLE OF CONTENTS

1. Introduction 1

1.1 Purpose 1

1.2 Document organization 1

1.3 Document conventions 2

1.3.1 Notation 2

1.3.2 Notation for Courier examples 2

1.3.3 Notation for C examples 3

2. XNS protocol dependencies 5

2.1 Internet Transport Protocols 5

2.1.1 Relationship of transport to FilingSubset session 5

2.2 Courier and Bulk Data Protocols 6

2.3 Clearinghouse Protocol 6

2.3.1 Implementation with a Clearinghouse service 6

2.3 2 Implementation without a Clearinghouse service 7

2.4 Authentication Protocol 8

2.4.1 Implementation with an Authentication Service 8

2.4.2 Implementation without an Authentication Service 9

2.5 Time Protocol 10

3. Client implementation 11

3 1 Opening a session 11

3 3.1 Maintaining an open session 12

3 2 Closing a session 13

3.3 Enumerating a file(s) 13

3.4 Storing a file(s) 14

3 4.1 Overwriting an existing remote file 14

3 4 1 Bulk data transfer 15

3.5 Retrieving a file(s) 15

3.5.1 Overwriting an existing local file 16

3 5.2 Bulk data transfer 16

3.6 Deleting a file(s) 16

3.7 Creating a directory 17

4. Service implementation 19

4.1 Common data structures 19

4.1.1 Session handle 20

4.1.2 File handle 20

4.2 Common support 21

4.2.1 Session validation 21

4.2.2 Use of the continuance timer 21

4.2.3 File handle validation 22

4.3 Procedures 22

4.3.1 Logon 22

4.3.2 Logoff 24

4.3.3 Continue 24

4.3.4 Open 24

4.3.5 Close 26

4.3.6 List 26

4.3.7 Store 27

4.3.8 Retrieve 30

4.3.9 Delete 31

5. UNIX system interface 33

5.1 Attribute support 33

5.1.1 Mandatory attributes 34

5.1.2 Implied attributes 39

5.1.3 Optional attributes 39

5.2 Client procedure support 40

5.2.1 Continuance timer support 40

5.2.2 Determining mandatory attribute values 41

5.3 Service procedure support 42

5.3.1 Logon 43

5.3.2 Continue 45

5.3.3 Open 45

5.3.4 List 47

5.3.5 Store 48

5.3.6 Retrieve 51

5.3.7 Delete 52

6. VMS system interface 55

6.1 Attribute support 55

6.1.1 Mandatory attributes 56

6.1.2 Implied attributes 64

6.1.3 Optional attributes 64

6.2 Client procedure support 65

6.2.1 Continuance timer support 65

6.2.2 Determining mandatory attribute values 66

6.3 Service procedure support 68

6.3.1 Logon 69

6.3.2 Continue 69

6.3.3 Open 70

6.3.4 List 72

6.3.5 Store 76

6.3.6 Retrieve 79

6.3.7 Delete 79

Appendices:

A. References 83

LIST OF TABLES

Tables:

4.1	Primary and secondary credentials combinations	23
5.1	UNIX supported values for implied attributes	39
6.1	VMS supported values for implied attributes	65

1. INTRODUCTION

The FilingSubset Protocol defines a minimal capability to store, retrieve, enumerate and delete files of a remote service. Maximum interconnectivity is ensured when both client and service implementations support this specified minimum level of service and make no assumptions regarding the availability of a broader functionality

The FilingSubset Protocol has been designed to provide XNS access to native file systems on heterogeneous hosts in a straightforward and easily implementable fashion The FilingSubset specification defines the behaviour between clients and services without respect to a specific implementation.

1.1 Purpose

The FilingSubset Implementor's Guide describes a framework for implementation of the FilingSubset Protocol which can serve as a handbook for future implementors. This document presents:

- a client mapping of common user functions to FilingSubset procedures
- service support for the FilingSubset
- recommended use of and support for FilingSubset procedures and attributes on the UNIX™ 4.2BSD, UNIX™ 4.3BSD, UNIX™ System V and VAX/VMS operating systems

Through the use of specific implementation examples, the reader will be shown a common method for implementing the protocol within the above operating systems and thereby further interconnectivity and reduce the potential for implementation inconsistencies

The examples presented in this guide describe a consistent implementation of the required functionality of the FilingSubset Protocol. These examples are by no means the only method for providing the facility desired; they have been chosen because they offer a simple and clearly understood framework for an actual implementation, regardless of the interface to the lower level XNS protocols.

1.2 Document organization

Chapter 2 of this document describes the relationship between the FilingSubset Protocol and other XNS protocols in terms of the support required for FilingSubset implementations Chapter 3 describes a client implementation for translating common user functions into the appropriate FilingSubset procedures Chapter 4 presents a service implementation at a level independent of the underlying file system interface Each of these chapters deals with the recommended use of the FilingSubset for interaction between client and server. Chapter 5 details the support for FilingSubset procedures and attributes with regard to the UNIX 4.2BSD, UNIX 4.3BSD and UNIX System V operating systems. Support for these same procedures and attributes for the VAX/VMS operating system is described in Section 6.

1.3 Document conventions

Courier text and examples are depicted in special fonts, and generally conform to a certain style. Examples illustrated through the use of C code are also depicted in a special font. The rules and style are set forth below.

1.3.1 Notation

Throughout this document, special fonts are used to depict Courier text/examples and C examples instead of using quote marks or other delimiters. This convention also aids the eye in discriminating between various examples and the exposition.

Items in THIS FONT indicate elements of the Courier language and are almost always in upper case. **This font** indicates items that are defined using the Courier language. Identifiers will have their first letter capitalized if they are the name of a type, error or procedure; identifiers with a lowercase first letter are usually the names of variables, arguments or results.

Items in `this font` indicate C code examples. Identifiers which are entirely uppercase are the names of user defined C constants or macros. Identifiers will have their first letter capitalized if they represent the name of a structure type, constant Courier value or Courier defined procedure. Those identifiers with a lowercase first letter are usually the names of user defined variables, arguments or results.

1.3.2 Notation for Courier examples

In the examples that follow, a call to a remote procedure is denoted by the name of the procedure followed by the arguments supplied to it. A return from a remote procedure is denoted simply by the results, preceded – when confusion might otherwise result – by the keyword RETURNS. The argument or result list is modeled as a record; the arguments or results as the record's components. Accordingly, Courier's standard notation for record constants is used to specify arguments and results lists.

For example, if the procedure **Add** is defined as:

Add: PROCEDURE [first, second: CARDINAL]
RETURNS [sum: CARDINAL] = 99;

then a call to that procedure would be denoted by:

Add [first: 7, second: 5]

and the call would yield the result

[sum: 12] or **RETURNS [sum: 12]**

Fine point: The above notation for procedure calls should not be confused with the standard notation for a record constant selected by means of a choice data type. The two are similar in appearance but otherwise unrelated.

Examples of remote errors are either just the name of the error, if it is defined without arguments:

OVERFLOW

or the same as a procedure call if it is defined with arguments. For example, if **Overflow** were defined as:

Overflow: ERROR [carry: CARDINAL] = 99;

then an example of its use might be:

Overflow [carry: 1]

indicating that **Overflow** was reported with argument **carry** having the value 1

Courier requires values for a **SEQUENCE OF UNSPECIFIED** to be a sequence of numbers. So as to retain readability in examples, the content of a **SEQUENCE OF UNSPECIFIED** is described using Courier notation. The reader should understand that the numeric representation of these types is what should be used as the content of the sequence.

1.3.3 Notation for C examples

Code examples are used in this document to describe the interface to the native file system. All examples are written in the C language as described in "The C Programming Language," Kernighan and Ritchie, Prentice-Hall, 1978.

The examples in Chapters 5 and 6 will present routines or portions of routines which make use of the resident system interface to provide the necessary support for attributes or procedures. These examples are intended to be working examples; however, the procedure and variable names are chosen for maximum clarity and may not necessarily adhere to the restrictions of a particular compiler

2. XNS PROTOCOL DEPENDENCIES

All implementations of the FilingSubset Protocol require, as a prerequisite, working implementations or at least knowledge of several other XNS protocols Specifically, the FilingSubset is dependent upon the XNS Internet Transport, Courier, Bulk Data, Clearinghouse, Authentication and Time Protocols.

Although the intent of this document is not to describe actual implementations of these supporting protocols, this section discusses specific portions of these protocols which must be implemented and recommends certain implementation restrictions which will further the interconnectivity of FilingSubset implementations.

2.1 Internet Transport Protocols

Any FilingSubset implementation requires a functional implementation of the following Internet Protocols [8]: Internet Datagram Protocol, Sequenced Packet Protocol (SPP), Routing Protocol and Error Protocol.

Although the Packet Exchange Protocol (PEP) is not essential to the implementation of the FilingSubset, a PEP implementation is recommended since it should be used for locating Clearinghouse and Authentication services on the network.

2.1.1 Relationship of transport connection to FilingSubset session

Implicit within the layered architecture of the XNS protocols is the notion that a higher level connection exists independent of the transport connection supporting it The XNS architecture allows complete independence between transport and session connections, so that one or more transports may be used to communicate procedure calls to a single session, and a single transport may be used to communicate procedure calls to one or more sessions With this in mind, some FilingSubset implementations may wish to restrict a session connection to a single transport connection In order to provide the greatest degree of interconnectivity, FilingSubset clients should restrict all operations pertaining to a given FilingSubset session to a single transport connection. That is to say, clients should endeavor to keep the transport alive during the life of the session, and should not divide operations on a given session among different transport connections.

The XNS architecture permits a further distinction to be made between the SPP and Courier connections. Within this model, a single transport connection implies both a single Courier connection and a single SPP connection.

2.2 Courier and Bulk Data Protocols

All FilingSubset procedures are defined in the Courier language and also pass arguments and convey results as Courier data types; therefore, implementation of the Courier Remote Procedure Call Protocol [6] is required by all FilingSubset implementations.

The FilingSubset is an application level protocol based on the Courier remote procedure call model. As such, all subset clients issue the initial connection request to the Courier well known socket. Implementation of a FilingSubset service implies the existence of such a Courier listener which accepts incoming requests and creates a connection which the subset service subsequently uses.

The FilingSubset Protocol uses the Bulk Data Protocol [2] to transfer file contents and enumerated lists. All FilingSubset implementations must support, at a minimum, the **BulkData.immediate** and **BulkData.null** transfer choices. Third party bulk data transfers need not be supported for operation of the FilingSubset.

2.3 Clearinghouse Protocol

The Clearinghouse Protocol [4] is used to interrogate a Clearinghouse service for information about objects within the network, such as users, services, etc. It is recommended that FilingSubset implementations use the Clearinghouse service when those functions are required; however, a subset implementation can perform without a Clearinghouse service. Alternative methods are presented for those cases where a Clearinghouse service does not exist or the implementation of such a service is non-trivial.

2.3.1 Implementation with a Clearinghouse service

FilingSubset client implementations may make use of the Clearinghouse Protocol for two specific functions: 1) location of Clearinghouse servers and 2) location and description of file services.

2.3.1.1 Location of a Clearinghouse server

Locating a Clearinghouse server is a prerequisite to the use of a Clearinghouse for other activities. The **BroadcastForServers** operation described in Section 3.8 and Appendix E of the *Clearinghouse Protocol* [4] is the recommended procedure for locating a Clearinghouse server.

Use of the **BroadcastForServers** procedure implies that a functional implementation of the Packet Exchange Protocol exists.

2.3.1.2 Location and description of file services

In most instances, a FilingSubset client will possess the name of the service for which a connection is desired. The client must translate this name into the unique network addre which is used by the Internet Transport Protocols. In addition, the client must determine whether the name identifies a properly registered file service and if so, what level of the Filing Protocol are provided, what level of Authentication is supported and the required secondary credentials item types.

This is accomplished by issuing a Courier **RetrieveItem** procedure call to a Clearinghouse service requesting the **fileService** property. The values returned from this procedure are described in *Clearinghouse Entry Formats* [5] and contain the following:

- the distinguished object name of the server, type **Clearinghouse.ObjectName**
- a description of the file service, type **STRING**
- a list of network addresses, **SEQUENCE OF Clearinghouse.NetworkAddress**
- the Authentication levels supported by the service, type **AuthenticationLevelValue**
- the level of Filing Protocol support provided by the service
- the secondary credentials item types required by that service

The Clearinghouse service may report an error indicating that the name supplied does not identify a file service.

2.3.2 Implementation without a Clearinghouse service

Under some circumstances, the use of a Clearinghouse service may not be possible or the implementation costs too great. Alternatives to Clearinghouse use are presented here, however, their use will result in a lesser degree of functionality, robustness and security. Each of these methods may be used as individual or collective replacements for the respective Clearinghouse procedures.

2.3.2.1 Location of a Clearinghouse server

A simpler mechanism of caching Clearinghouse server addresses may be used to avoid implementation of the **BroadcastForServers** procedure. This requires maintaining a single file which contains the host name and network address of the Clearinghouse servers within commonly used domains.

For example, the file `Sales.map` could contain entries for the clearinghouse servers which service the `sales` domain, as follows:

```
sales-clearinghouse1   1#1-123-456-789
sales-clearinghouse2   1#1-987-654-321
```

This mechanism is quite easily implemented and can provide service for the more commonly used domains. However, it is not reasonable to employ this mechanism as a means to access all domains on the network since the volume of data would be quite large and the data itself would be subject to change as the network changes.

2.3.2.2 Location and description of file services

A mechanism similar to that described above for locating Clearinghouse servers could also be employed for locating file services. However, this will only provide the name to address translation and will not allow the client to determine the file service's requirements regarding levels of Authentication support, protocol support and secondary credentials. A client should be prepared to receive appropriate error conditions from the service if the service does not support the FilingSubset Protocol or requires credentials different from those supplied.

If a Clearinghouse service does exist and its address can be ascertained with either of the previously mentioned methods, then location of the file service as specified in Section 2.3.1.2 is preferable to maintaining a large and dynamic file of file service addresses.

2.4 Authentication Protocol

FilingSubset clients and services rely on the Authentication Protocol. This defines 1) the format of the user's network credentials and verifiers and 2) the protocol to use when communicating with Authentication services to create and validate these credentials and verifiers.

FilingSubset services should provide support for immediate credentials of which there are two types: **simple** or **strong**. Clients may use either of these types although the use of strong credentials is encouraged because they incorporate a greater level of network security. However, support for strong credentials requires the use of an Authentication Service.

Subset clients provide both primary and secondary credentials and a verifier on a **Logon**. Primary credentials are those credentials that resolve a client's identity to a Clearinghouse name. Validation of primary credentials is accomplished through use of the *Authentication Protocol* [1] unless a client uses **nullPrimaryCredentials** which indicates that network authentication is not to be performed.

Secondary credentials communicate host-specific authentication information. These credentials are validated according to the mechanisms defined by the host operating system for the service. As such, the format of secondary credentials are service-specific. *Secondary Credentials Formats* [9] describes a set of well-known secondary item types to be employed by services.

2.4.1 Implementation with an Authentication Service

Successful use of the Authentication Protocol is predicated on interaction with an Authentication Service. An Authentication Service is located in much the same way as a Clearinghouse service. The **BroadcastForServers** operation, as described in Section 3.6 of the *Authentication Protocol*, is used. This operation requires a working implementation of the Packet Exchange Protocol; however, an alternate mechanism, similar to that suggested in Section 2.3.2.1 of this document, may be employed.

2.4.1.1 Primary credentials

A client does not require interaction with an Authentication Service to create simple credentials and a verifier. The credentials consist of the user's distinguished Clearinghouse name or alias, of type **Clearinghouse.Name**, while the verifier is of type **HashedPassword** The verifier value is computed using the algorithm in Section 5.1 of the *Authentication Protocol*.

A FilingSubset service validates simple credentials by issuing a **CheckSimpleCredentials** call to an Authentication Service. The subset service passes the client supplied credentials and verifier and receives a boolean response, which if TRUE indicates a valid verifier Appropriate errors are returned if the verifier is invalid.

Strong primary credentials are manufactured by an Authentication Service at the request of a client initiating a conversation. These credentials are then passed to a FilingSubset service which performs the validation using the procedure described in Section 2.9.1 of the *Authentication Protocol*.

2.4.1.2 Secondary credentials

Secondary credentials are created by a client depending upon the set of **SecondaryItemType** values required by the FilingSubset service. The client determines the necessary types by issuing a request to a Clearinghouse service. The required items of **SecondaryItemType** are then combined to form the secondary credentials passed to the service If a Clearinghouse service is not available, the service will reject a **Logon** when a client supplies the wrong set of secondary item types for the service. In this case, the item types required by the service will accompany the error so that the client may use these to repeat the **Logon** with the correct item types.

Secondary credentials are also available in the **simple** and **strong** types and it is recommended that a FilingSubset service support both of these types. Simple secondary credentials are validated by the service using the mechanisms supplied by the host operating system.

Strong secondaries are simple secondaries encrypted with the client's conversation key, as used to form the strong primary credentials. The unencrypted simple secondary value is formed, then padded with zero bits to a multiple of 64 bits and encrypted using the client's conversation key as described in Section 5.3 of the *Authentication Protocol*.

2.4.2 Implementation without an Authentication Service

FilingSubset clients and services can operate successfully without the use of an Authentication Service by relying on validation of the secondary credentials only.

2.4.2.1 Primary credentials

The use of simple and strong primary credentials is precluded if use of an Authentication Service is not possible, since the use of either type of credentials involves interaction with the service.

A client can, instead, use **nullPrimaryCredentials** which indicates to the service that network authentication is not to be performed.

2.4.2.2 Secondary credentials

Secondary credentials of strength **none** or **simple** can be employed by subset clients and services without requiring an Authentication Service. Strong secondaries cannot be used since they are encrypted with a conversation key which is created by the Authentication Service.

The use of simple secondaries is identical to that described in Section 2.4.1.2.

The use of secondaries of strength **none** is not encouraged since a client must use **nullPrimaryCredentials** when an Authentication Service is not available. This would provide no user authentication within the network or on the specific service.

2.5 Time Protocol

FilingSubset implementations do not explicitly require use of the Time Protocol as it applies to the use of network time servers. However, several FilingSubset attributes are defined in XNS Time format, which will imply a conversion to/from the native operating system time format.

3. CLIENT IMPLEMENTATION

A FilingSubset client interacts with a FilingSubset service on behalf of a user. This user may be a human being, where commands are input from an interactive user interface, or another software entity, where actions are requested via a procedural interface. In all cases, the user initiates the interaction between client and service: the service never initiates activity with a client.

The client is responsible for translation of user requests into FilingSubset procedures to effect the desired user action. The FilingSubset procedures, in turn, provide the client with low level access to the file system of remote hosts. It is the client's responsibility to sequence these procedure calls and maintain an appropriate control state to provide the desired action.

The FilingSubset client presented in this section allows the user to perform the following actions:

- open a session
- close a session
- enumerate a file or files in a directory
- store a file or files on a remote service
- retrieve a file or files from a remote service
- delete a file or files on a remote service
- create an empty directory on a remote service

Only those functions supported by the FilingSubset are used by this client. All pathnames presented to the service are specified in absolute syntax, where the **nullHandle** is used to specify the parent directory. Attribute integrity is assured by conveying all legal mandatory attributes to the service on a **Store** and retaining all mandatory attributes in the local file system on a **Retrieve**. The use of a single transport connection for the session implies that the client cannot enumerate the candidate files for retrieval or deletion on one connection, then simultaneously open a second connection to perform the retrieval or deletion. Instead, the client must save the enumerated list returned by the service and use this list when performing the retrieval or deletion later.

3.1 Opening a session

Prior to accessing any files on a file service, a client must open a Courier connection to a subset service and perform a **Logon**. This procedure returns a session handle which is used on subsequent procedure calls until a **Logoff** is issued or the connection is closed. Upon return from the **Logon**, the client may issue other procedure calls to the service by including the returned session handle on those calls.

In some scenarios, a user may specify either a file service name or a network address for the intended service. In the case where a name is specified, this name must be translated to the associated network address via the procedure outlined in Section 2.3 before the **Logon** can be performed.

User credentials are created as defined by the Authentication Protocol. Clients must supply both primary and secondary credentials and a verifier to the service. The client should use the appropriate primary and secondary credentials based upon the Authentication level supported by the service, as determined by the procedure outlined in Section 2.3. Secondary credentials are created according to the procedure outlined in Section 2.4.1.2.

Once the **Logon** has successfully completed, the client may open a default directory, generally the root. This is not necessary when the client uses the absolute pathname syntax for all file identification, since specification of the **nullHandle** as a directory handle implies the root.

The root file may be opened by specifying the **nullHandle** for the directory file handle along with an empty attribute sequence, **[SEQUENCE 0 OF UNSPECIFIED]**. The use of **nullHandle** with the empty attribute sequence will imply the root directory regardless of any service-specific pathname syntax.

3.1.1 Maintaining an open session

Once a session has been successfully established, the client is responsible for keeping that session open, especially during long periods of inactivity. This is accomplished by issuing **Continue** procedures at specific intervals to ensure that the service does not terminate the session.

Once the **Logon** has completed, the client issues a **Continue** to the service to determine the service specific continuance value. The value returned is specified in seconds, so the client should decrease this by some factor (i.e., $\frac{1}{2}$) and establish a timeout mechanism which will issue another **Continue** at the expiration of the interval. This allows the client to issue the next **Continue** well before the service timeout interval.

Once the timeout mechanism is in place, any FilingSubset call including the **Continue**, is considered to be activity and causes the service to reset its timer. The client should cancel any pending timer prior to issuing any procedure and reset the timer upon successful completion of each procedure. Once the **Logoff** has successfully completed, the client should cancel any pending timer without reestablishing it.

The XNS architecture allows any given session to exist over multiple transport connections. The definition of the FilingSubset does not preclude use of this facility; however, it is recognized that not all subset services can support this function. Clients should not assume that this facility exists and, should be able to operate correctly with only a single transport connection for each session. Likewise, clients should, where possible, prevent an early termination of the transport connection.

3.2 Closing a session

A user will typically close a connection once the desired interaction with that service has been completed. Closing a connection requires a **Logoff** to release the session handle followed by a close of the Courier connection used for that session and usually, the underlying SPP connection. After successfully completing the **Logoff**, the client should also cancel any pending continuance timer alarms.

3.3 Enumerating a file(s)

It is often useful to enumerate the pathnames of files in a given directory and optionally to retrieve additional attributes of those files. This is accomplished through the **List** procedure A client is responsible for specifying those attributes which will be returned along with the search criteria to be used. Subset services are only required to provide support for mandatory and implied attributes; therefore, the client should restrict the requested attribute types to the set of mandatory attributes: **createdOn**, **dataSize**, **isDirectory**, **modifiedOn**, **pathname** and **type**. A request for other attribute types may result in return values which are either null or constant for the service implementation. A client may also specify **allAttributeTypes** to request that values for all attributes supported by the service be returned.

Clients should only request those attributes which are of interest to the user Asking for unnecessary attributes may result in more performance overhead on the service and undoubtedly results in a larger amount of transferred data.

The FilingSubset allows use of the **pathname** attribute in the specification of the selection criteria and requires all services to support the absolute pathname syntax. Clients should specify a **scope** of type **filter**, with a **filter** type of **matches** on the attribute **pathname** which has a value in the absolute form. This guarantees that the service will accept the specification criteria. Since the pathname value specified in the filter is in absolute form, the **nullHandle** should be supplied as the **directory** handle on the **List**. The service will return appropriate errors if the **pathname** value specified is non-existent or inaccessible.

The stream of enumerated data returned to the client is of type **StreamofAttributeSequence** The client interprets this stream and present the results to the user. This stream contains one **AttributeSequence** for each file listed where each **AttributeSequence** contains an attribute value for each requested attribute. Those attributes defined by the FilingSubset to be mandatory or implied will contain a non-null value whereas those attributes defined as optional may have a null value (**Attribute**: **[type**: **AttributeType**, **value**: **SEQUENCE 0 OF UNSPECIFIED]**) if the service does not support the requested attribute.

Due to restrictions in the underlying operating system, a given service may actually perform the enumeration as a two step process: 1) enumerate the candidate files and 2) determine the requested attributes. This implies that an individual file may be deleted and/or inaccessible at the time the service determines the attribute values. If a file no longer exists, that file will simply not be returned by the service in the enumerated list. If a file has become inaccessible, all attributes except **pathname** will have null values.

3.4 Storing a file(s)

A new file is created on a remote service through use of the **Store** procedure. The client is responsible for specifying all mandatory attributes (except **modifiedOn**, which is illegal) on the **Store**. In addition, the client must determine if the file exists on the service and delete the existing file, if desired, when the service does not support multiple versions of a file with the same name.

Several FilingSubset procedures may be executed during the course of storing a file or files on a remote service. Since a user may provide a file specification which contains wildcard characters, the client must first enumerate the possible files on the local file system and then store each file individually, with or without user confirmation.

The client lists the specified files on the local file system using the standard host operating system facility. For each file listed, the values for the corresponding mandatory FilingSubset attributes (**createdOn**, **dataSize**, **isDirectory**, **modifiedOn**, **pathname** and **type**) should be determined. Having accomplished this, the client can create the file on the remote service by issuing a **Store** followed by a **Close** to release the file handle created on the **Store**.

The **Store** procedure should specify the following arguments: the directory handle **nullHandle**, an **AttributeSequence** containing values for all mandatory attributes except **modifiedOn**, the empty sequence for **controls**, the bulk data stream type **BulkData.immediateSource** and the session handle returned on the **Logon**. The file is read from the local system and transferred via a bulk data stream to the service. If the file is successfully created, a file handle is returned by the service.

If the **Store** was successful, a **Close** is issued to release the returned file handle. This will specify the file handle and the current session handle. Once the file has been closed, the sequence of **Store** and **Close** can be repeated for each file to be stored.

3.4.1 Overwriting an existing remote file

The possibility exists that a given FilingSubset service implementation does not support multiple versions of similarly named files and will not allow the client to overwrite an existing file on the service. In this case, the error **InsertionError [problem: fileNotUnique]** is returned by the service. A client who wishes to achieve the effect of overwriting an existing file on a service which does not support multiple versions must first delete the existing file and then perform the **Store**.

A client can determine if the file exists on the service and if it should be deleted by enumerating the desired file requesting the **childrenUniquelyNamed** attribute. If the file is not found, the service returns the error **AccessError [problem: fileNotFound]** and the client may continue with the **Store**. If the file does exist and the value returned for **childrenUniquelyNamed** is FALSE, then the client may store the file and the service will create the next highest version. If the value for **childrenUniquelyNamed** is TRUE, then the client may choose to either not perform the **Store** or first delete the existing file and then do the **Store**.

When the existing file must be deleted, the client should issue an **Open** and **Delete** similar to the scenario described in Section 3.6.

3.4.2 Bulk data transfer

File content is transferred from the client to service in a bulk data stream. The format of the data in this stream will vary depending upon whether the transferred file is of type **tAsciiText**.

A file of type **tAsciiText** is transmitted as a **StreamofAsciiText**. This represents an encoding of the records within the file, where a record is determined by the native operating system definition. The client must strip any operating system specific data from the record along with the record delimiter, if one exists, and transmit this as type **AsciiString**. The boolean **lastByteSignificant** must also be set to reflect whether the length of the record was odd or even, since **AsciiString** is actually a **SEQUENCE OF UNSPECIFIED**. These lines are then formatted into the **StreamofAsciiText** and transmitted.

Any files which are not of type **tAsciiText** are simply sent as a single uninterpreted stream of bytes. The service writes this stream to the local file system with no change of format.

3.5 Retrieving a file(s)

Similar to storing files, the client is responsible for retaining all mandatory attributes of a file retrieved from a remote service. On operating systems where multiple versions are not supported, the client must also determine if the local file exists and needs to be deleted before the file can be retrieved. A client also has the option to override the service's notion of the file type and, in turn, force the service to transfer the file as a specified type

A user may provide a file specification which contains wildcard characters. This implies that the client must first enumerate the possible files and then retrieve each one individually. This enumeration has another purpose, in that it retrieves the attributes of the file as stored on the remote service, so that the client can retain these attributes on the local file system.

Initially, the client performs a **List** in a manner similar to Section 3.3 The user-supplied file specification is provided as the **pathname** attribute value and all mandatory attributes are requested in **types**. The service returns a bulk data stream containing a **SEQUENCE OF AttributeSequence** for each file found which matches the **pathname** value No subsequent procedures can be issued before the entire bulk data stream is received, so the data received by the client must be retained until it can be used for the retrieval sequence later

For each file to be retrieved, the client issues an **Open** to obtain a file handle, a **Retrieve** to transfer the file and a **Close** to release the file handle. The **Open** requires the following arguments: an **AttributeSequence** containing the **pathname** attribute value returned from the **List**, the **directory** handle **nullHandle**, the empty sequence for **controls** and the session handle returned from the previous **Logon**. If the client wishes to request a particular type of transfer, the desired value for the **type** attribute would also be included in **AttributeSequence**. The file handle returned from the **Open** is then used on the subsequent **Retrieve** and **Close** calls.

The **Retrieve** procedure requires the file handle returned from the **Open**, a bulk data stream type of **BulkData.immediateSink** and the session handle obtained on the **Logon**. The resulting bulk data stream is received from the service and written to the local file. All attributes returned from the **List** should be retained along with the file in the local file system. Sections 5 and 6 of this document describes how this is done on the UNIX and VMS operating systems,respectively.

Once the **Retrieve** has completed, either successfully or unsuccessfully, a **Close** procedure is issued specifying the file handle and the current session handle. Once the file has been closed, the next file can be retrieved as determined by the enumerated data returned from the **List**.

3.5.1 Overwriting an existing local file

Some operating systems do not support the ability to create multiple versions of the same named file. On those systems, the client may wish to allow the user to decide whether to overwrite an existing local file or not. To accomplish this, the client must determine if the specified local file exists. If the file does not exist, the client may continue with the **Retrieve**. If the file does exist, the user may be prompted for a response. If the file is not to be overwritten, the client will not retrieve this file and continue with the next file in the enumerated list. Otherwise, the file is deleted via the local mechanisms and the file subsequently retrieved.

3.5.2 Bulk data transfer

File content is transferred from the service to client in a bulk data stream. The format of the data in this stream will vary depending upon whether the transferred file is of type **tAsciiText**.

A file of type **tAsciiText** is transmitted as a **StreamofAsciiText**. This represents an encoding of the records within the file, where a record is determined by the native operating system definition. The client must format the data from each **AsciiString** within this stream according to the local operating system conventions and write it to the local file Specifically, any line delimiters required by the local system will have to be added to the string as it is written. Since the string is transmitted as a **SEQUENCE OF UNSPECIFIED**, the boolean **lastByteSignificant** is used to determine if the client should ignore the last byte of the data string. Decoding of the **StreamofAsciiText** is operating system specific and implies that the true value for the **dataSize** attribute may be different than that supplied by the service.

Any files which are not of type **tAsciiText** are sent as a single uninterpreted stream of bytes The client writes this stream to the local file system with no change of format.

3.6 Deleting a file(s)

File deletion is accomplished in a manner similar to that of storage and retrieval. A **List**, requesting the **pathname** attribute, is performed to enumerate the candidates for deletion For each file returned, the file is deleted by the sequence of procedures: **Open** and **Delete**.

The **List** is executed specifying the arguments: the **directory** handle **nullHandle**, an **AttributeTypeSequence** containing only the **pathname** attribute, a **scope** of type filter with a filter type of **matches** including the user supplied file specification as the **pathname** attribute value, the bulk data stream type **BulkData.immediateSink** and the current session handle. The bulk data stream returned will contain a pathname attribute value for each file matching the user specification. The entire bulk data transfer must complete before the client can continue with the file deletion. This implies that the enumerated list will have to be retained for use later.

Each file in the returned bulk data list is opened via **Open**, specifying an **AttributeSequence** containing the **pathname** attribute value returned from the **List**, the **directory** handle **nullHandle**, the empty sequence for **controls** and the current session handle. Upon successful completion, a file handle is returned which is used by the client on the subsequent **Delete**. Appropriate errors will be returned from the service if the file does not exist or is inaccessible.

The **Delete** requires the file handle returned from the **Open** and the current session handle. Once the file is deleted, the file handle associated with that file is invalidated by the service A **Close** should be issued to release the associated file handle if an error occurs on the deletion.

The client should be aware that a given service may or may not support deletion of directory files and their descendants. If a service does not support deletion of directory files, the service will return the error **AccessError [problem: accessRightsInsufficient]** A client should not assume that the file was in fact deleted unless the **Delete** returns successfully.

A service may not always guarantee that all descendants of a directory file will, in turn, be deleted. When the service cannot support this feature or encounters a problem deleting the descendants, the error **AccessError [problem: accessRightsInsufficient]** is reported. Clients should be aware that when this error is reported, a portion of the directory tree may still remain on the service.

3.7 Creating a directory

The FilingSubset allows directory files to be created by using the **Store** procedure and providing an **isDirectory** attribute value of TRUE. A given service may allow or disallow the creation of a directory file and, if allowed, may also only allow the creation of empty directory files.

The sequence of steps used to create a directory file is almost identical to that of storing a file. The client supplies the following arguments on the **Store** procedure: the **directory** handle **nullHandle**, an **AttributeSequence** containing the set of mandatory attributes where the value for **isDirectory** must be TRUE and the value for **type** should be **tDirectory**, the empty sequence for **controls**, the bulk data stream type of **BulkData.nullSource** and the current session handle. No bulk data is transferred to the service since the source stream specified is of type **BulkData.nullSource**. The service returns a file handle upon successful completion

Once the directory is successfully created, a **Close** must be issued to release the file handle

4. SERVICE IMPLEMENTATION

A FilingSubset service interprets Courier procedures and provides the necessary interfaces to the local operating system. As such, a service implementation must accept the procedures defined by the FilingSubset: **Logon**, **Logoff**, **Continue**, **Open**, **Close**, **Store**, **Retrieve** and **Delete**.

This section describes how to support these procedures in a FilingSubset service independent of any underlying file system. Each procedure is discussed in detail, describing the actions required to interface to the local file system, acceptable procedure argument values and the use of specific error return values.

The service implementation described in this section provides support for the minimal functionality defined by the FilingSubset as summarized in Section E.3.7 of the *Filing Protocol*. Specifically, all file identification is performed with the **pathname** attribute in the absolute form with an accompanying **nullHandle** directory handle. All mandatory and implied attributes are supported and retained with stored files. The creation of empty directory files is supported, although not required by the FilingSubset; however, creation of non-empty directories and retrieval of directories is not supported. Additional comments may also be provided for common functions which are above the minimal functionality but may be of merit to specific implementation schemes.

Several sections of the implementation scheme presented here assume that a FilingSubset session is supported by a single transport connection, where loss of the transport implies loss of the session. This implementation also relies on the premise that a single instance of a process (as defined by the local operating system) will service a single session from the initial establishment of a Courier connection to the subsequent termination of the session. This allows the state of the session to be maintained internal to the process and eliminates a reliance on interprocess communication. The scenarios described here would need to be enhanced to remove these restrictions.

4.1 Common data structures

A service is responsible for creation and maintenance of several data structures which reflect the current state of a client's interaction with the service. The *session handle* is used to maintain the state of a FilingSubset session over the life of the session. The *file handle* maintains the state of a file that a client has opened on the remote service.

4.1.1 Session handle

The session handle contains two items: a **token** which is a unique service specific value representing the session, and a **verifier** which is an Authentication verifier used to enforce security on consecutive session procedure calls.

The item **token** is generated by the service when the session handle is created. The value given to **token** is an identifier which is used by the service to point to a *session context block*. This context block actually contains various entries relevant to the associated session, such as:

- the state of the session (i.e., logged on, file currently open, store in progress, retrieve in progress, etc.)
- identification of the underlying Courier connection
- the primary credentials of the user who is logged on
- the current verifier
- a list of handles for files which are currently open in the session

The session state is updated by each procedure processing routine to reflect the current activity of the session. This updating ensures consistency across procedure calls and allows errors to be returned for inappropriate calls sequences.

4.1.2 Fi handle

A file handle is used by the client and service to identify files which are to be accessed on the service. Upon completion of a successful **Open** or **Store**, a file handle is returned which identifies the file to the service on subsequent calls. This handle value is used to point to a *file context block* which contains information relevant to the associated file such as:

- the **pathname** attribute value for the file as specified on the **Open** or **Store**
- the **type** attribute value specified by the client on the **Open**
- an appropriate entry for each of the mandatory attribute values, **createdOn**, **dataSize**, **isDirectory**, **modifiedOn** and **type**
- a field indicating whether the file is physically open, closed, etc.
- any operating system specific structures as needed by the implementation

An **Open** procedure sets the **pathname** and **type** fields to the values specified in the **AttributeTypeSequence** provided. The remaining mandatory attribute values are determined and set appropriately. Any operating system specific structures are also initialized at that time. The values contained in the context block allow subsequent procedures to discern relevant information about the file by looking in this context block rather than interacting with the local file system.

A **Store** sets the **pathname** field and all attribute value fields to those values provided on the **Store**. The service will then retain these values in the local file system.

4.2 Common support

Many of the procedure routines perform a common set of actions in addition to any procedure specific processing required. All routines, with the exception of **Logon**, must verify the session and reset the continuance mechanism prior to other actions. Also, all routines which specify a file handle (**Close**, **Retrieve** and **Delete**) must check the file handle for validity.

4.2.1 Session validation

The session handle provided on each call subsequent to the **Logon**, must be validated by the service. Specifically, two functions are accomplished by this validation: 1) the verifier included in the session handle is revalidated and 2) the internal state of the session is checked for consistency..

4.2.1.1 Verifier validation

The verifier included in the session handle may be one of two types: **simple** or **strong**, depending upon the primary credentials type provided by the **Logon** that created the session handle. The implementation presented here uses **simple** credentials, the validation of **strong** credentials is described in the *Authentication Protocol* [1].

The mechanism for validating a simple verifier involves saving the **Logon** verifier within the session context block. Each subsequent procedure call simply compares the verifier within the session handle against the saved verifier and returns the error **AuthenticationError [problem: verifierInvalid]** if they do not match

4.2.1.2 Session consistency validation

The session context block created at **Logon** is used to validate the internal state of the session. The **token** within the session handle points to the session context block corresponding to the associated session. Specifically, the service verifies that the session state reflects an open session. If the **token** value is invalid or the context block pointed to represents a session which is not open, the error **SessionError [problem: tokenInvalid]** is returned.

4.2.2 Use of the continuance timer

A service cannot always expect that a client will terminate a session explicitly. The service should also maintain the option of terminating an open session after some specified period of inactivity. To enforce this, a service specific continuance value is maintained. This value defines, in seconds, the interval which must elapse between successive client procedure calls before the service will terminate the session.

Upon completion of a successful **Logon**, the service establishes an internal continuance timer which will cause the execution of a routine to terminate the session upon expiration of this interval. During each successive procedure call from the client, the processing routine cancels the previous timer and rearms the mechanism again. After a **Logoff** is successfully completed, the service cancels the previous timer and does not reset the interval.

If the continuance interval expires before the client issues its next procedure call, the service can terminate the session by closing any open files, releasing the associated file context blocks, closing the underlying Courier connection and releasing the session context block.

4.2.3 File handle validation

The **Close**, **Retrieve** and **Delete** procedures require the file handle for a file previously opened on the service. To maintain consistency, the service should perform a verification of the file handle in each of these routines.

These routines do not allow the specified file handle to be **nullHandle**. If **nullHandle** is used, the error **HandleError [problem: nullDisallowed]** is returned.

The state entry within the file context block is also checked to insure that the file was previously opened. The error **HandleError [problem: invalid]** is returned if the file pointed to by the file handle is not open.

Some FilingSubset implementations may not guarantee that a previously opened file is not deleted by another utility on the system. These services should check for file existence each time the file handle is used and should report the error **HandleError [problem: invalid]** if the file no longer exists.

The ownership of a previously opened file may also be altered by other utilities even though the client has a valid file handle. If the service is presented with a valid file handle, but cannot access the file that the handle references, then the error **AccessError [problem: fileChanged]** is reported.

4.3 Procedures

4.3.1 Logon

Logon: PROCEDURE [service: Clearinghouse.Name, credentials: Credentials, verifier: Verifier]
RETURNS [session: Session]
REPORTS [AuthenticationError, ServiceError, SessionError, UndefinedError] = Filing.Logon;

Logon establishes a session which is used to control the subsequent interaction between client and service. This procedure is accompanied by three arguments: **service**, **credentials** and **verifier**. **Service** is the distinguished name of the service being connected to while **credentials** and **verifier** describe the credentials to be used in validating a user.

This procedure initially verifies that the service being connected to is in fact the service currently processing the procedure. It is possible for multiple FilingSubset services to reside on the same network server, where each service has the same or a different root directory. Each service should maintain an internal tag to identify itself. This tag is used to validate the service name provided on the **Logon**.

Secondary / Primary	none	simple	strong
nullPrimaryCredentials	legal	legal	illegal
simple	legal	legal	illegal
strong	legal	illegal	legal

Table 4.1 Primary and secondary credentials combinations

User credentials are validated according to the type and strength of credentials supplied. The error **AuthenticationError [problem: secondaryCredentialsTypeInvalid]** should be returned if the combination of primaries and secondaries is not allowed as shown in Table 4.1.

The primary user credentials and verifier are validated as specified by the *Authentication Protocol*. Credentials of type **nullPrimaryCredentials** are not subject to any validation. A set of simple primary **credentials** and **verifier** are validated by passing them to an Authentication Service on a **CheckSimpleCredentials** procedure. A return value of **TRUE** indicates that the **verifier** is good. If the Authentication Service returns an **AuthenticationError**, the accompanying problem can be returned to the FilingSubset client as **AuthenticationError [problem: *problem*]**. The error **ServiceError [problem: cannotAuthenticate]** is returned if the Authentication Service can not be contacted. A set of strong primary **credentials** and **verifier** are validated as described in the *Authentication Protocol* with an appropriate error being returned if the credentials are invalid.

Secondary credentials are validated via the host operating system validation procedures, with appropriate errors being returned to the client if the validation fails. If a required **SecondaryItemType** is not supplied by the client, the error **AuthenticationError [problem: secondaryCredentialsTypeInvalid]** is returned, indicating the item types required. Subset services should report **AuthenticationError [problem: secondaryCredentialsRequired]** if secondary credentials of strength **none** are used in conjunction with **nullPrimaryCredentials**.

On hybrid hosts, the **Logon** routine may also have to alter the effective identity of the process to be that of the user specified in the secondary credentials. This action is dependent upon the host operating system and is accomplished by the local mechanisms as required. This alteration would be performed to ensure that user access to and ownership of files can be handled by the standard host mechanisms.

To be consistent with the Filing Protocol, the process should not position itself in a default working directory other than the root for the given service. It is the client's responsibility to perform any positioning subsequent to a successful **Logon**; this implies that the client may open the root explicitly. On some hybrid host services, it may be advantageous for the service to position itself to the appropriate root directory during the **Logon** since use of the

nullHandle by a client implies the service root directory. The error **ServiceError [problem: serviceUnavailable]** should be reported if this positioning fails.

Once this has been accomplished, the procedure creates a session handle, initializes the state of this handle and returns the handle to the client. If an error occurs in creating the session handle, the error **ServiceError [problem: serviceUnavailable]** should be returned. In a case where a single service process is responsible for a session, the error **ServiceError [problem: serviceFull]** should be reported if a **Logon** is attempted prior to the **Logoff** which terminates the current session.

4.3.2 Logoff

```
Logoff: PROCEDURE [session: Session]
REPORTS [AuthenticationError, ServiceError, SessionError, UndefinedError] = Filing.Logoff;
```

Logoff indicates that the client is terminating the session. This procedure has one argument: **session** which is the handle of the session to be ended

This procedure initially verifies that the session handle supplied is indeed valid using the procedure in Section 4.2. The **Logoff** routine not only verifies that the session is currently open, it also has to determine if any other actions are in progress. Subset clients are encouraged to maintain a single Courier connection for each session, so the service is not required to support simultaneous actions. When the **Logoff** is issued while another operation is in progress, the error **ServiceError [problem: sessionInUse]** is returned.

During the existence of the session, it is possible that some files have been opened and not subsequently closed. Prior to returning to the client, all files which are currently open within this session are closed and the associated file context blocks released.

4.3.3 Continue

```
Continue: PROCEDURE [session: Session]
RETURNS [continuance: CARDINAL]
REPORTS [AuthenticationError, SessionError, UndefinedError] = Filing.Continue;
```

Continue registers a client's interest in maintaining an open session during a long period of inactivity. This procedure passes **session**, the handle of the session to be continued.

Processing of a **Continue** involves verification of the session handle and resetting of the continuance mechanism, as explained in Section 4.2.

4.3.4 Open

```
Open: PROCEDURE [attributes: AttributeSequence, directory: Handle,
    controls: ControlSequence, session: Session]
RETURNS [file: Handle]
REPORTS [AccessError, AttributeTypeError, AttributeValueError,
    AuthenticationError, ControlTypeError, ControlValueError, HandleError,
    SessionError, UndefinedError] = Filing.Open;
```

Open makes a file available for use by the client. The following arguments are passed in the **Open** procedure: **attributes** identifies the file to be opened; **directory** specifies the starting directory in which to look for the file; **controls** specify the controls applied to the resulting file handle and **session** is the client's session handle.

Initially, the session handle is verified and the continuance timer reset. Argument values and attribute types and values contained in **attributes** are then checked for validity. The FilingSubset defines restrictions on the argument values and attribute types and values provided on the **Open**. The following errors are returned for the respective conditions:

AttributeTypeError [problem: disallowed, type: ***attribute type*****]**
an attribute type other than **parentID**, **pathname**, **type** or **version** is specified

AttributeTypeError [problem: duplicated, type: ***attribute type*****]**
the **parentID, pathname, type** or **version** is specified more than once

AttributeTypeError [problem: illegal, type: ***attribute type*****]**
an attribute type not defined by the Filing Protocol is specified

AttributeValueError [problem: illegal, type: ***attribute type*****]**
an illegal attribute value is specified

AttributeValueError [problem: unimplemented, type: parentID]
a **parentID** value other than **nullFileID** is specified

AttributeValueError [problem: unimplemented, type: version]
a **version** value other than **lowestVersion** or **highestVersion** is specified

ControlTypeError [problem: disallowed]
controls is not the empty sequence

HandleError: [problem: invalid]
directory is not **nullHandle**

A file handle is allocated and initialized by setting the **pathname** and **type** entries from the corresponding attribute values. If no values are specified, an appropriate default is used (i.e., the root pathname for the service and the actual file type as determined by the service). The **pathname** value is used to identify the file when it is opened. The **type** attribute conveys the client's intention to have the file content transfer be of this type when retrieved. To be consistent with its treatment of directory files, a service may only allow the **tAsciiText** and **tUnspecified** type values to be specified. The error **AttributeValueError [problem: disallowed, type: type]** would be returned if another type was requested.

The service then verifies that the file exists and the user has permission to access the file. The error **AccessError [problem: accessRightsInsufficient]** is returned if the user does not have access to the file. **AccessError [problem: fileNotFound]** is returned to indicate that the file does not exist. The service determines the values for the **isDirectory** and **type** attributes and saves these in the file context block. The operating system specific structures are also initialized once the file is opened. If successful, the file handle is inserted into the open file queue in the session context block and returned to the client.

A service should allow the specification of an empty **AttributeSequence** in conjunction with use of **nullHandle** for **directory**. This is used to open the root of the file service regardless of any service specific pathname syntax. The file handle returned to the client, in this case, should not be **nullHandle**.

4.3.5 Close

Close: PROCEDURE [file: Handle, session: Session]
REPORTS [AuthenticationError, HandleError, SessionError, UndefinedError] = Filing.Close;

Close indicates that a client no longer needs a file handle within the specified session. Arguments to this procedure are **file**, the handle to be closed and **session**, the session handle.

The accompanying session handle is validated and the continuance mechanism reset. The file handle is checked for validity as described in Section 4.2.3 and, if successful, the file is closed and the handle removed from the open queue in the session context block.

4.3.6 List

List: PROCEDURE [directory: Handle, types: AttributeTypeSequence,
scope: ScopeSequence, listing: BulkData.Sink, session: Session]
REPORTS [AccessError, AttributeTypeError, AuthenticationError,
ConnectionError, HandleError, ScopeTypeError, ScopeValueError, SessionError,
TransferError, UndefinedError] = Filing.List;

List enumerates files within a directory and returns the requested attributes associated with those files. This procedure include the following arguments: **directory**, the handle for the directory to be enumerated; **types**, the attribute types to be returned; **scope**, the selection criteria for enumeration; **listing**, the bulk data sink to receive the attribute list and session, the handle of the session to be continued.

The **List** procedure initially verifies the session handle and resets the continuance timer. The argument values and attribute types provided on the call are validated and the following errors reported if the specified conditions occur:

AttributeTypeError [problem:duplicate, type: ***attribute type*****]**
an attribute type is specified more than once

AttributeTypeError [problem:illegal, type: ***attribute type*****]**
an attribute type not defined in the Filing Protocol is specified

ScopeTypeError [problem:illegal, type: ***scope type*****]**
a scope type not defined in the Filing Protocol is specified

ScopeTypeError [problem:missing, type: ***scope type*****]**
a scope type of **count** or **filter** is not specified

ScopeTypeError [problem:unimplemented, type: ***scope type*****]**
a scope type other than **count** or **filter** is specified

ScopeValueError [problem: illegal, type: ***scope type*****]**
an illegal **pathname** attribute value is specified
an illegal **count** value is specified

ScopeValueError [problem: unimplemented, type: filter]
a **filter** type other than **matches** is specified
a **matches** attribute type other than **pathname** is specified

TransferError [problem: aborted]
a bulk data **sink** type other than **BulkData.immediateSink** or **BulkData.nullSink** is specified

The routine can return if **listing** specifies **BulkData.nullSink** If **BulkData.immediateSink** is specified, the **pathname** attribute value is then used to enumerate the candidate files by the host operating system. The attributes requested are retrieved for each file enumerated and transferred as a bulk data stream to the client.

The stream is formatted as a **StreamofAttributeSequences** with a single **AttributeSequence** for each file enumerated. The ordering of the **AttributeSequence** types within the stream is determined by the associated **ordering** value for the directory listed. If the **ordering** attribute is not supported by the service, the ordering will be **defaultOrdering** (**[key: name, ascending: TRUE, interpretation: string]**).

The FilingSubset states that values must be returned for all attributes requested. If the attribute is mandatory or implied, a non-null value is returned. If an implied attribute is not supported, the value returned is the service default value for that attribute. The value returned for unsupported optional attributes will be null (**attribute: [type: attribute type, value: SEQUENCE 0 OF UNSPECIFIED]**); optional attributes which are supported return valid values. If **types** requests **allAttributeTypes**, then the service must return non-null values for all mandatory, implied and supported optional attributes.

The error **AccessError [problem:accessrightsInsufficient]** is returned if the requested file is inaccessible by the user. **AccessError [problem:fileNotFound]** is returned if the pathname value results in no files being enumerated.

Some service implementations may perform the file and attribute enumeration in two steps. Thus, the possibility exists that the service can enumerate the directory. but may not be able to access individual files later to determine values for the requested attributes. If an individual file has been deleted. then the file will not be included in the bulk data stream returned to the client. If the user no longer has permission to access the file. the service will return null values for all attributes except the **pathname** attribute This implies to the client that the requested attribute values are not accessible on the service even though the parent directory is accessible.

4.3.7 Store

```
Store: PROCEDURE [directory: Handle, attributes: AttributeSequence,
    controls: ControlSequence, content: BulkData.Source, session: Session]
RETURNS [file: Handle]
REPORTS [AccessError, AttributeTypeError, AttributeValueError,
    AuthenticationError, ConnectionError, ControlTypeError, ControlValueError,
    HandleError, InsertionError, SessionError, SpaceError, TransferError,
    UndefinedError] = Filing.Store;
```

Store creates a file with the specified content and the specified attributes. **Store** uses five arguments: **directory** specifies the directory in which to insert the file; **attributes**. the attributes to give to the created file; **controls**. the controls to be applied to the resulting file handle; **content**, the bulk data source used to send the file contents and **session**, the current session handle.

Store verifies the session handle and resets the continuance mechanism. The following errors are returned if the associated restrictions on argument values and attribute types occur:

AttributeTypeError [problem: disallowed, type: ***attribute type*****]**
an attribute type of **fileID**, **modifiedBy**, **modifiedOn**, **name**, **numberOfChildren**, **parentID**, **readBy**, **readOn**, **storedSize** or **subtreeSize** is specified

AttributeTypeError [problem: duplicated, type: ***attribute type*****]**
a valid attribute is specified more than once

AttributeTypeError [problem: illegal, type: ***attribute type*****]**
an attribute type not defined by the Filing Protocol is specified

AttributeTypeError [problem: missing, type: pathname]
a **pathname** attribute value is not specified

AttributeTypeError [problem: unimplemented type: ***attribute type*****]**
an attribute type of **checksum**, **createdBy** or **position** is specified

AttributeTypeError [problem: unreasonable, type: type]
the **isDirectory** value is TRUE and the **type** value is not **tDirectory**
the **type** value is **tDirectory** and the **isDirectory** value is FALSE

AttributeValueError [problem: illegal, type: ***attribute type*****]**
an illegal attribute value is specified

AttributeValueError [problem: unimplemented, type: accessList]
an **accessList** value other than **[defaulted:** TRUE**]** is specified

AttributeValueError [problem: unimplemented, type: childrenUniquelyNamed]
a **childrenUniquelyNamed** value other than the service specific value is specified

AttributeValueError [problem: unimplemented, type: defaultAccessList]
a **defaultAccessList** value other than **[defaulted:** TRUE**]** is specified

AttributeValueError [problem: unimplemented, type: isTemporary]
an **isTemporary** value other than FALSE is specified

AttributeValueError [problem: unimplemented, type: ordering]
an **ordering** value other than **defaultOrdering** is specified

AttributeValueError [problem: unimplemented, type: subtreeSizeLimit]
a **subtreeSizeLimit** value other than **nullsubtreesizeLimit** is specified

AttributeValueError [problem: unimplemented, type: type]
a **type** value other than **tAsciiText**, **tDirectory** or **tUnspecified** is specified

AttributeValueError [problem: unimplemented, type: version]
a **version** value other than **highestVersion** is specified

ControlTypeError [problem: disallowed]
controls is not the empty sequence

HandleError: [problem: invalid]
directory is not **nullHandle**

TransferError: [problem: aborted]
a bulk data **source** type other than **BulkData.immediateSource** or **BulkData.nullSource** is specified

If **content** specifies **BulkData.nullSink**, the routine returns to the client. Otherwise, a file handle is created and the values supplied for the mandatory attributes cached in the file context block. Table E 4 of the *Filing Protocol* [7] describes the values to be given to any mandatory attributes not specified on the procedure call. The value for the **dataSize** attribute should be the number of bytes as stored on the local file system once the bulk data transfer has completed.

The specified file is created using the local operating system procedures and the necessary operating system specific structures initialized in the file context block. The error **AccessError [problem: accessRightsInsufficient]** is reported if the user does not have permission to create the file. If the file exists, the error **InsertionError [problem: fileNotUnique]** is reported to the client. If no space exists on the service to create the file, the error **SpaceError [problem: mediumFull]** is returned. The previously allocated file context block is released if an error is reported.

The content of the file is read from the bulk data stream and written to the file on the local file system. If any problems are encountered while reading the bulk data stream or writing to the file system, the service sends an out-of-band notification to the client to abort the bulk data transfer, reports the error **TransferError [problem: aborted]**, and deletes the partial file. If the client aborts the bulk data transfer, the same error is also reported and the partial file deleted.

Upon successful completion of the bulk data transfer, the attribute values contained in the file context block are stored with the file through the use of the local file system mechanisms The file handle is queued onto the session context block and the file handle returned to the client. If an error is reported after the file handle has been created, the associated context block is freed.

Files of type other than **tAsciiText** are transferred as an uninterpreted sequence of bytes and are written to the local file system with no formatting. The data transferred in the bulk data stream will be of type **StreamofAsciiText** for a file of type **tAsciiText**. Each **AsciiString** within this stream will be written to the file in the appropriate format for the local operating system. The value of **lastByteSignificant** will indicate whether the last byte in each **AsciiString.bytes** should be written to the file.

To indicate that the file to be created is a directory, a client will set the **isDirectory** value to TRUE. A TRUE value for the **isDirectory** attribute also implies a **type** value of **tDirectory** if the **type** value is not specified; however, a **type** of **tDirectory** does not imply an **isDirectory** value of TRUE. If both the **isDirectory** and **type** values are specified and they are in conflict, the error **AttributeTypeError [problem: unreasonable, type: type]** should be reported. This error would also be reported if a **type** value of **tDirectory** is specified with no associated **isDirectory** value.

A subset service is not required to support the creation of directory files and will report **AccessError [problem: accessRightsInsufficient]** if directory creation is not allowed. Furthermore, a service which does support directory creation is not required to allow the creation of non-empty directory files. A service which does not support this feature reports the error **AttributeTypeError [problem: unreasonable, type: isDirectory]** if the client specifies an **isDirectory** value of TRUE in conjunction with **BulkData.immediateSource** and a non-zero length data transfer.

4.3.8 Retrieve

Retrieve: PROCEDURE [file: Handle, content: BulkData.Sink, session: Session]
REPORTS [AccessError, AuthenticationError, ConnectionError, HandleError,
SessionError, TransferError, UndefinedError] = Filing.Retrieve;

Retrieve transfers the contents of a file on the service to the client. Three arguments accompany the **Retrieve** procedure: **file**, the handle of the file to be transfered, **content**, the bulk data sink to receive the file contents and **session**, the handle of the session to be continued.

The **Retrieve** routine verifies the session handle and resets the continuance timer as described in Section 4.2. The supplied file handle is verified, as described in Section 4.2.3, and the following error reported if the corresponding restriction on argument values occur:

TransferError: [problem: aborted]
a bulk data **sink** type other than **BulkData.immediateSink** or **BulkData.nullSink** is specified

If **content** specifies **BulkData.nullSink,** the procedure returns. If **BulkData.immediateSink** is specified, then the file identified by the file handle is read from the local file system and written to a bulk data stream for transfer to the client. If an error is encountered while either reading the file or writing to the bulk data stream, an out-of-band notification is sent to abort the transfer and the error **TransferError [problem: aborted]** is reported to the client If the client, for some reason, aborts the transfer, then the same error is reported.

The bulk data stream may be formatted depending upon the type of the file being transferred. The type is determined from a combination of the **type** attribute value as it was specified on the previous **Open** and the **type** attribute value of the file as it exists on the local file system. If the client specified a **type** on the **Open**, the file content is transferred as that type. If **type** was not specified, the locally determined file type is used. The service determines the correct transfer type by examining the respective values in the session context block at the time of the transfer.

Files of a type other than **tAsciiText** are transferred as a single uninterpreted stream of bytes. A file of type **tAsciiText** will be transferred in the bulk data stream as type **StreamofAsciiText**. Each line of the input file is stripped of any operating system specific data, including line delimiters, and encoded into an **AsciiString**. If the number of characters in the line is odd then **lastByteSignificant** is set to FALSE; otherwise it is set to TRUE.

FilingSubset services are not required to permit the retrieval of directory files. A service which does not allow this reports the error **AccessError [problem: accessRightsInsufficient]**

The **isDirectory** entry in the file context block is used to determine if the file is indeed a directory.

4.3.9 Delete

```
Delete: PROCEDURE [file: Handle, session: Session]
REPORTS [AccessError, AuthenticationError, HandleError, SessionError,
    UndefinedError] = Filing.Delete;
```

Delete deletes an existing file. The following arguments are passed in the **Delete** procedure: **file**, the handle of the file to be deleted and **session**, the current session handle.

The session handle is verified, the continuance mechanism rearmed and the file handle verified as described in Section 4 2. The file specified by the file handle will then be deleted Different actions may be taken depending upon whether the file is a directory as determined by examining the **isDirectory** entry in the file context block. Upon successful deletion of the file, the associated file handle is removed from the open file queue in the session context block and released.

FilingSubset services are not required to allow deletion of directory files. If directory deletion is not supported, then the error **AccessError [problem: accessRightsInsufficient]** is reported. A service that does in fact support deletion of directories may not be able to guarantee that all descendants of that directory will in fact be deleted, in accordance with the Filing Protocol. The error **AccessError [problem: accessRightsInsufficient]** should also be reported for this condition. Clients should recognize that in the situation where this error is reported, the portion of the directory structure that cannot be deleted, along with other files which would have been encountered had the deletion continued may be retained on the service.

5. UNIX SYSTEM INTERFACE

Implementation of the FilingSubset under UNIX requires both procedure and attribute support within the native operating and file systems. This section presents an implementation scenario which describes the necessary interactions with the UNIX system.

Theis section describes those interface procedures required by the client and service implementations presented in Sections 3 and 4. These are by no means the only method for providing the facility desired; they have been chosen either because they have actually been tested or are more likely to be portable between various versions of UNIX In those cases where differences arise between implementations on UNIX 4.2BSD, UNIX 4 3BSD and UNIX System V, these differences are noted.

In several instances, the examples presented will be identical to the VMS counterparts presented in Section 6. This replication is done in an effort to make both the UNIX and the VMS sections complete stand-alone sections.

Several of the examples presented are predicated on the assumption that a separate UNIX process instance handles all procedure calls from the time the Courier connection has been established on the initial **Logon** call until the subsequent **Logoff** call. The examples also assume the definition of Filing defined constants and Courier defined data types. In the examples, the string "Filing_" is prepended to structure and variable names which are defined by the Filing Protocol.

5.1 Attribute Support

The FilingSubset Protocol distinguishes three classes of attributes: mandatory, implied and optional. This section describes specific scenarios under the UNIX operating system for

- services to retain attributes so that they may be interpreted by other native operating system utilities and returned when requested by network clients
- clients to retrieve and retain the attributes when dealing with remote services

All attributes presented here are discussed with respect to two areas: 1) where attributes must be retained in the native file structures and 2) how they may be retrieved from these structures and transferred to other FilingSubset clients and services Retention of attributes is of importance to FilingSubset clients when retrieving files from a service and by services when a client requests creation of a file on the service. Likewise, retrieval of attributes from the native file structures is used by clients when issuing a **Store** and by services when returning attributes on a **List** procedure.

5.1.1 Mandatory Attributes

Mandatory attributes are those attributes which must be interpreted by all FilingSubset implementations. These attributes are guaranteed to be retained by any service implementing the FilingSubset Protocol and must be accepted in specific procedure calls to the extent that they are legal arguments of the corresponding procedure in the Filing Protocol. Additionally, clients may wish to to retain these attributes when retrieving files from a service.The FilingSubset defines the following mandatory attributes: **createdOn**, **dataSize**, **isDirectory**, **modifiedOn**, **pathname** and **type**.

Each of these attributes is discussed with respect to the areas of retention and retrieval. Retention of an attribute value describes a mechanism for saving the specified XNS attribute value within the UNIX file system along with the file contents. Retrieval of attribute values presents methods for deriving the XNS value from the UNIX file system. In each of these cases, the values may need to be converted from one form to the other

In the case of the **createdOn** and **modifiedOn** attributes, the retention and retrieval of attribute values requires a conversion between the UNIX and XNS formats. The **createdOn** and **modifiedOn** values are always specified in XNS Time format [10]. XNS time is based on the number of seconds since 00:00:00 Jan. 1, 1901 Greewich Mean Time. The UNIX operating system maintains time in a form specifying seconds since 00:00:00 GMT, Jan. 1, 1970. To convert XNS time values to UNIX time values, the constant 2177452800 must be subtracted from the XNS value. Note that this constant is the XNS encoding for the UNIX time 00:00:00 GMT, Jan. 1, 1970 [((1970-1901) years * 365 days/year + 17 leap days) * 24 hours/day * 60 minutes/hour * 60 seconds/minute]. Conversion from UNIX format to XNS format simply requires adding the constant to the UNIX value.

5.1.1.1 createdOn

The **createdOn** attribute is useful in determining if similarly named files on different file servers within the network are identical. This is especially true on systems such as UNIX where versions are not supported. The ability to retain the **createdOn** date must be coupled with a mechanism for native utilities to provide this date on demand. This can be accomplished on UNIX by retaining the **createdOn** value in the file status field stat.st_mtime. This allows non-network UNIX users to access this date easily also allows the network client and service to determine and modify this date If this file is modified by local UNIX uilities, the date will change, in effect implying a new version to network users.

[Retention]

The **createdOn** value is first converted to UNIX form as described above and then retained in the UNIX file status block field, stat.st_mtime, by issuing a utimes call (4.2BSD or 4.3BSD) or a utime call (4.2BSD, 4.3BSD and System V).

The following example illustrates use of the utime procedure for retaining the **createdOn** and **modifiedOn** values:

```
#include    <sys/types.h>

#define     XNS_TIME_DIFFERENCE    2177452800   /* difference between base times */

/*
    routine:
        set_create_time
    input:
        pointer to file context block
            where
                if no createdOn value was specified on Store. createdon == 0
                if createdOn value was specified on Store, createdOn != 0, value is
                in XNS time format
    returns:
        none
*/

set_create_time(file_context_block)
file_handle     *file_context_block;
{

    time_t      time_buffer[2];
    time_t      time();

    if (file_context_block->createdon)        /* save createdOn if specified */
        time_buffer[0]= file_context_block->createdon - XNS_TIME_DIFFERENCE;
    else                                     /* else, set to current date/time */
        time_buffer[0]= time(0);

    time_buffer[1]= time(0);                 /* set modifiedOn to current date/time */

    utime(file_context_block->pathname,time_buffer);
```

[Retrieval]

Network processes can retrieve the **createdOn** value by issuing a stat call on the file and returning the stat.st_mtime value after adding the conversion constant described above.

5.1.1.2 dataSize

The FilingSubset defines the value of the **dataSize** attribute to be an estimate of the number of eight-bit bytes within the file content. The UNIX file system maintains a file size, in bytes, which can be used for the **dataSize** value.

[Retention]

Since the **dataSize** value is regarded as an estimate of the native storage size, a UNIX service does not need to explicitly save this value. It will be retained by the UNIX file system once the file is created.

[Retrieval]

The **dataSize** value can be returned by issuing a stat call on the desired file and returning the stat.st_size value.

5.1.1.3 isDirectory

The **isDirectory** is a boolean designating whether the file is a directory or not. Since UNIX differentiates between directory and non-directory files, this value is retained in the format of the file and derived from the stat file structure field, stat.st_mode.

[Retention]

Retention of the **isDirectory** attribute implies that the file be created differently based on the attribute value. When the value is **FALSE**, the standard UNIX file creation routines (open, creat, fopen, etc.) can be used. If the value is **TRUE**, the directory file can be created with the mkdir system call (4.2BSD and 4.3BSD) or the mkdir command (4.2BSD, 4.3BSD and System V)

[Retrieval]

The **isDirectory** attribute value can be determined by issuing a call to the stat routine. This returns a file status block which contains the field stat.st_mode. The **isDirectory** value will be TRUE if the returned stat.st_mode value is TRUE when logically anded with the constant S_IFDIR.

5.1.1.4 modifiedOn

The **modifiedOn** attribute is retained in the UNIX file status field stat.st_atime.

[Retention]

The **modifiedOn** attribute is retained in the stat.st_atime field by a call to utimes (4.2BSD and 4.3BSD) or utime (4.2BSD, 4.3BSD and System V). When a file is created by a FilingSubset client or service, the **modifiedOn** value becomes the current date and time. If no value is specified for the modified date on the utimes routine, the current date and time will be used.

[Retrieval]

The **modifiedOn** value is returned to network processes by issuing a stat call on the file and returning the stat.st_atime value added to the UNIX to XNS time conversion constant described in Section 5.1.1.

5.1.1.5 pathname

The FilingSubset requires all service implementations to allow the specification of files by the **pathname** attribute value. The syntax of the attribute value is defined to be service specific, which implies that the **pathname** value will in fact be the UNIX file name. Likewise, the **pathname** value can be easily derived from the UNIX file name when listing the parent directory.

The context for use of the **pathname** attribute within the FilingSubset restricts the use of wildcard characters to the **matches** attribute value on the List procedure.

[Retention]

The **pathname** attribute value specified will be used as the UNIX file name when actually creating the file. This value is retained by the parent directory file once the file is successfully created.

[Retrieval]

A FilingSubset service is allowed to require the **pathname** attribute when accessing a file. As such, the value is always specified by the client, except on a List when the service must enumerate the parent directory. The mechanism presented in Section 5.3.4 using the `ls` command will always return a fully specified UNIX filename to the service.

5.1.1.6 type

The ability to transfer files between systems and retain generic file types is advantageous to the users of a heterogeneous network. In particular, the ability to transfer a text file to another system and preserve the editability of that file by the native text editors on the receiving system without explicit conversion is especially beneficial.

All FilingSubset implementations must support the type attribute values: **tAsciiText**, **tDirectory** and **tUnspecified**. The UNIX operating system does not provide an explicit mechanism to distinguish between text and binary files, so support for this distinction must rely on the client or service making a good guess as to the file type based upon analysis of the file content.

Generally, the distinction can be made that files containing only Ascii characters will be treated as **tAsciiText** and all other files (excluding directories)' will be treated as **tUnspecified**.

[Retention]

The **tDirectory** file type is retained in a manner similar to the **isDirectory** attribute. When the attribute value is **tDirectory,** the directory is created via a the system call `mkdir` (4.2BSD and 4.3BSD) or the command `mkdir` (4.2BSD, 4.3BSD and System V).

Since the UNIX operating system does not create text and non-text files differently, the service does not explicitly retain the attribute value when storing the file. Instead, the distinction is made when the **type** attribute is retrieved.

[Retrieval]

The **tDirectory** file type can be determined in a manner similar to that of the **isDirectory** attribute. A call to `stat` will return a file status block which contains the field `stat.st_mode`. The type value will be set to **tDirectory** if the returned `stat.st_mode` value is TRUE when logically anded with the consant `S_IFDIR`.

Since the UNIX file system does not provide explicit file types to distinguish between **tAsciitext** and **tUnspecified**, this distinction must be made based on the file content. A simple, but effective method for determining the file type is to read a selected number of bytes from the file and look for any byte sequences which contain non-ASCII characters (i.e., any character is 0 or has the high-order bit set). If any non-ASCII characters are found, then the file can be assumed to be **tUnspecified**; if only ASCII characters are found, then the **tAsciiText** type can be assumed. It should be noted that this method will not discern the correct type in all cases; however, it is possible for the client to override the service determined value by specifying the desired **type** on the **Open** call.

The routine `get_type` is defined to return the file type.

```
#include   <stdio.h>

#define    CHARS_TO_READ  2048

/*
    routine:
        get_type
    input:
        pointer to pathname of file

    returns:
        Cardinal containing Filing defined file type
*/

Cardinal get_type(pathname)
char        *pathname;
{
    FILE        *file_desc;
    char        buffer[CHARS_TO_READ];
    int         count;
    char        *ptr;
    Cardinal    type;

    if ( (file_desc= fopen(pathname,"r")) ) {
        type= Filing_tUnspecified;                      /* if error, assume tUnspecified */
        return(type);
    }

    if ( (count= fread(buffer,sizeof(char),CHARS_TO_READ,file_desc)) != 0 )
        type= Filing_tUnspecified;                      /* if error, assume tUnspecified */
    else {
        type= Filing_tAsciiText;
                                                                /* assume tAsciiText */
        for ( ptr= buffer; ptr < buffer + count - 1; )  {  /* for each character */
            if ( (*ptr == 0 ) | (*ptr++ & 0200) ) {        /* if 0 or high order bit */
                type= Filing_tUnspecified;                 /* assume tUnspecified */
                break;
            }
        }
    }

    fclose(file_desc);                                     /* close file */
    return(type);
}
```

5.1.2 Implied attributes

Implied attributes are those attributes which obtain an implicit value when a new file is created. All subset implementations are required to permit the specification of the implied (default) value for these attributes. A service implementation may reject a **Store** procedure if the value for an implied attribute is not the default value and the service does not support the retention of non-default values for the attribute.

The implied attributes defined in the FilingSubset are **accessList**, **childrenUniquelyNamed**, **defaultAccessList**, **isTemporary**, **ordering**, **subtreeSizeLimit** and **version**.

Table 5.1 specifies the default values for these attributes on the UNIX operating system. Since the attribute values are identical for every file unless otherwise supported, no explicit provision for retention and retrieval of these attributes is needed. The service should verify that the associated value is indeed the default on a **Store** and return the default values when requested on a **List** procedure.

Attribute	Supported Values
accessList	**[defaulted:** TRUE**]**
childrenUniquelyNamed	TRUE
defaultAccessList	**[defaulted:** TRUE**]**
isTemporary	FALSE
ordering	**defaultOrdering**
subtreeSizeLimit	**nullSubtreeSizeLimit**
version	**highestVersion**

Table 5.1 UNIX supported values for implied attributes

5.1.3 Optional attributes

Those attributes which are defined as interpreted in the Filing Protocol but are not defined as either mandatory or implied within the FilingSubset are classified as optional attributes. These attributes are not required to be supported by any FilingSubset service. Conventions for retaining and retrieving values for these attributes are not discussed here, since they are outside the definition for required functionality in the FilingSubset.

5.2 Client procedure support

Client routines require various UNIX system calls to perform functions specific to the UNIX operating system and to access the UNIX file system. Examples of this interaction are discussed in this section.

5.2.1 Continuance timer support

A FilingSubset client must issue a **Continue** procedure at specific time intervals to prevent the service from terminating the session for lack of activity. This mechanism is implemented via use of the `alarm` and `signal` UNIX routines. Three routines are defined for use by the client: `set_continuance_timer`, `reset_continuance_timer` and `cancel_continuance_timer`. In addition, the routine `send_continue` is referenced. This routine will send a **Continue** to the service to maintain the open session.

`set_continuance_timer` calls `send_continue` to determine the service continuance value then initializes the timer mechanism to send a SIGALRM signal before the expiration of that interval.

```
#include    <signal.h>

extern      send_continue();        /* expiration routine, will send continue */

Cardinal    continuance;                      /* continuance value, in seconds */
                                              /* returned from service */

/*
    routine:
        set_continuance_timer

    called after a successful Logon
*/

set_continuance_timer()
{
    continuance= send_continue();             /* get service value */
    continuance= continuance/3;               /* insure we expire before service */

    alarm(0);                                 /* cancel any previous alarm */
    signal(SIGALRM,send_continue);            /* set routine to catch alarm */
    alarm(continuance);                       /* set alarm */
}
```

reset_continuance_timer cancels any pending timer and reissues a new timer request.

```
/*
    routine:
        reset_continuance_timer

    called after any FilingSubset procedure call
*/

reset_continuance_timer()
{
    alarm(0);                                   /* cancel previous alarm */
    alarm(continuance);                         /* then, reset alarm */
}
```

cancel_continuance_timer cancels the previous request and turns off handling of the SIGALRM signal.

```
/*
    routine:
        cancel_continuance_timer

    called after a successful Logoff
*/

cancel_continuance_timer()
{
    alarm(0);                                   /* cancel any previous alarm */
    signal(SIGALRM,SIG_IGN);                    /* set routine to ignore alarm */
}
```

5.2.2 Determining mandatory attribute values

When a client performs a **Store**, values for the mandatory attributes may accompany the remote procedure call Each of these values, with the exception of **pathname** and **type**, can be obtained locally by using the stat system call. The routine get_attributes illustrates how to accomplish this.

```
#include    <sys/types.h>
#include    <sys/stat.h>

extern  LongCardinal    createdon;
extern  LongCardinal    modifiedon;
extern  Boolean         isdirectory;
extern  Cardinal        datasize;
extern  Cardinal        type;
```

```
/*
    routine:
        get_attributes
    input:
        pointer to pathname of file
    returns:
        -1    success
         1    error
*/

get_attributes(pathname)
char        *pathname;                              /* file name */
{
    struct stat    file_stat;

    if ( stat(pathname,&file_stat) == -1 )                       /* stat file */
        return(1);

    createdon= file_stat.st_mtime + XNS_TIME_DIFFERENCE;         /* createdOn */
    modifiedon= file_stat.st_atime + XNS_TIME_DIFFERENCE;        /* modifiedOn */

    datasize= file_stat.st_size;                                 /* dataSize */

                                                    /* type and isDirectory */
    if ( (file_stat.st_mode & S_IFDIR) != 0 ) {
        isdirectory= TRUE;
        type= Filing_tDirectory;
    } else {
        isdirectory= FALSE;
        type= get_type(pathname);
    }

return(-1);
}
```

5.3 Service procedure support

A FilingSubset service implemented on the UNIX operating system will need to use various system calls to access the local file system and provide UNIX specific procedure support. This section presents detailed examples of this interaction.

Client access to files on a subset service is controlled through the use of a file handle. The implementation presented in Section 4 describes the value of the file handle as a pointer to a *file context block*. To provide the necessary functionality, this context block will contain some items which are operating system specific.

For the implementation presented here, the following items are contained in the file context block:

- a copy of the **pathname** attribute value as specified on the **Open** or **Store**
- a cardinal identifying the file type requested by the client on the **Open**
- a cardinal specifying the file type as determined by the service
- a cardinal specifying the **dataSize** value for the file
- a boolean specifying the **isDirectory** value for the file
- a long cardinal specifying the **createdOn** value for the file in XNS format
- a long cardinal specifying the **modifiedOn** value for the file in XNS format
- a FILE file descriptor used to access the opened file

The following C structure defines the structure used in this section:

```
typedef file_handle {
    char           *pathname;      /* pointer to pathname value */
    Cardinal       type;           /* client requested type (from Open) */
    Cardinal       truetype;       /* file system file type */
    Cardinal       datasize;       /* dataSize value */
    Boolean        isdirectory;    /* isDirectory */
    LongCardinal   createdon;      /* createdOn  value */
    LongCardinal   modifiedon;     /* modifiedOn value */
    FILE           *file_desc;     /* ptr to file descriptor for open file */
};
```

5.3.1 Logon

The **Logon** procedure is responsible for validating the user attempting the connection and, if successful, altering the process ownership to that of the user. This alteration of ownership ensures that the process is subject to the normal access/protection mechanisms employed by the UNIX operating system when subsequent procedure calls request access to files on the service.

The user name and password entries of the secondary credentials supplied on the **Logon** are validated against the standard UNIX account file (/etc passwd). Once this has been completed, the user ID and group ID of the process is changed to that of the respective user as determined from the password file entry for the user. The process is also positioned to the appropriate root file for the service, generally the UNIX root "/". This provides a working directory which can be associated with **nullHandle**.

The verifyandposition_user routine is defined to perform these functions.

```
#include    <pwd.h>

#define     SERVICE_ROOT    "/"

/*
    routine:
        verifyand position_user
    input:
        user name        derived from secondary credentials
        user password    derived from secondary credentials
    returns:
        -1  - success
        else Filing Error, Problem
*/

Filing_Error   verifyandposition_user(user_name, user_password)
char    *user_name;         /* user name derived from secondary credentials */
char    *user_password;     /* user password derived from secondary credentials */
{
    struct passwd      *pwd_entry;
    struct passwd      *getpwnam();
    char               *crypt;
    Filing_Error       error_value;           /* Filing error, problem pair */

                                           /* set to Filing AuthenticationError */
    error_value.error= Filing_AuthenticationError;
    error_value.problem= Authentication_secondaryCredentialsInvalid;

                                              /* determine if user is valid */
    if ( (pwd_entry= getpwnam(user_name)) == (struct passwd *)0 )
        return(error_value);
                                              /* determine if password is valid */
    if ( strcmp(pwd_entry->pw_passwd,crypt(user_password,pwd_entry->pw_passwd)) )
        return(error_value);

                                              /* set process user ID */
    if ( setuid(pwd_entry->pw_uid) == -1 )
        return(error_value);

                                               /* set process group ID */
    if ( setgid(pwd_entry->pw_gid) == -1 )
        return(error_value);

                                               /* position in service root */
    if ( chdir(SERVICE_ROOT) == -1 ) {
        error_value.error= Filing_ServiceError;
        error_value.problem= Filing_serviceUnavailable;
        return(error_value);
    }

    return(-1)
}
```

5.3.2 Continue

The continuance mechanism is defined to allow services to close a session if it has been idle for a long period of time or the session needs to be terminated for other reasons. Each service maintains a continuance value which is the number of seconds that it will keep a session open between successive procedure calls. This allows the service to set a timeout mechanism to notify it when this time interval has passed and allow it to disconnect the active session.

This mechanism is armed once a session has been successfully established by a **Logon** and is terminated once the session is ended with a **Logoff**. Additionally, each routine which processes a FilingSubset procedure, as described in Section 4, should rearm the timer.

The `alarm` and `signal` routines are used to implement this mechanism for UNIX services. `alarm` is used to set the timer mechanism for the specified interval while `signal` is used to indicate whether the service is to handle or ignore the alarm.

The routines `set_continuance_timer`, `reset_continuance_timer` and `cancel_continuance_timer` are defined. The service routine `continuance_expiration` is referenced by `set_continuance_timer` and would execute at the expiration of a timeout interval. At that time, this routine would close the current session in a manner similar to that proposed for the **Logoff** procedure in Section 4.4.

`set_continuance_timer` initially establishes the timeout mechanism

```
#include    <signal.h>

Cardinal    continuance;                        /* continuance value, in seconds */
extern      continuance_expiration();           /* expiration routine */

/*
    routine:
        set_continuance_timer
*/

set_continuance_timer()
{
    alarm(0);                                   /* cancel any previous alarm */
    signal(SIGALRM.continuance_expiration);     /* set routine to catch alarm */
    alarm(continuance);                         /* set alarm */
}
```

The `reset_continuance_timer` and `cancel_continuance_timer` routines are identical to the client routines specified in Section 5.2.1.

5.3.3 Open

The **Open** procedure opens a file for subsequent access by the client. The file is identified by the value specified for the **pathname** attribute. UNIX does not support multiple versions, so the **version** values **lowestVersion** and **highestVersion** are accepted but indicate the same file.

With respect to the UNIX file system, there is no guarantee that the file cannot be deleted by other utilities running outside of the process that has the file open. Since there is no benefit to physically opening the file during processing of an **Open**, the **Open** routine will simply determine if the file exists and the user has permission to access the file. Subsequent procedures which require physical access to the file will be responsible for actually performing the open.

The `stat_file` routine is defined to accomplish this. The UNIX routine `stat` is used to fill in the attribute entries within the file context block. In addition, a call to `get_type` is issued to determine the file type as stored on the UNIX file system. This allows subsequent file transfer procedures to determine values for the mandatory attributes **dataSize**, **isDirectory** and **type** simply by examining the file context block. The possible error returns are: **accessRightsInsufficient** if the file cannot be accessed, **fileNotFound**, if the file or some component of the pathname does not exist and **accessRightsIndeterminate** if any other error occurs.

```
#include    <errno.h>
#include    <sys/types.h>
#include    <sys/stat.h>

/*
    routine:
        stat_file
    input:
        pointer to file handle
    returns:
        -1    success
        else Filing Error, Problem

        file_context_block entries filled in
*/

Filing_Error    stat_file(file_context_block)
file_handle     *file_context_block;
{
    struct stat    file_stat;
    Filing_Error   error_value;                    /* Filing error, problem pair */

    error_value.error= Filing_AccessError;         /* default to AccessError */

    if ( stat(file_context_block->pathname,&file_stat) == -1 ) {
        switch (errno) {
            case EACCES:                           /* user has no access */
                error_value.problem= Filing_accessRightsInsufficient;
                return(error_value);

            case ENOTDIR:                          /* directory doesn't exist */
            case ENOENT:                           /* file doesn't exist */
                error_value.problem= Filing_fileNotFound;
                return(error_value);
            default:                                   /* all other errors */
                error_value.problem= Filing_accessRightsIndeterminate;
                return(error_value);
        }
    }

    file_context_block->datasize= file_stat.st_size;              /* dataSize */
                                                                  /* file type */

    if ( (file_stat.st_mode & S_IFDIR) != 0 ) {         /* type and isDirectory */
        file_context_block->isdirectory= TRUE;
        truetype= Filing_tDirectory;
    } else {
        file_context_block->isdirectory= FALSE;
        file_context_block->truetype= get_type(file_context_block->pathname);
    }

return(-1);
}
```

5.3.4 List

The **List** procedure enumerates a directory looking for the specified file or files and returns the requested attributes for each file found. The file specification to be listed is specified in the **pathname** attribute value on a **filter** of type **matches** This procedure is unique in that it is the only procedure which will allow wildcard characters in the pathname syntax which is interpreted by the service.

This function is easily accomodated through the use of the UNIX ls command which lists a directory and returns the files matching some file name criteria The service uses the popen routine to execute the ls command and read the subsequent output. Use of the -ld switches result in the output being formatted one file name per line with the file name being fully specified from the UNIX root ("/"). The file names are also returned in ascending order by name which is the **defaultOrdering** value for the UNIX implementation Each file name returned can then be used to determine the attribute values as requested on the **List** If for any reason the popen routine is not successful, the **AccessProblem accessRightsInsufficient** is returned.

The routine list_directory is defined to perform this function. [Note: A slightly altered version of the get_attributes routine presented in Section 5 2.2 can be used to determine the mandatory attributes for a file.]

```
#include    <stdio.h>
#include    <errno.h>

/*
    routine:
        list_directory
    input:
        pointer to UNIX file specification
        returns:
            -1    success
            else Filing Error, Problem
*/

Filing_Error list_directory(file_spec)
char        *file_spec;        /* pathname attribute from filter of type matches */
{
        Filing_AttributeSequence      attribute_sequence;
        FILE            *pipe_desc;
        FILE            *popen();
        char            command[256];
        Filing_Error    error_value;                    /* Filing error, problem pair */

        error_value.error= Filing_AccessError;          /* default to AccessError */

        strcpy(command,"/bin/ls -ld ");                 /* form appropriate command */
        strcat(command,file_spec);

        if ( (pipe_desc= popen(command)) == NULL ) {       /* issue command */
            error_value.problem= Filing_accessRightsInsufficient;
            return(error_value);
        }
                                                            /* read each file name */
        while ( fgets(filename,MAX_FILENAME_LENGTH,pipe_desc) != NULL ) {
/*
            insert implementation specific routines here:
                determine the values for the requested attributes
                make an attribute sequence
                write the attribute sequence to the bulk data stream
*/
        }

        pclose(pipe_desc);
        return(-1);
}
```

5.3.5 Store

The **Store** procedure is used to create both directory and non-directory files. A different system call is used to create directory files under UNIX, so the service will take an appropriate action based on the values of the **isDirectory** and **type** attribute values as stored in the file context block.

Non-directory files are stored by creating the specified file, reading the bulk data stream, writing to the file and closing the file. The **createdOn** and **modifiedOn** attribute values are retained once the file is closed as described in Section 5.1.

After the **Store** routine has validated the argument and attribute values, a file handle is allocated. The `create_file` routine is then called to physically create the file. Appropriate values for **AccessProblem** are returned if the file cannot be created for any reason. The service does not allow overwriting an existing file and returns an error if the file exists.

```
#include   <stdio.h>
#include   <errno.h>

/*
    routine:
        create_file
    input:
        pointer to file handle
    returns:
        -1   success
        else Filing Error, Problem

        file_context_block->file_desc filled in
*/

Filing_Error create_file(file_context_block)
file_handle    *file_context_block;
{

    FILE            *fopen();
    Filing_Error    error_value;                 /* Filing error, problem pair */

                                                 /* open file for write */
    if ( file_context_block->file_desc=
                        fopen(file_context_block->pathname,"w") ) {
        switch (errno) {
            case EACCES:                         /* user has no access */
                error_value.error= Filing_AccessError;
                error_value.problem= Filing_accessrightsInsufficient;
                return(rror_value);

            case EEXIST:                         /* file exists */
                error_value.error= Filing_InsertionError;
                error_value.problem= Filing_fileNotUnique;
                return(error_value);

            case ENOENT:                         /* no such file, OK */
                break;

            case ENOTDIR:                        /* no such directory */
                error_value.error= Filing_AccessError;
                error_value.problem= Filing_fileNotFound;
                return(error_value);

            case EMFILE:                         /* process file table full */
            case ENFILE:                         /* system file table full */
                error_value.error= Filing_SpaceError;
                error_value.problem= Filing_allocationExceeded;
                return(error_value);

            default:                             /* any other error */
                error_value.problem= Filing_accessRightsIndeterminate;
                return(error_value);
        }
        }

        return(-1);
    }
```

Once the file has been successfully created, the **Store** routine will save the attribute specified on the procedure call in the file context block. Default values will be assigned for all mandatory attributes not specified. The bulk data stream will be read and written to the file. If the file transfer is **tAsciiText**, then the appropriate decoding of **AsciiString** must be performed to allow the UNIX line delimiters (the linefeed character, octal 012) to be added to the file. This can be accomplished by writing the contents of **AsciiString.bytes** to the file followed by a call to `fputc` as follows:

```
    {
        int count:


        ....


                                        /* character count is sequence length * 2 */
        count= sequence_length(AsciiString.bytes)/2:
        if ( !AsciiString.lastByteSignificant )              /* if count is odd. */
            count--:                                          /* decrement by 1 */

                                                              /* write characters */
        fwrite(AsciiString.bytes.sizeof(char).count.file_context_block->file_desc):
        fputc('\n'.file_context_block->file_desc):            /* then line feed */


        ....
    }
```

FilingSubset services are not required to support directory creation. If directory creation is supported, the service may optionally restrict this to only allow the creation of empty directories. Directory files can be created easily on UNIX with the `mkdir` command: however, the format of directory files is operating system dependent and therefore does not encourage the transfer of directory file contents. The `create_directory` routine is provided to illustrate the creation of empty directory files.

```
    /*
        routine
            create_directory
        input:
            pointer to file handle
        returns:
            -1    success
            else Filing Error. Problem
    */


    Filing_Error   create_directory(file_context_block)
    file_handle    *file_context_block:
    {
        int            status;
        Filing_Error   error_value:                  /* Filing error. problem pair */

        error_value.error= Filing_AccessError:       /* default to AccessError */
        status= 0;
        if ( fork() == 0 ) {                          /* execute command */
               execl("/bin/mkdir","mkdir",file_context_block->pathname,0);
               exit(-1);
        }

        wait(&status);
        if ( status ) {                   /* error reports accessRightsInsufficient */
            error_value.problem= Filing_accessRightsInsufficient;
            return(error_value);
        }

        return(-1);
}
```

5.3.6 Retrieve

The **Retrieve** procedure transfers a file from a service to the calling client. FilingSubset services are not required to allow the retrieval of directory files. This is true of the implementation presented here, however, the fact that a file is a directory or not is determined at a higher level. If the file is not a directory, then the file is opened and the content transferred via a bulk data stream to the client.

The open_file routine physically opens the file for reading via the fopen subroutine. Any errors encountered during this are returned as type **AccessProblem**

```
#include    <stdio.h>
#include    <errno.h>

/*
    routine
        open_file
    input:
        pointer to file handle
    returns:
        -1    success
        else Filing Error, Problem

        file_context_block->file_desc filled in
*/

Filing_Error open_file(file_context_block)
file_handle     *file_context_block;
{

    FILE            *fopen();
    Filing_Error    error_value;                  /* Filing error problem pair */

    error_value= Filing_AccessError;              /* default to AccessError */
```

```
                                                    /* open file */
        if ( file_context_block->file_desc=
                                fopen(file_context_block->pathname."r") ) {

            switch (errno) {
                case EACCES:                        /* user has no access */
                    error_value.problem= Filing_accessrightsInsufficient:
                    return(error_value);

                case ENOENT:                        /* no such file */
                case ENOTDIR:                       /* no such directory */
                    error_value.problem= Filing_fileNotFound:
                    return(error_value);

                default:                            /* all other errors */
                    error_value.problem= Filing_accessRightsIndeterminate;
                    return(error_value);
            }
        }

        return(-1):
    }
```

The content of the file is then read and written to the bulk data stream. Files of type **tAsciiText** are transferred as a **StreamofAsciiText**. The content of the file as read from the file must be encoded into this form for transmission to the client. This involves removing the UNIX line delimiter (the linefeed character, octal 012) before representing the data as an **AsciiString**.

5.3.7 Delete

The **Delete** procedure is used by clients to delete files. If the specified file is a directory, a FilingSubset service is not required to support the deletion of that file and optionally all descendants of the file. The UNIX `rm` command provides a relatively simple mechanism for providing this facility. The `delete_file` routine decides on the required processing based upon whether the file is a directory or not. Directory files are deleted by specifying the `-r` switch on `rm`. The `-f` switch is also used to force the deletion, if necessary. Errors encountered during the deletion of the directory and its descendants are returned as **AccessProblem accessRightsInsufficient**.

Non-directory files are deleted with the UNIX `unlink` routine. Appropriate errors are reported as type **AccessProblem**.

```
    #include    <errno.h>

    /*
        routine:
            delete_file
        input:
            pointer to file handle
        returns:
            -1    success
            else Filing Error. Problem
    */
```

```
Filing_Error   delete_file(file_handle)
file_handle    *file_context_block;
{
    int status;
    Filing_Error   error_value;                      /* Filing error,*problem pair */

    error_value.error= Filing_AccessError;           /* default to AccessError */

    if ( file_context_block->isdirectory ) {
        if ( fork() == 0 ) {                         /* use rm -rf for directories */
            execl("/bin/rm","rm","-rf",file_context_block->pathname,0);
            exit(-1);
        }

        wait(&status);
        if ( status ) {                /* error reports accessRightsInsufficient */
            error_value.problem= Filing_accessRightsInsufficient;
            return(error_value);
        }
    } else {                                     /* use unlink for non-directories */
        if ( unlink(file_context_block->pathname) == -1 ) {
            switch (errno) {
                case EACCES:                         /* user has no access */
                    error_value.problem= Filing_accessRightsInsufficient;
                    return(error_value);

                case ENOENT:                         /* no such file */
                case ENOTDIR:                        /* no such directory */
                    error_value.problem= Filing_fileNotFound;
                    return(error_value);

                default:                             /* all other errors */
                    error_value.problem= Filing_accessRightsIndeterminate;
                    return(error_value);
            }
        }
    }

    return(-1);
}
```

6. VMS SYSTEM INTERFACE

Implementation of the FilingSubset under VMS requires both procedure and attribute support within the native operating and file systems. This section presents an implementation scenario which describes the necessary interactions with the VMS system.

This section describes those interface procedures required by the client and service implementations presented in Sections 3 and 4. These are by no means the only method for providing the facility desired; they have been chosen because they are consistent with the UNIX routines of the previous section, except in those instances where appropriate VAX C routines do not provide the necessary funtionality. Several procedures are alluded to but cannot be provided, due to the lack of support for certain features from the VAX C run-time library and the proprietary nature of the VMS operating system. In these cases, it is assumed that the appropriate functions can be provided through the use of internal VMS functions.

In several instances, the examples presented will be identical to the UNIX counterparts presented in Section 5. This replication is done in an effort to make both the UNIX and the VMS sections complete stand-alone sections.

Several of the examples presented are predicated on the assumption that a single VMS process instance handles all procedure calls from the time the Courier connection has been established on the initial **Logon** call until the subsequent **Logoff** call. The examples also assume the definition of Filing defined constants and Courier defined data types. In the examples, the string "Filing_" is prepended to structure and variable names which are defined by the Filing Protocol.

6.1 Attribute Support

The FilingSubset Protocol distinguishes three classes of attributes: mandatory, implied and optional. This section describes specific scenarios under the VMS operating system for

- services to retain attributes so that they may be interpreted by other native operating system utilities and returned when requested by network clients
- clients to retrieve and retain the attributes when dealing with remote services

All attributes presented here are discussed with respect to two areas: 1) where attributes must be retained in the native file structures and 2) how they may be retrieved from these structures and transferred to other FilingSubset clients and services. Retention of attributes is of importance to FilingSubset clients when retrieving files from a service and to services when a client requests creation of a file on the service. Likewise, retrieval of attributes from the native file structures is used by clients when issuing a **Store** and by services when returning attributes on a **List** procedure.

6.1.1 Mandatory Attributes

Mandatory attributes are those attributes which must be interpreted by all FilingSubset implementations. These attributes are guaranteed to be retained by any service implementing the FilingSubset Protocol and must be accepted in specific procedure calls to the extent that they are legal arguments of the corresponding procedure in the Filing Protocol. Additionally, clients may wish to retain these attributes when retrieving files from a service. The FilingSubset defines the following mandatory attributes: **createdOn**, **dataSize**, **isDirectory**, **modifiedOn**, **pathname** and **type**.

Each of these attributes is discussed with respect to the areas of retention and retrieval. Retention of an attribute value describes a mechanism for saving the specified XNS attribute value within the VMS file system along with the file contents. Retrieval of attribute values presents methods for deriving the XNS value from the VMS file system. In each of these cases, the values may need to be converted from one form to the other.

In the case of the **createdOn** and **modifiedOn** attributes, the retention and retrieval of attribute values requires a conversion between the VMS and XNS formats. The **createdOn** and **modifiedOn** values are always specified in XNS Time format [10]. XNS time is based on the number of seconds since 00:00:00 Jan. 1, 1901 Greewich Mean Time. The VMS operating system maintains time in a 64 bit quadword specifying 100 nano-second intervals from 00:00:00, Nov. 17, 1858 in local time. VMS has no knowledge of offset from Greenwich Mean Time nor daylight saving time (DST) adjustments. Therefore, the conversion mechanism must adjust according to the local values for these offsets.

For a given machine, the difference between the Jan. 1968 and Nov 1858 base values, and the local difference from GMT are constants. Thus the combined offset can be calculated at process initialization. The `set_base_time` routine converts the earliest representable XNS time (00:00:00 Jan. 1, 1968) to VMS format and adjusts it by the appropriate GMT offset, as expressed in VMS format.

```
#include   rms
#include   ssdef
#include   descrip

double xns_base_time:          /* VMS value for XNS earliest time */

/*
    routine:
        set_base_time
    input
        gmt_difference       local GMT offset in VMS ASCII time format
                               maximum offset is ± 12 hours from GMT
                               (i.e.. for EST. "0 5:0:0.0")
        east_of_gmt          Boolean representing east/west of GMT
                               (TRUE   east. FALSE   west)
    returns
        xns_base_time set to appropriate vms time value
        -1    if unsuccessful
*/

set_base_time(gmt_difference,east_of_gmt)
struct dsc$descriptor     *gmt_difference:               /* local GMT offset */
Boolean                   east_of_gmt:                   /* direction */
```

```
{
    double time, gmt_offset;

    static $DESCRIPTOR(XNS_EARLIEST_TIME,"01-JAN-1968 0:0:0.0");

                        /* convert earliest representable XNS time to VMS format */
    if ((error= sys$bintim(&XNS_EARLIEST_TIME,&time)) != SS$_NORMAL)
        return(-1);

                        /* convert GMT offset to VMS format */
    if ((error= sys$bintim(gmt_difference,&gmt_offset)) != SS$_NORMAL)
        return(-1);

    if  (east_of_gmt) {         /* if east, subtract offset; if west, add offset */
        if ((error= lib$subx(&time,&gmt_offset,&xns_base_time)) != SS$_NORMAL)
            return(-1);
    } else {
        if ((error= lib$addx(&time,&gmt_offset,&xns_base_time)) != SS$_NORMAL)
            return(-1);
    }
}
```

The conversion from XNS time to VMS time is then accomplished by subtracting the XNS representation for earliestTime (2114294400) from the XNS time and adjusting for any daylight savings time offset. The resulting value will be the number of seconds from 00:00:00 Jan. 1, 1968. This value is multiplied by 10 million, to convert to 100 nano-second intervals and the previously computed VMS constant for the XNS earliest time added to create an VMS value.

The routine convert_xns_time illustrates this:

```
#include   ssdef

#define    XNS_EARLIEST_TIME  2114294400

double xns_base_time;          /* VMS value for XNS earliest time */

/*
    routine:
        convert_xns_time
    input:
        xns_time    - XNS time value
        IS_DST        function which will indicate whether daylight savings time is
                        in effect on local machine (TRUE   dst in effect)
    returns
        corresponding VMS time value (64 bit quadword)
        -1    if error occurs
*/

double convert_xns_time(xns_time)
LongCardinal   xns_time;
{

    double     vms_time;
```

```
        long        ten_million= 10000000;
        long        addend= 0;

                                            /* get difference from XNS earliest time */
        xns_time   xns_time   XNS_EARLIEST_TIME;

        if (IS_DST)                         /* adjust for daylight savings time */
            xns_time= xns_time   3600;

                                            /* convert to 100 nano-second intervals */
        if ((error= lib$emul(&xns_time,&ten_million,&addend,&vms_time)) != SS$_NORMAL)
            return(-1);

                                /* make relative to Nov. 17, 1858 local standard time */
        if ((error= lib$addx(&vms_time,xns_base_time,&vms_time)) != SS$_NORMAL)
            return(-1);

        return(vms_time);
    }
```

Retrieval of the **createdOn** or **modifiedOn** attributes involves using the reverse of the above conversion. The routine `convert_vms_time` illustrates the conversion from VMS to XNS format.

```
    #include   ssdef

    #define XNS_EARLIEST_TIME     2114294400

    double     xns_base_time;          /* VMS value for XNS earliest time */

    /*
        routine:
            convert_vms_time
        input:
            vms_value       pointer to 64 bit quadword containing VMS value
            xns_value       pointer to LongCardinal to receive XNS value
            IS_FILE_DST     function which will indicate whether daylight savings
                              time is in effect for the specified vms_time
                              (TRUE   dst in effect)
        returns:
            xns_value       XNS time value
            -1  - if error occurs
    */

    convert_vms_time(vms_value,xns_value)
    double          *vms_value;
    LongCardinal    *xns_value;
    {

        double      date;
        long        ten_million= 10000000;
        long        remainder;
        int         error;
```

```
                                    /* get difference from earliest time */
    if ((error= lib$subx(vms_value,&xns_base_time,&date)) != SS$_NORMAL)
        return(-1);

                                    /* convert to seconds (divide by ten million) */
    if ((error= lib$ediv(&ten_million,&date,xns_value,
                                            &remainder)) != SS$_NORMAL)
        return(-1);
                                    /* relative to earliest XNS time */
    xns_value= xns_value + XNS_EARLIEST_TIME;

    if (IS_FILE_DST)          /* adjust for local DST offset, when in effect */
        xns_value = xns_value + 3600;         /* add 1 hour (60 min * 60 sec) */

}
```

6.1.1.1 createdOn

The **createdOn** attribute is useful in determining if similarly named files on different file systems within the network are identical. The ability to retain the **createdOn** date must be coupled with a mechanism for native utilities to provide this date on demand. This can be accomplished on VMS by setting the XAB$Q_CDT field of the XABDAT file structure to the **createdOn** value prior to creating the file. This allows non-network VMS users to access this date easily and also allows the network client and service to determine and modify this date.

[Retention]

The **createdOn** value is converted from XNS format to VMS format using the convert_xns_time routine described in Section 6.1.1. The VMS value can then be retained by placing the value in the XAB$Q_CDT field before creating the file, as illustrated below:

```
#include    rms
#include    ssdef

struct xab_date_format {                /* xab defined format for date/time */
    unsigned: 32;                       /* alleviates compiler typing problems */
    unsigned: 32;
};

/*
    routine:
        set_create_time
    input:
        pointer to file context block
            where
                if no createdOn value was specified on Store, createdon == 0
                if createdOn value was specified on Store, createdon != 0, value is
                in XNS time format
    returns:
        sets xab$q_cdt if appropriate
*/

set_create_time(file_context_block)
file_handle     *file_context_block;
```

```
{
        union {                              /* this alleviates compiler typing problems */
            struct xab_date_format      date_format;
            double                      date_double;
        } vms_time;

        if (file_context_block->createdon) {        /* save createdOn if specified */
            if ((vms_time.date_double=
                    convert_xns_time(file_context_block->createdon)) != -1)
                file_context_block->xab.xab$q_cdt= vms_time.date_format;
        }
}
```

[Retrieval]

Network processes can retrieve the **createdOn** value by requesting the file creation date (ATR$C_CREDATE) when performing an IO$_ACCESS QIO to the disk ACP and converting from VMS to XNS format using the convert_vms_time routine from Section 6.1.1.

The VAX C stat routine can also be used to determine the value for the **createdOn** attribute. In this case, the returned value stat.st_ctime is converted to XNS time in a manner similar to the UNIX mechanism. The value returned from stat is specified as seconds since 00:00:00 GMT, Jan. 1, 1970. To convert to XNS format, the constant 2177452800 must be added to this value. Note that this constant is the XNS encoding for the time 00:00:00 GMT, Jan. 1, 1970 [((1970-1901) years * 365 days/year + 17 leap days) * 24 hours/day * 60 minutes/hour * 60 seconds/minute].

6.1.1.2 dataSize

The FilingSubset defines the value of the **dataSize** attribute to be an estimate of the number of eight-bit bytes within the file content. The VMS file system maintains a file size, in bytes, which can be used for the **dataSize** value. Since the VMS value accounts for appropriate formatting overhead, this value may not be equivalent to the actual file content size

[Retention]

Since the **dataSize** value is regarded as an estimate of the native storage size, a VMS service does not need to explicitly save this value. An appropriate value will be retained by the VMS file system once the file is created.

[Retrieval]

The **dataSize** value can be determined in one of two ways: issuing an IO$_ACCESS QIO to the Files-11 ACP requesting the file attributes (ATR$C_RECATTR) or invoking the VAX C stat routine. The **dataSize** value can be computed by combining the FAT$L_EFBLK and FAT$W_FFBYTE values returned from the QIO as follows:

```
#include fatdef

/*
    routine:
        compute_datasize
    input:
        file_attributes          FAT structure containing returned file attributes
    returns:
        XNS dataSize value as a LongCardinal
*/

LongCardinal compute_datasize(file_attributes)
struct fat          vms_value;
{

    int block_count;

                  /* need to swap 16-bit words */
    block_count= (file_attributes.fat$l_efblk << 16) |
                   (file_attributes.fat$w_efblk >> 16);

                  /* dataSize is (block_count-1) * 512 + #bytes used in last block */
    return (((block_count-1)*512)+file_attributes.fat$w_ffbyte);
}
```

The `stat.st_size` value returned from `stat` will yield the **dataSize** value directly

6.1.1.3 isDirectory

The **isDirectory** is a boolean designating whether the file is a directory or not Since VMS differentiates between directory and non-directory files, this value is retained in the format of the file and retrieved in one of several ways.

[Retention]

Retention of the **isDirectory** attribute implies that the file be created differently based on the attribute value. When the value is **FALSE**, the standard RMS file creation routines (`sys$create` or `sys$open`) or VAX C file creation routines (`open`, `creat`, `fopen`, etc.) can be used. If the value is **TRUE**, the directory file can be created with the VAX C `mkdir` or `LIB$CREATE_DIR` routine

[Retrieval]

The **isDirectory** attribute value can be determined by issuing an `IO$_ACCESS` QIO to the Files-11 ACP requesting the file characteristics of the file or through use of the VAX C `stat` routine The **isDirectory** value will be **TRUE** if the value from the QIO is **TRUE** when logically anded with the constant `FCH$M_DIRECTORY`. Likewise, if the `stat.st_mode` value returned from `stat` is **TRUE** when logically anded with `S_IFDIR`, the **isDirectory** value will be **TRUE**.

6.1.1.4 modifiedOn

The **modifiedOn** attribute is retained in the XAB$Q_RDT field of the XAB file structure.

[Retention]

The **modifiedOn** attribute is set to the current date and time when a file is created by a FilingSubset client or service. VMS will set the XAB$Q_RDT field to the current date and time when a file is created unless otherwise specified.

[Retrieval]

The **modifiedOn** value is returned to network processes by requesting the revision date (ATR$C_REVDATE) on an IO$_ACCESS QIO to the Files-11 ACP. The returned value can then be converted to XNS time as described in Section 6.1.1. The VAX C stat routine cannot be used to determine a value for the **modifiedOn** attribute since it does not return the XAB$Q_RDT value.

6.1.1.5 pathname

The FilingSubset requires all service implementations to allow the specification of files by the **pathname** attribute value. The syntax of the attribute value is defined to be service specific, which implies that the **pathname** value will in fact be the VMS file name. Likewise, the **pathname** value can be easily derived from the VMS file name when listing the parent directory.

The context for use of the **pathname** attribute within the FilingSubset restricts the use of wildcard characters to the **matches** attribute value on the **List** procedure.

[Retention]

The **pathname** attribute value specified on a **Store** will be used as the VMS file name when actually creating the file. This value is retained in the VMS file system once the file is successfully created.

[Retrieval]

A FilingSubset service is allowed to require the **pathname** attribute for accessing a file. As such, the value is always specified by the client, except on a **List** when the service must enumerate the parent directory. The mechanism presented in Section 6.3.4 using the IO$_ACCESS QIO to the Files-11 ACP will return a fully specified VMS filename that the service can return to the client.

6.1.1.6 type

The ability to transfer files between systems and retain generic file types is advantageous to the users of a heterogeneous network. In particular, the ability to transfer a text file to another system and preserve the editability of that file by the native text editors on the receiving system without explicit conversion is especially beneficial.

All FilingSubset implementations must support the type attribute values: **tAsciiText**, **tDirectory** and **tUnspecified**. The VMS operating system provides an explicit mechanism to distinguish between various file types; however, it is possible that several VMS file types will map to a single Filing **type** value. In general, a VMS client or service will choose a single VMS file type to represent the various Filing **type** values when creating files on either the **Retrieve** or **Store** procedures. This may result in a given implementation not creating the file in the correct format as desired by the user. However, without support for VMS-specific attributes, this cannot be avoided. Generally, files containing only Ascii characters will be treated as **tAsciiText** and all other non-directory files will be treated as **tUnspecified**.

[Retention]

The **tDirectory** file type is retained in a manner similar to the **isDirectory** attribute. When the attribute value is **tDirectory,** the directory is created via the VAX C `mkdir` or the `LIB$CREATE_DIR` routine.

It is possible to represent the **tAsciiText** and **tUnspecified** file types as one of several VMS file types. Without explicit support for VMS-specific attributes, the client or service implementation must make a choice, which may be what the user wants or not. One solution is to create **tAsciiText** files as VMS files with the following VMS attributes: sequential organization, variable record format with implied carriage control Files of type **tUnspecified** can be created as VMS files with sequential organization and undefined record format.

[Retrieval]

Values for the **type** attribute can be determined in one of two ways, either a call to `stat` or an `IO$_ACCESS` QIO to the Files-11 ACP asking for both file characteristics (`ATR$C_UCHAR`) and record attributes (`ATR$C_RECATTR`).

The **tDirectory** file type can be determined in a manner similar to that of the **isDirectory** attribute. The type value will be set to **tDirectory** if the file characteristics value returned from the QIO is `TRUE` when logically anded with the consant `FCH$M_DIRECTORY` Likewise, if the value `stat.st_mode` value returned from `stat` is `TRUE` when logically anded with `S_IFDIR`, the **type** value will be set to **tDirectory**.

The file organization and record format values (`stat.st_fab_rfm` and `stat st_fab_rat` returned from `stat` or `FAT$B_RTYPE` and `FAT$B_RATTRIB` returned from the QIO) can be used to determine non-directory values for the **type** attribute. The type **tAsciiText** can be assumed if the file has the following VMS record attributes: variable or fixed record format with implied carriage control or stream, streamlf or streamcr record formats Files of any other record format can be assumed to be of type **tUnspecified** The following routine illustrates this process:

```
#include    fatdef

/*
    routine:
        get_type
    input:
        record_format           VMS record format (FAB$B_RFM)
        record_attributes       VMS record attributes (FAB$B_RAT)
        returns:
            Cardinal containing XNS type value
*/

Cardinal get_type(record_format,record_attributes)
int     record_format;
int     record_attributes;
{

                    /* stream, streamlf and streamcr assumed tAsciiText */
    if ( (record_format == FAT$M_STM) ||
            (record_format == FAT$C_STMLF) ||
            (record_format == FAT$C_STMCR) )
        return(Filing_tAsciiText);
                    /* variable with implied carriage control assumed tAsciiText */
    else if ( (record_format == FAT$C_VAR) &&
            (record_attributes == FAT$M_CR) )
        return(Filing_tAsciiText);
                    /* fixed with implied carriage control assumed tAsciiText */
    else if ( (record_format == FAT$C_FIX) &&
            (record_attributes == FAT$M_CR) )
        return(Filing_tAsciiText);
                                        /* all else, assume tUnspecified */
    else return(Filing_tUnspecified);
}
```

The contents of files which are determined to be of type **tAsciiText** will be transferred in the form **StramofAsciiText**. The specific encoding/decoding of the bulk data stream is discussed in Sections 6.3.6 (**Store**) and 6.3.7 (**Retrieve**).

6.1.2 Implied attributes

Implied attributes are those attributes which obtain an implicit value when a new file is created. All subset implementations are required to permit the specification of the implied (default) value for these attributes. A service implementation may reject a **Store** procedure if the value for an implied attribute is not the default value and the service does not support the retention of non-default values for the attribute.

The implied attributes defined in the FilingSubset are **accessList**, **childrenUniquelyNamed**, **defaultAccessList**, **isTemporary**, **ordering**, **subtreeSizeLimit** and **version**.

Table 6.1 specifies the default values for these attributes on the VMS operating system. Since the attribute values are identical for every file unless otherwise supported, no explicit provision for retention and retrieval of these attributes is needed. The service should verify that the associated value is indeed the default on a **Store** and return the default values when requested on a **List** procedure.

6.1.3 Optional attributes

Those attributes which are defined as interpreted in the Filing Protocol but are not defined as either mandatory or implied within the FilingSubset are classified as optional attributes.

Attribute	Supported Values
accessList	[defaulted: TRUE]
childrenUniquelyNamed	TRUE
defaultAccessList	[defaulted: TRUE]
isTemporary	FALSE
ordering	defaultOrdering
subtreeSizeLimit	nullSubtreeSizeLimit
version	highestVersion

Table 6.1 VMS supported values for implied attributes

These attributes are not required to be supported by any FilingSubset service. Conventions for retaining and retrieving values for these attributes are not discussed here, since they are outside the definition for required functionality in the FilingSubset.

6.2 Client procedure support

Client routines require various VMS, RMS and VAX C routines to perform functions specific to the VMS operating system and to access the VMS/RMS file system. This interaction is discussed in this section.

6.2.1 Continuance timer support

A FilingSubset client must issue a **Continue** procedure at specific time intervals to prevent the service from terminating the session for lack of activity. This mechanism is implemented via use of the `alarm` and `signal` VAX C routines. Three routines are defined for use by the client: `set_continuance_timer`, `reset_continuance_timer` and `cancel_continuance_timer`. In addition, the routine `send_continue` is referenced. This routine will send a **Continue** to the service to maintain the open session.

`set_continuance_timer` calls `send_continue` to determine the service continuance value and then initializes the timer mechanism to send a SIGALRM signal before the expiration of that interval.

```
#include    signal

extern      send_continue();        /* expiration routine, will send continue */

Cardinal    continuance;                    /* continuance value, in seconds */
                                            /* returned from service */


/*
    routine:
        set_continuance_timer
        called after a successful Logon
    */

    set_continuance_timer()
    {
        continuance= send_continue();         /* get service value */
        continuance= continuance/3;           /* insure we expire before service */

        alarm(0);                             /* cancel any previous alarm */
        signal(SIGALRM,send_continue);        /* set routine to catch alarm */
        alarm(continuance);                   /* set alarm */
    }
```

reset_continuance_timer cancels any pending timer and reissues a new timer request.

```
    /*
        routine:
            reset_continuance_timer

        called after any FilingSubset procedure call
    */

    reset_continuance_timer()
    {
        alarm(0);                             /* cancel previous alarm */
        alarm(continuance);                   /* reset alarm */
    }
```

cancel_continuance_timer cancels the previous request and turns off handling of the SIGALRM signal.

```
    /*
        routine:
            cancel_continuance_timer

        called after a successful Logoff
    */

    cancel_continuance_timer()
    {
        alarm(0);                             /* cancel any previous alarm */
        signal(SIGALRM.SIG_IGN);              /* set routine: to ignore alarm */
    }
```

6.2.2 Determining mandatory attribute values

When a client performs a **Store**, values for the mandatory attributes may accompany the remote procedure call. Most of these values, with the exception of **pathname** and **type**, can be obtained locally by using the stat system call. The routine get_attributes illustrates how to accomplish this.

The stat routine returns the various file dates in a form similar to UNIX. The conversion mechanism described in Section 6.1.1 is not required to convert this value to XNS format

Instead the conversion mechansim for use with the stat routine described in Section 6.1.1.1 is used. The XNS time is computed by adding the returned value to the constant 2177452800, which represents the base time (00:00:00 GMT Jan 1, 1970).

```
#include stat
 -
#define    XNS_TIME_DIFFERENCE   2177452800      /* seconds between base times */

extern LongCardinal      createdon;
extern LongCardinal      modifiedon;
extern Boolean           isdirectory;
extern Cardinal          datasize;
extern Cardinal          type;

/*
    routine
        get_attributes
    input:
        pathname     service-specific pathname of file
    returns:
        -1    success
         1    error
*/

get_attributes(pathname)
char    *pathname;
{
    struct stat    file_stat;

    if ( stat(pathname,&file_stat) == -1 )                    /* stat file */
        return(1);

    createdon= file_stat.st_ctime + XNS_TIME_DIFFERENCE;      /* createdOn */
    modifiedon= file_stat.st_mtime + XNS_TIME_DIFFERENCE;     /* modifiedOn */

    datasize= file_stat.st_size;                              /* dataSize */

    if ( file_stat.st_mode & S_IFDIR ) {                      /* directory file */
        isdirectory= TRUE;
        type= tDirectory;
    } else {                                                  /* non-directory */
        isdirectory= FALSE;
        type= get_type(file_stat.st_fab_rfm,file_stat.st_fab_rat);
    }

    return(-1);
}
```

6.3 Service procedure support

A FilingSubset service implemented on the VMS operating system may use various VMS, RMS and VAX C routines to access the local file system and provide VMS-specific procedure support. This section presents detailed examples of this interaction.

Client access to files on a subset service is controlled through the use of a file handle. The implementation presented in Section 4 describes the value of the file handle as a pointer to to a *file context block*. To provide the necessary functionality, this context block will contain some items which are operating system specific.

For the implementation presented here, the following items are contained in the file context block:

- a copy of the **pathname** attribute value as specified on the **Open** or **Store**
- a cardinal identifying the file type requested by the client on the **Open**
- a cardinal specifying the file type as determined by the service
- a cardinal specifying the **dataSize** value for the file
- a boolean specifying the **isDirectory** value for the file
- a long cardinal specifying the **createdOn** value for the file in XNS format
- a long cardinal specifying the **modifiedOn** value for the file in XNS format
- FAB, XABDAT, RAB and NAM file structures used when accessing a file
- an appropriate buffer for use with the RAB structure

The following C structure defines the structure used in this section:

```
typedef file_handle {
    char            *pathname;             /* pointer to pathname value */
    Cardinal        type;                  /* client requested type (from Open) */
    Cardinal        truetype;              /* file system file type */
    Cardinal        datasize;              /* dataSize value */
    Boolean         isdirectory;           /* isDirectory */
    LongCardinal    createon;              /* createdOn value */
    LongCardinal    modifiedon;            /* modifiedOn value */
    struct fab      file_fab;              /* file access block (FAB) */
    struct rab      file_rab;              /* record access block (RAB) */
    struct xabdat   file_xab;              /* extended attribute block (XABDAT) *
    struct nam      file_nam;              /* name block (NAM) */
    char            file_buffer[32767];    /* input/output buffer 32767= max size */
};
```

6.3.1 Logon

The **Logon** procedure is responsible for validating the user attempting the connection and, if successful, altering the process ownership to that of the user. This alteration of ownership ensures that the process is subject to the normal access/protection mechanisms employed by the VMS operating system when subsequent procedure calls request access to files on the service. The user name and password entries of the secondary credentials supplied on the **Logon** are validated against the standard VMS UAF file. Once this has been completed, the UIC and privileges of the process are changed to that of the respective user as determined from the authorization file entry for the user.

The process is also positioned to the appropriate root directory for the service, which corresponds to a VMS disk/directory pair (generally the VMS root, [000000], on a specific disk). This provides a VMS disk and directory which can be associated with **nullHandle** as the root for the service. The examples in this section use the external variable `service_root_device` to specify the default device for the service. The **Logon** procedure will set this variable to the appropriate value.

VMS does not export routines to perform these services and the nature of these routines is such that they are proprietary to VMS. Because of this, no routines are presented here. Instead it is assumed that implementors of a VMS service will have access to VMS internal documentation which describes the VMS mechanisms for performing the required functions.

6.3.2 Continue

The continuance mechanism is defined to allow services to close a session if it has been idle for a long period of time or the session needs to be terminated for other reasons. Each service maintains a continuance value which is the number of seconds that it will keep a session open between successive procedure calls. This allows the service to set a timeout mechanism to notify it when this time interval has passed and allow it to disconnect the active session.

This mechanism is armed once a session has been successfully established by a **Logon** and is terminated once the session is ended with a **Logoff**. Additionally, each routine which processes a FilingSubset procedure, as described in Section 4, should rearm the timer.

The `alarm` and `signal` routines are used to implement this mechanism for VMS services. `alarm` is used to set the timer mechanism for the specified interval while `signal` is used to indicate whether the service is to handle or ignore the alarm.

The routines `set_continuance_timer`, `reset_continuance_timer` and `cancel_continuance_timer` are defined. The service routine `continuance_expiration` is referenced by `set_continuance_timer` and would execute at the expiration of a timeout interval. At that time, this routine would close the current session in a manner similar to that proposed for the **Logoff** procedure in Section 4.4.

`set_continuance_timer` initially establishes the timeout mechanism.

```
#include    signal

Cardinal    continuance;                  /* continuance value, in seconds */
extern      continuance_expiration();     /* expiration routine */
```

```
/*
    routine:
        set_continuance_timer
*/

set_continuance_timer()
{
    alarm(0);                                    /* cancel any previous alarm */
    signal(SIGALRM,continuance_expiration);      /* set routine to catch alarm */
    alarm(continuance);                          /* set alarm */
}
```

The reset_continuance_timer and cancel_continuance_timer routines are identical to the client routines specified in Section 6.2.1.

6.3.3 Open

The **Open** procedure opens a file for subsequent access by the client. The file is identified by the value specified for the **pathname** attribute. VMS supports multiple versions, so the **version** values **lowestVersion** and **highestVersion** will indicate different files if more than one version exists. If the pathname does not contain a version specification, the string ";-0" or ";0" can be catenated to the **pathname** value to indicate the lowest version or highest version of a file, respectively. When an explicit **version** value is specified along with a **pathname** value that contains a version, the explicit **version** value will take precedence over the **pathname** value.

The **Open** routine first performs a call to stat_file to determine values for the **dataSize**, **isDirectory** and **type** attributes for the desired file. This allows the subsequent file transfer procedures to access necessary information simply by examining the file context block.

```
#include    stat

/*
    routine:
        stat_file
    input:
        pointer to file handle
    returns:
        -1    success
        1     failure (specific errors will be determined on the subsequent file
              open)

        file_context_block filled in
*/

stat_file(file_context_block)
file_handle    *file_context_block;
{
    struct stat    file_stat;
    /* stat does not return detailed errors so specific errors will be returned
    when the actual open is attempted */
    if ( stat(file_context_block->pathname,&file_stat) == -1 )
        return(1);

    file_context_block->datasize= file_stat.st_size;         /* dataSize */

    if ( file_stat.st_mode & S_IFDIR ) {
        file_context_block->isdirectory= TRUE;                /* directory */
        file_context_block->truetype= Filing_tDirectory;
    } else {
        file_context_block->isdirectory= FALSE;               /* non-directory */
        file_context_block->truetype= get_type(file_stat.st_fab_rfm,
                                               file_stat.st_fab_rat);
    }

    return(-1);
}
```

The routine open_file is subsequently called to open the file. This routine will be called regardless of any errors that are returned from stat_file, since specific error conditions cannot be determined until the open is atempted. The only possible errors are: **accessRightsInsufficient** if the file cannot be accessed, **fileNotFound** if the file or some component of the pathname does not exist, and **accessRightsIndeterminate** if any other error occurs.

```
#include    rms

/*
    routine:
        open_file
    input:
        pointer to file handle
    returns:
        -1    success
        else Filing Error, Problem

        file_context_block entries filled in
*/

Filing_Error    open_file(file_context_block)
file_handle     *file_context_block;
{

    int error;
    Filing_Error    error_value;            /* Filing error, problem pair */

    error_value.error= Filing_AccessError;  /* set to Filing AccessError */

    file_context_block->file_fab= cc$rms_fab;                    /* initialize FAB */
    file_context_block->file_fab.fab$l_fna= cbptr->pathname;
    file_context_block->file_fab.fab$b_fns= strlen(cbptr->pathname);
    file_context_block->file_fab.fab$b_fac= FAB$M_GET | FAB$M_BRO;
```

```
        file_context_block->file_fab.fab$b_shr= FAB$M_NIL;

        error= sys$open(&file_context_block->file_fab);        /* open file */
        if ( error != RMS$_NORMAL ) {
            if ( error == RMS$_FNF )                           /* no such file */
                error_value.problem= Filing_fileNotFound;
            else if ( error == RMS$_PRV )                      /* user has no access */
                error_value.problem= Filing_accessRightsInsufficient;
            else                                               /* all other errors */
                error_value.problem= Filing_accessRightsIndeterminate;
            return (error_value);
        }

        file_context_block->file_rab= cc$rms_rab;                 /* initialize RAB */
        file_context_block->file_rab.rab$l_fab= &file_context_block->file_fab;
        file_context_block->file_rab.rab$l_ubf= file_context_block->file_buffer;
        file_context_block->file_rab.rab$w_usz= MAX_RECORD_SIZE;

        error= sys$connect(&file_context_block->file_rab);    /* connect rab to fab */

        if ( error != RMS$_NORMAL ) {
            error_value.problem= Filing_accessRightsIndeterminate;
            return (error_value);
        }

    return(-1);
    }
```

6.3.4 List

The **List** procedure enumerates a directory looking for the specified file or files and returns the requested attributes for each file found. The file specification to be listed is specified in the **pathname** attribute value on a **filter** of type **matches**. This procedure is unique in that it is the only procedure which will allow wildcard characters in the pathname syntax which are interpreted by the service.

This function is easily accomodated through the use of the IO$_ACCESS QIO to the Files-11 ACP which lists a directory and returns the files matching some file name criteria along with various file characteristics requested. The file names are returned in ascending order by name which is the Filing Protocol **defaultOrdering**

The routine list_directory is defined to perform this function. This routine performs a VMS SYS$ASSIGN to the service disk device as specified by service_root_device. The routine get_directory_id is then called to parse the specified pathname to return the VMS file identifier for the appropriate directory. list_directory then repetitively performs an IO$ACCESS QIO to retrieve the next filename matching the **pathname** specification taken from the **filter** of type **matches**. Appropriate VMS file characteristics are requested so that the values for the FilingSubset defined mandatory attributes can be returned to the client ATR$C_UCHAR is used to determine the value for the **isDirectory** attribute. The **type** and **dataSize** values are determined from the values returned from ATR$C_RECATTR. The **createdOn** and **modifiedOn** values come from ATR$C_CREDATE and ATR$C_REVDATE respectively

The resulting filename buffer specified on the QIO will return the VMS specific **pathname** value.

The error **AccessError accessRightsInsufficient** is returned if an error is encountered during the SYS$ASSIGN or returned from get_directory_id. If an error occurs when accessing an individual file, that file is simply omitted from the list returned.

```
#include rms
#include ssdef
#include descrip
#include atrdef               /* assumes include files which define public */
#include fibdef               /* structures ATR, FIB, FAT, FCH and IOSB */
#include fatdef               /* these are not necessarily included in VAX C */
#include fchdef
#include iosb

#define    EVENT_FLAG    2        /* event flag for QIO use */

extern char    *service_root_device:      /* VMS device known as service root */

/*
    routine:
        list_directory
    input:
        file_spec    pointer to VMS file specification
    returns:
        -1    success
        else Filing Error, Problem
*/

Filing_Error   list_directory(file_spec)
char    *file_spec:            /* pathname attribute from filter of type matches */

{
    int            error:
    struct fib     fib= 0:
    struct atr     attributes[5]:
    char           file_name[NAM$C_MAXRSS]:
    char           result_name[NAM$C_MAXRSS]:
    long           length:
    double         creation_date:
    double         revision_date:
    struct fat     record_attributes:
    long           file_characteristics:
    short          channel:
    struct iosb    io_status:
    Filing_Error   error_value:

            /* following are used to hold mandatory attributes until added to
            outgoing bulk data stream */
    LongCardinal   createdon:
    LongCardinal   modifiedon:
    Cardinal       datasize:
    Boolean        isdirectory
```

```
Cardinal        type;
String          pathname;

struct dsc$descriptor_s result_name_descriptor;
struct dsc$descriptor_s device_descriptor;
struct dsc$descriptor_s fib_descriptor;
struct dsc$descriptor_s file_descriptor;

error_value.error= Filing_AccessError;          /* set to Filing AccessError */

fib.fib$w_nmctl= FIB$M_WILD;                   /* turn on wildcard mechanism */
                                                   /* fill in directory id */
if ( get_directory_id(file_spec,&fib) == 1 ) {
    error_value.problem= Filing_accessRightsInsufficient;
    return(error_value);
}

                              /* VMS descriptor setup */
                                            /* file name returned from QID */
result_name_descriptor.dsc$a_pointer= result_name;       /* pathname value */
result_name_descriptor.dsc$w_length= NAM$C_MAXRSS+1;
result_name_descriptor.dsc$b_class= DSC$K_CLASS_S;
result_name_descriptor.dsc$b_dtype= DSC$K_DTYPE_T;


                                            /* device known as service root */
device_descriptor.dsc$a_pointer= service_root_device;
device_descriptor.dsc$w_length= strlen(service_root_device);
device_descriptor.dsc$b_class= DSC$K_CLASS_S;
device_descriptor.dsc$b_dtype= DSC$K_DTYPE_T;


                                                   /* file identifier block */
fib_descriptor.dsc$a_pointer= &fib;
fib_descriptor.dsc$w_length= FIB$K_SMALLSIZE;
fib_descriptor.dsc$b_class= DSC$K_CLASS_S;
fib_descriptor.dsc$b_dtype= DSC$K_DTYPE_T;


                                                   /* input pathname value */
file_descriptor.dsc$a_pointer= file_spec;
file_descriptor.dsc$w_length= strlen(file_spec);
file_descriptor.dsc$b_class= DSC$K_CLASS_S;
file_descriptor.dsc$b_dtype= DSC$K_DTYPE_T;

                         /* set up VMS attribute structures to be returned */
attributes[0].atr$w_size= ATR$S_UCHAR;         /* file characteristics */
attributes[0].atr$w_type= ATR$C_UCHAR;         /* isDirectory value */
attributes[0].atr$l_addr= &file_characteristics;

attributes[1].atr$w_size= ATR$S_RECATTR;       /* record attributes */
attributes[1].atr$w_type= ATR$C_RECATTR;       /* type and dataSize values */
attributes[1].atr$l_addr= &record_attributes;

attributes[2].ATR$W_SIZE= ATR$S_CREDATE;       /* creation date */
attributes[2].atr$w_type= ATR$C_CREDATE;       /* createdOn value */
attributes[2].atr$l_addr= &creation_date;
```

```
attributes[3].atr$w_size= ATR$S_REVDATE:        /* revision date */
attributes[3].atr$w_type= ATR$C_REVDATE:        /* modifiedOn value */
attributes[3].atr$l_addr= &revision_date:

attributes[4].atr$w_size= attributes[4].atr$w_type= 0:

if ( (error= sys$assign(&device_descriptor,&channel,0,0)) != SS$_NORMAL) {
    error_value.problem= Filing_accessRightsInsufficient:
    return(error_value):
}

while (TRUE) {                  /* get each file matching file_spec */
    fib.fib$w_fid[0]= fib.fib$w_fid[1]= fib.fib$w_fid[2]= 0:

                                /* returns next file and characteristics */
    sys$qiow(EVENT_FLAG,channel,IO$_ACCESS|IO$M_ACCESS,&io_status,0,0,
        &fib_descriptor,&file_descriptor,&length,&result_name_descriptor,
        attributes,0):

                                    /* break out when no file returned */
    if ( io_status.s_status != SS$_NORMAL ) {
        if ( fib.fib$l_wcc && (io_status.s_status != SS$_NOMOREFILES) )
            continue:           /* any other error simply omit file from list */
        else    break:
    }
                                    /* determine mandatory attribute values */
    strncpy(pathname,result_name,length):      /* pathname from result_name */
    *(pathname+length)= '/0':

                                /* convert createdOn, modifiedOn to XNS format */
    convert_vms_time(&creation_date,createdon):
    convert_vms_time(&revision_date,modifiedon):

                                /* compute dataSize value */
    datasize= compute_datasize(record_attributes):

                                /* set isDirectory and type as appropriate */
    if ( file_characteristics & FCH$M_DIRECTORY ) {
        isdirectory= TRUE:
        type= tDirectory:
    } else {
        isdirectory= FALSE:
        type= get_type(record_attributes.fat$b_rtype,
                                    record_attributes.fat$b_rattrib):

    }
/* insert implementation specific routines here:
          make an attribute sequence from createdon, datasize, isdirectory,
             modifiedon, pathname and type variables
            (if other non-mandatory attributes arerequested, appropriate values
            must also be returned)
          write the attribute sequence to the bulk data stream
*/
    return(-1);
}

/*
    routine:
        get_directory_id
    input:
        file_spec     pointer to pathname value
        file_fib      pointer to fib structure to fill in directory id info
    returns:
        -1    success
        1     error occurred
*/

get_directory_id(file_spec,file_fib)
char            *file_spec;
struct FIB      *file_fib;

{
    struct fab     file_fab;
    struct nam     file_nam;

    file_fab= cc$rms_fab;                               /* initialize fab */
    file_fab.fab$l_fna= file_spec;
    file_fab.fab$b_fns= strlen(file_spec);

    file_nam= cc$rms_nam;                               /* initialize nam */
    file_fab.fab$l_nam= &file_nam;

                                      /* use sys$parse to obtain directory id */
    if ( sys$parse(&file_fab) != SS$_NORMAL )
        return(1);

    file_fib->fib$w_did[0]= file_nam.nam$w_did[0];
    file_fib->fib$w_did[1]= file_nam.nam$w_did[1];
    file_fib->fib$w_did[2]= file_nam.nam$w_did[2];

    return(-1);
}
```

6.3.5 Store

The **Store** procedure is used to create both directory and non-directory files. A different method is used to create directory files under VMS, so the service will take an appropriate action based on the values of the **isDirectory** and **type** attribute values as stored in the file context block.

After the **Store** routine has validated the argument and attribute values, a file handle is allocated. The `create_file` routine is then called to actually create the file with the specified attributes and default values for unspecified mandatory attributes. A FilingSubset service will always create a new version of a file. If a specified pathname includes a specific version and that version already exists, the service will return **InsertionError fileNotUnique**. Appropriate values for **AccessProblem** are returned if the file cannot be created for any other reason.

```
#include    rms

#define     MAX_RECORD_SIZE    32767              /* maximum VMS record size */
/*
    routine:
        create_file
    input:
        pointer to file handle
    returns:
        -1    success
        else Filing Error, Problem

        file_context_block structures filled in
*/

Filing_Error create_file(file_context_block)
file_handle    *file_context_block;
{

    int             error;
    Filing_Error    error_value;                  /* Filing error, problem pair */

    error_value.error= Filing_AccessError;        /* set default Filing error */

    file_context_block->file_fab= cc$rms_fab;          /* initialize FAB */
    file_context_block->file_fab.fab$l_fna= file_context_block->pathname;
    file_context_block->file_fab.fab$b_fns= strlen(file_context_block->pathname);
    file_context_block->file_fab.fab$l_alq= file_context_block->dataSize/512 + 1;
    file_context_block->file_fab.fab$l_fop= FAB$M_MXV;
    file_context_block->file_fab.fab$w_mrs= MAX_RECORD_SIZE;

                                                /* pick VMS type from type value */
    if ( file_context_block->type == Filing_tAsciiText ) {
        file_context_block->file_fab.fab$b_rfm= FAB$C_VAR;
        file_context_block->file_fab.fab$b_rat= FAB$M_CR;
    } else if ( file_context_block->type == Filing_tUnspecified ) {
        file_context_block->file_fab.fab$b_rfm= FAB$C_UDF.
    }
                                                     /* initialize XAB */
    file_context_block->file_fab.fab$l_xab= &file_context_block->file_xab;
    file_context_block->file_xab= cc$rms_xabdat;
                                     /* save createdOn, per Section 5.1.1.1 */
    set_create_time(file_context_block);
```

```
        error= sys$create(&file_context_block->file_fab);      /* create file */
        if ( error != RMS$_NORMAL ) {                          /* check for errors */
            if ( error == RMS$_PRV )                           /* no privilege */
                error_value.problem= Filing_accessRightsInsufficient;
            else if ( error == RMS$_FEX ) {                    /* file exists */
                error_value.error= Filing_InsertionError;
                error_value.problem= Filing_fileNotUnique;
            } else                                             /* all others */
                error_value.problem= Filing_accessRightsIndeterminate;
            return(error_value);
        }

        file_context_block->file_rab= cc$rms_rab;              /* initialize RAB */
        file_context_block->file_rab.rab$l_fab= &file_context_block->file_fab;
        file_context_block->file_rab.rab$rbf= file_context_block->file_buffer;
        file_context_block->file_rab.rab$w_rsz= MAX_RECORD_SIZE;

                                                               /* connect RAB to FAB */
        if (error= sys$connect(file_context_block->file_rab)) != RMS$_NORMAL )
            error_value.problem= Filing_accessRightsIndeterminate;
            return(error_value);
        }

        return(-1);
    }
```

After the file is successfully created by `create_file`, the bulk data stream will be read and written to the file and the file is closed. Files of type **tAsciiText** will have to be decoded to determine the correct record size before writing the record as illustrated below:

```
    {
        int count;

        ....

                                    /* character count is sequence length * 2 */
        count= sequence_length(AsciiString.bytes)/2;
        if ( !AsciiString.lastByteSignificant )           /* if count is odd. */
            count--;                                      /* decrement by 1 */

        file_context_block->rab.rab$b_rsz= count;         /* set count in rab */
                                                          /* write characters */
        sys$put(&file_context_block->fab);

        ....

    }
```

FilingSubset services are not required to support directory creation. If directory creation is supported, the service may optionally restrict this to only allow the creation of empty directories. Directory files can be created easily on VMS with the VAX C `mkdir` or the `LIB$CREATE_DIR` routines; however, the format of directory files is operating system dependent and therefore does not encourage the transfer of directory file contents. The create_directory routine is provided to illustrate the creation of empty directory files.

```
/*
    routine:
        create_directory
    input:
        pointer to file handle
    returns:
        -1    success
        else Filing Error, Problem
*/

Filing_Error   create_directory(file_context_block)
file_handle    *file_context_block;
{
    int            status;
    Filing_Error   error_value;                  /* Filing error, problem pair */

    error_value.error= Filing_AccessError;       /* default to AccessError */

                                                 /* create directory */
    if (mkdir(file_context_block->pathname,0) == -1) {
        error_value.problem= Filing_accessRightsInsufficient;
        return(error_value);
    }

    return(-1);
}
```

6.3.6 Retrieve

The **Retrieve** procedure transfers a file from a service to the calling client. FilingSubset services are not required to allow the retrieval of directory files. The VMS implementation does not allow the retrieval of directory files since the format of these files is VMS-specific. If the file is not a directory, then the file is opened and the content transferred via a bulk data stream to the client.

The **Retrieve** procedure assumes that the file was previously opened by an **Open** procedure and that the appropriate `file_context_block` fields have been initialized. The content of the file is then read and written to the bulk data stream.

Files of type **tAsciiText** are transferred as a **StreamofAsciiText**. The content of the file as read from the file must be encoded into this format for transmission to the client.

6.3.7 Delete

The **Delete** procedure is used by clients to delete files. If the specified file is a directory, a FilingSubset service is not required to support the deletion of that file and all its descendants. VMS does not provide a simple mechanism for supporting the deletion of non-empty directories, therefore the service returns **AccessProblem accessRightsInsufficient** if the directory file cannot be deleted.

The VMS `delete` routine is used to delete the specified file. The `delete` routine will return an error if the file is a non-empty directory. Appropriate Filing errors are reported as type **AccessProblem**.

The following procedure illustrates the file deletion mechanism:

```
#include   errno

/*
    routine:
        delete_file
    input:
        pointer to file handle
    returns:
        -1   success
        else Filing Error, Problem

*/

Filing_Error   delete_file(file_context_block)
file_handle    *file_context_block;
{
    int status;
    Filing_Error   error_value;                    /* Filing error, problem pair */

    error_value.error= Filing_AccessError;         /* set to Filing AccessError */
                    /* attempt delete of directory, success only if empty */
    if (file_context_block->isdirectory ) {
        if ( delete(file_context_block->pathname) == -1 ) {
            error_value.problem= Filing_accessRightsInsufficient;
            return(error_value);
        }
    } else {                                       /* use delete for non-directories */
        if ( delete(file_context_block->pathname) == -1 ) {
            switch (errno) {
                case EACCES:                       /* user has no access */
                case EPERM:                        /* user has no access */
                    error_value.problem= Filing_accessRightsInsufficient;
                    return(error_value);

                case ENOENT:                       /* no such file */
                case ENOTDIR:                      /* no such directory */
                    error_value.proolem= Filing_fileNotFound;
                    return(error_value);

                default:                           /* all other errors */
                    error_value.problem= Filing_accessRightsIndeterminate;
                    return(error_value);
            }
        }
    }
```

```
        return(-1);
    }
```

A. REFERENCES

The following documents describe those protocols and data structures referenced within this guide.

[1] Xerox Corporation. *Authentication Protocol*. Xerox Network Systems Standard Stamford, Connecticut: April 1984: XNSS 098404 (XSIS 098404).
This reference defines the Authentication Protocol upon which the Filing and FilingSubset Protocols rely for authentication.

[2] Xerox Corporation. *Bulk Data Transfer*. Xerox Network Systems Standard. Stamford, Connecticut: April 1984: XNSS 038112 (XSIS 038112): Addendum 1a. Augments [6]
This reference defines the Bulk Data Transfer Protocol upon which the Filing and FilingSubset Protocols rely for bulk data transfer

[3] Xerox Corporation. *Character Code Standard*. Xerox Network Systems Standard. Stamford, Connecticut: April 1984: XNSS 058404 (XSIS 058404).
This reference defines the character set and the string format which provide the basis for Courier's string data type.

[4] Xerox Corporation. *Clearinghouse Protocol*. Xerox Network Systems Standard Stamford, Connecticut: April 1984: XNSS 078404 (XSIS 078404).
This reference defines the protocol which FilingSubset implementations use to provide various services. It also defines the structure of user names which appear as various file attributes

[5] Xerox Corporation. *Clearinghouse Entry Formats*. Xerox Network Systems Standard Stamford, Connecticut. April 1984. XNSS 168404 (XSIS 168404)
This document defines Clearinghouse property types and the structure of their entries in terms of Courier data types

[6] Xerox Corporation. *Courier: The Remote Procedure Call Protocol*. Xerox Network Systems Standard. Stamford, Connecticut: December 1981. XNSS 038112 (XSIS 038112).
This reference defines the Courier language, in terms of which the Filing and FilingSubset Protocols are defined.

[7] Xerox Corporation. *Filing Protocol*. Xerox Network Systems Standard Stamford, Connecticut. May 1986. XNSS 108605 (XSIS 108605)
This reference defines the Filing and the FilingSubset Protocols

[8] Xerox Corporation. *Internet Transport Protocols*. Xerox Network Systems Standard Stamford, Connecticut: December 1981: XNSS 028112 (XSIS 028112)
This reference defines the Sequenced Packet Protocol upon which Courier relies for data transport.

[9] Xerox Corporation. *Secondary Credentials Formats*. Xerox Network Systems Standard. Stamford, Connecticut: May 1986; XNSS 258605 (XSIS 258605).
This reference documents specific type assignments and data formats for secondary credentials. Implementations of FilingSubset on hybrid hosts may require secondary authentication information.

[10] Xerox Corporation. *Time Protocol*. Xerox Network Systems Standard. Stamford, Connecticut: October, 1982; XNSS 088210 (XSIS 088210).
This reference defines the Time Standard upon which the Filing and FilingSubset Protocols rely for the definition of the format for time and date quantities.

B. Filing Protocol

A copy of the Xerox Filing Protocol is included for reference purposes. This document defines both the Filing and FilingSubset Protocols.

Filing Protocol

Xerox System Integration Standard

XEROX

XNSS 108605
May 1986

FILING PROTOCOL

XEROX

XNSS 108605
May 1986

Notice

This *Xerox Network Systems Standard* describes the Filing Protocol.

All text and graphics prepared on the Xerox 8010 Information System.

PREFACE

This document is one of the family of publications that describe the network protocols underlying Xerox Network Systems (XNS).

Xerox Network Systems comprise a variety of digital processors interconnected by means of a variety of transmission media. System elements communicate both to transmit information between users and to economically share resources. For system elements to communicate with one another, certain standard protocols must be observed.

Comments and suggestions on this document and its use are encouraged. Please address communications to:

Xerox Corporation
Xerox Network Systems Institute
2300 Geng Road
Palo Alto, California 94303

TABLE OF CONTENTS

1. Introduction 1

1.1 Purpose 1

1.2 Relationship to other protocols 1

1.3 Documentation organization 2

1.4 Document conventions 2

1.4.1 Notation 2

1.4.2 Notation for examples 2

2. Overview 5

2.1 Clients and services 5

2.2 Users, authentication, and sessions 5

2.3 Files, content, and attributes 6

2.4 Directories 7

2.5 Handles and controls 7

2.6 Creating, deleting, and accessing files 8

2.7 Enumerating and locating files in directories 8

2.8 Serializing and deserializing files 9

2.9 Transferring data 9

3. Remote procedures 11

3.1 Logging on and off 11

3.1.1 Credentials 11

3.1.2 Sessions 14

3.1.3 Logon 14

3.1.4 Logoff 15

3.1.5 Continue 15

3.2 Opening and closing files 16

3.2.1 File handles 16

3.2.2 Opening files 17

3.2.3 Closing files 19

3.3 Accessing and modifying controls 19

3.3.1 Controls 19

3.3.2 Accessing controls 22

3.3.3 Modifying controls 22

3.4 Accessing and modifying attributes 23

3.4.1 Attributes 23

3.4.2 Accessing attributes 24

3.4.3 Modifying attributes 25

3.5 Locating and listing files in directories 26

3.5.1 Scopes 26

3.5.2 Locating files 31

3.5.3 Listing files 31

3.6 Accessing and modifying the content of files 33

3.6.1 Uninterpreted file format 33

3.6.2 Storing files 33

3.6.3 Retrieving files 34

3.6.4 Replacing files 35

3.6.5 Random access to files 35

3.7 Creating and deleting files 37

3.7.1 Creating files 37

3.7.2 Deleting files 38

3.8 Copying and moving files 38

3.8.1 Copying files 39

3.8.2 Moving files 39

3.9 Serializing and deserializing files 40

3.9.1 Serialized file format 40

3.9.2 Serialize 41

3.9.3 Deserialize 41

3.10 Procedures and attributes 43

4. Attributes 59

4.1 Classes of attributes 59

4.1.1 Interpreted vs. uninterpreted 59

4.1.2 Environment vs. data 60

4.1.3 Primary vs. derived 61

4.2 Definition of attributes 61

4.2.1 How attributes are defined 61

4.2.2 Identification-related attributes 62

4.2.3 Content-related attributes 65

4.2.4 Parent-related attributes 66

4.2.5 Event-related attributes 67

4.2.6 Directory-related attributes 69

4.2.7 Access-related attributes 71

5. Remote errors 75

5.1 Access errors 75

5.2 Argument errors 77

5.3 Authentication errors 78

5.4 Connection errors 80

5.5 Handle errors 81

5.6 Insertion errors 81

5.7 Range errors 82

5.8 Service errors 83

5.9 Session errors 84

5.10 Space errors 84

5.11 Transfer errors 85

5.12 Undefined errors 86

Appendices:

A. References 87

B. Type assignment procedures 89

B.1 Common file types 89

C. Program declaration 91

D. Examples 103

E. FilingSubset 109

E.1 Overview 109

E.2 Definition 110

E.3 Procedures 111

E.4 Attributes 116

E.5 Remote errors 118

E.6 Procedures and attributes 118

E.7 Program declaration 123

F. Pathname syntax 129

LIST OF TABLES

3.1	ChangeAttributes	44
3.2	Copy	45
3.3	Create	46
3.4	Delete	47
3.5	Deserialize	48
3.6	Move	49
3.7	Open	50
3.8	Replace	51
3.9	ReplaceBytes	52
3.10	Retrieve	53
3.11	RetrieveBytes	54
3.12	Serialize	55
3.13	Store	56
3.14	UnifyAccessLists	57
E.1	FilingSubset and Filing attribute classes	116
E.2	List	120
E.3	Open	121
E.4	Store	122

1. INTRODUCTION

In any information handling system, storage of information is one of the most basic functions. A component of the system may store information in order to protect it from certain failures, release a more crucial storage area, or communicate the information to some other part of the system. In a distributed system, these goals are even more fundamental. Storage of information in another physical location can protect it against even catastrophic failures at the original location, and communication of information to other parts of the system is crucial.

In a distributed system, filing functionality is provided by a file service. A file service is similar to a conventional file system, the principal difference being that it offers its services to clients residing in other system elements. The file service defined here provides such features as:

- storage and retrieval of files
- hierarchically structured directories
- searching and sorting by arbitrary file attributes
- operations on subtrees of files
- recording of file activity

1.1 Purpose

This document defines the *Xerox Filing Protocol*, the protocol for interaction between clients and file services. It is both a guide for using a file service, and a specification for the implementation of such a service. It does not describe any particular implementation of the protocol.

Typically a file service is associated with other support mechanisms such as backup and archival features. These features are considered to be invisible to the network client, and are not addressed in this specification.

This protocol provides a general filing facility to support a wide variety of applications. However, it is *not* intended to directly support network administration functions, printing, electronic mail, or other distributed activities. These are subjects of other specifications.

1.2 Relationship to other protocols

The protocol defined in this document is an application-level protocol. It employs several other protocols. Requests of a file service are communicated using the request-reply or transaction discipline defined by Courier [6]. Every type of request is modeled as a remote procedure as defined in Courier; every exceptional condition that may arise is modeled as a remote error. All parameters transferred between the client and the file service obey the conventions described in the Courier specification. This present specification, therefore, constitutes a Courier remote program.

The contents of files and other large data items are transferred using the protocol described in the *Bulk Data Transfer Protocol* addendum [3].

The Filing Protocol also depends upon the standard time format defined in the Time Protocol [8], and on the Clearinghouse Protocol [5] and the Authentication Protocol [2].

1.3 Document organization

Chapter 2 of this document gives an overview of the concepts defined in the standard. Chapter 3 defines the remote procedures needed to interact with a file service. Chapter 4 defines the meaning and use of the attributes that are interpreted by the file service. Chapter 5 defines the remote errors reported by the file service when exceptional conditions occur.

Appendix A lists other documents which supplement the specification. Appendix B describes the administrative procedures for obtaining ranges of file types and attribute types. Appendix C lists the Filing remote program in its entirety. Appendix D gives examples of interactions between a client and a file service. Appendix E defines a subset of the Filing Protocol. Appendix F defines a common syntax for the expression of file pathnames.

1.4 Document conventions

Courier text and examples are depicted in special fonts, and generally conform to a certain style. The rules and style are set forth below.

1.4.1 Notation

Throughout this document, special fonts are used to depict Courier text instead of using quote marks or other delimiters. This convention also aids the eye in discriminating between Courier text and the exposition. Items in THIS FONT indicate elements of the Courier language and are almost always in upper case. **This font** indicates items that are defined using the Courier language.

Identifiers that are defined in this protocol (as opposed to being defined by Courier) will have their first letter capitalized if they are the name of a type, error, or procedure; identifiers with a lowercase first letter are usually the names of variables, arguments, or results.

1.4.2 Notation for examples

In the examples that follow, a call to a remote procedure is denoted by the name of the procedure followed by the arguments supplied to it. A return from a remote procedure is denoted simply by the results, preceded—when confusion might otherwise result—by the keyword RETURNS. The argument or result list is modeled as a record; the arguments or results as the record's components. Accordingly, Courier's standard notation for record constants is used to specify argument and result lists.

For example, if the procedure **Add** is defined as:

Add: PROCEDURE **[first, second**: CARDINAL**]**
RETURNS **[sum**: CARDINAL**] = 99;**

then a call to that procedure would be denoted by:

Add [first: 7, second: 5]

and the call would yield the result:

[sum: 12] or RETURNS **[sum: 12]**

Fine point: The above notation for procedure calls should not be confused with the standard notation for a record constant selected by means of a choice data type. The two are similar in appearance but otherwise unrelated.

Examples of remote errors are either just the name of the error, if it is defined without arguments:

Overflow

or the same as a procedure call if it is defined with arguments. For example, if **Overflow** were defined as:

Overflow: ERROR **[carry**: CARDINAL**] = 99;**

then an example of its use might be:

Overflow [carry: 1]

indicating that **Overflow** was reported with argument **carry** having the value 1.

Courier requires values for a SEQUENCE OF UNSPECIFIED to be a sequence of numbers. So as to retain readability in examples, the content of a SEQUENCE OF UNSPECIFIED is described using Courier notation. The reader should understand that the numeric representation of these types is what should be used as the content of the sequence.

2. OVERVIEW

To better understand the description of the file service, it is necessary to understand a number of concepts and terms. Most of the concepts described below will be familiar as those of conventional file systems, but there are a few that are considerably different.

2.1 Clients and services

This standard defines a protocol for the communication of filing requests. Requests to store a file, to delete a file, or to list a directory are all examples of filing requests.

A *service* is an entity (software or hardware) that accepts and responds to submitted requests for some type of service. A *file service* is a service that handles filing requests. A *client* of a service is an entity that submits requests to that service. In this document, where the service is not otherwise specified, the service is assumed to be a file service. A client may or may not be operating on behalf of a human being.

All interaction between the client and the service is initiated by the client. The service never spontaneously interacts with a client.

2.2 Users, authentication, and sessions

A client always interacts with a service on behalf of a *user*. The user may be a human being, or may be some other entity (such as another service). In any event, the user has a *user name* that distinguishes him from other users.

Before making use of a file service a client must *log on*. The client presents *credentials* which identify him to the file service. The service responds by establishing a *session* and returning a *session handle* which is used to identify the client (and the state of this interaction) in future requests.

Credentials represent the client's proof of identity and permit the service to identify the party initiating the interaction. Where only a Clearinghouse distinguished name is required to identify the initiator, the standard mechanisms of the *Authentication Protocol* [2] are used. Credentials that resolve a client's identity to a Clearinghouse name are the client's *primary credentials*.

A file service implemented on a host whose authentication requirements go beyond those satisfied by the primary authentication mechanisms of the *Authentication Protocol* may require additional authentication information. This additional authentication information is referred to as *secondary credentials*. A host requiring secondary credentials is a *hybrid* host.

Once appropriate credentials have been provided by the client, a session is established. A session encapsulates the state of the client with respect to the interaction then being initiated. For example, the session keeps track of files that are open, locks that are held, and the name of the user on whose behalf the client is operating. The session handle is included in all subsequent requests in order to identify the client and its state. When interaction is complete, the client *logs off*. This terminates the session, freeing any allocated resources and invalidating the session handle. The client must log on again before any further interaction may occur.

Sessions may vary greatly in duration. In some patterns of use a session is established to perform a single operation and then terminated. In others, a session may last a very long time even though it is largely inactive. The file service reserves the right to terminate a session at any time that a remote procedure call is not in progress. This might occur if a session remains inactive for a long period, or if the system element supporting the file service has to be shut down.

There may be several sessions simultaneously in existence for the same user whether or not they were established by the same client.

2.3 Files, content, and attributes

The file service stores and operates on *files*. A file is a body of data that has been grouped and provided to the file service for the purpose of short- or long-term storage. Every file is either *temporary* or *permanent*. A permanent file resides in a directory and exists until it is explicitly deleted. A temporary file does not reside in a directory. It exists only until it is closed by all sessions that have opened it.

A file consists of two types of information, content and attributes. The *content* of a file is the data actually contained within the file. Usually, the content is the file's reason for existence. The content is a series of eight-bit bytes, uninterpreted by the file service. The content of a file is obtained or modified only by explicit action.

Attributes are data items that identify the file, describe its content, or are in some other way associated with the file. Attributes vary widely in purpose, structure, and behavior. Some attributes have a particular meaning to the file service, and specifying such an attribute results in a defined behavior in the file service. These attributes are said to be *interpreted*. All other attributes are *uninterpreted*. Such attributes, if specified, are associated with the file and, when requested, are returned unchanged.

Every attribute is identified by its *attribute type*. A number of attribute types are defined by this standard. All attributes having these types must be interpreted by every file service implementing this standard. A client may also define attributes that are useful in its particular application.

A number of procedures accept arbitrary attributes. However, not all attributes are allowed in all contexts. Where an attribute is allowed, and the default behavior when it is unspecified are given in chapter 3.

Attributes may be obtained or modified by explicit action. In addition, attributes are obtained when a directory is listed, and interpreted attributes are modified implicitly by many procedures.

Attributes are described in detail in chapter 4, but in order to give some of the flavor of them, a few are mentioned here.

The *fileID* attribute uniquely names the file within the file service (no other file stored in the file service has the same value of fileID), and may therefore be used to identify the file to file service procedures. This attribute is not human-sensible, and its structure is private to the particular implementation of the file service. For a given file, the value of this attribute remains constant for the duration of a session.

The *name* attribute associates a human-sensible string name with a file. This attribute may be used to identify a file (as in conventional file systems), or it may merely be a descriptive string. Several files may have the same name, even within the same directory. The *version*, also an attribute, is a number that is assigned to the file in such a way that the name-version pair is unique within the file's containing directory.

The *type* attribute is intended to describe the nature of the content or attributes of the file in order to communicate how this file is to be interpreted by potential users of the file. The type of a given file is specified by the client when the file is created. The file service does not enforce any interpretation of the type on the content or attributes. Several frequently-used types are defined in this standard. A client may also assign types of its own.

2.4 Directories

Every file is either a *directory* or a *non-directory*. A directory is a special type of file which can reference other files. A directory also has all of the characteristics of a non-directory in that it can have content and attributes. However, a directory cannot be temporary.

Within a file service, files exist in a hierarchical structure. Every permanent file resides at some level in this hierarchy. The files directly referenced by a directory are its *children*. The *descendants* of a directory include its children and the children of its descendants. The directory which directly references a file is that file's *parent*. The *ancestors* of a file include its parent and the parents of its ancestors. In each file service, there is a *root* file which is the file that has no parent and is an ancestor of all other permanent files.

2.5 Handles and controls

To manipulate a file, a client must *open* that file. The file is then said to be *open within the session*, and will stay open until the session ends or the file is explicitly *closed*. Opening a file marks it as "in use" (so that other clients cannot delete it, for example), and indicates that the file will be used in some way in the near future. Closing a file clears this "in use" mark and indicates that the file will not be used in the near future.

The file to be opened may be specified by giving either its fileID or its name. Since a file's fileID is unique within the file service, no other qualification is necessary. The name-version pair, however, is directory-relative. Therefore, the ID of the parent must also be specified when opening a file with a name-version pair. A file may also be opened by specifying some condition on the attributes of the file.

When a new file is created or an existing file is opened, the file service returns a *handle*. The structure of a handle is private to the implementation of the file service. This handle is presented in subsequent operations to identify this file to the file service, and remains valid until either the session ends or the file is closed using this handle. The handle is relative to the session and so cannot be used in conjunction with any session other than the one used to obtain it.

A client may wish to explicitly specify certain characteristics of its intended interaction with the file. These characteristics are called *controls*. For example, a client may obtain a share lock, specifying that no other clients are allowed to modify the file while it is in use. Or, a client may specify that if the file is in use in a way that conflicts with its own use, it wishes to be notified immediately rather than waiting for access to the file. Controls persist only as long as the handle exists; they are lost when the file is closed.

If a file is opened several times, several handles result. These handles are distinct, and the file remains open within the session until all handles have been presented in requests to close the file. Controls applied to a file are associated with a particular handle. If several handles for the same file exist, a change to the controls of one handle does not affect the others. Also, locks obtained on multiple handles to a file within the same session do not conflict with one another. However, for a session, the effective lock for a file is the most restrictive one obtained for that file within the session.

2.6 Creating, deleting, and accessing files

A number of procedures exist for creating new files, deleting files that are no longer needed, and modifying files in various ways.

A file can be created without storing its content. This is especially useful for the creation of directories. A file can also be created and filled with data transferred to the file service. Finally, a file can be created that is a copy of an existing file.

An existing file may be deleted. The attributes of a file may be accessed or modified, and the content of a file may be accessed, modified, or replaced. In addition, a file may be moved to another directory.

Since directories are also files, all of these procedures may be applied to directories as well as non-directories. When directories are copied, moved or deleted, all descendants are copied, moved or deleted as well.

2.7 Enumerating and locating files in directories

Several procedures enumerate files in a directory, performing some action when files are encountered that satisfy some criteria. The procedures differ in the action taken. If the client *lists* files in a directory, the attributes of all files that satisfy the criteria are furnished to the client. If the client searches for a file, the first file that satisfies the criteria is opened and its handle returned to the client.

The arguments that describe how enumeration is to proceed, and the criteria to be satisfied, are *scopes*. Scopes include the direction of enumeration (front-to-back or back-to-front), a condition on the attributes of the files, and the maximum number of files that may satisfy the condition.

2.8 Serializing and deserializing files

The subtree of files consisting of a particular file and all of its descendants is sometimes a useful entity with which to work, and the file service's procedures are designed to make it easy to operate on such subtrees. However, there are times when it is useful to encapsulate all of the information in such a subtree so that the information can be stored or manipulated outside the file service.

A *serialized file* is a series of eight-bit bytes that encapsulates a file's content, its attributes, and its descendants. The file service provides a procedure that *serializes* a file, producing such a series of bytes, and another procedure that *deserializes* the series of bytes, reconstructing the file's content, attributes, and descendants.

2.9 Transferring data

For those filing procedures that intrinsically require the transmission of a large amount of data, the *Bulk Data Transfer Protocol* [3] is employed. Basically it works as follows: between the call to a remote procedure and the return from that procedure, the sender (either client or service) uses the bulk data transfer mechanism to send to the receiver the attributes or the content of the designated file(s). Note that the *Bulk Data Transfer Protocol* allows the data to be sent to, or retrieved from, a system element different than that of the client.

3. REMOTE PROCEDURES

The Filing Protocol is a Courier-based definition of a file service. It defines the data structures and procedures that constitute the Filing remote program. To be considered a file service, an implementation must implement each of these procedures.

Each procedure description includes a declaration of the procedure in Courier's standard notation, a description of the procedure's arguments and results, and frequently an example of its use. The interaction of these procedures with attributes is described in chapter 4, and the errors these procedures report are described in chapter 5.

The following definition gives the program and version numbers of the Filing Protocol and lists all other Courier-based protocols which are referenced from this program.

```
Filing: PROGRAM 10 VERSION 6 =
BEGIN
    DEPENDS UPON
        BulkData (0) VERSION 1,
        Clearinghouse (2) VERSION 3,
        Authentication (14) VERSION 3,
        Time (15) VERSION 2;
    ...
END.
```

This indicates that Filing is program number 10. This document defines version 6. Filing references some types and constants that are defined in other protocols as shown in the above declaration. These protocols are documented in the *Authentication Protocol* [2], the *Bulk Data Transfer Protocol* [3], the *Clearinghouse Protocol* [5], and the *Time Protocol* [8].

Fine Point: The Filing Protocol definition depends upon the Clearinghouse and Authentication remote program declarations only for types defined by these programs. All of the references to the Clearinghouse and Authentication Protocols within the Filing Protocol are compatible with earlier versions of those protocols.

3.1 Logging on and off

Before making use of a file service, a client must log on. When interaction is complete, it logs off. After logging on, a client is given a session handle which identifies the state of its interaction with the file service.

3.1.1 Credentials

When a client makes use of an application protocol such as Filing, the client is required to present *credentials* to the service. Credentials represent the client's proof of identity and permit the service to identify the initiator of the interaction. Where only a Clearinghouse distinguished name is required to identify the initiator, the standard mechanisms of the *Authentication Protocol* [2] are used. Credentials that resolve a client's identity to a Clearinghouse name are the client's *primary credentials*.

Services implemented on hosts whose authentication requirements go beyond those satisfied by the primary authentication mechanisms of the *Authentication Protocol* may require additional authentication information. This additional authentication information is referred to as *secondary credentials*. A host requiring secondary credentials is a *hybrid* host.

```
Credentials: TYPE = RECORD [
    primary: PrimaryCredentials,
    secondary: SecondaryCredentials];
```

Credentials are used by the client to communicate primary and secondary authentication information to a service. The **primary** portion of **Credentials** is used by the client to supply primary credentials as specified in the *Authentication Protocol*. The **secondary** portion of **Credentials** permits additional secondary authentication information to be supplied.

3.1.1.1 Primary credentials

Primary credentials resolve a client's identity to a Clearinghouse name. This form of authentication information is defined in the *Authentication Protocol* [2]. The definition of **PrimaryCredentials** reflects this relationship.

```
PrimaryCredentials: TYPE = Authentication.Credentials;
nullPrimaryCredentials: PrimaryCredentials = Authentication.nullCredentials;
```

Two levels of authentication security are defined for primary credentials, *simple* and *strong*. Simple credentials encapsulate an identity using straightforward encoding and hashing functions. Encryption is used in creating strong credentials to provide a much greater level of security. Strong credentials make it impossible for one client to impersonate another. Refer to the *Authentication Protocol* documentation for a more complete explanation of these credentials forms. The primary credentials constant **nullPrimaryCredentials** is used to denote that no primary credentials information is being specified.

3.1.1.2 Secondary credentials

Secondary credentials are used to communicate host-specific authentication information. They are similar to primary credentials in that a *strength* corresponding to each primary authentication level is defined. The strength of an instance of secondary credentials is linked to the authentication level of the associated primary credentials.

```
Strength: TYPE = {none(0), simple(1), strong(2)};
SecondaryCredentials: TYPE = CHOICE Strength OF {
    none => [],
    simple => Secondary,
    strong => EncryptedSecondary};
```

SecondaryCredentials defines the data type for secondary authentication information. Secondary credentials of strength **none** are defined to permit a client to avoid the specification of any secondary authentication information if that is appropriate or desired.

A secondary of **simple** strength represents the most basic form of secondary authentication information. The secondary authentication information is encoded in a straightforward way using Courier conventions. No encryption is used (see further details below), therefore information contained in a simple secondary is not immune to eavesdropping threats. A simple secondary may only be specified if the acompanying primary credentials are simple or null.

Secondaries of strength **strong** are used to encapsulate secondary authentication information in a secure manner. Strong secondary credentials are identical to corresponding simple secondary credentials except that the authentication information is encrypted after being encoded using standard Courier conventions. A **strong** secondary may only be specified if the accompanying primary credentials are also strong. The conversation key supplied with the associated primary credentials is used to encrypt the secondary authentication information contained within the strong secondary.

Secondary / Primary	none	simple	strong
nullPrimaryCredentials	legal	legal	illegal
simple	legal	legal	illegal
strong	legal	illegal	legal

Table 3.1 Primary and secondary credentials combinations

Table 3.1 summarizes the valid combinations of primary and secondary credentials information which may appear in **Credentials**.

SecondaryType: TYPE = SEQUENCE 10 OF **SecondaryItemType**;
SecondaryItemType: TYPE = LONG CARDINAL;

The secondary authentication requirements of a hybrid host are described by a value of **SecondaryType**. A secondary type is made up of a set of item types. Individual secondary item types are used to define the structure and interpretation of an element of secondary authentication information. Well-known secondary item types are described in *Secondary Credentials Formats* [10] along with the administrative procedure used to define new item types.

```
Secondary: TYPE = SEQUENCE 10 OF SecondaryItem;
SecondaryItem: TYPE = RECORD [
    type: SecondaryItemType,
    value: SEQUENCE OF UNSPECIFIED];
```

Secondary defines the structure of secondary authentication information. A secondary value comprises a set of secondary items, each designating an item type and a corresponding value of that type. Clients are expected to supply a hybrid host an appropriate set of secondary authentication items. A hybrid service which does not receive the correct set of secondary items indicates the nature of the problem and the secondary item types required by reporting **AuthenticationError** (see §5.3).

EncryptedSecondary: TYPE = SEQUENCE OF Authentication.Block;

EncryptedSecondary is an encrypted form of a secondary. An encrypted secondary is obtained by padding an unencrypted secondary with an appropriate number of zero bits and encrypting using the algorithm outlined in the *Authentication Protocol* (the unencrypted value must be a multiple of 64 bits, hence the zero-padding). The key to be used in the encryption process is the conversation key supplied in the associated strong primary credentials.

3.1.2 Sessions

The **Logon** procedure returns a session handle which is then used as a parameter in calls to almost all other Filing procedures. This structure identifies the state of the client's interaction with the file service.

```
Session: TYPE = RECORD [token: ARRAY 2 OF UNSPECIFIED, verifier: Verifier];

Verifier: TYPE = Authentication.Verifier;
```

token identifies the session to the file service, thereby identifying the user and the state of his interaction with the file service. It does not change during the life of the session and is uninterpretable by the client. **Verifier** is defined by the *Authentication Protocol* [2]. It is included in order to substantiate that all procedure calls using the session handle originated from the same client that established the session and are not replays of previous calls. Note that while the token remains unchanged within a session, the verifier may change with each new call.

3.1.3 Logon

A session is used to access the files of a file service. **Logon** is called to begin a session. The client identifies a particular service to be accessed by supplying the distinguished name of the service as an argument. When an explicit service is not specified (the supplied name is null), the distinguished default service provided by the system element is assumed. In either case, the file service verifies that the request is valid, creates a session, and returns the session handle.

```
Logon: PROCEDURE [
    service: Clearinghouse.Name, credentials: Credentials, verifier: Verifier]
RETURNS [session: Session]
REPORTS [AuthenticationError, ServiceError, SessionError, UndefinedError] = 0;
```

Arguments: **service** is the distinguished name of the service to be accessed with the session; **credentials** identify the client to the service (see §3.1.1); **verifier** is described in the *Authentication Protocol* [2].

Results: **session** is a session handle. **session.token** is to be used in subsequent calls to the file service within this session. If **session.verifier** is a simple verifier, then a simple verifier must be used in all subsequent interactions with the file service within this session.

Example:

A typical log-on call might be the following:

Logon [service: [organization: "Xerox", domain: "Office Systems", object: "TestFS"], credentials: [*credentials-object*], verifier: simpleVerifier]

RETURNS [session: [token: [11B, 27734B], verifier: simpleVerifier]]

The specific value for the **credentials** argument would be of type **Credentials** (which can take several forms). The result of this call is that the client is logged on, a session is created, and a session handle is returned to the client. The **token** of the session handle has the value [11B, 27734B] (the numbers here have no intrinsic meaning; they are provided for illustrative purposes only). The **verifier** argument is either created as defined in the *Authentication Protocol* [2], or obtained from an authentication service. The **verifier** result is used throughout the rest of the session as a means of continuing the authentication of the session. This and the remaining examples denote this verifier value as **simpleVerifier**. The session handle result is used in all subsequent calls to the file service until a logoff request is made.

3.1.4 Logoff

Logoff is called to end a session. The file service verifies that the request is valid, destroys the session, releases any allocated resources, and invalidates the session handle.

Logoff: PROCEDURE [session: Session]
REPORTS [AuthenticationError, ServiceError, SessionError, UndefinedError] = 1;

Arguments: **session** identifies the session to be ended.

Example:

To end the session that was created in the Logon example, the client would make the request:

Logoff [session: [token: [11B, 27734B], verifier: simpleVerifier]]

Notice that the same session handle token is specified that was returned by the **Logon** request. The effect of the logoff for the client is that the session handle is no longer an acceptable argument for file service requests. To obtain another valid session handle, the client must log on again.

3.1.5 Continue

Continue registers interest in a session. A client who wishes a session to remain in existence through some period of inactivity may call **Continue** to prevent the file service from terminating it due to inactivity.

Continue: PROCEDURE [session: Session]
RETURNS [continuance: CARDINAL]
REPORTS [AuthenticationError, SessionError, UndefinedError] = 19;

Arguments: **session** refers to the session that is to be continued.

Results: continuance is in seconds. Under normal conditions, the file service will not terminate the session unless it has been inactive for longer than this number of seconds. The call to Continue, as well as all other remote procedure calls, registers as activity.

Example:

If a client wanted to discover what the timeout period for a file server was, it could make the following request:

Continue [session: [token [11B, 27734B], verifier: simpleVerifier]]

RETURNS [continuance: 600]

The returned value of 600 (seconds) indicates the frequency with which the client must poll or make file service requests to avoid the session being terminated due to inactivity. In this example, to avoid timing out a file service request must be made at least every 10 minutes.

3.2 Opening and closing files

A file must be opened before it can be used. It should be closed when it is no longer needed. While open, a file handle is used to refer to it. The file handle encapsulates the state of the file within the session.

3.2.1 File handles

The file service returns a file handle when a file is opened.

Handle: TYPE = ARRAY 2 OF UNSPECIFIED;

The handle identifies the file to the file service. It is relative to the session. A handle created during one session cannot be used in conjunction with any other session. A handle remains valid until it is explicitly destroyed (by presenting the handle in a request to close or delete the file) or until the session ends, whichever comes first.

nullHandle: Handle = [0, 0];

The constant nullHandle is a reserved value of Handle. In certain procedures where a directory file may be specified, nullHandle is used to imply the *root* file (the file within a file service which has no parent and is an ancestor of all other permanent files).

Specific mention will be made where nullHandle is allowed as a procedure parameter. Unless otherwise indicated, it is disallowed.

The client may hold several handles for the same file in the same session. Each handle is distinct and has its own state; destroying one leaves the others intact. A file is not closed until all handles for it are destroyed.

3.2.2 Opening files

Open makes a file available for use. The attributes identify the desired file. The file service prepares it for use, applies the specified controls, and creates and returns a file handle for the file. The file is also marked "in use" so that it cannot be moved or deleted in other sessions.

```
Open: PROCEDURE [attributes: AttributeSequence, directory: Handle,
    controls: ControlSequence, session: Session]
RETURNS [file: Handle]
REPORTS [AccessError, AttributeTypeError, AttributeValueError,
    AuthenticationError, ControlTypeError, ControlValueError, HandleError,
    SessionError, UndefinedError] = 2;
```

Arguments: **attributes** identifies the file as described below; **directory** specifies a starting directory in which to look for the file (it may be the null handle); **controls** specifies the controls to be applied to the new handle; **session** is the client's session handle. Only the following interpreted attributes may be included in **attributes**:

parentID: the starting directory is the directory which has this **fileID**; if omitted, the starting directory is the root directory. Specifying a **directory** handle is equivalent to specifying its fileID as **parentID** in the attribute list. If both are explicitly specified, the corresponding fileIDs must be equal;

fileID: open the file which has this fileID. If **parentID** or **directory** is specified, the file must be a child of the starting directory (but if neither is specified, the file may be anywhere);

name: open the file which has this name and is a child of the starting directory;

pathname: open the file which has the specified **pathname**. The first component of the pathname must be a child of the starting directory (if the starting directory is omitted, the root directory is used). Every file included in the path except the last must be an accessible directory. The **version** attribute may also be specified, but is ignored if the last file named by **pathname** includes an explicit version specification.

type: open the file with the specified type;

version: open the file which has this version number; if omitted, the file with the highest version is opened.

Uninterpreted attributes are ignored. The attribute sequence may not include more than one of **fileID**, **name**, or **pathname**. If none are present, the root file is opened and **parentID** and **directory** must have null values or be omitted; otherwise an error will be reported. The **version** attribute may only be specified if **name** or **pathname** is specified. In summary, the attribute sequence must be equivalent to one of the following (where optional attributes are designated with brackets):

a) **fileID [parentID] [type]**
b) **name [parentID] [type] [version]**
c) **pathname [parentID] [type] [version]**

Results: **file** is the file handle for the file being opened.

Examples:

If a client wanted to open a file for which it already had the fileID, it might make the request:

```
Open [attributes: [[type: fileID, value: [17B, 33B, 744B, 6B, 225B]]],
    directory: nullHandle,
    controls: [],
    session: [token: [11B, 27734B], verifier: simpleVerifier]]

RETURNS [file: [7244B, 352B]]
```

The file is specified by a **fileID** attribute (**type** = **fileID**) where the fileID was [17B, 33B, 744B, 6B, 225B] (here again the numbers are only illustrative). No controls were specified. The session handle came from a previous **Logon** request (in this case, from the Logon example). The file service returned a file handle, [7244B, 352B], that must be used when accessing this file in other file service requests.

To open a file named "Letters" within the directory opened in the above example, the following remote procedure call could be made:

```
Open [
    attributes: [
        [type: parentID, value: [17B, 33B, 744B, 6B, 225B]],
        [type: name, value: "Letters"]],
    directory: nullHandle,
    controls: [],
    session: [token: [11B, 27734B], verifier: simpleVerifier]]

RETURNS [file: [7247B, 1B]]
```

Here the directory file was specified by a **parentID** attribute, and the file name by a **name** attribute. Again, no controls were specified. The file service located at least one file name "Letters" in the designated directory, opened the one with the highest version number, and returned its file handle. Note that exactly the same effect could be achieved by specifying a value for **directory** instead of specifying a **parentID** attribute, as in the following call:

```
Open [
    attributes: [[type: name, value: "Letters"]],
    directory: [7244B, 352B],
    controls: [],
    session: [token: [11B, 27734B], verifier: simpleVerifier]]
```

The value used for directory was taken from the first Open example.

If the client wished to open a file for which it could provide an access path, it could make the request:

```
Open [
    attributes: [type: pathname, value: "Finance!1/Correspondence! + /Memo!4"],
    directory: nullHandle,
    controls: [],
    session: [token: [11B, 27734B], verifier: simpleVerifier]]

RETURNS [file: [170956B, 3B]]
```

In this example, the client has supplied an access path (pathname) to the file named "Memo". The null value supplied for **directory** implies that the path is relative to the root **directory**. Implicitly, the service searches for a directory within the root named "Finance", the highest version of a directory within "Finance" named "Correspondence", and finally the file itself.

3.2.3 Closing files

Close is called to indicate that a handle to a file is no longer needed in the specified session. The file service releases acquired resources (such as locks associated with the handle) and invalidates the file handle. If the file is temporary and no other file handle exists for it, the file is deleted.

```
Close: PROCEDURE [file: Handle, session: Session]
REPORTS [AuthenticationError, HandleError, SessionError, UndefinedError] = 3;
```

Arguments: **file** is the handle to be closed; **session** is the client's session handle.

3.3 Accessing and modifying controls

A client may specify controls which characterize its intended use of a file handle. Controls may be specified when or after a file is opened. They apply only to a single file handle.

3.3.1 Controls

When a file is opened, a file handle is returned which has some assumed characteristics. For example, possession of a handle by one client prevents other clients from moving or deleting a file, but does not prevent them from reading or modifying the file. This characteristic of a handle is an example of a control. Controls define the nature of file access that a handle gives to the client who holds it.

```
ControlType: TYPE = {lock(0), timeout(1), access(2)};
ControlTypeSequence: TYPE = SEQUENCE 3 OF ControlType;

Control: TYPE = CHOICE ControlType OF {
    lock => Lock,
    timeout => Timeout,
    access => AccessSequence};
ControlSequence: TYPE = SEQUENCE 3 OF Control;
```

Controls may be specified in any procedure that returns a file handle. The specified controls apply only to the returned handle. There exist procedures to obtain and modify controls applying to a specific handle.

3.3.1.1 Locks

A lock on a file is a restriction on the use of the file by other sessions. A client might specify a lock if it wishes to prevent certain types of access to the file while it is operating on it:

Lock: TYPE = **{none(0), share(1), exclusive(2)}**;

An **exclusive** lock is more restrictive than a **share** lock, which is more restrictive than a **none** lock.

If a session has opened a file but no lock has been applied, then other sessions are prevented from moving or deleting the file.

If a session has opened a file and a **share** lock has been applied, then other sessions are prevented from moving or deleting the file, and from acquiring an **exclusive** lock on the file.

If a session has opened a file and an **exclusive** lock has been applied, then other sessions are prevented from moving or deleting the file, and from acquiring a **share** or **exclusive** lock on the file.

The file service acquires the locks that it needs to ensure correct execution of procedures called by the client. It always acquires a **share** lock when a client explicitly reads the content of a file, and an **exclusive** lock when a client explicitly changes the content or attributes of a file, or when children are added to or removed from a directory. Depending on the implementation, it may also acquire other locks as necessary to ensure its own correct operation. Since the file service guarantees it will obtain the locks it requires, the client never needs to explicitly acquire locks unless it wants additional protection. For example, if a client wishes to prevent modification of a file by other sessions during execution of a procedure that reads the file, it need not acquire a lock. The file service acquires a **share** lock and holds it for the duration of the procedure. However, if the client wishes to prevent modification of a file *between* calls to two procedures that read the file, a **share** lock should be obtained before the first procedure is called and released after the second procedure returns.

Locks are maintained on a per-session basis; the lock effectively held by a session is the most restrictive lock held on any handle to a file within the session. For example, if two handles to a file exist in the same session and a **share** lock is applied to one while an **exclusive** lock is applied to the other, then an **exclusive** lock for the file is held by the session. These locks do *not* provide any protection between file accesses made in the same session. The client must provide such protection if it is needed, although the file service will prevent conflicting requests from damaging data (for example, by serializing requests within a session where necessary).

A lock on a file provides no protection for the path to that file. Without specifically protecting the path, it is possible for a separate session to modify an ancestor of a locked file.

If no lock is specified, **none** is assumed.

3.3.1.2 Timeouts

When a client requests a lock that is unavailable, the file service waits until it becomes available or until the timeout expires, whichever occurs first. If the lock becomes available, it is acquired and execution continues. If the timeout expires, an error is reported. The length of the wait is ordinarily an implementation-dependent constant. However, clients who wish to specify a particular value may do so.

```
Timeout: TYPE = CARDINAL;
```

The timeout value is given in seconds. The timeout associated with a handle applies to any request to acquire a lock on that handle. If a timeout of zero is specified, the file service does not wait. In this case if the requested lock is unavailable, an error is immediately reported. Conversely, a very large timeout may cause the file service to wait a very long time for a lock to become available. Such timeouts should be used with care.

If **defaultTimeout** is specified, an implementation-dependent default is applied. When the current timeout value is requested from the file service, the actual timeout value, rather than **defaultTimeout**, is returned:

```
defaultTimeout: Timeout = 177777B;
```

If no timeout is specified, **defaultTimeout** is assumed.

3.3.1.3 Access

Access determines what operations are allowed for a particular file handle. An **Access** is a set of permissions, each of which enables a particular form of access to a file (or its children). If a particular access has not been enabled, the handle may not be used in any operation which would require that access to the file.

```
AccessType: TYPE = {
   -- all files -- read(0), write(1), owner(2),
   -- directories -- add(3), remove(4)};

AccessSequence: TYPE = SEQUENCE 5 OF AccessType;
fullAccess: AccessSequence = [177777B];
```

Each type of access enables particular forms of access to a file as follows:

read	The client may read the file's content and attributes. If the file is a directory, the client may also enumerate its children and search for files in the directory.
write	The client may change the file's content and data attributes, and may delete the file. If the file is a directory, the client may also change environment attributes and access lists of the directory's children.
owner	The client may change the file's access list.
add	If the file is a directory, the client may add children to it (using any of the operations that create files).
remove	If the file is a directory, the client may remove children from it.

The *effective* access available to a client is the logical AND of the access last specified for the handle (with **ChangeControls** or in the operation which returned the handle) and the access allowed by the file's access control list (see §4.2.7). A client's effective access may be empty. The constant **fullAccess**, when specified as a control value, requests that all access permissions be permitted. In operations which return handles, **fullAccess** is assumed if a specification of access mitted.

3.3.2 Accessing controls

GetControls returns the controls in effect for a given handle. Only the values of the specific controls requested are returned. Since different controls may be obtained with varying degrees of difficulty, the client should request only those controls that it needs.

```
GetControls: PROCEDURE [
    file: Handle, types: ControlTypeSequence, session: Session]
RETURNS [controls: ControlSequence]
REPORTS [AccessError, AuthenticationError, ControlTypeError,
    HandleError, SessionError, UndefinedError] = 6;
```

<u>Arguments</u>: **file** i file handle of interest; **types** is a sequence of the types of control items that are desired; n is the client's session handle.

<u>Results</u>: **controls** quence of control items correspon ng one-for-one with the items specified in **types** .

<u>Example</u>:

To obtain the values for **timeout** and **lock** for a file the following request should be made:

```
GetControls [file: [7244B, 352B], types: [timeout, lock],
    session: [token: [11B, 27734B], verifier: simpleVerifier]]

RETURNS [controls: [timeout 60, lock none]]
```

The file handle has a timeout value of one minute and a lock of none. The file service could also have returned the results:

```
[controls: [lock none, timeout 60]]
```

The order of both types and controls is not significant and, in particular, they do not have to match.

3.3.3 Modifying controls

ChangeControls modifies the controls that apply to a given handle. The specified control values are changed. If a lock is specified, the file service attempts to acquire it, and if successful, any prior lock is released.

```
ChangeControls: PROCEDURE [
    file: Handle, controls: ControlSequence, session: Session]
REPORTS [AccessError, AuthenticationError, ControlTypeError,
    ControlValueError, HandleError, SessionError, UndefinedError] = 7;
```

Arguments: **file** is the file handle whose controls are to be modified; **controls** is a sequence of the control items to be set; **session** is the client's session handle.

3.4 Accessing and modifying attributes

Attributes are data that describe a file or are otherwise associated with it. They are obtained when a directory is listed, and may be modified implicitly by many procedures. In addition, they may be obtained or modified by explicit action. Attributes vary widely in purpose, structure, and behavior.

3.4.1 Attributes

An attribute is a data item that is associated with a file. Attributes may identify the file, describe its structure, record historical activity, or perform any other desired function. Some attributes have a particular meaning to the file service and specifying such an attribute results in a defined behavior in the file service. Such attributes are said to be interpreted. All other attributes are uninterpreted. Uninterpreted attributes, when specified, are stored with the file and, when requested, returned unchanged.

```
AttributeType: TYPE = LONG CARDINAL;
AttributeTypeSequence: TYPE = SEQUENCE OF AttributeType;
allAttributeTypes: AttributeTypeSequence = [37777777777B];

Attribute: TYPE = RECORD [
    type: AttributeType, value: SEQUENCE OF UNSPECIFIED];
AttributeSequence: TYPE = SEQUENCE OF Attribute;
```

Attributes may be specified when a file is created and explicitly changed at any time. In addition, some procedures allow certain attributes to be specified. For example, when a file is copied, the resulting file may be given a different name.

Every attribute has an attribute type which identifies the purpose and structure of the attribute. Some attribute types are defined by this standard. All attributes having these defined types *must* be interpreted by every file service. Chapter 4 contains a comprehensive discussion of interpreted attributes. A customer or vendor may also define attributes that are of use in his particular application. Such attributes have types allocated from a range assigned to the customer or vendor.

An attribute's value should be a Courier representation appropriate to the type of the attribute. The file service enforces this for interpreted attributes. For example, an attribute sequence containing a name and a version might appear as follows:

```
[[type: name, value: "Annual Report"], [type: version, value: 1]]
```

Conceptually, every file has a value for every attribute. If the attribute is uninterpreted and it has never been set, or if an interpreted attribute is not meaningful for the file, then the value is [], a zero-length sequence. By convention, this is taken to mean the attribute is not set and the attribute is said to be null. The attribute can be explicitly put in this state by specifying a value of []. The constant **allAttributeTypes**, when a parameter to a procedure that returns the attributes of a file, requests that all attributes that are not null be returned.

Attributes that are zero-length sequences are not returned. However, an attribute whose type has been named in an attribute type sequence is returned, even if it is null.

Many procedures take an **AttributeSequence** as an argument. However, the set of allowed attributes in the sequence varies from procedure to procedure. The restrictions on the various attributes are described for each procedure.

3.4.2 Accessing attributes

GetAttributes returns attributes of the specified file. The file service obtains the requested attributes and returns them to the client. Since different attributes may be obtained with varying degrees of difficulty, the client should request only those attributes that it needs.

```
GetAttributes: PROCEDURE [file: Handle,
    types: AttributeTypeSequence, session: Session]
RETURNS [attributes: AttributeSequence]
REPORTS [AccessError, AttributeTypeError,
    AuthenticationError, HandleError, SessionError, UndefinedError] = 8;
```

Arguments: **file** is a file handle for the file whose attributes are to be examined; **types** is a sequence of the types of attributes that are desired; **session** is the client's session handle.

Results: **attributes** is a sequence of attributes corresponding one-for-one with the items specified in **types** (or containing all non-null attributes if **types** is **allAttributeTypes**).

Access: read access to **file** (or to file's parent).

Examples:

To obtain a file's **name** and **isDirectory** attributes, the following request would be made:

```
GetAttributes [file: [7244B, 352B], types: [name, isDirectory],
    session: [token: [11B, 27734B], verifier: simpleVerifier]]

RETURNS [
    attributes: [
        [type: name, value: "Old Letters"],
        [type: isDirectory, value: TRUE]]]
```

The name of the file turned out to be "Old Letters", and it is a directory-type file. Note that the components of **attributes** could have been returned in either order.

To obtain the uninterpreted attribute whose type is 733B, the following request would be made:

```
GetAttributes [file: [7244B, 352B], types: [733B],
    session: [token: [11B, 27734B], verifier: simpleVerifier]]

RETURNS [attributes: [[type: 733B, value: [] ]]]
```

The result indicates that the value of the attribute was null; that is, it had never been set.

3.4.3 Modifying attributes

3.4.3.1 ChangeAttributes

ChangeAttributes modifies attributes of the specified file. The changes may have other effects on the file depending on the attribute.

```
ChangeAttributes: PROCEDURE [file: Handle,
    attributes: AttributeSequence, session: Session]
REPORTS [AccessError, AttributeTypeError, AttributeValueError,
    AuthenticationError, HandleError, InsertionError, SessionError, SpaceError,
    UndefinedError] = 9;
```

Arguments: **file** is a file handle for the file to be modified; **attributes** is a sequence of the attributes to be modified; **session** is the client's session handle.

Access: write access is required for **file** if only data attributes are changed; write access to **file**'s parent is required for environment attribute changes. If access list attributes are changed, write access to **file**'s parent or owner access to **file** is required as well.

Example:

To change a file's name to "Design Memo", and the value of an uninterpreted attribute type 733B to the two words [644B, 3217B], the following request would be made:

```
ChangeAttributes [file: [7244B, 1B],
    attributes: [
        [type: name, value: "Design Memo"],
        [type: 733B, value: [644B, 3217B]]],
    session: [token: [11B, 27734B], verifier: simpleVerifier]]
```

3.4.3.2 UnifyAccessLists

Access attributes (**accessList** and **defaultAccessList**) may be modified for a given file using **ChangeAttributes**, but it is sometimes necessary or useful to unify the effective access lists of an entire subtree of files. **UnifyAccessLists** is used for this purpose.

```
UnifyAccessLists: PROCEDURE [directory: Handle, session: Session]
REPORTS [AccessError, AuthenticationError, HandleError, SessionError,
    UndefinedError] = 20;
```

Arguments: A handle to the subtree of files whose access lists are to be unified is given by **directory**; **session** is the client's session handle.

Access: Write access is required to **directory**.

The **accessList** and **defaultAccessList** attributes of each descendant file within the subtree rooted by **directory** are given defaulted values. The cumulative result is that all files within the subtree obtain the same effective access controls as those in place for **directory**.

Changes to a file's access list attributes, whether by **ChangeAttributes** or **UnifyAccessLists**, take immediate effect for all handles to the file within the client's session and all new handles acquired by the client's session or other sessions. Access list changes within one session are *not* guaranteed to affect clients of other existing sessions until those sessions end.

3.5 Locating and listing files in directories

A client may examine the files in a directory. Scope information describes the files of interest, and how they are to be examined. Depending on the specific procedure called, either the attributes of files of interest are returned to the client or the first file of interest is opened.

3.5.1 Scopes

Scope items determine what files in a directory are of interest to the client and how they are to be examined. The client may specify: the direction of listing or searching, what files are to be examined, and in the case of listing, the maximum number of files. Scope-type parameters are effective only in the procedure to which they are arguments.

```
ScopeType: TYPE = {count(0), direction(1), filter(2), depth(3)};

Scope: TYPE = CHOICE ScopeType OF {
    count = > Count,
    depth = > Depth,
    direction = > Direction,
    filter = > Filter};

ScopeSequence: TYPE = SEQUENCE 4 OF Scope;
```

3.5.1.1 Count

Count specifies the maximum number of files the client wishes to see.

```
Count: TYPE = CARDINAL;
```

For example, if a directory is being listed and the client specifies a **Count** of five, no more than five sequences of attributes will be returned, even if there are more than five files in the directory. The constant **unlimitedCount** should be used if no restriction is desired (the client wishes to see all files that satisfy the other criteria).

```
unlimitedCount: Count = 177777B;
```

If count is not specified, **unlimitedCount** is assumed. When searching for a file, count is ignored.

3.5.1.2 Depth

Depth specifies to what depth the client wishes descendant files to be considered.

```
Depth: TYPE = CARDINAL;
```

Specifying a **Depth** of one includes only the immediate descendants of the directory being enumerated; a depth of two includes the immediate descendants of directory files referenced by the directory being enumerated. In general, a file is included in the enumeration if fewer than **depth** ancestors separate it from the directory being enumerated. A descendant directory is always considered before its descendants within the enumeration.

```
allDescendants: Depth = 177777B;
```

The constant **allDescendants** should be used if no restriction is desired (the client wishes to consider all descendants).

If no enumeration depth is specified, a **Depth** of one is assumed.

3.5.1.3 Direction

Direction specifies whether enumeration of the directory is to proceed from beginning to end or from end to beginning. The actual order of files is determined by the **ordering** attribute:

```
Direction: TYPE = {forward(0), backward(1)};
```

If the direction is **forward**, enumeration starts with the first file in the ordering. If the direction is **backward**, enumeration starts with the last file. Direction affects both listing (files are listed in the specified direction) and searching (the first encountered file that matches the specified criteria is returned).

If no direction is specified, **forward** is assumed.

3.5.1.4 Filters

Filter specifies a condition that distinguishes files of interest from other files in the directory. The condition is one of: the constants TRUE or FALSE; a relation between an attribute and a constant; a logical combination of conditions.

```
FilterType: TYPE = {
    -- relations --
        less(0), lessOrEqual(1), equal(2),
        notEqual(3), greaterOrEqual(4), greater(5),
    -- logical --
        and(6), or(7), not(8),
    -- constants --
        none(9), all(10),
    -- patterns --
        matches(11)};
```

```
Filter: TYPE = CHOICE FilterType OF {
    less, lessOrEqual, equal, notEqual, greaterOrEqual, greater = >
        RECORD [attribute: Attribute, interpretation: Interpretation],
    and, or = > SEQUENCE of Filter,
    not = > Filter,
    none, all = > RECORD [],
    matches = > [attribute: Attribute]};

Interpretation: TYPE = {none(0), boolean(1), cardinal(2), longCardinal(3),
    time(4), integer(5), longInteger(6), string(7)};
```

A filter whose value is **and** [*filter*$_1$, *filter*$_2$, ..., *filter*$_n$] is satisfied only if all of *filter*$_1$, *filter*$_2$, ..., *filter*$_n$ are satisfied.

A filter whose value is **or** [*filter*$_1$, *filter*$_2$, ..., *filter*$_n$] is satisfied when at least one of *filter*$_1$, *filter*$_2$, ..., *filter*$_n$ is satisfied.

A filter whose value is **not** *filter* is satisfied when *filter* is not satisfied.

A filter whose value is **none** [] is never satisfied, while a filter whose value is **all** [] is always satisfied.

A filter whose value is **matches** [] is satisfied if the corresponding string attribute of a file satisfies the string pattern of the filter. Two wildcard characters are defined: asterisk (*) and sharp sign (#). An asterisk within a string pattern matches zero or more characters within a string attribute; a sharp sign matches any single character. Wildcard characters meant to be interpreted literally within a pattern must be escaped. A wildcard character is escaped by preceding it with the apostrophe character ('). To include the escape character literally in a string pattern, it must be escaped as well.

For example, consider a directory that references five files with the following attributes:

#	name	version
1	Alpha	1
2	Beta	1
3	Beta	2
4	Delta	1
5	Gamma	6

The following filters will select the files mentioned in the comment:

```
matches [attribute: [type: name, value: "B*"]]
    -- files 2 and 3 --

matches [attribute: [type: name, value: "#####"]]
    -- files 1, 4 and 5 --
```

Within a **pathname** attribute, the above wildcard characters may be used to specify string pattern matches of individual pathname components; the wildcard characters are used to match only the name portion of a pathname component. Two consecutive asterisk characters within a wildcarded pathname match multiple components. In both cases, all versions of a file with a given name are considered to match. Explicit version specifications may be included using any of the version designators (see §4.2.2.5). Pathname syntactical characters may be included in pathname filters with appropriate escaping (preceding individual characters with the escape character).

For example, consider a subtree of ten files with the following attributes:

#	name	version
1	Profit-Loss Statements	1
2	Fiscal 1983	1
3	First Quarter	1
4	Second Quarter	1
5	Third Quarter	1
6	Fourth Quarter	1
7	Fourth Quarter	2
8	Fiscal 1984	2
9	First Quarter	1
10	Second Quarter	1

The following filters will select the files mentioned in the comment:

```
matches [attribute: [type: pathname, value: "Profit-Loss Statements/*"]]
    -- files 2 and 8 --

matches [attribute: [type: pathname, value: "Profit-Loss Statements/**"]]
    -- files 2, 3, 4, 5, 6, 7, 8, 9, and 10 --

matches [attribute: [type: pathname, value: "Profit-Loss Statements/**First*"]]
    -- files 3 and 9 --

matches [attribute: [type: pathname, value: "Profit-Loss Statements/**F*!-"]]
    -- files 2, 3, 6, 8, and 9 --
```

All other filters are relations between a constant attribute value and the corresponding attribute of a file. Each of these filters is satisfied if the file's attribute, when interpreted in an appropriate way and compared to the constant value given in the filter, satisfies the specified relation.

The **interpretation** component provides the file service with the information it needs to properly compare the attribute in the file to the constant value. The file service needs this information only for uninterpreted attributes. For attributes that the file service itself interprets the standard interpretation is used, and any specified interpretation is ignored (if the standard interpretation is not one of the values of **Interpretation**, it is assumed to be **none**). Attribute values with the given interpretation are compared as follows:

none	Values are compared word-by-word, starting with the first. That is, corresponding sixteen-bit words are compared as though they were of type CARDINAL, starting with the first, until an unequal pair is found The relationship of this unequal pair is considered to be the relationship of the two attributes. If the attributes are equal up to the length of the shorter, the longer attribute is considered to be greater.
boolean	TRUE is greater than FALSE.
cardinal	Values are compared as unsigned sixteen-bit numbers.

longCardinal	Values are compared as unsigned thirty-two-bit numbers.
time	Values are compared as points in a linear time span where a later time is considered to be greater than an earlier time. Because of the time encoding, this comparison is not the same as for **longCardinal**.
integer	Values are compared as signed sixteen-bit numbers.
longInteger	Values are compared as signed thirty-two-bit numbers.
string	Values are compared according to an implementation-dependent string-sorting algorithm. It is recommended that strings be sorted in a way that allows direct presentation of strings to human users (for example, in alphabetical order) and that essentially equivalent strings (for example, strings that differ only in case) be considered to be equal.

If the value of an attribute is not a valid representation of a value of the stated interpretation, that attribute is considered to be less than any attributes that are valid representations.

If no filter is specified, **nullFilter** is assumed.

nullFilter: Filter = all [];

Example:

Consider a directory that references five files with the following attributes:

#	**name**	**version**
1	Alpha	1
2	Beta	1
3	Beta	2
4	Delta	1
5	Gamma	6

The following filters will select the files mentioned in the comment:

all [] -- *all five of the files* --

none [] -- *none of the five files* --

equal [attribute: [type: name, value: "Beta"], interpretation: string]
-- *files 2 and 3; note that interpretation is ignored* --

greaterOrEqual [attribute: [type: version, value: 2], interpretation: none]
-- *files 3 and 5; note that interpretation is ignored* --

not or [equal [attribute: [type: name, value: "Beta"], interpretation: string],
greater [attribute: [type: version, value: 1], interpretation: cardinal]]
-- *files 1 and 4; note that interpretations are ignored* --

An implementation is not required to support all possible attributes in filters. If a particular value of a filter is not supported then the implementation may report **ScopeValueError** [unimplemented, filter] when that value is specified. However, every implementation must support all possible combinations of relations on the **name**, **position**, and **version** attributes.

3.5.2 Locating files

Find is called to locate and open a particular file in a directory. The file service enumerates the directory's children in the specified direction (the ordering is determined by the **ordering** attribute of the directory) and opens the first file that meets the specified criteria, reporting an error if there is none.

```
Find: PROCEDURE [directory: Handle, scope: ScopeSequence,
    controls: ControlSequence, session: Session]
RETURNS [file: Handle]
REPORTS [AccessError, AuthenticationError, ControlTypeError, ControlValueError,
    HandleError, ScopeTypeError, ScopeValueError, SessionError, UndefinedError] = 17;
```

Arguments: **directory** is a file handle for the directory whose children are to be enumerated (the null handle may be specified); **scope** specifies characteristics of the enumeration and the search criteria; **controls** specifies the controls to be applied to the new handle; **session** is the client's session handle.

Results: **file** is a file handle for the file that was found.

Access: Read access is required to **directory**.

Example:

If one wanted to find in a directory the last occurring file whose **name** attribute is "Notice", the following call would be made:

```
Find [directory: [7244B, 352B],
     scope: [direction backward,
         filter equal
             [attribute: [type: name, value: "Notice"], interpretation: string]],
    controls: [],
    session: [token: [11B, 27734B], verifier: simpleVerifier]]

RETURNS [file: [31B, 6435B]]
```

The scope specifies that the directory is to be searched from the end to the beginning looking for a file whose name equals "Notice". No controls are to be applied. Such a file was found; it was opened and its handle was returned.

3.5.3 Listing files

List enumerates the files in a directory, returning some of their attributes. The file service enumerates the directory in the specified direction (the ordering is determined by the **ordering** attribute of the directory), and sends the requested attributes for files that meet the specified criteria. Since different attributes may be obtained with varying degrees of difficulty, the client should request only the attributes that are needed.

The files in the directory may change while the operation is in progress so that the set of attributes returned may not reflect the state of the directory at any single point in time. The client may prevent such changes, if necessary, by acquiring a share lock on the directory before calling List. Also, the client may call other procedures while listing, but if one of these procedure calls affects the directory being listed, the effects may or may not be reflected in the remainder of the list. Note that if a depth greater than one has been specified, descendants of the directory being listed must also be considered.

```
List: PROCEDURE [directory: Handle, types: AttributeTypeSequence,
    scope: ScopeSequence, listing: BulkData.Sink, session: Session]
REPORTS [AccessError, AttributeTypeError, AuthenticationError,
    ConnectionError, HandleError, ScopeTypeError,
    ScopeValueError, SessionError, TransferError, UndefinedError] = 18;
```

Arguments: directory is a file handle for the directory to be enumerated (the null handle may be specified); types is a sequence of the types of attributes that are desired; scope specifies characteristics of the enumeration and the search criteria; listing specifies the sink that is to receive the requested attributes in accordance with the *Bulk Data Transfer Protocol* [3]; session is the client's session handle.

Access: Read access is required to directory.

The transferred bulk data is a single object of type StreamOfAttributeSequence.

```
StreamOfAttributeSequence: TYPE = CHOICE OF {
    nextSegment (0) = > RECORD [
        segment: SEQUENCE OF AttributeSequence,
        restOfStream: StreamOfAttributeSequence],
    lastSegment (1) = > SEQUENCE OF AttributeSequence};
```

There is one AttributeSequence for each file listed, and each AttributeSequence is a sequence of attributes, corresponding one-for-one with the items specified in types or containing all non-null attributes if allAttributeTypes was specified.

Example:

If a client wants to enumerate the children of a directory backward, obtaining name and version attributes for files whose version attribute is greater than 1, it would make the following call:

```
List [directory: [7244B, 352B], types: [name, version],
    scope:
        [direction backward,
        filter greater
        [attribute: [type: version, value: 1], interpretation: cardinal],
    listing: sampleSink,
    session: [token: [11B, 27734B], verifier: simpleVerifier]]
```

Before List returns, the list of files satisfying the criteria would be sent via bulk data transfer as a StreamOfAttributeSequence from the file service. The destination of the data would be determined by the sampleSink. The data might have the following form:

```
nextSegment [
    segment: [[
        [type: name, value: "Report To Management"],
        [type: version, value: 3]]],
    restOfStream:

        nextSegment [
            segment: [[
                [type: name, value: "Quarterly Performance"],
                [type: version, value: 2]]],
            restOfStream:

                lastSegment [[
                    [type: name, value: "Personnel Summary"],
                    [type: version, value: 2]]]
                ]]
```

3.6 Accessing and modifying the content of files

The content of a file may be set to a value by storing it or replacing it. The content may be obtained by retrieving it.

3.6.1 Uninterpreted file format

Procedures in this section transfer the content of a file using the bulk data transfer mechanism. The transferred data is a single uninterpreted data object consisting of a sequence of eight-bit bytes. The length of the file is exactly the number of bytes transferred

3.6.2 Storing files

Store creates a file with a specified content. When a new file is created with the specified attributes in the specified directory, it is filled with data sent by the client using bulk data transfer, and a file handle for the file is returned.

```
Store: PROCEDURE [directory: Handle, attributes: AttributeSequence,
    controls: ControlSequence, content: BulkData.Source, session: Session]
RETURNS [file: Handle]
REPORTS [AccessError, AttributeTypeError, AttributeValueError,
    AuthenticationError, ConnectionError, ControlTypeError, ControlValueError,
    HandleError, InsertionError,
    SessionError, SpaceError, TransferError, UndefinedError] = 12;
```

Arguments: **directory** is a file handle for the directory into which the new file is to be placed (the null handle may be specified); **attributes** specifies the characteristics of the new file. **controls** specifies the controls to be applied to the returned handle; **content** specifies the source that is to supply the content of the file in accordance with the *Bulk Data Transfer Protocol* [3]; **session** is the client's session handle.

<u>Results</u>: file is a file handle for the newly created file. Between the call to the remote procedure and the return, the file service uses the bulk data transfer mechanism to retrieve the content of the new file.

<u>Access</u>: Add access is required to directory (if it is not the null handle).

<u>Example</u>:

A client wanting to store the data obtained from a source into a file called "Document" in a directory, and to acquire an exclusive lock on the returned file handle, would make the call:

```
Store [directory: [7244B, 352B],
    attributes: [
        [type: name, value: "Document"],
        [type: dataSize, value: 275B]],
    controls: [lock exclusive],
    content: sampleSource,
    session: [token: [11B, 27734B], verifier: simpleVerifier]]
```

Before the procedure returned to the client the file service would retrieve the data from sampleSource using bulk data transfer and store it in the file:

... 275B eight-bit bytes transferred to the file ...

```
RETURNS [file: [71B, 2133B]]
```

The file handle returned has an exclusive lock applied to it.

3.6.3 Retrieving files

Retrieve transfers to the client the content of an existing file.

```
Retrieve: PROCEDURE [file: Handle, content: BulkData.Sink, session: Session]
REPORTS [AccessError, AuthenticationError, ConnectionError, HandleError,
    SessionError, TransferError, UndefinedError] = 13;
```

<u>Arguments</u>: file is a file handle for the file whose content is being retrieved; content specifies the sink that is to receive the content of the file in accordance with the *Bulk Data Transfer Protocol* [3]; session is the client's session handle.

<u>Access</u>: Read access is required to file.

<u>Example</u>:

To reverse the process of the previous example and retrieve a file from the file service, a typical call would be:

```
Retrieve [file: [71B, 2133B],
    content: sampleSink,
    session: [token: [11B, 27734B], verifier: simpleVerifier]]
```

Before the file service returns from the remote procedure call it transfers the requested data via bulk data transfer from the file server to the destinations specified by **sampleSink**:

> ... *275B eight-bit bytes transferred to sampleSink* ...

3.6.4 Replacing files

Replace replaces the content of an existing file with data received from the specified source

```
Replace: PROCEDURE [file: Handle, attributes: AttributeSequence,
    content: BulkData.Source, session: Session]
REPORTS [AccessError, AttributeTypeError, AttributeValueError,
    AuthenticationError, ConnectionError, HandleError,
    SessionError, SpaceError, TransferError, UndefinedError] = 14;
```

Arguments: **file** is a file handle for the file whose content is being replaced; **attributes** specifies characteristics of the resulting file; **content** specifies the source that is to supply the new content of the file in accordance with the *Bulk Data Transfer Protocol* [3]; **session** is the client's session handle.

Access: Write access is required to **file**.

3.6.5 Random access to files

3.6.5.1 Byte ranges

A *byte range* specifies a contiguous sequence of bytes within the content of a file. A range is defined by a byte offset within the content of the file and a count of the bytes in the range.

```
ByteAddress: TYPE = LONG CARDINAL;
ByteCount: TYPE = LONG CARDINAL;
```

A **ByteAddress** specifies a byte offset within the content of a file. A **ByteAddress** is valid for a given file if included in the interval [0..**dataSize**-1], where **dataSize** is the value of the file's **dataSize** attribute as returned by the **GetAttributes** operation. A **ByteCount** is a non-zero count used to specify a number of bytes.

```
endOfFile: LONG CARDINAL = 37777777777B;
```

The constant **endOfFile** is defined for use in referring to the logical end of a file. As a byte address, **endOfFile** is used to refer to the byte position at the end of a file where new data can be appended. As a byte count, **endOfFile** can be used to represent a count of bytes ending with the last byte defined for a file, regardless of the file's exact size.

```
ByteRange: TYPE = RECORD [firstByte: ByteAddress, count: ByteCount];
```

A contiguous sequence of bytes within a file is defined by a **ByteRange**; **firstByte** specifies the starting byte of this sequence; **count** specifies the number of bytes in the sequence.

3.6.5.2 RetrieveBytes

RetrieveBytes allows clients to read a range of bytes within a file. The requested bytes of file content are returned as a result of the call.

```
RetrieveBytes: PROCEDURE [file: Handle, range: ByteRange,
    sink: BulkData.Sink, session: Session]
REPORTS [AccessError, HandleError, RangeError, SessionError, UndefinedError] = 22;
```

Arguments: **file** is a handle to the file whose data is to be read; **range** defines the sequence of bytes to be returned; **sink** specifies the sink that is to receive the requested data bytes in accordance with the *Bulk Data Transfer Protocol* [3]; **session** is the client's session handle.

Access: Read access is required to **file**.

Example:

To obtain the ten bytes of a file's content beginning with its fifteenth byte, the client could make the request:

```
RetrieveBytes [
    file: [4117B, 256B],
    range: [firstByte: 14, count: 10],
    sink: sampleSink,
    session: [token: [11B, 27734B], verifier: simpleVerifier]]
```

The data is transferred by means of bulk data transfer to the specified sink.

3.6.5.3 ReplaceBytes

ReplaceBytes is used to change the content of a file. The operation may be used to overwrite existing data of a file or append new data to a file.

```
ReplaceBytes: PROCEDURE [
    file: Handle, range: ByteRange, source: BulkData.Source, session: Session]
REPORTS [AccessError, HandleError, RangeError, SessionError, SpaceError,
    UndefinedError] = 23;
```

Arguments: **file** is a handle to the file whose data is to be replaced; **range** specifies the region of the file to be written; **source** specifies the source that is to supply the data bytes which are to be used to replace or extend those of the file; **session** is the client's session handle.

Access: Write access is required to **file**.

The **range** argument and the data supplied via **source** must be consistent; otherwise, an error is reported. If the **firstByte** component of **range** is equal to **endOfFile**, the supplied data is appended to the file; otherwise, the supplied data replaces data within the specified byte range of the file. In case of append, **ReplaceBytes** must guarantee that all of the supplied data is appended successfully, or none of it is.

Examples:

To overwrite the first nine bytes of data within a file, a client would make the request:

```
ReplaceBytes[
    file: [4117B, 256B],
    range: [firstByte: 0, count: 9],
    source: sampleSource,
    session: [token: 11B, 27734B], verifier: simpleVerifier]]
```

To append six bytes of new data to a file, a client would make the request:

```
ReplaceBytes[
    file: [4117B, 256B],
    range: [firstByte: endOfFile, count: 6],
    source: sampleSource,
    session: [token: 11B, 27734B], verifier: simpleVerifier]]
```

3.7 Creating and deleting files

A file may be created without transferring any data. Any existing file may be deleted, freeing the resources assigned to the file and removing any association with a directory.

3.7.1 Creating files

Create creates a file. A new file is created with the specified attributes in the specified directory and a handle for the file is returned. **Create** is particularly useful for creating directories. Usually, a non-directory has content, making **Store** a more appropriate operation.

```
Create: PROCEDURE [directory: Handle, attributes: AttributeSequence,
    controls: ControlSequence, session: Session]
RETURNS [file: Handle]
REPORTS [AccessError, AttributeTypeError, AttributeValueError, AuthenticationError,
    ControlTypeError, ControlValueError, HandleError, InsertionError, SessionError,
    SpaceError, UndefinedError] = 4;
```

Arguments: **directory** is a file handle for the directory into which the created file is placed (the null handle may be specified); **attributes** specifies the characteristics of the new file; **controls** specifies the controls to be applied to the returned handle; **session** is the client's session handle.

Results: **file** is a file handle for the newly-created file.

Access: Add access is required to **directory**.

Examples:

To create a temporary file with default values for all attributes, the following call would be made:

```
Create [directory: nullHandle, attributes: [[type: isTemporary, value: TRUE]], controls: [],
    session: [token: [11B, 27734B], verifier: simpleVerifier]]

RETURNS [file: [4661B, 361B]]
```

To create a new directory with a **name** attribute of "Financial Documents" and a **childrenUniquelyNamed** attribute of FALSE, a client would make the call:

```
Create [directory: [7244B, 352B],
    attributes: [
        [type: name, value: "Financial Documents"],
        [type: isDirectory, value: TRUE],
        [type: childrenUniquelyNamed, value: FALSE]],
    controls: [lock share],
    session: [token: [11B, 27734B], verifier: simpleVerifier]]

RETURNS [file: [4661B, 372B]]
```

The resulting file handle has a share lock applied to it.

3.7.2 Deleting files

Delete deletes an existing file. The file is closed and deleted, freeing the resources allocated to the file and removing any association with a directory. If the file is a directory, all descendants are also deleted.

```
Delete: PROCEDURE [file: Handle, session: Session]
REPORTS [AccessError, AuthenticationError, HandleError, SessionError,
    UndefinedError] = 5;
```

Arguments: **file** is a file handle for the file to be deleted (it must be the session's only file handle for this file); **session** is the client's session handle.

Access: Remove access to **file**'s parent is required; write access to **file** (and each descendant).

3.8 Copying and moving files

A file which is identical to an existing file may be created by copying the existing file. The new file may be temporary, or it may be inserted in a directory. An existing file may also be moved to a directory. The file is removed from its old directory if it resided in one.

3.8.1 Copying files

Copy creates a file which is a copy of an existing one. If the existing file has descendants, they are copied as well. The file service creates a set of files which are copies of the specified file and all of its descendants, and inserts the new structure into the specified directory A file cannot be copied into itself or any of its descendants.

```
Copy: PROCEDURE [file, destinationDirectory: Handle, attributes:
    AttributeSequence, controls: ControlSequence, session: Session]
RETURNS [newFile: Handle]
REPORTS [AccessError, AttributeTypeError, AttributeValueError,
    AuthenticationError, ControlTypeError, ControlValueError, HandleError,
    InsertionError, SessionError, SpaceError, UndefinedError] = 10;
```

Arguments: **file** is a file handle for the file to be copied; **destinationDirectory** is a file handle for the directory into which the copy is to be placed (the null handle may be specified); **attributes** specifies the characteristics of the new file and overrides those of the original file; **controls** specifies the controls to be applied to the returned handle; **session** is the client's session handle.

Results: **newFile** is a file handle for the newly-created file.

Access: Add access to **destinationDirectory** is required, and read access to **file** (and each descendant of **file**).

Example:

The following call will copy a file along with any of its descendants, changing its name to "Summary, Section 2":

```
Copy [file: [4661B, 361B],
    destinationDirectory: [7244B, 352B],
    attributes: [[type: name, value: "Summary, Section 2"]],
    controls: [],
    session: [token: [11B, 27734B], verifier: simpleVerifier]]

RETURNS [newFile: [37B, 1627B]]
```

3.8.2 Moving files

Move changes the directory structure of the file service without creating or deleting any files. The file service moves the specified file into the specified directory. If it was previously a child of another directory, it is removed from that directory. If the file was temporary, it becomes permanent. If the file has descendants, they are moved as well (that is, they remain descendants of the file). A file may not be moved into itself or any of its descendants.

```
Move: PROCEDURE [file, destinationDirectory: Handle,
    attributes: AttributeSequence, session: Session]
REPORTS [AccessError, AttributeTypeError, AttributeValueError, AuthenticationError,
    HandleError, InsertionError, SessionError, SpaceError, UndefinedError] = 11;
```

Arguments: **file** is a file handle for the file to be moved (it must be the session's only file handle for this file); **destinationDirectory** is a file handle for the directory into which the file is to be placed (the null handle *may not* be specified); **attributes** modify the characteristics of the file; **session** is the client's session handle.

Access: Read and write access are required to **file**; remove access is required for **file**'s parent and add access is required to **destinationDirectory**.

Example:

To move a file and all of its descendants to a new directory, make the call:

```
Move [file: [37B, 1627B],
    destinationDirectory: [4661B, 372B],
    attributes: [[type: 351B, value: TRUE]],
    session: [token: [11B, 27734B], verifier: simpleVerifier]]
```

This call will change attribute 351B of the file to TRUE in the process.

3.9 Serializing and deserializing files

At times, it is useful to compress all of the information contained in a file and all of its descendants into a series of eight-bit bytes, in order to transfer it to another file service, store it on some other medium, or manipulate it in some other way. The format of data in this series of bytes is the serialized file format. Serializing a file produces a series of bytes which contains all of the information in the file and its descendants, while deserializing such a series of bytes recreates a file and its descendants.

3.9.1 Serialized file format

Procedures in this section transfer a serialized file to a sink or from a source using the bulk data transfer mechanism. The data is a single object of type **SerializedFile**, encoded in its standard representation. A serialized file starts with the version number of the serialized format to distinguish it from other versions of serialized files.

```
SerializedFile: TYPE = RECORD [version: LONG CARDINAL, file: SerializedTree];
```

```
currentVersion: LONG CARDINAL = 3;
```

Each file consists of its attributes, its content, and all of its children. The attribute sequence contains attributes that apply to this file, in arbitrary order. The sequence of children is in the order of the directory.

```
SerializedTree: TYPE = RECORD [
    attributes: AttributeSequence,
    content: RECORD [data: BulkData.StreamOfUnspecified,
        lastByteIsSignificant: BOOLEAN],
    children: SEQUENCE OF SerializedTree];
```

The content of a file is represented as a stream of sixteen-bit words followed by an indication of whether or not the last byte of the last word is significant (that is, whether or not the length in bytes is even). If not, the last byte has the value zero and should be ignored.

3.9.2 Serialize

Serialize encodes all of the information of a file and its descendants into a series of bytes. The file (including its attributes and content) and all descendants are serialized into a series of bytes.

```
Serialize: PROCEDURE [file: Handle, serializedFile: BulkData.Sink, session: Session]
REPORTS [AccessError, AuthenticationError, ConnectionError, HandleError, SessionError,
    TransferError, UndefinedError] = 15;
```

Arguments: file is a file handle for the file which is being serialized; serializedFile specifies the sink that is to receive the serialized file in accordance with the *Bulk Data Transfer Protocol* [3]; session is the client's session handle.

Access: Read access is required to file and each of its descendants.

Example:

To transfer a file in serialized form to another system element, make the call:

```
Serialize [file: [71B, 2133B],
    serializedFile: sampleSink, session: [token: [11B, 27734B], verifier: simpleVerifier]]
```

The file is transferred as a SerializedFile by means of bulk data transfer to the specified sink. It has the following form:

```
[version: 3, file: [
    attributes: [
        [type: checksum, value: 27451676B],
        ... other attributes ...
        [type: modifiedOn, value: 22635230000B]],
    content: [data: lastSegment [ ... 276 bytes ...], lastByteIsSignificant: FALSE],
        children: [] ]]
```

The file was 275 bytes long, and notice that there were no descendants of the specified file.

3.9.3 Deserialize

Deserialize reconstructs a file and its descendants from a serialized representation. A new file is created in the specified directory, its attributes, content and descendants are constructed from the serialized file, and a file handle for the file is returned. During deserialization, some attributes (for example, numberOfChildren) are ignored because the attribute duplicates information that is implicit in the rest of the data. If the deserialized file duplicates an existing file (in name), the deserialized file is created with an appropriate version number. It does not replace the existing file.

```
Deserialize: PROCEDURE [directory: Handle, attributes: AttributeSequence,
    controls: ControlSequence, serializedFile: BulkData.Source, session: Session]
```

```
RETURNS [file: Handle]
REPORTS [AccessError, AttributeTypeError, AttributeValueError,
    AuthenticationError, ConnectionError, ControlTypeError, ControlValueError,
    HandleError, InsertionError, SessionError, SpaceError, TransferError,
    UndefinedError] = 16;
```

Arguments: **directory** is a file handle for the directory into which the file is to be placed (the null handle may be specified); **attributes** specify the characteristics of the new file (overriding corresponding attributes specified in the serialized file); **controls** specifies the controls to be applied to the returned handle; **serializedFile** specifies the source that is to supply the file in accordance with the *Bulk Data Transfer Protocol* [3]; **session** is the client's session handle.

Deserialize ignores attributes in the serialized file that are not allowed to be specified rather than reporting an error. Attributes that are not specified are given default values.

Results: **file** is a file handle for the newly created file.

Access: Add access is required to **directory** (if it is not the null handle).

Example:

To deserialize a serialized file, and in the process, change its name to "Old Letters", the client should make the call:

```
Deserialize [directory: [7244B, 352B],
    attributes: [[type: name, value: "Old Letters"]],
    controls: [],
    serializedFile: sampleSource,
    session: [token: [11B, 27734B], verifier: simpleVerifier]]
```

The serialized file is transferred as a **SerializedFile** via bulk data transfer from the source specified by **sampleSource** to the file service:

```
[version: 3, file:
    [attributes: [
        [type: checksum, value: 0],
        ... other attributes ...
        [type: modifiedOn, value: 22635237112B]],
    content: [data: lastSegment [] , lastByteIsSignificant: TRUE],
    children:
        [attributes: [
            [type: checksum, value: 547375333B],
            ... other attributes ...
            [type: modifiedOn, value: 2263522457B]],
        content: [data: lastSegment [ ... 24B bytes ...], lastByteIsSignificant: TRUE],
        children: [] ]]]

RETURNS [file: [4117B, 256B]]
```

The serialized file was a directory that had one child.

3.10 Procedures and attributes

The tables on the following pages show the effects that the procedures described above have on the interpreted attributes. The tables are in alphabetical order by procedure name. If a procedure never modifies interpreted attributes, no table is given. If an entry in the table is empty, the corresponding attribute is never changed. Otherwise, a brief indication of the change is given. The tables do not attempt to describe the restrictions on specifying various combinations of attributes.

Effects of Filing Operations on Attributes

Table 3.1 **ChangeAttributes**

Attribute	If a Parameter	If not a Parameter	Descendants
accessList	set		
checksum	set		
childrenUniquelyNamed	set		
createdBy	set		
createdOn	set		
dataSize	set		
defaultAccessList	set		
fileID	illegal		
isDirectory	illegal		
isTemporary	illegal		
modifiedBy	illegal	currently logged-in user	
modifedOn	illegal	current date and time	
name	set		
numberOfChildren	illegal		
ordering	set		
parentID			
pathname	illegal	consistent with ancestry	
position	at specified point	if key in parent's ordering changed	if parent's ordering changed
readBy	illegal		
readOn	illegal		
storedSize	illegal		
subtreeSize	illegal		
subtreeSizeLimit	set		
type	set		
uninterpreted	set		
version	set		

Table 3.2 **Copy**

Attribute	If a Parameter	If not a Parameter	Descendants	Dest. Parent	Source File	Source File Descend.
accessList	set					
checksum	illegal					
childrenUniquelyNamed	illegal					
createdBy	illegal					
createdOn	illegal					
dataSize	illegal					
defaultAccessList	set					
fileID	illegal	system-assigned value	system-assigned value			
isDirectory	illegal					
isTemporary	set	**FALSE**	**FALSE**			
modifiedBy	illegal	currently logged-in user	currently logged-in user	currently logged-in user		
modifedOn	illegal	current date and time	current date and time	current date and time		
name	set					
numberOfChildren	illegal			incremented		
ordering	illegal					
parentID	illegal	**fileID** of resulting parent	**fileID** of resulting parent			
pathname	set	consistent with new ancestry	consistent with ancestry			
position	at specified point	depends on parent's ordering	same relative point			
readBy	illegal	""	""		currently logged-in user	currently logged-in user
readOn	illegal	**nullTime**	**nullTime**		current date and time	current date and time
storedSize	illegal					
subtreeSize	illegal			total content in subtree		
subtreeSizeLimit	set					
type	illegal					
uninterpreted	set					
version	set	next available				

Table 3.3 **Create**

Attribute	If a Parameter	If not a Parameter	Parent
accessList	set	set to **[defaulted: TRUE]**	
checksum	set	**unknownChecksum**	
childrenUniquelyNamed	set	implementation dependent	
createdBy	set	currently logged-in user	
createdOn	set	current date and time	
dataSize	set	0	
defaultAccessList	set	set to **[defaulted: TRUE]**	
fileID	illegal	system-assigned value	
isDirectory	set	**FALSE**	
isTemporary	set	**FALSE**	
modifiedBy	illegal	currently logged-in user	currently logged-in user
modifedOn	illegal	current date and time	current date and time
name	set	implementation dependent	
numberOfChildren	illegal	0	incremented
ordering	set	**defaultOrdering**	
parentID	illegal	**fileID** of resulting parent	
pathname	set	consistent with ancestry	
position	at specified point	depends on parent's ordering	
readBy	illegal	""	
readOn	illegal	**nullTime**	
storedSize	illegal	0	
subtreeSize	illegal	0	new total content in subtree
subtreeSizeLimit	set	0	
type	set	**tUnspecified** or **tDirectory**	
uninterpreted	set	null	
version	set	next available	

3.4 Delete

Attribute	Parent
accessList	
checksum	
childrenUniquelyNamed	
createdBy	
createdOn	
dataSize	
defaultAccessList	
fileID	
isDirectory	
isTemporary	
modifiedBy	currently logged-in user
modifedOn	current date and time
name	
numberOfChildren	decremented
ordering	
parentID	
pathname	
position	
readBy	
readOn	
storedSize	
subtreeSize	new total content in subtree
subtreeSizeLimit	
type	
uninterpreted	
version	

3.5 **Deserialize**

Attribute	If a Parameter	If not a Parameter	Descendants	Parent
accessList	set			
checksum	illegal	set appropriately	set appropriately	
childrenUniquelyNamed	illegal			
createdBy	illegal			
createdOn	illegal			
dataSize	illegal			
defaultAccessList	set			
fileID	illegal	system-assigned value	system-assigned value	
isDirectory	illegal			
isTemporary	set	**FALSE**	**FALSE**	
modifiedBy	illegal	currently logged-in user	currently logged-in user	currently logged-in user
modifedOn	illegal	current date and time	current date and time	current date and time
name	set			
numberOfChildren	illegal			incremented
ordering	illegal			
parentID	illegal	**fileID** of resulting parent	**fileID** of resulting parent	
pathname	set	consistent with new ancestry	consistent with new ancestry	
position	at specified point	depends on parent's ordering	same relative point	
readBy	illegal	""	""	
readOn	illegal	**nullTime**	**nullTime**	
storedSize	illegal			
subtreeSize	illegal			new total content in subtree
subtreeSizeLimit	set	**nullSubtreeSizeLimit**		
type	illegal			
uninterpreted	set			
version	set	next available		

3.6 **Move**

Attribute	If a Parameter	If not a Parameter	Source Parent	Destination Parent
accessList	set			
checksum	illegal			
childrenUniquelyNamed	illegal			
createdBy	illegal			
createdOn	illegal			
dataSize	illegal			
defaultAccessList	set			
fileID	illegal			
isDirectory	illegal			
isTemporary	must be **FALSE**	**FALSE**		
modifiedBy	illegal	currently logged-in user	currently logged-in user	currently logged-in user
modifedOn	illegal	current date and time	current date and time	current date and time
name	set			
numberOfChildren	illegal		decremented	incremented
ordering	illegal			
parentID	illegal	**fileID** of resulting parent		
pathname	illegal	consistent with new ancestry		
position	at specified point	depends on parent's ordering		
readBy	illegal			
readOn	illegal			
storedSize	illegal			
subtreeSize	illegal		new total content of subtree	new total content of subtree
subtreeSizeLimit	set			
type	illegal			
uninterpreted	set			
version	set	next available		

Table 3.7 **Open**

Attribute	If a Parameter	If not a Parameter
accessList	illegal	
checksum	illegal	
childrenUniquelyNamed	illegal	
createdBy	illegal	
createdOn	illegal	
dataSize	illegal	
defaultAccessList	illegal	
fileID	file with this value is opened	
isDirectory	illegal	
isTemporary	illegal	
modifiedBy	illegal	
modifedOn	illegal	
name	file with this value is opened	
numberOfChildren	illegal	
ordering	illegal	
parentID	**fileID** of directory to search	
pathname	file with this value is opened	
position	illegal	
readBy	illegal	
readOn	illegal	
storedSize	illegal	
subtreeSize	illegal	
subtreeSizeLimit	illegal	
type	file with this value is opened	
uninterpreted	ignored	
version	file with this value is opened	

3.8 Replace

Attribute	If a Parameter	If not a Parameter	Parent
accessList	illegal		
checksum	set	set appropriately	
childrenUniquelyNamed	illegal		
createdBy	set	currently logged-in user	
createdOn	set	current date and time	
dataSize	initial allocation	number of bytes transferred	
defaultAccessList	illegal	.	
fileID	illegal		
isDirectory	illegal		
isTemporary	illegal		
modifiedBy	illegal	currently logged-in user	
modifedOn	illegal	current date and time	
name	illegal		
numberOfChildren	illegal		
ordering	illegal		
parentID	illegal		
pathname	illegal		
position	illegal		
readBy	illegal		
readOn	illegal		
storedSize	illegal	set appropriately	
subtreeSize	illegal	set appropriately	new total content in subtree
subtreeSizeLimit	illegal		
type	illegal		
uninterpreted	illegal		
version	illegal		

Table 3.9 **ReplaceBytes**

Attribute	File	Parent
accessList		
checksum	**unknownChecksum**	
childrenUniquelyNamed		
createdBy	currently logged-in user	
createdOn	current date and time	
dataSize	increased by extension amount	
defaultAccessList		
fileID		
isDirectory		
isTemporary		
modifiedBy	currently logged-in user	
modifedOn	current date and time	
name		
numberOfChildren		
ordering		
parentID		
pathname		
position		
readBy		
readOn		
storedSize	set appropriately	
subtreeSize	new total content in subtree	new total content in subtree
subtreeSizeLimit		
type		
uninterpreted		
version		

Table 3.10 **Retrieve**

Attribute	File
accessList	
checksum	set if previously unknown
childrenUniquelyNamed	
createdBy	
createdOn	
dataSize	
defaultAccessList	
fileID	
isDirectory	
isTemporary	
modifiedBy	
modifedOn	
name	
numberOfChildren	
ordering	
parentID	
pathname	
position	
readBy	currently logged-in user
readOn	current date and time
storedSize	
subtreeSize	
subtreeSizeLimit	
type	
uninterpreted	
version	

Table 3.11 **RetrieveBytes**

Attribute	File
accessList	
checksum	
childrenUniquelyNamed	
createdBy	
createdOn	
dataSize	
defaultAccessList	
fileID	
isDirectory	
isTemporary	
modifiedBy	
modifedOn	
name	
numberOfChildren	
ordering	
parentID	
pathname	
position	
readBy	currently logged-in user
readOn	current date and time
storedSize	
subtreeSize	
subtreeSizeLimit	
type	
uninterpreted	
version	

Table 3.12 **Serialize**

Attribute	File	Descendants
accessList		
checksum	set if previously unknown	set if previously unknown
childrenUniquelyNamed		
createdBy		
createdOn		
dataSize		
defaultAccessList		
fileID		
isDirectory		
isTemporary		
modifiedBy		
modifedOn		
name		
numberOfChildren		
ordering		
parentID		
pathname		
position		
readBy	currently logged-in user	currently logged-in user
readOn	current date and time	current date and time
storedSize		
subtreeSize		
subtreeSizeLimit		
type		
uninterpreted		
version		

Table 3.13 **Store**

Attribute	If a Parameter	If not a Parameter	Parent
accessList	set	set to [**defaulted: TRUE**]	
checksum	set	set appropriately	
childrenUniquelyNamed	set	implementation dependent	
createdBy	set	currently logged-in user	
createdOn	set	current date and time	
dataSize	initial allocation	number of bytes transferred	
defaultAccessList	set	set to [**defaulted: TRUE**]	
fileID	illegal	system-assigned value	
isDirectory	set	**FALSE**	
isTemporary	set	**FALSE**	
modifiedBy	illegal	currently logged-in user	currently logged-in user
modifedOn	illegal	current date and time	current date and time
name	set	implementation dependent	
numberOfChildren	illegal	0	incremented
ordering	set	**defaultOrdering**	
parentID	illegal	**fileID** of resulting parent	
pathname	set	consistent with ancestry	
position	at specified point	depends on parent's ordering	
readBy	illegal	""	
readOn	illegal	**nullTime**	
storedSize	illegal	set appropriately	
subtreeSize	illegal	set appropriately	new total content in subtree
subtreeSizeLimit	set	**nullSubtreeSizeLimit**	
type	set	**tUnspecified** or **tDirectory**	
uninterpreted	set	null	
version	set	next available	

Table 3.14 **UnifyAccessLists**

Attribute	File	Descendants
accessList		Set to **[defaulted: TRUE]**
checksum		
childrenUniquelyNamed		
createdBy		
createdOn		
dataSize		
defaultAccessList		Set to **[defaulted: TRUE]**
fileID		
isDirectory		
isTemporary		
modifiedBy		currently logged-in user (if changed)
modifedOn		current date and time (if changed)
name		
numberOfChildren		
ordering		
parentID		
pathname		
position		
readBy		
readOn		
storedSize		
subtreeSize		
subtreeSizeLimit		
type		
uninterpreted		
version		

An *attribute* is a data item that is associated with a file. Any information associated with a file, but which is not a part of the file's content, is contained in the file's attributes Attributes may help to identify the file so that it can be distinguished from other files. describe the structure or behavior of the file, record information about certain events in the life of the file, or perform any other desired function.

Every attribute has an *attribute type* (or simply *type*) which identifies the attribute. Certain types are defined in this standard. Other types may be defined by customers or vendors Types to be defined in this way must be allocated by means of the administrative procedures described in Appendix B. A customer or vendor is the *type owner* of attribute types that have been assigned to him.

Every interpreted attribute has a *data type* which can be described in the language of Courier, and every instance of such an attribute should be a well-formed Courier representation of a value of that data type. To promote sharing of information, uninterpreted attributes should also be represented according to Courier conventions. however, this is not mandatory.

Not every attribute is meaningful for all files For example, directory-related attributes have no meaning for files that are not directories. Such attributes may not be specified when they are inappropriate, and are always null when examined

A file service may set an implementation-dependent limit on the total amount of attribute data which may be stored in a single file. This limit must not be less than 32,768 sixteen-bit words. A file service may also set an implementation-dependent limit on the total number of attributes which may be stored in a single file. This limit must not be less than 128 attributes. Clients should not expect to be able to store more than 32,768 words of attribute data, nor more than 128 attributes, on a single file.

4.1 Classes of attributes

Since attributes serve a wide variety of purposes, their behavior varies a great deal. Certain classifications, however, are helpful in pointing out similarities between attributes.

4.1.1 Interpreted vs. uninterpreted

Many attributes have a particular meaning to the file service and specifying such an attribute results in a defined behavior in the file service. These attributes are said to be *interpreted*. All other attributes are *uninterpreted*, or client-defined. The set of interpreted attributes varies among file services. This section specifies attributes that must be interpreted and attributes that cannot be interpreted.

An interpreted attribute has a Courier data type that is defined in a standard such as this one, and all values of the attribute conform to that data type. The file service enforces this constraint. When an interpreted attribute is specified during a procedure call, it results in some defined behavior on the part of the file service, and this behavior may affect other attributes or even other files. The value of an interpreted attribute may change, even when it has not been specified during a procedure call, as a side-effect of that procedure. Various restrictions may be imposed on the use of an interpreted attribute in certain procedures. In general, a client cannot always expect an interpreted attribute to remain unchanged during arbitrary procedure calls.

An uninterpreted attribute should have a defined data type, but the file service does not know what this data type is and, therefore, cannot enforce it. When an uninterpreted attribute is specified during a procedure call, it is stored with the file but causes no other action. In particular, other attributes are unaffected except those that indicate file activity (**modifiedBy**, **modifiedOn**) or position within a parent (**position**). The values of uninterpreted attributes do not change except when they are changed explicitly. Uninterpreted attributes may be passed to any procedure that expects a sequence of attributes. The value of an uninterpreted attribute is always exactly the value to which it was explicitly set by the client.

A file service *must* interpret any attribute type described as interpreted in this standard. A file service *may not* interpret any attribute type that is considered to be uninterpreted by the type owner. In practice, this means that a file service shouldn't ordinarily interpret any attribute type other than those defined in this standard, since the implementor would not normally know whether or not the attribute type's owner considers it to be interpreted.

As a result of these rules, a client is guaranteed that any attribute type described as interpreted in this standard is always interpreted, and that any attribute type considered to be uninterpreted by the type owner is always uninterpreted. In particular, the client's own uninterpreted types are guaranteed to be uninterpreted by any file service. The client would not normally know whether the owner of some other attribute type considers it to be uninterpreted.

4.1.2 Environment vs. data

An environment attribute describes the relationship of a file to its environment, such as its name or parent directory. A data attribute describes aspects of the file that are contained entirely within the file. This distinction is useful because it determines many of the differences in attribute behavior.

A data attribute is tightly bound to the file. Data attributes may be thought of as extensions of the file's content. Data attributes are always carried along when a file is moved, copied, or deserialized. Data attributes may not be explicitly changed during those procedures which change the context in which a file resides but not the file itself. In addition, data attributes may not be used to identify a file when opening it. Examples of data attributes are: **checksum**, **type**, and **numberOfChildren**.

An environment attribute is much more loosely bound to the file. Environment attributes may be thought of as part of the file's parent directory. It is common to want to change these attributes when a file's context changes, as in moving, copying, or deserializing. Some environment attributes may be used to identify a file when opening it. For example, **fileID**, **name**, and **version** are environment attributes. An uninterpreted attribute may be considered to be a data or an environment attribute depending on the client's use of the attribute

4.1.3 Primary vs. derived

A primary attribute is an attribute that carries information for which the attribute is the only source. **name** and **ordering** are primary attributes.

A derived attribute carries information that is derived from other characteristics of the file. For example, **dataSize** records the length of the file's content, and **numberOfChildren** records the number of children in a directory.

4.2 Definition of attributes

The attributes in this standard are defined in a standard format. Certain attributes are related and use common definitions.

4.2.1 How attributes are defined

Each attribute definition includes a description of the meaning and purpose of the attribute, a declaration in Courier notation of the attribute type and of the attribute's data type, a description of the use of the attribute, a declaration in Courier notation of significant values of the attribute, a description of those values, and a statement of where it is legal to specify the attribute.

Several attributes record the date and time at which some event occurred. Time and date values are encoded in conformance with the *Time Protocol* [8].

```
Time: TYPE = Time.Time;
```

Time attributes can be set to null with the value:

```
nullTime: Time = Time.earliestTime;
```

For each date-and-time attribute, there is a corresponding user attribute. This attribute records the name of the user on whose behalf the client was operating when the particular event occurred (the name presented when the session began)

```
User: TYPE = Clearinghouse.Name;
```

User names are always fully-qualified clearinghouse names. These names conform to the conventions described in the *Clearinghouse Protocol* [5].

4.2.2 Identification-related attributes

Identification-related attributes are used to identify the file or to describe major characteristics of the file.

4.2.2.1 fileID

fileID is an environment attribute that unambiguously and uniquely identifies the file within the file service.

fileID: AttributeType = 4;
FileID: TYPE = ARRAY 5 OF UNSPECIFIED;

This attribute names a file within a file service, independent of its parent directory. The value for a given file is guaranteed to remain constant as long as the file remains in the file service. The attribute is human-insensible, and its interpretation is implementation-dependent. In general, clients cannot understand its internal structure. The fact that it is small and fixed in length makes it more convenient than a conventional string name in many applications. The distinguished value, **nullFileID**, of this attribute is never assigned to any file.

nullFileID: FileID = [0, 0, 0, 0, 0];

4.2.2.2 isDirectory

isDirectory is a data attribute that indicates whether the file is a directory or a non-directory. Certain procedures may not be applied to a file that is a non-directory. Directories cannot be temporary files.

isDirectory: AttributeType = 5;
IsDirectory: TYPE = BOOLEAN;

4.2.2.3 isTemporary

isTemporary is an environment attribute that indicates whether the file is temporary or permanent. A temporary file, which can never be a directory, has no parent directory, and it is deleted as soon as all file handles are closed. A permanent file resides in a directory and is not deleted until there is an explicit request to do so.

isTemporary: AttributeType = 6;
IsTemporary: TYPE = BOOLEAN;

4.2.2.4 name

name is an environment attribute that contains a human-sensible string name for the file.

name: **AttributeType** = **9**;
Name: TYPE = STRING;

This attribute may identify a file or it may be merely a description of the file. The name of a file is not necessarily unique within its parent. However, the name-version pair is always unique within its parent. No **name** attribute may have a length of zero, and the length of the Courier representation must not exceed 100 bytes (depending on the character representation in the attribute, the maximum number of characters may be considerably more or less). Capitalization is ignored when names are compared.

It is strongly recommended that file names not contain the characters: apostrophe ('), asterisk (*), sharp sign (#), comma (,), diagonal slash (/), or exclamation point (!). These characters are intended for use within filter specifications and file pathnames (see Appendix F).

4.2.2.5 pathname

pathname is an environment attribute specifying an access path to the file relative to the root file of the service.

pathname: **AttributeType** = **21**;
Pathname: TYPE = STRING;

A *pathname* specifies a hierarchical access path to a file by encoding the **name** and version attributes of a set of the file's ancestors. It describes an access path to the file relative to an assumed *base directory*. The order of the encoded name-version pairs is significant; the first specifies the file's ancestor which is a child of the base directory, the second pair identifies a file whose parent is named by the first, and so forth. The last name and version pair names the file itself. The **pathname** attribute of a file is a pathname which assumes the root file as a base directory.

Name and version portions within a pathname are distinguished by an exclamation character (!). On retrieval, reserved characters within each name portion are escaped by preceding them with an escape character, the apostrophe (') (see §4.2.2.4 for the complete list of reserved characters). As with the **name** attribute, a name portion may not have a length of zero or exceed 100 bytes in the Courier representation of its unescaped form.

A version is specified using a numeric constant, the plus (+) character (designating the highest version of a file), or the minus (-) character (designating the lowest version). If omitted, a designation of highest version is assumed. Each name-version pair is delimited from the next by a diagonal slash character (/).

The following grammar summarizes the syntax of pathname strings:

```
Pathname : = NameVersionPairList

NameVersionPairList : = NameVersionPair | NameVersionPair/NameVersionPairList

NameVersionPair : = Name | Name!Version

Name : = [ string with reserved characters escaped not exceeding 100 bytes in
           unescaped form]
```

Version : = [*string of digits with numeric value in the range (0..65535)*]
| ' + –*i.e. highestVersion*
| '- –*i.e. lowestVersion*

It is recommended that the notation for qualified pathnames defined in Appendix F be used at any human interface with the file service, such as at an administrative console.

4.2.2.6 type

type is a data attribute that describes the nature of the content or attributes of the file in order to communicate to potential users of the file how this file is to be interpreted.

type: **AttributeType = 17;**
Type: TYPE = LONG CARDINAL;

A customer or vendor may define types for files of his own that he wishes to distinguish. Types to be defined in this way must be allocated by means of the administrative procedures described in Appendix B.

The file service interprets neither the type nor the content of a file. In particular, the type may not be used to determine whether a file is a directory or a non-directory. This information is determined by the **isDirectory** attribute.

Several commonly-used type values are defined in this standard. Clients are encouraged to use these types to identify files that have the specified characteristics in order to promote information sharing. However, *the file service does not enforce the specified semantics*. See Appendix B for details.

4.2.2.7 version

version is an environment attribute that distinguishes different files that have the same **name** attribute within the same directory. The name-version pair is always unique across the children of a directory.

version: **AttributeType = 18;**
Version: TYPE = CARDINAL;

This attribute may be specified by the client when a new file is created. Ordinarily, however, it is omitted, and the new file acquires the next version number. If there are files in the specified directory with the same name as the new file, the next version number is one greater than the highest version number associated with any of those files. If there are no such files, the next version number is 1.

If **lowestVersion** or **highestVersion** is specified, the file to be accessed is the one having the specified name and the lowest or highest version number within the directory, respectively:

lowestVersion: Version = 0;
highestVersion: Version = 177777B;

Because an error is reported when the client attempts to create a file with a non-unique name-version pair, a client should not ordinarily specify either **lowestVersion** or **highestVersion** when creating a file. Within a filter, **lowestVersion** and **highestVersion** may be specified only when the order of enumeration (in procedures **Find** or **List**) is by the name attribute.

4.2.3 Content-related attributes

Content-related attributes describe the content of the file.

4.2.3.1 checksum

checksum is a data attribute that helps to verify the validity of the content of the file It is intended to detect file damage that may occur while the file is stored by a file service.

checksum: AttributeType = 0;
Checksum: TYPE = CARDINAL;

The file service computes a checksum whenever the content of a file is transferred This occurs in **Store**, **Retrieve**, **Replace**, **Serialize** and **Deserialize**. When the content is transferred to the file service, the computed value is saved in the **checksum** attribute. If the client has specified a value for the attribute, it is compared to the computed value and an error is reported if there is a mismatch. When the content is transferred from the file service, the computed value is compared with the **checksum** attribute and an error is reported if there is a mismatch.

If the checksum is not known because, for example, the file has been damaged while stored by the file service, the value of the **checksum** attribute is set to **unknownChecksum** The client may also set this value explicitly via **ChangeAttributes** Any computed value of **checksum** is always considered to match **unknownChecksum** A client might explicitly set the value to **unknownChecksum** if the **checksum** attribute does not match the file's content due to file damage and the client wishes to retrieve the file without checksum validation

unknownChecksum: Checksum = 177777B;

The checksum is a one's complement, add-and-cycle checksum that is computed over the sixteen-bit words comprising the file's content. Specifically, the checksum is calculated by initializing it to zero and then, for each successive data word, adding the word (using one's complement addition) and performing a left cycle of the result. If an odd number of bytes is transmitted, a last byte of zero is assumed for purposes of the checksum computation If the result is the ones-complement value minus zero (177777B), it must be converted to plus zero (0B) so it won't be interpreted as the **unknownChecksum** value.

4.2.3.2 dataSize

dataSize is a data attribute that records the number of eight-bit bytes in the content of the file.

dataSize: AttributeType = 16;
DataSize: TYPE = LONG CARDINAL;

This attribute is not intended to describe the total amount of physical space occupied by the file when stored in the file service. For example, it does not include the space required to store attributes, or space overhead required by the file service.

4.2.3.3 storedSize

storedSize is an attribute that records the number of eight-bit bytes occupied by the file when stored in the file service. The attribute includes the space required to store attributes and any other overhead associated with storing the file in the service.

storedSize: AttributeType = 26;
StoredSize: TYPE = LONG CARDINAL;

Note that the value of this attribute will normally be a multiple of a service's underlying unit of allocation.

4.2.4 Parent-related attributes

Parent-related attributes describe a file's relationship to its parent directory. These attributes are always null in a file which has no parent (for example, a temporary file).

4.2.4.1 parentID

parentID is an environment attribute that is equal to the **fileID** attribute of the file's parent.

parentID: AttributeType = 12;
ParentID: TYPE = FileID;

4.2.4.2 position

position is an environment attribute that specifies a file's position within its parent directory. It is used to indicate starting and ending points for listing and locating files in a directory, and to specify the insertion point when creating a file in a directory that is ordered by position (q.v.).

position: AttributeType = 13;
Position: TYPE = SEQUENCE 100 OF UNSPECIFIED;

A position defines a point within the linear span of a directory at which there is at most one file. A file's **position** attribute specifies that file's position within its parent directory.

A position value remains valid even if the file to which it applies is moved or deleted. The position then refers to the point where the file resided. However, a position value is tied to the ordering of the directory into which it points; therefore, it cannot be used after the directory has been reordered (by changing its **ordering** attribute), and it cannot be used to specify a position within any other directory.

A position value is uninterpretable by the client. Its internal structure is implementation-dependent. Because the file service may embed arbitrary information in the position, the client may not compare positions, even for equality.

Two special values identify distinguished points within a directory. The constant **firstPosition** specifies a point before the first file in the directory, and **lastPosition** specifies a point after the last file. "First" and "last" are determined by the directory's ordering

firstPosition: Position = [0];
lastPosition: Position = [177777B];

4.2.5 Event-related attributes

Event-related attributes record the date and time of significant events in the life of a file, and the name of the user on whose behalf an event occurred.

For performance reasons, the file service does not necessarily change these times and names exactly when the related event occurs. Rather, it may cache changes for later application, or may group several changes together. The file service guarantees that if an event occurs during a session, then the times and names will be updated appropriately sometime during that session. The file service also guarantees that explicitly-requested changes to times and names, where allowed, occur immediately.

4.2.5.1 createdBy

createdBy is a data attribute that records the creator of the file's content. It is the name of the user who last modified the content of the file.

createdBy: AttributeType = 2;
CreatedBy: TYPE = User;

If the client does not specify this attribute during **Create**, **Store** or **Replace**, the file service will set it to the name of the current user. However, since the attribute is intended to be the name of the creator of the content of the file (rather than the physical file itself), it is strongly recommended that all clients maintain this name with the file and specify it when transferring the file to a file service.

4.2.5.2 createdOn

createdOn is a data attribute that records the time of creation of the file's content. This attribute is used to maintain the generation time of the file in order to determine the relative ages of similar files.

createdOn: AttributeType = 3;
CreatedOn: TYPE = Time;

If the client does not specify this attribute during **Create**, **Store** or **Replace**, the file service will set it to the current date and time. However, since the attribute is intended to be the time of creation of the *content* of the file (rather than the physical file itself), it is strongly recommended that all clients maintain this time with the file and specify it when transferring the file to a file service.

4.2.5.3 readBy

readBy is a data attribute that records the name of the user who last examined the content of the file.

readBy: AttributeType = 14;
ReadBy: TYPE = User;

When a new file is created, this attribute is set to empty strings to indicate that the file has never been read. Subsequently, the file service maintains the attribute.

4.2.5.4 readOn

readOn is a data attribute that records the time at which the content of the file was last examined.

readOn: AttributeType = 15;
ReadOn: TYPE = Time;

When a new file is created, this attribute is set to **nullTime** to indicate that the file has never been read. Subsequently, the file service maintains the attribute.

4.2.5.5 modifiedBy

modifiedBy is a data attribute that records the name of the last user who changed the file in any way.

modifiedBy: AttributeType = 7;
ModifiedBy: TYPE = User;

When a new file is created, this attribute is set to the name of the current user. Subsequently, the file service maintains the attribute.

4.2.5.6 modifiedOn

modifiedOn is a data attribute that records the time at which the file was last changed in any way.

modifiedOn: AttributeType = 8;
ModifiedOn: TYPE = Time;

When a new file is created, this attribute is set to the current time. Subsequently, the file service maintains the attribute.

4.2.6 Directory-related attributes

Directory-related attributes describe certain aspects of directories. In non-directories, directory-related attributes are always null.

4.2.6.1 childrenUniquelyNamed

childrenUniquelyNamed is a data attribute that specifies whether the children of this directory are constrained to have distinct **name** attributes. The default value of this attribute is implementation dependent.

```
childrenUniquelyNamed: AttributeType = 1;
ChildrenUniquelyNamed: TYPE = BOOLEAN;
```

When this attribute is TRUE, no two children of the directory may have the same **name** attribute, and the file service rejects any attempt to add a file with the same **name** attribute as an existing file. When this attribute is FALSE, this restriction is not enforced. Files that have the same **name** attribute are distinguished by their **version** attributes. Comparison of **name** attributes is described in §3.5.1.4.

4.2.6.2 numberOfChildren

numberOfChildren is a data attribute that is a count of the directory's children. Note that this is not a count of the directory's descendants.

```
numberOfChildren: AttributeType = 10;
NumberOfChildren: TYPE = CARDINAL;
```

4.2.6.3 ordering

ordering is a data attribute that specifies the order of enumeration of files in the directory during certain procedures.

```
ordering: AttributeType = 11;
Ordering: TYPE = RECORD [
    key: AttributeType, ascending: BOOLEAN, interpretation: Interpretation];
```

Except when ordering by position (described below), the placement of files in a directory is determined by the relative values of a particular attribute. The component **key** specifies which attribute is to be used for ordering. **ascending** determines whether ordering is to be in ascending order of the attribute, and **interpretation** specifies how the file service should interpret the attribute if there is not a standard interpretation for the attribute. For example, a value of

```
[key: createdOn, ascending: FALSE, interpretation: time]
```

specifies that when the directory is listed, the first file to be delivered should be the one that has the highest (most recent) **createdOn** value.

When the attribute used for ordering is an uninterpreted one, the interpretation to be used must be specified so that the file service can determine the relative placement of files. If a file's attribute value is not a valid Courier representation of the type specified in **interpretation**, then it is placed before those files that do have valid values. When the attribute is an interpreted one, the interpretation specified in the **ordering** attribute is ignored; the file service uses the standard interpretation for the attribute. The comparison rules for various interpretations are described in §3.5.1.4. Interpreted attributes with standard interpretations other than those defined in this section are ordered as though the interpretation were **none**.

The behavior of a directory is somewhat different when the specified key is the **position** attribute. In all other cases, the relative placement of files is determined entirely by the value of the specified attribute. When ordering is by position, however, the relative placement of files is explicitly determined by the client. When adding a file to the directory, the client specifies the position at which it would like the file to reside. The following constants specify ordering by position:

```
byAscendingPosition: Ordering = [
    key: position, ascending: TRUE, interpretation: none];
byDescendingPosition: Ordering = [
    key: position, ascending: FALSE, interpretation: none];
```

If ordering is by ascending position, a file that is added without specifying its position is placed at the end of the directory. If ordering is by descending position, a file that is added without specifying its position is placed at the beginning. Otherwise, there is no difference between these values.

When the ordering of a directory is changed to an ordering by position, the relative placement of files in the directory *is not affected*. In other words, when ordering by position, the files are initially placed according to their placement in the previous ordering. Subsequent additions need not conform to the previous ordering.

After a number of additions at the same point within a directory ordered by position, the density of files in that area may become too great to allow further additions. When this condition occurs, a procedure attempting to insert a file reports:

InsertionError [problem: positionUnavailable]

The client should call **ChangeAttributes** specifying an ordering that is the same as the current ordering. This action redistributes the files without changing their relative placement.

When no ordering is specified during creation of a new directory, **defaultOrdering** is used. When the **ordering** attribute has this value, or the corresponding value with **ascending** equal to FALSE, the ordering is actually based on ascending or descending values of first, the **name** attribute, and second, the **version** attribute, rather than just the name alone:

```
defaultOrdering: Ordering = [key: name, ascending: TRUE, interpretation: string];
```

File service implementations are not required to support all possible values of this attribute; however, every file service implementation must support **defaultOrdering**.

4.2.6.4 subtreeSize

subtreeSize is a data attribute that records the number of eight-bit bytes occupied by a directory and all its descendants.

subtreeSize: AttributeType = 27;
SubtreeSize: TYPE = LONG CARDINAL;

This attribute is equivalent to the sum of the **storedSize** attributes of a directory and each of its descendants.

4.2.6.5 subtreeSizeLimit

subtreeSizeLimit records the maximum number of eight-bit bytes which may be allocated to a directory and all files it directly or indirectly contains.

subtreeSizeLimit: AttributeType = 28;
SubtreeSizeLimit: TYPE = LONG CARDINAL;

This attribute is equivalent to the sum of the **storedSize** attributes of a directory and each of its descendants. An operation is rejected if it would cause the value of a directory's subtree size to exceed the limit specified by the directory's **subtreeSizeLimit** attribute. The client is permitted to change the value of a directory's **subtreeSizeLimit** attribute at any time even if this would cause it to obtain a value less than the current value of the directory's **subtreeSize** attribute.

nullSubtreeSizeLimit: SubtreeSizeLimit = 37777777777B;

When a directory is created and no **subtreeSizeLimit** is specified, **nullSubtreeSizeLimit** is assumed. This value is used to specify that a directory has no cumulative limit on the amount of physical space the directory and its descendants may require.

4.2.7 Access-related attributes

Access-related attributes are used to control access to a file, its content and attributes, and descendants. Every file has an **accessList** attribute, which may be defaulted. In addition, each directory file has a **defaultAccessList** attribute, which specifies the access for files within the directory having explicitly defaulted access lists. The ability to modify a file's access attributes is subject to the access granted the client by the access list in effect for the file.

When a file is created, it receives defaulted values for its access lists or those specified by the client, if supplied. When a file is inserted into a directory, the file receives access lists as specified by the client; if an access list or default access list is not specified during the insertion, the respective access list remains unchanged. Access lists of descendants of the inserted file are not affected by the insertion.

When the access list of a file must be determined, the **accessList** attribute of the file is consulted. If this value has been defaulted, then the **defaultAccessList** attribute of the file's parent directory is retrieved. If the **defaultAccessList** attribute of the parent is defaulted, the parent's access list is used. The method of determining the access list of the parent is the same as for the original file; this process proceeds recursively until a non-defaulted list is encountered or the root directory is reached.

4.2.7.1 accessList

accessList is an environment attribute that specifies the access permissions to be granted to particular clients. Each enabled permission permits particular types of access to the specified client. Clients not represented in the access list of a file are denied any access to the file. The access granted a particular client with respect to a file is the union of the permissions specified in all entries containing a key *representing* the client.

```
accessList: AttributeType = 19;
AccessList: TYPE = RECORD [entries: SEQUENCE OF AccessEntry, defaulted: BOOLEAN];

AccessEntry: TYPE = RECORD [key: Clearinghouse.Name, access: AccessSequence];
```

An access list is comprised of a set of key/access permission pairs. If a session's user can be identified with the **key** portion of an entry, then the permissions specified by the entry are granted to the session. During retrieval, the **defaulted** component of a file's **accessList** attribute specifies whether the attribute was explicitly set with the file or was defaulted to that of its parent. On input, the **defaulted** component is used to explicitly default the value of an access list attribute, and in that case, **entries** must be empty.

The **key** portion of an individual **AccessEntry** will typically denote the name of a user or group of users defined via the *Clearinghouse* [5]. A limited form of wildcarding is also permitted within the **key** of an access list entry with the use of the asterisk character (*). A wildcard may replace the object portion, the object and domain portions, or all three portions of a **key**. These specifications imply respectively: all users within a domain and organization; all users within all domains of an organization; and all clients.

If the access list for a file has no entries, it is said to be empty and no access to the file is allowed to any client. If the **accessList** attribute of a file is explicitly defaulted, access to the file is determined by the **defaultAccessList** attribute of the file's parent directory (see below). See §3.3.1.3 for an explanation of the access permissions.

Example:

The following access list specifies **read** and **write** access to the user "John Q. Public" of the "Office Systems" organization within "Xerox"; **read** access for the group of users designated by the Clearinghouse group "UserGroup1"; **add** access for the user whose alias is "UserAlias1"; and **read** access to any user within the "Xerox" organization.

```
AccessList [entries: [
    [key: ["John Q. Public", "Office Systems", "Xerox"], access: [[read], [write]]],
    [key: ["UserGroup1", "Office Systems", "Xerox"], access: [[read]]],
    [key: ["UserAlias1", "Office Systems", "Xerox"], access: [[add]]],
    [key: ["*", "*", "Xerox"], access: [[read]]]],
```

4.2.7.2 defaultAccessList

defaultAccessList is an environment attribute that applies only to directories. This attribute specifies the access controls for files within the directory which have explicitly defaulted

accessList values. If the **defaultAccessList** of a directory is given the defaulted value, then the directory's **accessList** value is used instead.

```
defaultAccessList: AttributeType = 20;
DefaultAccessList: TYPE = AccessList;
```

When a remote procedure completes successfully, it returns results as specified in the definition of the procedure. However, conditions can arise before or during execution of the procedure that make successful completion of the request impossible. For example, the client may have specified incorrect arguments in a remote procedure call, or some required resource may be unavailable.

When such conditions occur, an error is reported to communicate to the client the nature of the problem. Each error encompasses an entire class of possible conditions and the specific problem is further described by the arguments of the error. For example, **HandleError** indicates that something is wrong with a file handle specified in the arguments of a procedure. The particular problem with the file handle is specified by the argument which is of type **HandleProblem**.

When an exceptional condition arises *during* execution of a remote procedure, the file service makes every effort to undo the effects of the partial execution so that the file service appears to the client as though the procedure had never been called. However, the file service does not guarantee that such effects can always be reversed. Therefore, when an error is reported, the client must be prepared for the possibility that the procedure was partially executed. In any event, no files are lost unless deletion was requested.

Each error definition includes a declaration of the error in Courier notation, a description of its arguments, and examples of conditions that cause the error to be reported.

5.1 Access errors

AccessError may be reported by any procedure that requires access to a file. It indicates that access to the file is not possible. The inaccessible file is not necessarily the one whose handle was specified as an argument to the procedure call because some procedures operate on additional files. For example, **Delete** deletes the descendants of a specified file as well as the file itself.

AccessError: ERROR [**problem**: **AccessProblem**] = **6**;

The argument **problem** describes the problem in greater detail.

AccessProblem: TYPE = {

accessRightsInsufficient(0), -- *the user does not have the access rights needed to satisfy the request; consult description of individual procedures for specific requirements* --

accessRightsIndeterminate(1), -- *the file service could not determine whether the user has the access rights needed to satisfy the request; consult description of individual procedures for specific requirements --*

fileChanged(2), -- *while the procedure was executing, the file changed in such a way that execution could not continue; this condition can occur during List if the ordering of the directory changes --*

fileDamaged(3), -- *a file was found to be internally damaged in some way, but not badly enough to require shutdown of the file service --*

fileInUse(4), -- *even after expiration of the timeout, the file service could not acquire a lock it needed to satisfy the request --*

fileNotFound(5), -- *a file was not found in the context in which it was expected --*

fileOpen(6)}; -- *during an attempt to move or delete a file, another file handle for the file was found to exist in the same session --*

Examples:

If one session calls **Retrieve** while another session is calling **Replace** for the same file, and the timeout on the **Retrieve** procedure expires before the **Replace** procedure completes, the following error is reported:

AccessError [problem: fileInUse]

If **List** is called and during the execution of **List** some other session changes the **ordering** attribute of the directory being listed, the following error is reported:

AccessError [problem: fileChanged]

If **Open** is called specifying a **parentID** and a **name** and there is no file with that name in the directory identified by **parentID**, the following error is reported:

AccessError [problem: fileNotFound]

If **Open** is called specifying a **fileID** and there is no file in the file service with that **fileID**, the following error is reported:

AccessError [problem: fileNotFound]

If **Delete** is called specifying a file handle for a directory and the client has opened but not yet closed a descendant of that directory, the following error is reported:

AccessError [problem: fileOpen]

If **Delete** is called specifying a file handle for a directory and *the client of another session* has opened but not yet closed a descendant of that directory, the following error is reported:

AccessError [problem: fileInUse]

5.2 Argument errors

There are argument errors for each class of Filing procedure argument: attributes, controls, and scopes. A given argument error may be reported by any procedure that has an argument of the corresponding type. Each class contains two errors. The type-related error indicates that specifying that attribute type resulted in a problem; the value-related error indicates that the attribute type was legitimate, but the specified value caused a problem.

AttributeTypeError is reported whenever the attribute type specified in an **AttributeTypeSequence** or an **AttributeSequence** causes some kind of problem. **AttributeValueError** is reported whenever the attribute value specified in an **AttributeSequence** causes a problem. The argument **type** indicates the offending attribute type or the type of the offending attribute value.

```
AttributeTypeError: ERROR [
    problem: ArgumentProblem, type: AttributeType] = 0;

AttributeValueError: ERROR [
    problem: ArgumentProblem, type: AttributeType] = 1;
```

ControlTypeError is reported when a control type specified in a **ControlTypeSequence** or **ControlSequence** causes a problem. **ControlValueError** is reported when a control value specified in a **ControlSequence** causes a problem. The argument **type** indicates the offending control type or the type of the offending control value.

```
ControlTypeError: ERROR [
    problem: ArgumentProblem, type: ControlType] = 2;

ControlValueError: ERROR [
    problem: ArgumentProblem, type: ControlType] = 3;
```

ScopeTypeError is reported when a scope type specified in a **ScopeSequence** causes a problem. **ScopeValueError** is reported when a scope value specified in a **ScopeSequence** causes a problem. The argument **type** indicates the offending scope type or the type of the offending scope value.

```
ScopeTypeError: ERROR [
    problem: ArgumentProblem, type: ScopeType] = 4;

ScopeValueError: ERROR [
    problem: ArgumentProblem, type: ScopeType] = 5;
```

In all of the above errors, the argument **problem** describes the problem in greater detail.

```
ArgumentProblem: TYPE = {

    illegal(0), -- this value is never allowed; this condition can only occur for attribute
        values --

    disallowed(1), -- this type or value is sometimes allowed, but is never allowed by this
        remote procedure; this condition can occur for attribute types and values --
```

unreasonable(2), -- *this type or value is sometimes allowed by this procedure, but not in the context in which it was supplied; for example, it may conflict with other arguments; this condition can occur for attribute types and values* --

unimplemented(3), -- *this type or value is not supported by this implementation of the file service; this condition can only occur for certain values of the* **filter** *scope and the* **ordering** *attribute, but never occurs for types* --

duplicated(4), -- *this type is specified more than once in a sequence; this condition never occurs for values* --

missing(5)}; -- *this type or value is missing in a context in which it is required; this condition can occur for certain attribute types in Open* --

Examples:

If the **name** attribute is specified in **Create** and the specified string contains a character that is illegal in names, the following error is reported:

AttributeValueError [problem: illegal, type: name]

If the **dataSize** attribute is specified in **Copy**, the following error is reported:

AttributeTypeError [problem: disallowed, type: dataSize]

If the **ordering** attribute is specified in **Create** and the **isDirectory** attribute has not been specified (and therefore defaults to FALSE), the following error is reported:

AttributeTypeError [problem: unreasonable, type: ordering]

If a filter based on the value of the **fileID** attribute is specified in **List** and the file service does not support this value of filter, the following error is reported:

ScopeValueError [problem: unimplemented, type: filter]

If the lock control is specified twice in the sequence of controls that is an argument to **Store**, the following error is reported:

ControlTypeError [problem: duplicated, type: lock]

If no attributes are specified in **Open**, the following error is reported:

AttributeTypeError [problem: missing, type: fileID]

If only the **version** attribute is specified in **Open**, the following error is reported:

AttributeTypeError [problem: missing, type: name]

5.3 Authentication errors

AuthenticationError may be reported by any procedure. The most common occurance of this error is in response to a **Logon** operation. The service may detect some problem with the client's primary or secondary credentials. Later in the interaction, **AuthenticationError** is used to report a problem with the authentication verifier contained in the session handle.

```
AuthenticationError: ERROR [
    problem: AuthenticationProblem, type: SecondaryType] = 7;
```

The argument **problem** describes the problem in greater detail. The argument **type** describes the format of secondary credentials information expected by the service. Details of specific secondary types is documented in *Secondary Credentials Formats* [10]. Where no interpretation for the **type** field is indicated, this error argument has the null value and should be ignored by the client.

```
AuthenticationProblem: TYPE = {

    primaryCredentialsInvalid(0), -- decryption failed or Clearinghouse name was invalid --

    verifierInvalid(1), -- decryption failed or simple password was invalid --

    verifierExpired(2), -- a strong verifier was too old --

    verifierReused(3), -- service has either seen the same strong verifier before or one
        generated more recently --

    primaryCredentialsExpired(4), -- expiration date and time of the supplied primary
        credentials has been exceeded --

    inappropriatePrimaryCredentials(5), -- primary credentials were not of the appropriate
        strength (strong may be required where simple were supplied) --

    secondaryCredentialsRequired(6), -- secondary authentication information required but
        none was supplied; type indicates the type of secondary credentials required --

    secondaryCredentialsTypeInvalid(7), -- the type of the supplied secondary credentials
        was incorrect or the secondary credentials value was improperly formatted for
        the specified type; type indicates the type of secondary credentials required --

    secondaryCredentialsValueInvalid(8)}; -- the specified secondary authentication
        information was not acceptable to the service; type indicates the type of secondary
        credentials required --
```

Examples:

If the service requires that clients supply strong primary credentials and only simple credentials are supplied, the following error is reported:

AuthenticationError [problem: inappropriatePrimaryCredentials, type: []]

If a client specifies null primary credentials with a strong secondary credentials value in **Logon**, the following error is reported:

AuthenticationError [problem: inappropriatePrimaryCredentials, type: []]

If a file service requires its clients to supply secondary authentication information and none is supplied during a **Logon** request, the following error is reported:

AuthenticationError [problem: secondaryCredentialsRequired, type: [2B, 3B]]

Note that the **type** argument in the error report indicates the expected format of the required secondary credentials information.

If a client supplies secondary credentials in a call to **Logon**, but their format is not that required by the service, the following error is reported:

AuthenticationError [problem: secondaryCredentialsTypeInvalid, type: [2B, 3B, 8B]]

Note that the **type** argument in the error report indicates the expected format of the required secondary credentials information.

5.4 Connection errors

ConnectionError may be reported by any procedure that takes an argument of type **BulkData.Source** or **BulkData.Sink**. It indicates that there is a problem with establishing the connection for transferring the bulk data.

ConnectionError: ERROR [problem: ConnectionProblem] = 8;

ConnectionProblem: TYPE = {

-- *communication problems* --

noRoute(0), -- *no route to the other party could be found* --
noResponse(1), -- *the other party never answered* --
transmissionHardware(2), -- *some local transmission hardware was inoperable* --
transportTimeout(3), -- *the other party responded but later failed to respond* --

-- *resource problems* --

tooManyLocalConnections(4), -- *no additional connection is possible* --
tooManyRemoteConnections(5), -- *the other party rejected the connection attempt* --

-- *remote program implementation problems* --

missingCourier(6), -- *the other party had no Courier implementation* --
missingProgram(7), -- *the other party did not implement the bulk data program* --
missingProcedure(8), -- *the other party did not implement the procedure* --
protocolMismatch(9), -- *the two parties have no Courier version in common* --
parameterInconsistency(10), -- *a protocol violation occurred in parameters* --
invalidMessage(11), -- *a protocol violation occurred in message format* --
returnTimedOut(12), -- *the procedure call never returned* --

-- *miscellaneous* --

otherCallProblem(177777B) }; -- *some other protocol violation during a call* --

5.5 Handle errors

HandleError may be reported by any procedure that takes an argument of type **Handle**. It indicates that there is a problem with the specified file handle.

HandleError: ERROR [**problem**: **HandleProblem**] = **9**;

The argument **problem** describes the problem in greater detail.

HandleProblem: TYPE = {

invalid(0), -- *an invalid file handle was specified; it may be an obsolete handle in the current session or it may be a valid file handle in another session* --

nullDisallowed(1), -- *the null handle was specified as a value for an argument that requires a valid handle to a file* --

directoryRequired(2)}; -- *the null handle or a handle to a non-directory was specified as a value for an argument that requires a handle to a directory* --

Examples:

If a handle to a non-directory is specified as the value of **destinationDirectory** in **Move**, the following error is reported:

HandleError [problem: directoryRequired]

If a null handle is specified as the value of **file** in **Copy**, the following error is reported:

HandleError [problem: nullDisallowed]

5.6 Insertion errors

InsertionError may be reported by any procedure that inserts a file into a directory whether the file being inserted is a new file or is being moved from somewhere else. It indicates that the directory could not accommodate the file.

InsertionError: ERROR [**problem**: **InsertionProblem**] = **10**;

The argument **problem** describes the problem in greater detail.

InsertionProblem: TYPE = {

positionUnavailable(0), -- *the directory is ordered by position, and the density of files in the area surrounding the specified position is so great that no point for insertion is available; the directory must be reorganized as described in §4.2.6.3* --

fileNotUnique(1), -- *the directory already references a file with the same name (if the directory's* ***childrenUniquelyNamed*** *attribute is* TRUE*) or the same name and version (if the directory's* ***childrenUniquelyNamed*** *attribute is* FALSE*)* --

loopInHierarchy(2)}; -- *the directory is a descendant of the file being moved or copied* --

Examples:

If many files are inserted at the same point in a directory that is ordered by position and an attempt is made to insert another file at a point at which there is no position available, the following error is reported:

InsertionError [problem: positionUnavailable]

If a directory whose **childrenUniquelyNamed** attribute is TRUE references a file named "Product Specification" and an attempt is made to insert another file with the same name, the following error is reported:

InsertionError [problem: fileNotUnique]

If directory A is a child of directory B and an attempt is made to move directory B into directory A, the following error is reported:

InsertionError [problem: loopInHierarchy]

5.7 Range errors

RangeError may be reported by the random access procedures **RetrieveBytes** and **ReplaceBytes**. It indicates an inconsistency or other problem with the **range** argument specified by the client.

RangeError: ERROR [problem: ArgumentProblem] = 16;

RangeError results if an improper range specification is supplied to a random access procedure. Two problem types are reported: **illegal** (such a range can never be specified), and **unreasonable** (a range was not valid for a given file).

Examples:

If the client supplies a range with the value **[EndOfFile, EndOfFile]** to a random access operation, the service reports the error:

RangeError [problem: illegal]

If the client supplies a byte range for a given file with a **firstByte** specification that exceeds the size of the file, the service reports the error:

RangeError [problem: unreasonable]

5.8 Service errors

ServiceError may be reported by **Logon** or **Logoff**. It indicates that the service encountered a problem while attempting to create or destroy a session.

ServiceError: ERROR [problem: ServiceProblem] = 11;

The argument **problem** describes the problem in greater detail.

ServiceProblem: TYPE = {

> **cannotAuthenticate(0),** -- *the client may not log on because the file service is unable to determine whether the user's credentials are valid; this could occur if the file service needs to contact some service that is unavailable* --
>
> **serviceFull(1),** -- *the client may not log on because creation of another session would cause the number of sessions to exceed an implementation-dependent limit* --
>
> **serviceUnavailable(2),** -- *the file service is currently unavailable for use by new clients*
>
> **sessionInUse(3),** -- *the client may not log off because a remote procedure is still executing* --
>
> **serviceUnknown(4)};** -- *the requested service is not supported by this system element* --

Examples:

If the file service must contact another system to determine whether or not the user's credentials are valid and that system is unavailable, the following error is reported:

ServiceError [problem: cannotAuthenticate]

If the file service allows a maximum of ten concurrent sessions and a client tries to log on when there are already ten sessions, the following error is reported:

ServiceError [problem: serviceFull]

If the file service operator has entered a command to prevent further use of the file service and a client tries to log on, the following error is reported:

ServiceError [problem: serviceUnavailable]

If the client has called **List** and then tries to log off while execution of the procedure is in progress, the following error is reported:

ServiceError [problem: sessionInUse]

If the client attempts a **Logon** specifying a name of a service which a given system element does not support, the following error is reported:

ServiceError [problem: serviceUnknown]

5.9 Session errors

SessionError may be reported by any procedure. It indicates that the session handle is invalid.

SessionError: ERROR [problem: SessionProblem] = 12;

The argument **problem** describes the problem in greater detail.

SessionProblem: TYPE = {

tokenInvalid(0)}; -- *the token component of the session handle does not specify a currently valid session on this file service; the client may have already called Logoff or the session may have been forcibly terminated by the file service* --

Examples:

If the client calls a procedure that requires a session handle and the specified token value was never issued by **Logon**, the following error is reported:

SessionError [problem: tokenInvalid]

5.10 Space errors

SpaceError may be reported by any procedure that must allocate physical space for the storage of information. It indicates that the request for space could not be satisfied.

SpaceError: ERROR [problem: SpaceProblem] = 13;

The argument **problem** describes the problem in greater detail.

SpaceProblem: TYPE = {

allocationExceeded(0), -- *the space required by the procedure caused a directory's space limit to be exceeded (the total space occurpied by the directory and all of its descendants would have exceeded the directory's* **subtreeSizeLimit** *attribute)* --

attributeAreaFull(1), -- *there was not enough space in the attribute area to satisfy the request; the limits described in Chapter 4 would have been exceeded* --

mediumFull(2)}; -- *there was not enough space in the file service to satisfy the request* --

Examples:

If the client tries to store a file and its size causes an allocation limit to be exceeded even though enough physical space is available in the file service, the following error is reported:

SpaceError [problem: allocationExceeded]

If the client tries to store a file and there is physically no space available in the file service even though no allocation limit has been exceeded, the following error is reported:

SpaceError [problem: mediumFull]

If a file has 128 attributes that are set and the client tries to set another attribute, the following error is reported:

SpaceError [problem: attributeAreaFull]

5.11 Transfer errors

TransferError may be reported by any procedure that sends data to a sink or receives data from a source. It indicates that a problem occurred during the transfer.

TransferError: ERROR **[problem: TransferProblem] = 14;**

The argument **problem** describes the problem in greater detail.

TransferProblem: TYPE = {

aborted(0), -- *the bulk data transfer was aborted by the party at the other end of the connection* --

checksumIncorrect(1), -- *after transfer of a file's content to a sink, the checksum computed over the data did not match the file's stored* **checksum** *attribute* --

formatIncorrect(2), -- *the bulk data received from the source did not have the expected format* --

noRendezvous(3), -- *the identifier from the other party never appeared* --

wrongDirection(4)}; -- *the other party wanted to transfer the data in the wrong direction* --

Examples:

If the client calls **List** and then aborts the bulk data transfer because it has already received enough data, the following error is reported:

TransferError [problem: aborted]

If the client calls **Retrieve** and the checksum computed over the data transferred from the file service does not match the file's **checksum** attribute, the following error is reported:

TransferError [problem: checksumIncorrect]

If the client calls **Deserialize** and the transferred data is not a valid serialized file, the following error is reported:

TransferError [problem: formatIncorrect]

5.12 Undefined errors

UndefinedError may be reported by any procedure. It indicates that some implementation-dependent problem occurred that is not covered by another error. This error is normally reported only when the file service is malfunctioning. The client has no way of recovering from undefined errors.

UndefinedError: ERROR [problem: UndefinedProblem] = 15;

The argument **problem** describes the problem in greater detail. The meanings of specific values of this argument are implementation-dependent.

UndefinedProblem: TYPE = CARDINAL;

Examples:

If the file service encounters a disk error in directory structures and it has assigned the value 7 to this error condition, the following error is reported:

UndefinedError [problem: 7]

A. REFERENCES

The following documents supplement this protocol specification. References [1 and 7] are informational; they contain helpful motivational and explanatory material, but the Filing Protocol can be understood without them. References [2-6, 8] are mandatory; they describe other protocols upon which the Filing Protocol depends.

[1] Digital Equipment Corporation; Intel Corporation; Xerox Corporation. *The Ethernet, A Local Area Network: Data Link Layer and Physical Layer Specifications*. September 1980; Version 1.0.
This reference contains the data link and physical layer specifications for the Ethernet, the transmission medium for which Courier's standard representations are optimized.

[2] Xerox Corporation. *Authentication Protocol*. Xerox Network Systems Standard. Stamford, Connecticut; May 1986; XNSS 098605.
This reference defines the Authentication Protocol upon which the Filing Protocol relies for authentication.

[3] Xerox Corporation. *Bulk Data Transfer*. Xerox Network Systems Standard. Stamford, Connecticut; April 1984; XNSS 038112 (XSIS 038112); Addendum 1a. Augments [6].
This reference defines the Bulk Data Transfer Protocol upon which the Filing Protocol relies for bulk data transfer.

[4] Xerox Corporation. *Character Code Standard*. Xerox Network Systems Standard. Stamford, Connecticut; April 1984; XNSS 058404 (XSIS 058404).
This reference defines the character set and the string format which provide the basis for Courier's string data type.

[5] Xerox Corporation. *Clearinghouse Protocol*. Xerox Network Systems Standard. Stamford, Connecticut; April 1984; XNSS 078404 (XSIS 078404).
This reference defines the structure of user names which appear as various file attributes.

[6] Xerox Corporation. *Courier: The Remote Procedure Call Protocol*. Xerox Network Systems Standard. Stamford, Connecticut; December 1981; XNSS 038112 (XSIS 038112).
This reference defines the Courier language, in terms of which the Filing Protocol is defined.

[7] Xerox Corporation. *Internet Transport Protocols*. Xerox Network Systems Standard. Stamford, Connecticut; December 1981; XNSS 028112 (XSIS 028112).
This reference defines the Sequenced Packet Protocol upon which Courier relies for data transport.

[8] Xerox Corporation. *Time Protocol*. Xerox Network Systems Standard. Stamford, Connecticut; October 1982; XNSS 088210 (XSIS 088210).
This reference defines the Time Standard upon which the Filing Protocol relies for the definition of the format of time and date quantities.

[9] American National Standards Institute. *American National Standard Code for Information Interchange*. X3.4-1977.
This reference defines the character code assignments useful for text file interchange.

[10] Xerox Corporation. *Secondary Credentials Formats*. Xerox Network Systems Standard. Stamford, Connecticut; May 1986; XNSS 258605.
This reference documents specific type assignments and data formats for secondary credentials. Implementations of Filing or FilingSubset on hybrid hosts may require secondary authentication information.

B. TYPE ASSIGNMENT PROCEDURES

As stated in this document, file types and file attributes are assigned 32-bit numbers that are unique throughout the distributed system. These file type and attribute number spaces are administered by Xerox Corporation. To obtain a block of numbers, submit a written request to:

Xerox Corporation
Xerox Network Systems Institute
2300 Geng Road
Palo Alto, California 94303

Filing Protocol implementors are encouraged to apply for unique blocks of numbers for their particular applications. Uniqueness allows systems to freely interconnect without having to worry about overlapping values for critical fields.

Both file type and file attribute numbers should be used with economy as the total number of blocks is limited. If a Filing Protocol client or implementation is using a large quantity of either of these type numbers, the designer has probably misunderstood their utility.

B.1 Common file types

Commonly-used values of the **type** file attribute are defined in this appendix. Clients are encouraged to use these types to identify files that have the specified characteristics in order to promote information sharing. However, *the file service does not enforce the specified semantics*.

Files that have a format private to a single client, or for which the format is unknown or uninteresting, are conventionally given type:

tUnspecified: Type = 0;

Files that are directories with no additional semantics (and no content) are conventionally given type:

tDirectory: Type = 1;

Non-directory files containing text conforming to the *Character Code Standard* [4], including the Xerox String Encoding defined there (except that no length information is in the content) are conventionally given type:

tText: Type = 2;

Files that are non-directories containing a single data structure of type SerializedFile are conventionally given type:

tSerialized: Type = 3;

Files consisting of attribute information only and no content are conventionally given the type:

```
tEmpty: Type = 4;
```

Non-directory text files whose content can be interpreted as standard ASCII [9] are conventionally given the type:

```
tAscii: Type = 6;
```

StreamOfAsciiText defines a standard encoding for line-oriented ASCII textual information. In this approach to text encoding, individual lines are distinguished by the encoding syntax and not by specific codes embedded within the strings. This implies that line delimiter characters are absent from the text encoding; structural information is conveyed entirely by the encoding.

```
StreamOfAsciiText: TYPE = CHOICE OF {
    nextLine (0) => RECORD [
        line: AsciiString,
        restOfText: StreamOfAsciiText],
    lastLine (1) => AsciiString};

AsciiString: TYPE = RECORD [
    lastByteSignificant: BOOLEAN,
    bytes: SEQUENCE OF UNSPECIFIED];
```

AsciiString is used to represent a series of codes from the ASCII character set [9] as a sequence of sixteen-bit entities. Each sixteen-bit unspecified value contains two eight-bit ASCII codes. The boolean **lastByteSignificant** indicates whether or not the last byte of the last unspecified value is significant (that is, whether or not the length of the ASCII string in bytes is even). If **lastByteSignificant** is FALSE, then the last byte has a zero value and should be ignored.

Files whose content conforms to the **StreamOfAsciiText** definition are given the type:

```
tAsciiText: Type = 7;
```

The complete declaration of the Filing remote program is given below.

```
Filing: PROGRAM 10 VERSION 6 =
BEGIN
    DEPENDS UPON
        BulkData (0) VERSION 1,
        Clearinghouse (2) VERSION 3,
        Authentication (14) VERSION 3,
        Time (15) VERSION 2;

-- TYPES AND CONSTANTS --

-- Attributes (individual attributes defined later) --

AttributeType: TYPE = LONG CARDINAL;
AttributeTypeSequence: TYPE = SEQUENCE OF AttributeType;
allAttributeTypes: AttributeTypeSequence = [37777777777B];
Attribute: TYPE = RECORD [type: AttributeType, value: SEQUENCE OF UNSPECIFIED];
AttributeSequence: TYPE = SEQUENCE OF Attribute;

-- Controls --

ControlType: TYPE = {lock(0), timeout(1), access(2)};
ControlTypeSequence: TYPE = SEQUENCE 3 OF ControlType;
Control: TYPE = CHOICE ControlType OF {
    lock = > Lock,
    timeout = > Timeout,
    access = > AccessSequence};
ControlSequence: TYPE = SEQUENCE 3 OF Control;

Lock: TYPE = {none(0), share(1), exclusive(2)};

Timeout: TYPE = CARDINAL; -- in seconds --
defaultTimeout: Timeout = 177777B; -- actual value is implementation-dependent --

AccessType: TYPE = {
    -- all files -- read(0), write(1), owner(2),
    -- directories -- add(3), remove(4)};
AccessSequence: TYPE = SEQUENCE 5 OF AccessType;
fullAccess: AccessSequence = [177777B];

-- Scopes --

ScopeType: TYPE = {count(0), direction(1), filter(2), depth(3)};
Scope: TYPE = CHOICE ScopeType OF {
    count = > Count,
    depth = > Depth,
    direction = > Direction,
    filter = > Filter};
ScopeSequence: TYPE = SEQUENCE 4 OF Scope;

Count: TYPE = CARDINAL;
unlimitedCount: Count = 177777B;

Depth: TYPE = CARDINAL;
allDescendants: Depth = 177777B;

Direction: TYPE = {forward(0), backward(1)};

FilterType: TYPE = {
    -- relations --
    less(0), lessOrEqual(1), equal(2), notEqual(3), greaterOrEqual(4), greater(5),
    -- logical --
    and(6), or(7), not(8),
    -- constants --
    none(9), all(10),
    -- patterns --
    matches(11)};
Filter: TYPE = CHOICE FilterType OF {
    less, lessOrEqual, equal, notEqual, greaterOrEqual, greater = >
        RECORD [attribute: Attribute, interpretation: Interpretation], -- interpretation
            ignored if attribute interpreted by implementor --
    and, or = > SEQUENCE OF Filter,
    not = > Filter,
    none, all = > RECORD [],
    matches = > RECORD [attribute: Attribute]};
Interpretation: TYPE = {none(0), boolean(1), cardinal(2), longCardinal(3),
    time(4), integer(5), longInteger(6), string(7)};
nullFilter: Filter = all [];

-- Handles and Authentication --

Credentials: TYPE = RECORD [
    primary: PrimaryCredentials,
    secondary: SecondaryCredentials];

PrimaryCredentials: TYPE = Authentication.Credentials;
nullPrimaryCredentials: PrimaryCredentials = Authentication.nullCredentials;

Strength: TYPE = {none(0), simple(1), strong(2)};
SecondaryCredentials: TYPE = CHOICE Strength OF {
    none = > [],
    simple = > Secondary,
    strong = > EncryptedSecondary};

SecondaryItemType: TYPE = LONG CARDINAL;
SecondaryType: TYPE = SEQUENCE 10 OF SecondaryItemType;
```

```
Secondary: TYPE = SEQUENCE 10 OF SecondaryItem;
SecondaryItem: TYPE = RECORD [
    type: SecondaryItemType,
    value: SEQUENCE OF UNSPECIFIED];

EncryptedSecondary: TYPE = SEQUENCE OF Authentication.Block;

Handle: TYPE = ARRAY 2 OF UNSPECIFIED;
nullHandle: Handle = [0, 0]; -- meaning depends on operation --

Session: TYPE = RECORD [token: ARRAY 2 OF UNSPECIFIED, verifier: Verifier];

Verifier: TYPE = Authentication.Verifier;

-- Random access --

ByteAddress: TYPE = LONG CARDINAL;
ByteCount: TYPE = LONG CARDINAL;
endOfFile: LONG CARDINAL = 37777777777B; -- logical end of file --

ByteRange: TYPE = RECORD [firstByte: ByteAddress, count: ByteCount];

-- REMOTE PROCEDURES --

-- Logging On and Off --

Logon: PROCEDURE [
    service: Clearinghouse.Name, credentials: Credentials, verifier: Verifier]
RETURNS [session: Session]
REPORTS [AuthenticationError, ServiceError, SessionError, UndefinedError] = 0;

Logoff: PROCEDURE [session: Session]
REPORTS [AuthenticationError, ServiceError, SessionError, UndefinedError] = 1;

Continue: PROCEDURE [session: Session]
RETURNS [continuance: CARDINAL]
REPORTS [AuthenticationError, SessionError, UndefinedError] = 19;

-- Opening and Closing Files --

Open: PROCEDURE [attributes: AttributeSequence, directory: Handle,
    controls: ControlSequence, session: Session]
RETURNS [file: Handle]
REPORTS [AccessError, AttributeTypeError, AttributeValueError,
    AuthenticationError, ControlTypeError, ControlValueError, HandleError,
    SessionError, UndefinedError] = 2;

Close: PROCEDURE [file: Handle, session: Session]
REPORTS [AuthenticationError, HandleError, SessionError, UndefinedError] = 3;

-- Creating and Deleting Files --

Create: PROCEDURE [directory: Handle, attributes: AttributeSequence,
    controls: ControlSequence, session: Session]
RETURNS [file: Handle]
REPORTS [AccessError, AttributeTypeError, AttributeValueError,
     AuthenticationError, ControlTypeError, ControlValueError, HandleError,
     InsertionError, SessionError, SpaceError, UndefinedError] = 4;

Delete: PROCEDURE [file: Handle, session: Session]
REPORTS [AccessError, AuthenticationError, HandleError, SessionError,
     UndefinedError] = 5;

-- Getting and Changing Controls (Transient) --

GetControls: PROCEDURE [file: Handle, types: ControlTypeSequence,
     session: Session]
RETURNS [controls: ControlSequence]
REPORTS [AccessError, AuthenticationError, ControlTypeError, HandleError,
     SessionError, UndefinedError] = 6;

ChangeControls: PROCEDURE [file: Handle, controls: ControlSequence,
     session: Session]
REPORTS [AccessError, AuthenticationError, ControlTypeError,
     ControlValueError, HandleError, SessionError, UndefinedError] = 7;

-- Getting and Changing Attributes (Permanent) --

GetAttributes: PROCEDURE [file: Handle, types: AttributeTypeSequence,
     session: Session]
RETURNS [attributes: AttributeSequence]
REPORTS [AccessError, AttributeTypeError, AuthenticationError, HandleError,
     SessionError, UndefinedError] = 8;

ChangeAttributes: PROCEDURE [file: Handle, attributes: AttributeSequence,
     session: Session]
REPORTS [AccessError, AttributeTypeError, AttributeValueError,
     AuthenticationError, HandleError, InsertionError, SessionError, SpaceError,
     UndefinedError] = 9;

UnifyAccessLists: PROCEDURE [directory: Handle, session: Session]
REPORTS [AccessError, AuthenticationError, HandleError, SessionError,
     UndefinedError] = 20;

-- Copying and Moving Files --

Copy: PROCEDURE [file, destinationDirectory: Handle,
     attributes: AttributeSequence, controls: ControlSequence, session: Session]
RETURNS [newFile: Handle]
REPORTS [AccessError, AttributeTypeError, AttributeValueError,
     AuthenticationError, ControlTypeError, ControlValueError, HandleError,
     InsertionError, SessionError, SpaceError, UndefinedError] = 10;

Move: PROCEDURE [file, destinationDirectory: Handle,
     attributes: AttributeSequence, session: Session]
REPORTS [AccessError, AttributeTypeError, AttributeValueError,
     AuthenticationError, HandleError, InsertionError, SessionError, SpaceError,
     UndefinedError] = 11;
```

```
-- Transferring Bulk Data (File Content) --

Store: PROCEDURE [directory: Handle, attributes: AttributeSequence,
    controls: ControlSequence, content: BulkData.Source, session: Session]
RETURNS [file: Handle]
REPORTS [AccessError, AttributeTypeError, AttributeValueError,
    AuthenticationError, ConnectionError, ControlTypeError, ControlValueError,
    HandleError, InsertionError, SessionError, SpaceError, TransferError,
    UndefinedError] = 12;

Retrieve: PROCEDURE [file: Handle, content: BulkData.Sink, session: Session]
REPORTS [AccessError, AuthenticationError, ConnectionError, HandleError,
    SessionError, TransferError, UndefinedError] = 13;

Replace: PROCEDURE [file: Handle, attributes: AttributeSequence,
    content: BulkData.Source, session: Session]
REPORTS [AccessError, AttributeTypeError, AttributeValueError,
    AuthenticationError, ConnectionError, HandleError, SessionError, SpaceError,
    TransferError, UndefinedError] = 14;

-- Transferring Bulk Data (Serialized Files) --

Serialize: PROCEDURE [file: Handle, serializedFile: BulkData.Sink,
    session: Session]
REPORTS [AccessError, AuthenticationError, ConnectionError, HandleError,
    SessionError, TransferError, UndefinedError] = 15;

Deserialize: PROCEDURE [directory: Handle, attributes: AttributeSequence,
    controls: ControlSequence, serializedFile: BulkData.Source, session: Session]
RETURNS [file: Handle]
REPORTS [AccessError, AttributeTypeError, AttributeValueError,
    AuthenticationError, ConnectionError, ControlTypeError, ControlValueError,
    HandleError, InsertionError, SessionError, SpaceError, TransferError,
    UndefinedError] = 16;

-- Random Access to File Data --

RetrieveBytes: PROCEDURE [file: Handle, range: ByteRange,
    sink: BulkData.Sink, session: Session]
REPORTS [AccessError, HandleError, RangeError, SessionError, UndefinedError] = 22;

ReplaceBytes: PROCEDURE [
    file: Handle, range: ByteRange, source: BulkData.Source, session: Session]
REPORTS [AccessError, HandleError, RangeError, SessionError, SpaceError,
    UndefinedError] = 23;

-- Locating and Listing Files in a Directory --

Find: PROCEDURE [directory: Handle, scope: ScopeSequence,
    controls: ControlSequence, session: Session]
RETURNS [file: Handle]
REPORTS [AccessError, AuthenticationError, ControlTypeError,
    ControlValueError, HandleError, ScopeTypeError, ScopeValueError,
    SessionError, UndefinedError] = 17;
```

```
List: PROCEDURE [directory: Handle, types: AttributeTypeSequence,
    scope: ScopeSequence, listing: BulkData.Sink, session: Session]
REPORTS [AccessError, AttributeTypeError, AuthenticationError,
    ConnectionError, HandleError, ScopeTypeError, ScopeValueError, SessionError,
    TransferError, UndefinedError] = 18;

-- REMOTE ERRORS --

-- problem with an attribute type or value --

AttributeTypeError: ERROR [problem: ArgumentProblem,
    type: AttributeType] = 0;
AttributeValueError: ERROR [problem: ArgumentProblem,
    type: AttributeType] = 1;

-- problem with a control type or value --

ControlTypeError: ERROR [problem: ArgumentProblem, type: ControlType] = 2;
ControlValueError: ERROR [problem: ArgumentProblem, type: ControlType] = 3;

-- problem with a scope type or value --

ScopeTypeError: ERROR [problem: ArgumentProblem, type: ScopeType] = 4;
ScopeValueError: ERROR [problem: ArgumentProblem, type: ScopeType] = 5;

ArgumentProblem: TYPE = {
    illegal(0), -- this type or value is never allowed --
    disallowed(1), -- this type or value is not allowed in this procedure --
    unreasonable(2), -- this type or value does not make sense in this context --
    unimplemented(3), -- this type or value is not supported in this implementation --
    duplicated(4), -- this type or value is specified twice --
    missing(5)}; -- this type or value is required but not specified --

-- problem in obtaining access to a file --

AccessError: ERROR [problem: AccessProblem] = 6;
AccessProblem: TYPE = {
    accessRightsInsufficient(0), -- the user doesn't have access --
    accessRightsIndeterminate(1), -- cannot determine whether the user has access --
    fileChanged(2), -- in such a way that the procedure cannot continue --
    fileDamaged(3), -- this file should not be used --
    fileInUse(4), -- file is still unavailable after expiration of timeout --
    fileNotFound(5), -- the file does not exist in the specified context --
    fileOpen(6)}; -- file cannot be moved or deleted while another handle exists --

-- problem with a credentials or verifier --

AuthenticationError: ERROR [problem: AuthenticationProblem, type: SecondaryType] = 7;
AuthenticationProblem: TYPE = {
    primaryCredentialsInvalid(0), -- decryption failed or Clearinghouse name was invalid --
    verifierInvalid(1), -- decryption failed or simple password was invalid --
    verifierExpired(2), -- a strong verifier was too old --
    verifierReused(3), -- verifier has been reused or is out of sequence --
    primaryCredentialsExpired(4), -- validity of primary credentials has lapsed --
    inappropriatePrimaryCredentials(5), -- primary credentials were too weak --
```

```
        secondaryCredentialsRequired(6), -- secondary authentication information required but
            none was supplied; type indicates the type of secondary credentials required --
        secondaryCredentialsTypeInvalid(7), -- the specified secondary credentials type was
            incorrect or the secondary credentials value was improperly formatted for the
            specified type; type indicates the type of secondary credentials required --
        secondaryCredentialsValueInvalid(8)}; -- the specified secondary authentication
            information was not acceptible to the service; type indicates the type of secondary
            credentials required --

    -- problem with a bulk data transfer --

    ConnectionError: ERROR [problem: ConnectionProblem] = 8;
    ConnectionProblem: TYPE = {

     -- communication problems --

     noRoute(0), -- no route to the other party could be found --
     noResponse(1), -- the other party never answered --
     transmissionHardware(2), -- some local transmission hardware was inoperable --
     transportTimeout(3), -- the other party responded but later failed to respond --

     -- resource problems --

     tooManyLocalConnections(4), -- no additional connection is possible --
     tooManyRemoteConnections(5), -- the other party rejected the connection attempt --

     -- remote program implementation problems --

     missingCourier(6), -- the other party had no Courier implementation --
     missingProgram(7), -- the other party did not implement the bulk data program --
     missingProcedure(8), -- the other party did not implement the procedure --
     protocolMismatch(9), -- the two parties have no Courier version in common --
     parameterInconsistency(10), -- a protocol violation occurred in parameters --
     invalidMessage(11), -- a protocol violation occurred in message format --
     returnTimedOut(12), -- the procedure call never returned --

     -- miscellaneous --

     otherCallProblem(177777B) }; -- some other protocol violation during a call --

    -- problem with a file handle --

    HandleError: ERROR [problem: HandleProblem] = 9;
    HandleProblem: TYPE = {
        invalid(0), -- this file handle is not valid --
        nullDisallowed(1), -- the null handle is not allowed here --
        directoryRequired(2)}; -- the handle must designate a directory --

    -- problem during insertion in directory (or changing attributes) --

    InsertionError: ERROR [problem: InsertionProblem] = 10;
    InsertionProblem: TYPE = {
        positionUnavailable(0), -- no "point" at which to insert in order-by-position --
    fileNotUnique(1), -- identifying information (e.g. name) is not unique --
    loopInHierarchy(2)}; -- cyclic directory structures are illegal --

-- problem during random access operation --

RangeError: ERROR [problem: ArgumentProblem] = 16;

-- problem during logon or logoff --

ServiceError: ERROR [problem: ServiceProblem] = 11;
ServiceProblem: TYPE = {
    cannotAuthenticate(0), -- cannot reach authentication service, for example --
    serviceFull(1), -- no more logons can be accepted --
    serviceUnavailable(2), -- logons are not being accepted --
    sessionInUse(3)}; -- cannot logoff while an operation is in progress --

-- problem with a session --

SessionError: ERROR [problem: SessionProblem] = 12;
SessionProblem: TYPE = {
    tokenInvalid(0)}; -- the token is invalid --

-- problem obtaining space for file content or attributes --

SpaceError: ERROR [problem: SpaceProblem] = 13;
SpaceProblem: TYPE = {
    allocationExceeded(0), -- specifically-allocated file space exceeded --
    attributeAreaFull(1), -- no more attributes may be stored with file --
    mediumFull(2)}; -- no more room is available on the storage medium --

-- problem during bulk data transfer --

TransferError: ERROR [problem: TransferProblem] = 14;
TransferProblem: TYPE = {
    aborted(0), -- the transfer was aborted by the source or sink --
    checksumIncorrect(1), -- after transfer of a file's content to a sink, the checksum
        computed over the data did not match the file's stored checksum attribute --
    formatIncorrect(2), -- bulk data received from source did not have the expected format --
    noRendezvous(3), -- the identifier from the other party never appeared --
    wrongDirection(4)}; -- other party wanted to transfer the data in the wrong direction --

-- some undefined (and implementation-dependent) problem occurred --

UndefinedError: ERROR [problem: UndefinedProblem] = 15;
UndefinedProblem: TYPE = CARDINAL; -- implementation-dependent --

-- INTERPRETED ATTRIBUTE DEFINITIONS --

accessList: AttributeType = 19;
AccessEntry: TYPE = RECORD [key: Clearinghouse.Name, access: AccessSequence];
AccessList: TYPE = RECORD [entries: SEQUENCE OF AccessEntry, defaulted: BOOLEAN];
    -- specification of access permissions --
```

```
checksum: AttributeType = 0; -- checksum over content of file --
Checksum: TYPE = CARDINAL;
unknownChecksum: Checksum = 177777B;

childrenUniquelyNamed: AttributeType = 1; -- all children uniquely named --
ChildrenUniquelyNamed: TYPE = BOOLEAN; -- default value is implementation-dependent --

createdBy: AttributeType = 2; -- name of user whose action changed createdOn --
CreatedBy: TYPE = User;

createdOn: AttributeType = 3; -- date file's content was created --
CreatedOn: TYPE = Time;

dataSize: AttributeType = 16; -- number of bytes of data in file's content --
DataSize: TYPE = LONG CARDINAL;

defaultAccessList: AttributeType = 20;
    -- access inherited by children with defaulted access lists --
DefaultAccessList: TYPE = AccessList;

fileID: AttributeType = 4; -- ID of file --
FileID: TYPE = ARRAY 5 OF UNSPECIFIED; -- implementation-dependent --
nullFileID: FileID = [0, 0, 0, 0, 0];

isDirectory: AttributeType = 5; -- file is a directory (potentially has children) --
IsDirectory: TYPE = BOOLEAN;

isTemporary: AttributeType = 6; -- file is temporary (cannot be a directory) --
IsTemporary: TYPE = BOOLEAN;

modifiedBy: AttributeType = 7; -- name of user whose action changed modifiedOn --
ModifiedBy: TYPE = User;

modifiedOn: AttributeType = 8; -- date file was last modified --
ModifiedOn: TYPE = Time;

name: AttributeType = 9; -- descriptive name of file (relative to parent) --
Name: TYPE = STRING; -- must not exceed 100 bytes (not characters) --

numberOfChildren: AttributeType = 10; -- number of children in this directory --
NumberOfChildren: TYPE = CARDINAL;

ordering: AttributeType = 11; -- order of children for Find, List --
Ordering: TYPE = RECORD [
    key: AttributeType, ascending: BOOLEAN, interpretation: Interpretation];
    -- interpretation ignored if attribute interpreted by implementor --

defaultOrdering: Ordering = [key: name, ascending: TRUE, interpretation: string];
byAscendingPosition: Ordering = [key: position, ascending: TRUE, interpretation: none];
byDescendingPosition: Ordering = [key: position, ascending: FALSE, interpretation: none];

parentID: AttributeType = 12; -- ID of parent directory of this file --
ParentID: TYPE = FileID;
```

```
pathname: AttributeType = 21; -- access path to file relative to root file --
Pathname: TYPE = STRING;

position: AttributeType = 13; -- reference to position within directory --
Position: TYPE = SEQUENCE 100 OF UNSPECIFIED;
firstPosition: Position = [0];
lastPosition: Position = [177777B];

readBy: AttributeType = 14; -- name of user whose action changed readOn --
ReadBy: TYPE = User;

readOn: AttributeType = 15; -- date file's content was last read --
ReadOn: TYPE = Time;

storedSize: AttributeType = 26; -- number of bytes physically allocated to stored file --
StoredSize: TYPE = LONG CARDINAL;

subtreeSize: AttributeType = 27;
    -- number of content bytes in directory and all descendants --
SubtreeSize: TYPE = LONG CARDINAL;
nullSubtreeSizeLimit: SubtreeSizeLimit = 37777777777B;

subtreeSizeLimit: AttributeType = 28;
    -- limitation on number content bytes in directory and all descendants --
SubtreeSizeLimit: TYPE = LONG CARDINAL;

type: AttributeType = 17; -- file type; assigned by client --
Type: TYPE = LONG CARDINAL;

version: AttributeType = 18; -- version number of file (relative to parent) --
Version: TYPE = CARDINAL;
lowestVersion: Version = 0;
highestVersion: Version = 177777B;

-- Common Definitions --

Time: TYPE = Time.Time; -- seconds --

nullTime:  Time = Time.earliestTime;
User: TYPE = Clearinghouse.Name;

-- BULK DATA FORMATS --

-- Serialized File Format, used in Serialize and Deserialize --

SerializedFile: TYPE = RECORD [version: LONG CARDINAL, file: SerializedTree];

currentVersion: LONG CARDINAL = 3;

SerializedTree: TYPE = RECORD [
    attributes: AttributeSequence,
    content: RECORD [data: BulkData.StreamOfUnspecified,
        lastByteIsSignificant: BOOLEAN],
    children: SEQUENCE OF SerializedTree];
```

```
-- Attribute Series Format, used in List --

StreamOfAttributeSequence: TYPE = CHOICE OF {
    nextSegment (0) => RECORD [
        segment: SEQUENCE OF AttributeSequence,
        restOfStream: StreamOfAttributeSequence],
    lastSegment (1) => SEQUENCE OF AttributeSequence};

-- Line-oriented ASCII text file format, used in file interchange --

StreamOfAsciiText: TYPE = CHOICE OF {
    nextLine (0) => RECORD [
        line: AsciiString,
        restOfText: StreamOfAsciiText],
    lastLine (1) => AsciiString};

AsciiString: TYPE = RECORD [
    lastByteSignificant: BOOLEAN,
    bytes: SEQUENCE OF UNSPECIFIED];

-- FILE TYPES --

-- Clients are encouraged to use these predefined types to identify files that have the specified
characteristics, to promote information sharing --

tUnspecified: Type = 0; -- nothing is known about the content or attributes of a file of this
    type; it is useful for files that have a private format --

tDirectory: Type = 1; -- this file is a directory, and it has no content (only children) --

tText: Type = 2; -- this file is not a directory, and its content conforms to the string encoding
    described in the Character Code Standard --

tSerialized: Type = 3; -- this file is not a directory, and its content conforms to the serialized
    file format --

tEmpty: Type = 4; -- this file is not a directory, and is comprised of attribute information only
    and no content --

tAscii: Type = 6; -- this file is not a directory, and its content is comprised of ASCII data --

tAsciiText: Type = 7; -- this file is not a directory, and its content conforms to the stream of
    ASCII text definition --

END. -- of Filing --
```

This appendix gives an example of a complete session of interaction with the file service, annotated with the purpose of each procedure, the task to be performed, and the section of this document in which the procedure is explained. In this session, the client

- opens and lists the directory "Letters" within the directory "Development"
- retrieves a file and changes one of its attributes
- deletes all filès that meet a certain characteristic
- stores a new file, and copies it to a different directory
- moves the retrieved file to the same directory
- lists the two directories·

First, the client logs on to the file service (§3.1.3). The constant **userCredentials** denotes a value of credentials appropriate for this client to log on to the file service (obtained from an authentication service or by other means). A session handle is returned which is used in subsequent operations.

```
Logon [service: [organization: "Xerox," domain: "Office Systems," object: "TestFS"],
    credentials: userCredentials, verifier: simpleVerifier]

RETURNS [session: [token: [41B, 3B], verifier: simpleVerifier]]
```

Most clients wish to remain logged on to the file service until an explicit logoff occurs even if there are periods of inactivity. To "remind" the file service that the client is still interested in the session, the client probes the file service (§3.1.5).

```
Continue [session: [token: [41B, 3B], verifier: simpleVerifier]]

RETURNS [continuance: 600]
```

Because the client just logged on, the first **Continue** has little effect (the session is unlikely to be terminated so soon), but it does let the client know that it should **Continue** again before ten minutes (600 seconds) have elapsed. A client would typically instruct some background process to call **Continue** again within that period, however, in this example it is assumed that this session is logged off before the next **Continue** is necessary.

Next, the client opens a directory which resides in the root directory and has the name "Development" (§3.2.2). The file service opens a file within the root directory because no **parentID** attribute was specified.

```
Open [
    attributes: [[type: name, value: "Development"]],
    directory: nullHandle,
    controls: [],
    session: [token: [41B, 3B], verifier: simpleVerifier]]

RETURNS [file: [365B, 21B]]
```

The client needs the **fileID** attribute of this directory for use in future procedure calls so it specifies that attribute in a call to **GetAttributes** (§3.4.2).

```
GetAttributes [file: [365B, 21B], types: [fileID],
    session: [token: [41B, 3B], verifier: simpleVerifier]]

RETURNS [attributes: [[type: fileID, value: [0B, 0B, 2B, 33B, 6334B]]]]
```

The client again calls **Open** (§3.2.2), this time to open the directory "Letters" within the directory "Development." The **parentID** attribute is specified to indicate that the file being opened must be inside "Development."

```
Open [
    attributes: [
        [type: name, value: "Letters"],
        [type: parentID, value: [0B, 0B, 2B, 33B, 6334B]]],
    directory: nullHandle,
    controls: [],
    session: [token: [41B, 3B], verifier: simpleVerifier]]

RETURNS [file: [214B, 22B]]
```

Again, the **fileID** is determined for future reference.

```
GetAttributes [file: [214B, 22B], types: [fileID],
    session: [token: [41B, 3B], verifier: simpleVerifier]]

RETURNS [attributes: [[type: fileID, value: [0B, 0B, 477B, 1162B, 5B]]]]
```

Now, the client lists the files in "Letters" starting at the last file and moving toward the beginning. It is assumed that the directory's ordering is **defaultOrdering**. The attributes to be returned are **fileID**, **name**, **version**, and **type**. Notice that the attributes of the three files are sent to the client as bulk data. Before calling **List** (§3.5.3) the client must have constructed **bulkDataSink1**, a Bulk Data Transfer **Sink**, and this may have required making a remote procedure call according to the Bulk Data Transfer specification.

```
List [directory: [214B, 22B], types: [fileID, name, version, type],
    scope: [direction backward], listing: bulkDataSink1,
    session: [token: [41B, 3B], verifier: simpleVerifier]]

-- the following record is transferred as bulk data --

nextSegment [
    segment: [[
        [type: fileID, value: [0B, 0B, 65B, 11743B, 2634B]],
        [type: name, value: "Recent Purchases"],
        [type: version, value: 2],
        [type: type, value: tUnspecified]]],
    restOfStream:
        nextSegment [
            segment: [[
                [type: fileID, value: [0B, 0B, 1762B, 153B, 7775B]],
                [type: name, value: "Information Request"],
                [type: version, value: 1],
                [type: type, value: tText]]],
            restOfStream:

                lastSegment [[
                    [type: fileID, value: [0B, 0B, 3633B, 1102B, 5B]],
                        [type: name, value: "Expense Report"],
                        [type: version, value: 1],
                        [type: type, value: tUnspecified]]]
                ]]
```

The client now wishes to retrieve the content of the file named "Recent Purchases." To open it the client can directly specify the **fileID** obtained in the previous procedure (along with the **parentID** to make sure that the file has not moved in the meantime). The client also specifies a lock since the subsequent procedures need to proceed without interruption.

```
Open [attributes: [
        [type: fileID, value: [0B, 0B, 65B, 11743B, 2634B]],
        [type: parentID, value: [0B, 0B, 477B, 1162B, 5B]]],
    directory: nullHandle,
    controls: [lock exclusive],
    session: [token: [41B, 3B], verifier: simpleVerifier]]

RETURNS [file: [602B, 24B]]
```

The client now wishes to retrieve the content of the file. Before calling **Retrieve** (§3.6.3) the client must have constructed **bulkDataSink2**, a Bulk Data Transfer **Sink**, and this may have required making a remote procedure call according to the Bulk Data Transfer Protocol.

```
Retrieve [file: [602B, 24B], content: bulkDataSink2,
    session: [token: [41B, 3B], verifier: simpleVerifier]]
```

-- *the data is transferred as bulk data* --

It is assumed that attribute type 4416B is assigned to the client and that this attribute is used as a "marker" to indicate that the file has been retrieved. Since the file has now been retrieved, the client proceeds to set the attribute to TRUE using **ChangeAttributes** (§3.4.3).

```
ChangeAttributes [file: [602B, 24B],
    attributes: [[type: 4416B, value: TRUE]],
    session: [token: [41B, 3B], verifier: simpleVerifier]]
```

The client doesn't yet close the file (in case it is needed later), but the exclusive lock is no longer needed so the lock is changed (§3.3.3).

```
ChangeControls [file: [602B, 24B], controls: [lock none],
    session: [token: [41B, 3B], verifier: simpleVerifier]]
```

Now the client deletes all files that have types other than **tUnspecified**. This is done by repeatedly calling **Find** (§3.5.2) with the appropriate criteria and deleting the result, until

Find no longer succeeds. If there are many files in a directory, **List** (§3.5.3) might be a more appropriate procedure for this purpose.

```
Find [directory: [214B, 22B],
    scope: [filter notEqual
        [attribute: [type: type, value: tUnspecified], interpretation: longCardinal]],
    controls: [],
    session: [token: [41B, 3B], verifier: simpleVerifier]]

RETURNS [file: [710B, 27B]]
```

Now that the file has been found (the returned file handle refers to the file named "Information Request"), the client deletes it (§3.7.2).

```
Delete [file: [710B, 27B], session: [token: [41B, 3B], verifier: simpleVerifier]]
```

The client tries again to find a file that satisfies the criteria. This time, however, no such file exists. All files left in the directory are of type **tUnspecified**.

```
Find [directory: [214B, 22B],
    scope: [filter notEqual
        [attribute: [type: type, value: tUnspecified], interpretation: longCardinal]],
    controls: [],
    session: [token: [41B, 3B], verifier: simpleVerifier]]

REPORTS AccessError [problem: fileNotFound]
```

The next step is to store a new file in the "Letters" directory. The type is **tText** and the name is "August Progress Report." The creator and the date of creation are also specified. The specified **dataSize** is a hint. It could have been omitted or even specified incorrectly (within the available space on the file service) without affecting anything but performance. Notice that the file's content is sent to the file service as bulk data. Before calling **Store** (§3.6.2) the client must have constructed **bulkDataSource1**, a Bulk Data Transfer **Source**, and this may have required making a remote procedure call according to the Bulk Data Transfer Protocol.

```
Store [directory: [214B, 22B],
    attributes: [
        [type: name, value: "August Progress Report"],
        [type: type, value: tText],
        [type: createdBy: value:
            [organization: "Xerox," domain: "Office Systems," object: "Kabcenell"],
        [type: createdOn: value: 2263526570B],
        [type:dataSize, value: 50B]],
    controls: [], content: bulkDataSource1,
    session: [token: [41B, 3B], verifier: simpleVerifier]]

-- the data is transferred as bulk data --

RETURNS [file: [717B, 23B]]
```

The client then copies this file just stored in "Letters" into "Development" giving the copy the name "Current Progress" (§3.8.1).

```
Copy [file: [717B, 23B], destinationDirectory: [365B, 21B],
     attributes: [[type: name, value: "Current Progress"]],
     controls: [], session: [token: [41B, 3B], verifier: simpleVerifier]]

RETURNS [file: [3104B, 20B]]
```

The client then closes both the stored file and its copy (§3.2.3).

```
Close [file: [717B, 23B],
     session: [token: [41B, 3B], verifier: simpleVerifier]]

Close [file: [3104B, 20B],
     session: [token: [41B, 3B], verifier: simpleVerifier]]
```

The file retrieved earlier ("Recent Purchases"), which is still open, is now moved from "Letters" to "Development" without specifying any attributes (§3.8.2).

```
Move [file: [602B, 24B], destinationDirectory: [365B, 21B],
     attributes: [], session: [token: [41B, 3B], verifier: simpleVerifier]]
```

Finally, the files in directories "Development" and "Letters" are listed, from beginning to end this time. Notice that "Recent Purchases" has acquired a version number of 1; version numbers are not preserved when moving between directories. Also notice that the fileID of "Recent Purchases" has not changed.

```
-- "bulkDataSink3" is established for the bulk data transfer --

List [directory: [365B, 21B], types: [fileID, name, version, type],
     scope: [],
     listing: bulkDataSink3,
     session: [token: [41B, 3B], verifier: simpleVerifier]]

-- bulk data for "Development" --

nextSegment [
     segment: [[
          [type: fileID, value: [0B, 0B, 47103B, 511B, 60B]],
          [type: name, value: "Current Progress"]
          [type: version, value: 1],
          [type: type, value: tText]]],
     restOfStream:

          nextSegment [
               segment: [[
                    [type: fileID, value: [0B, 0B, 477B, 1162B, 5B]],
                    [type: name, value: "Letters"],
                    [type: version, value: 1],
                    [type: type, value: Type tDirectory]]],
               restOfStream:

                    lastSegment [[
                         [type: fileID, value: [0B, 0B, 65B, 11743B, 2634B]],
                         [type: name, value: "Recent Purchases"],
                         [type: version, value: 1],
                    [type: type, value: tUnspecified]]
                ]]

-- "bulkDataSink4" is established for the bulk data transfer --

List [directory: [214B, 22B], types: [fileID, name, version, type],
    scope: [], listing: bulkDataSink4,
    session: [token: [41B, 3B], verifier: simpleVerifier]]

-- bulk data for "Letters" --

nextSegment [
    segment: [[
        [type: fileID, value: [0B, 0B, 4267B, 315B, 5516B]],
        [type: name, value: "August Progress Report"],
        [type: version, value: 1],
        [type: type, value: tText]]],
    restOfStream:

        lastSegment [[
            [type: fileID, value: [0B, 0B, 3633B, 1102B, 5B]],
            [type: name, value: "Expense Report"],
            [type: version, value: 1],
            [type: type, value: tUnspecified]]]
        ]
```

The session is then logged off (§3.1.4). Termination of a session closes all remaining file handles opened within the session so it is not necessary to close them explicitly.

```
Logoff [session: [token: [41B, 3B], verifier: simpleVerifier]]
```

E. FILING SUBSET

E.1 Overview

The FilingSubset Protocol defines a minimal capability to store, retrieve, enumerate, and delete files of a remote service. Hosts whose primary role is not that of a network file service may support this protocol in order to provide a limited file transfer and file management capability.

Because the native file system interfaces of many systems may not easily support the general features of the Filing Protocol, the FilingSubset Protocol has also been designed to facilitate straightforward implementation on heterogeneous systems.

E.1.1 Motivation

The Filing Protocol provides a standardized means of accessing and transferring named collections of data between cooperating system elements in an internetwork. This protocol offers an exceedingly rich functionality; however, the extent of this richness may make the full protocol too difficult and expensive in development cost to justify in a host whose primary role is not that of a network file service.

In distributed processing applications, it is also desirable to enable two hosts to exchange files even though it is not intended that either system provide a comprehensive file service. Further, it is often desirable to provide network access to files residing on heterogeneous systems, if the addition of this feature does not require changes to native file system interfaces.

In summary, a number of desires motivate the definition of a simple filing capability:

- support file exchange without requiring an exceedingly rich functionality.
- provide XNS file access to the native file systems of heterogeneous network hosts.
- facilitate network access to the files on a system whose primary purpose is not that of a file service.
- ease the difficulty in supporting the Filing Protocol on native operating systems.
- permit implementation on diverse systems in a straightforward way.

E.1.2 Requirements and Goals

The definition of the FilingSubset Protocol is guided by a set of requirements and goals. In general, the requirements are to provide a minimal but useful level of service within the context of the Filing Protocol. The requirements guiding the definition of the FilingSubset Protocol are:

- provide the common file system functions of storage, retrieval, enumeration and deletion.
- foster compatibility by remaining a proper subset of the Filing Protocol.
- facilitate implementation on heterogeneous systems.
- ensure round-trip equality of file data.

A set of goals is also defined which, although not required nor guaranteed, are important to the overall usefulness for elements implementing the subset. The following goals are desirable in the definition of the Filing Subset:

- ease of implementation of service provider and client software on a variety of systems.
- round-trip preservation of attributes (the ability to store a file on a remote system and retrieve it at a later date with all attributes intact).
- the ability to perform common processing activities on a file regardless of which system it currently resides on (for example, text editing, data base listing and backup/restore).

E.2 Definition

The FilingSubset Protocol specifies a minimal level of file service which subset client and service implementations must support. Maximum interconnectivity is ensured when both client and service implementations support this minimum level of service and make no assumptions regarding the availability of a broader functionality. However, increasing levels of functionality may be supported by individual FilingSubset implementations. Corresponding client implementations must always be prepared to deal with only the minimum functionality defined here; in this way the client achieves the greatest level of compatibility with differing subset implementations, all of which support at least the minimum.

FilingSubset is defined as a subset of the Filing Protocol. This guarantees that the style of interaction between a subset client and service is consistent with that of the Filing Protocol. This method of definition also guarantees that clients which implement the subset may interact with a service implementing the Filing Protocol by issuing calls with the different Courier program number and specifying appropriate parameter values. In addition, a client using the Filing Protocol can interact with a FilingSubset service by restricting its use of remote operations and arguments to those defined here (again by using the appropriate Courier program number and specifying appropriate parameter values).

Note: Clients should not assume that an arbitrary FilingSubset service implementation will support multiple Courier connections for a single session; a servce may not allow a session obtained on one connection to be used on any other connection.

Note: In order to foster compatability between different FilingSubset client and service implementations, it is recommended that implementations avoid terminating the connection supporting their interaction too quickly or unnecessarily. Short periods of inactivity should not result in termination of the connection.

In all cases, the operations, arguments, and errors defined in the subset are identical to those in the Filing Protocol. In providing a minimal level of service, the subset does, however, restrict the choices available for argument and error values.

The complete Courier definition of the FilingSubset Protocol is presented in section E.7.

E.3 Procedures

The FilingSubset supports those Filing operations which provide the essential functions required for file storage, retrieval, enumeration, and deletion. These procedures are **Logon**, **Logoff**, **Continue**, **Open**, **Close**, **Retrieve**, **Store**, **List**, and **Delete**.

The FilingSubset Protocol also requires that all implementations permit file identification to be performed through the use of the **pathname** attribute. The syntax and interpretation of **pathname** attribute values is service-dependent.

E.3.1 Session support

The **Logon**, **Logoff**, and **Continue** operations are included in the FilingSubset and are identical to the corresponding operations of the Filing Protocol.

E.3.2 Opening and closing files

The **Open** operation is included in the FilingSubset and is identical to the corresponding operation of the Filing Protocol. All implementations must permit use of the **pathname** attribute for file identification in **Open**. The **parentID**, **type**, and **version** attributes must be supported in conjunction with the **pathname** attribute; however, the set of required values for each of these attributes may be limited (**nullFileID** for **parentID**; **tUnspecified**, **tAsciiText**, and **tDirectory** for **type**, and **lowestVersion** and **highestVersion** for **version**). This implies that a subset service implementation may not return an **AttributeTypeError** if the **parentID**, **type** or **version** attributes are specified on an **Open**; instead, an **AttributeValueError** may be returned if the value of the attribute is not one of the above.

The **Open** procedure may be rejected if **controls** is not the empty sequence or **directory** is not the **nullHandle**.

The Filing Protocol specifies that while a client has a file open, the file may not be deleted by other clients. The FilingSubset Protocol does not require this behavior; that is, a service implementation need not prevent a previously opened file from being deleted by other users, regardless of whether they are general interactive users or other network clients. The error to be reported by the service in this case is **HandleProblem[problem: invalid]**. Subset clients should be prepared to deal with directories or files which cannot be accessed even after a valid handle is obtained.

The **Close** procedure is defined to be identical to the Filing protocol.

E.3.3 Enumerating files

The FilingSubset Protocol defines a minimal file enumeration capability based on the **pathname** attribute. Attribute types and values are handled in a manner consistent with the Filing Protocol.

E.3.3.1 Scopes

The FilingSubset Protocol requires a minimum level of support for the scope types defined in Filing. Specifically, only the **count** and **filter** scope types must be supported. However, not all scope *values* for these types need be supported. At a minimum the **filter** scope type must permit the **matches** filter type and permit its use for at least the **pathname** attribute type.

Example:

The following is an example of the required **matches** filter type (assuming the Filing pathname syntax):

filter [matches [attribute: [type: pathname, value: "Document Archive/"]]]**
-- all files contained in the "Document Archive" directory --

E.3.3.2 Attributes

A FilingSubset implementation of the **List** operation must return a value for each of the attribute types requested by a client. Non-null values must always be returned for the attribute types **createdOn, dataSize, isDirectory, isTemporary, modifiedOn, pathname** and **type**. An appropriate non-null value must also be returned for the **childrenUniquelyNamed** attribute type if the listed file is a directory (a null value for this attribute is appropriate for non-directories).

Appropriate null values must be returned for all attribute types not supported by an implementation. Note that this behavior is consistent with Filing in that each attribute requested by a client is represented in the results of a **List** call whether or not the service can supply a meaningful value for the attribute.

Example:

An implementation which did not support the **position** attribute would return the following in response to a client request for this attribute's value:

attribute: [type: position, value: SEQUENCE 0 OF UNSPECIFIED]
-- an appropriate null-value response to an unsupported attribute --

E.3.3.3 Bulk data

Two *Bulk Data Transfer* [3] choices, **BulkData.immediateSink** and **BulkData.nullSink**, must be supported by all FilingSubset implementations of **List**. Other bulk data choices may be supported but are not required. In response to the use of an unsupported bulk data choice by a client, a subset service must report the error **TransferError [problem: aborted]**.

E.3.4 Storing files

The FilingSubset **Store** procedure is defined to be identical to the corresponding operation in Filing. At a minimum all implementations must permit the use of the **pathname** attribute for file identification. The **type** and **version** attributes must be permitted in conjunction with the **pathname** attribute; however, the set of required values for each of these attribute types is small: **tUnspecified**, **tAsciiText**, and **tDirectory** must be permitted for **type**, **highestVersion** for **version**.

Treatment of other client-supplied attributes depends on which attributes a subset service implements. A FilingSubset service must not reject a **Store** operation with an **AttributeTypeError** if the **createdOn**, **dataSize**, **isDirectory**, **pathname**, **type**, or **version** attributes is specified. An **AttributeValueError** may result if the accompanying value for any of these attributes is invalid.

Similarly, a FilingSubset service may not reject a **Store** operation with an **AttributeTypeError** if the **accessList**, **childrenUniquelyNamed**, **defaultAccessList**, **isTemporary**, **ordering**, or **subtreeSizeLimit** attributes is specified. The service must not report an **AttributeValueError** if the value of one of these attributes is as shown in the attribute tables of E.6. If other values are supplied and these are not supported by the subset implementation then an appropriate **AttributeValueError** must be reported.

A service implementation may support more than this minimal attribute capability but is not required to do so.

The **Store** procedure may be rejected if the **controls** argument is not the empty sequence or the **directory** argument is not the **nullHandle**.

The semantics of the **Store** operation permit both non-directory and directory files to be created. In the Filing Protocol a directory represents a special kind of file that may reference other files; a directory in Filing also has all of the characteristics of a non-directory file, namely attributes and content. FilingSubset implementations are not required to support this entire functionality.

A subset service may allow or refuse to allow directory files to be created by its clients at its discretion. If directory file creation is not permitted, the error **AccessError [problem: accessRightsInsufficient]** must be reported.

Note: The **Store** operation must always result in the creation of a *new* file or an error report; existing files are never overwritten by this operation.

Example:

A FilingSubset client attempting to create a new directory might specify (assuming the FilingProtocol pathname syntax):

```
Store [directory: nullHandle,
    attributes: [
        [type: pathname, value: "Projects/Correspondence/Pending"],
        [type: isDirectory, value: TRUE],
        [type: type, value: tDirectory]],
    controls: [],
    content: BulkData.nullSource,
    session: [token: [11B, 27734B], verifier: simpleVerifier]]
```

Note that the specification of **BulkData.nullSource** for **content** is equivalent to the specification of **BulkData.immediateSource** with zero bytes transferred.

A subset service supporting directory creation is not required to support directory files having data content. To reject a request to create a directory with content a service should report the error **AttributeValueError [problem: unreasonable, type: isDirectory]**.

E.3.4.1 Bulk data

Two *Bulk Data Transfer* [3] choices, **BulkData.immediateSource** and **BulkData.nullSource**, must be supported by all FilingSubset implementations of **Store**. Other bulk data choices may be supported but are not required. In response to the use of an unsupported bulk data choice by a client, a subset service must report the error **TransferError [problem: aborted]**.

E.3.5 Retrieving files

The FilingSubset **Retrieve** operation is identical to the Filing Protocol equivalent. This operation transfers the content of an existing file to the client.

E.3.5.1 Bulk data

Two *Bulk Data Transfer* [3] choices, **BulkData.immediateSink** and **BulkData.nullSink**, must be supported by all FilingSubset implementations of **Retrieve**. Other bulk data choices may be supported but are not required. In response to the use of an unsupported bulk data choice by a client, a subset service must report the error **TransferError [problem: aborted]**.

E.3.6 Deleting files

The FilingSubset **Delete** operation is identical to the Filing Protocol equivalent. This operation permits a client to delete existing files.

The Filing Protocol specifies that a **Delete** operation applied to a directory file will result in the deletion of the directory and all its descendants. This behavior is not required of all subset implementations, although consistency with Filing is desirable. If a subset service implementation does not support the Filing behavior then it should report the error **AccessError [problem: accessRightsInsufficient]**. Client implementations should always be prepared to deal with this failure report.

E.3.7 Summary of remote procedure restrictions

The FilingSubset Protocol defines the minimum capabilities which all implementations must provide. A subset client may attempt to use more than the minimum functionality required of a subset service but should not assume that the additional procedure or argument capabilities will be available. Similarly, a subset service implementation may support greater capabilities than those defined here, but must always provide the support expected by a client obeying the restrictions defined here.

This section summarizes the restrictions which govern the expected behavior of FilingSubset client and service implementations. The intent is to provide a convenient list of the argument values which must be allowed by all subset implementations and those exceptions which may validly result in an error response to the client. To assure maximum compatability, clients are advised to restrict their use of protocol capabilities to those listed here.

A subset implementation may report an appropriate error for a given procedure if any of the stated conditions is observed:

Open

- **directory** specifies a handle other than **nullHandle**.
- **controls** is not the empty sequence.
- **attributes** does not contain the **pathname** attribute type.
- **attributes** contains an attribute type other than **parentID**, **pathname**, **type**, or **version**.
- the **parentID** attribute specifies a value other than **nullFileID**.
- the **type** attribute specifies a type other than **tAsciiText**, **tDirectory**, or **tUnspecified**.
- the **version** attribute specifies a value other than **lowestVersion** or **highestVersion**.

Store

- **directory** specifies a handle other than **nullHandle**
- **controls** is not the empty sequence.
- **content** specifies a bulk data source type other than **BulkData.immediateSource** or **BulkData.nullSource**.
- **attributes** does not contain the **pathname** attribute type.
- **attributes** contains one of the attributes: **fileID**, **modifiedBy**, **modifiedOn**, **name**, **numberOfChildren**, **parentID**, **readBy**, **readOn**, **storedSize**, or **subtreeSize**.
- the **type** attribute specifies a type other than **tAsciiText**, **tDirectory**, or **tUnspecified**
- the **version** attribute specifies a value other than **highestVersion**.

Retrieve

- **content** specifies a bulk data sink type other than **BulkData.immediateSink** or **BulkData.nullSink**.

List

- **directory** specifies a handle other than **nullHandle**.
- **scope** includes a scope type other than **filter** or **count**.
- **filterType** specifies a filter type other than **matches**.
- a **matches** filter specifies an attribute type other than **pathname**.

- **listing** specifies a bulk data sink type other than **BulkData.immediateSink** or **BulkData.nullSink**.

E.4 Attributes

The Filing Protocol distinguishes two classes of attributes, namely *interpreted* and *uninterpreted* attributes. A service implementing Filing must support any attribute type described as interpreted in the standard and is required to preserve the values of uninterpreted attributes as explicitly set by the client. FilingSubset implementations are not required to conform to these requirements.

In order to specify the attribute requirements of FilingSubset implementations it is useful to distinguish three attribute classes rather than the two used by Filing. The FilingSubset attribute classes are *mandatory*, *implied*, and *optional*. The relationship of the attribute classes of the FilingSubset Protocol and those of the Filing Protocols is shown in Table E.1.

Filing	Attribute	FilingSubset
Interpreted	createdOn dataSize isDirectory modifiedOn pathname type	Mandatory
	accessList childrenUniquelyNamed defaultAccessList isTemporary ordering subtreeSizeLimit version	Implied
	checksum createdBy fileID modifiedBy name numberOfChildren ordering parentID position readBy readOn storedSize subtreeSize	Optional
Uninterpreted	uninterpreted	

Table E.1 Relationship of FilingSubset and Filing attribute classes.

E.4.1 Mandatory attributes

Mandatory attributes are those attributes which must be interpreted by all FilingSubset implementations. These attributes are guaranteed to be retained by any service implementing the FilingSubset and must also be accepted on specific procedure calls to the extent that they are legal arguments of the corresponding procedure of the Filing Protocol.

FilingSubset implementations must support the following mandatory attributes: **createdOn, dataSize, isDirectory, modifiedOn, pathname,** and **type**. Support for an attribute means that a service implementation will accept the attribute on a **Store** procedure, if properly specified, and will return the appropriate non-null value when requested with a **List** procedure.

E.4.1.1 createdOn

The **createdOn** attribute is as defined in the Filing Protocol.

E.3.1.2 dataSize

The Filing Protocol states that a file's **dataSize** attribute specifies the number of eight-bit bytes in the content of the file. The FilingSubset Protocol recognizes that it may not be straightforward for specific implementations to determine the actual content size of a file. Therefore, FilingSubset clients should regard the value of a file's **dataSize** attribute as an *estimate* of the file's size rather than the actual size itself.

E.4.1.3 isDirectory

The **isDirectory** attribute is as defined in the Filing Protocol. Typically this attribute need not be stored since it can be derived from context.

E.4.1.4 modifiedOn

The **modifiedOn** attribute is as defined in the Filing Protocol.

E.4.1.5 pathname

The FilingSubset Protocol requires that all implementations support the **pathname** attribute. This is the primary means by which a client may identify a file of interest. The value of the **pathname** attribute must specify the access path to a remote file in a form which is recognized by the particular service. This means that a FilingSubset client should make no assumptions regarding the syntax of this attribute since it may vary from service to service.

It should also be noted that the Filing Protocol permits its clients to make use of directory-relative pathnames in various operations. A FilingSubset client may *not* assume that this support will be provided by a given subset service; it is not required. All FilingSubset operations which accept directory handle arguments may report an error if a non-null directory handle is specified.

The syntax of the **pathname** attribute values returned by the **List** procedure should always be of an absolute form so that they may be used directly in subsequent operations.

E.4.1.6 type

All FilingSubset implementations must support at least the following values of the **type** attribute: **tAsciiText**, **tDirectory** and **tUnspecified** (refer to E.7 or appendix B for the definition of these types). The **type** attribute describes the nature of the content or attributes of a file in order to communicate to potential users of the file how the file is to be interpreted.

A service implementing the Filing Protocol interprets neither the type nor the content of a file. In order to facilitate the convenient interchange of text files between systems having different text file representations, the FilingSubset Protocol relaxes this behavior of the Filing Protocol.

A client may request that a file be transferred in a particular format by specifying the **type** attribute in the **Store** operation or in an **Open** call preceding a **Retrieve**. The **type** attribute value **tUnspecified** implies that the file content should be transferred uninterpreted. Round-trip data equality must be guaranteed by a subset service if the client specifies **tUnspecified** on storing and again on retrieval of a given file; this applies even to text files which the client designates as **tUnspecified**.

The **type** attribute value **tAsciiText** is used to indicate that a file's content should be interpreted as text and transferred using the **StreamOfAsciiText** encoding. This may require interpretation by the client or service to or from a native text representation. FilingSubset implementations should attempt to honor requests to interpret files as text files, since the mapping to and from native text format permits native mode text manipulation within each system. There are cases where round-trip data equality cannot be guaranteed for files of type **tAsciiText**; retrieving a file that has been stored as **tAsciiText** using the **tUnspecified** type may not have predictable results.

E.4.2 Implied attributes

Implied attributes are those non-mandatory attributes that obtain an implicit value when a new file is created using the Filing Protocol. To maintain consistency in the attribute behaviors defined in the Filing Protocol for this class of attribute, all subset implementations are required to permit the specification of the implied (default) value for each of these attributes (see E.6). The implied attribute types are: **accessList**, **childrenUniquely-Named**, **defaultAccessList**, **isTemporary**, **ordering**, **subtreeSizeLimit** and **version**.

A FilingSubset implementation of the **Store** procedure must always permit the specification of implied attributes. However, specification of an unsupported (non-default) value may validly be rejected with the error **AttributeValueError**.

E.4.3 Optional attributes

Optional attributes are those attributes which are uninterpreted by the Filing Protocol or which are not otherwise specified as mandatory or implied. If an implementation provides support for any of these additional attributes, that support must conform to the definition of the attribute in the Filing Protocol.

E.5 Remote errors

All errors from the Filing Protocol are similarly defined in the FilingSubset Protocol.

E.6 Procedures and attributes

The tables on the following pages describe the effects of FilingSubset procedures on attributes. If a procedure does not modify interpreted attributes, no table is shown. When a procedure modifies an attribute, a brief indication of the change is given. Where specification of an attribute will result in an error condition, the appropriate error is identified.

In the case of **List**, the table specifies the attribute values a service must return. Although this procedure does not modify attributes, its behavior is defined by the FilingSubset.

Table E.2 **List**

Attribute	If Requested[1]
accessList	returned
checksum	returned
childrenUniquelyNamed	returned (null for non-directory)
createdBy	returned
createdOn	non-null value must be returned
dataSize	non-null value must be returned[2]
defaultAccessList	returned
fileID	returned
isDirectory	non-null value must be returned
isTemporary	non-null value must be returned
modifiedBy	returned
modifiedOn	non-null value must be returned
name	returned
numberOfChildren	returned
ordering	returned
parentID	returned
pathname	non-null value must be returned
position	returned
readBy	returned
readOn	returned
storedSize	returned
subtreeSize	returned
subtreeSizeLimit	returned
type	non-null value must be returned
uninterpreted	returned
version	returned

[1] An appropriate value for each attribute supported by a FilingSubset implementation must be returned if requested. Even if an attribute is not supported, an appropriate *null* value must be returned (see E.3.3.2).

[2] The value returned for the **dataSize** attribute should be interpreted by the client as a close approximation of the actual content size.

Table E.3 **Open**

Attribute	If Requested[1]
accessList	illegal
checksum	illegal
childrenUniquelyNamed	illegal
createdBy	illegal
createdOn	illegal
dataSize	illegal
defaultAccessList	illegal
fileID	open if type supported[2]
isDirectory	illegal
isTemporary	illegal
modifiedBy	illegal
modifiedOn	illegal
name	open if type supported[2]
numberOfChildren	illegal
ordering	illegal
parentID	open if type supported[3]
pathname	file with this value is opened
position	illegal
readBy	illegal
readOn	illegal
storedSize	illegal
subtreeSize	illegal
subtreeSizeLimit	illegal
type	open if value supported[4]
uninterpreted	ignored
version	open if type supported[3]

1 A FilingSubset service implementation should report an **AttributeTypeError** if any of the attribute types designated as "illegal" is supplied in the arguments to **Open**.

2 FilingSubset implementations may support this attribute type but are not required to do so; if support is not provided, an **AttributeTypeError** should be reported when the client specifies the attribute.

3 FilingSubset implementations may report **AttributeValueError** if **parentID** is not equal to **nullFileID**, or **version** is not **lowestVersion** or **highestVersion** and the implementation does not support the attribute value.

4 The **type** attribute may be used to indicate a desired transfer format. This may imply a transformation of the actual file content as the file is transferred. If a specified **type** is not supported by the implementation, an **AttributeValueError** should be reported.

Table E.4 **Store**

Attribute	If a Parameter[1]	Supported Values	If not a Parameter[4]
accessList	set if value supported	**[defaulted:** TRUE**]**	set to **[defaulted:** TRUE**]**
checksum	set if type supported	**unknownChecksum**	set appropriately
childrenUniquelyNamed	set if value supported		implementation dependant
createdBy	set if type supported		currently logged-in user
createdOn	set		current date and time
dataSize	initial allocation (hint)		approximate file size
defaultAccessList	set if value supported	**[defaulted:** TRUE**]**	set to **[defaulted:** TRUE**]**
fileID	illegal, **AttributeTypeError**		system-assigned value
isDirectory	set		FALSE
isTemporary	set if value supported		FALSE
modifiedBy	illegal, **AttributeTypeError**		currently logged-in user
modifiedOn	illegal, **AttributeTypeError**		current date and time
name	set if type supported[2]		implementation dependant
numberOfChildren	illegal, **AttributeTypeError**		0
ordering	set if value supported	**defaultOrdering**	**defaultOrdering**
parentID	illegal, **AttributeTypeError**		**fileID** of resulting parent
pathname	set		consistent with ancestry
position	set if type supported		depends on parent's ordering
readBy	illegal, **AttributeTypeError**		""
readOn	illegal, **AttributeTypeError**		**nullTime**
storedSize	illegal, **AttributeTypeError**		set appropriately
subtreeSize	illegal, **AttributeTypeError**		set appropriately
subtreeSizeLimit	set if value supported	**nullSubtreeSizeLimit**	**nullSubtreeSizeLimit**
type	set if value supported[3]	**tUnspecified**, **tAsciiText**, or **tDirectory**	**tUnspecified** or **tDirectory**
uninterpreted	set if type supported		null
version	set if value supported	**highestVersion**	next available

1 FilingSubset implementations must treat attributes in one of four ways: 1) "illegal" attributes must be rejected with **AttributeTypeError**; 2) attributes designated "set" must not be rejected with **AttributeTypeError** and must normally accept non-null values (although an invalid *value* should be rejected, such as a string which is too long); 3) An attribute marked "set if value supported" must not result in an **AttributeTypeError**; the value of such an attribute may not result in an **AttributeValueError** if the value is one of the supported values shown above. Other values for these attributes may result in **AttributeValueError**; 4) An attribute designated "set if type supported" must be rejected with **AttributeTypeError** or **AttributeValueError** if the implementation does not fully support the type or value, respectively.

2 The **name** attribute, if supportecd, may not be specified with the **pathname** attribute.

3 The **type** attribute values **tAsciiText**, **tDirectory**, and **tUnspecified** must be supported.

4 These values must be assumed if the attribute type is supported by the implementation.

E.7 Program declaration

The complete declaration of the FilingSubset remote program is given below. All FilingSubset Courier definitions are identical to the corresponding definitions of the Filing Protocol.

```
FilingSubset: PROGRAM 1500 VERSION 1 =
BEGIN
    DEPENDS UPON
        BulkData (0) VERSION 1,
        Clearinghouse (2) VERSION 3,
        Filing (10) VERSION 6,
        Authentication (14) VERSION 3;

-- TYPES AND CONSTANTS --

-- Attributes (individual attributes defined later) --

AttributeSequence: TYPE = Filing.AttributeSequence;
AttributeTypeSequence: TYPE = Filing.AttributeTypeSequence;
allAttributeTypes: Handle = Filing.allAttributeTypes;

-- Controls --

ControlSequence: TYPE = Filing.ControlSequence;
ControlTypeSequence: TYPE = Filing.ControlTypeSequence;

-- Handles and Authentication --

Credentials: TYPE = Filing.Credentials;

SecondaryType: TYPE = Filing.SecondaryType;

Handle: TYPE = Filing.Handle;
nullHandle: Handle = Filing.nullHandle;

Session: TYPE = Filing.Session;
Verifier: TYPE = Authentication.Verifier;

-- Scopes --

ScopeSequence: TYPE = Filing.ScopeSequence;

-- REMOTE PROCEDURES --

-- Logging On and Off --

Logon: PROCEDURE [
    service: Clearinghouse.Name, credentials: Credentials, verifier: Verifier]
RETURNS [session: Session]
REPORTS [AuthenticationError, ServiceError, SessionError, UndefinedError] =
    Filing.Logon;
```

```
Logoff: PROCEDURE [session: Session]
REPORTS [AuthenticationError, ServiceError, SessionError, UndefinedError] =
    Filing.Logoff;

Continue: PROCEDURE [session: Session]
RETURNS [continuance: CARDINAL]
REPORTS [AuthenticationError, SessionError, UndefinedError] = Filing.Continue;

-- Opening and Closing Files --

Open: PROCEDURE [attributes: AttributeSequence, directory: Handle,
    controls: ControlSequence, session: Session]
RETURNS [file: Handle]
REPORTS [AccessError, AttributeTypeError, AttributeValueError, AuthenticationError,
    ControlTypeError, ControlValueError, HandleError, SessionError, UndefinedError] =
    Filing.Open;

Close: PROCEDURE [file: Handle, session: Session]
REPORTS [AuthenticationError, HandleError, SessionError, UndefinedError] =
    Filing.Close;

-- Deleting Files --

Delete: PROCEDURE [file: Handle, session: Session]
REPORTS [AccessError, AuthenticationError, HandleError, SessionError, UndefinedError] =
    Filing.Delete;

-- File Transfer --

Store: PROCEDURE [directory: Handle, attributes: AttributeSequence,
    controls: ControlSequence, content: BulkData.Source, session: Session]
RETURNS [file: Handle]
REPORTS [AccessError, AttributeTypeError, AttributeValueError, AuthenticationError,
    ConnectionError, ControlTypeError, ControlValueError, HandleError, InsertionError,
    SessionError, SpaceError, TransferError, UndefinedError] = Filing.Store;

Retrieve: PROCEDURE [file: Handle, content: BulkData.Sink, session: Session]
REPORTS [AccessError, AuthenticationError, ConnectionError, HandleError, SessionError,
    TransferError, UndefinedError] = Filing.Retrieve;

-- Listing Files in a Directory --

List: PROCEDURE [directory: Handle, types: AttributeTypeSequence,
    scope: ScopeSequence, listing: BulkData.Sink, session: Session]
REPORTS [AccessError, AttributeTypeError, AuthenticationError, ConnectionError,
    HandleError, ScopeTypeError, ScopeValueError, SessionError, TransferError,
    UndefinedError] = Filing.List;

-- REMOTE ERRORS --

-- problem with an attribute type or value --

AttributeTypeError: ERROR [problem: ArgumentProblem,type: AttributeType] =
    Filing.AttributeTypeError;
```

```
AttributeValueError: ERROR [problem: ArgumentProblem,type: AttributeType] =
    Filing.AttributeValueError;

-- problem with a control type or value --

ControlTypeError: ERROR [problem: ArgumentProblem, type: ControlType] =
    Filing.ControlTypeError;
ControlValueError: ERROR [problem: ArgumentProblem, type: ControlType] =
    Filing.ControlValueError;

-- problem with a scope type or value --

ScopeTypeError: ERROR [problem: ArgumentProblem, type: ScopeType] =
    Filing.ScopeTypeError;
ScopeValueError: ERROR [problem: ArgumentProblem, type: ScopeType] =
    Filing.ScopeValueError;

ArgumentProblem: TYPE = Filing.ArgumentProblem;

-- problem in obtaining access to a file --

AccessError: ERROR [problem: AccessProblem] = Filing.AccessError;
AccessProblem: TYPE = Filing.AccessProblem;

-- problem with a credentials or verifier --

AuthenticationError: ERROR [problem: Authentication.Problem, type: SecondaryType] =
    Filing.AuthenticationError;

-- problem with a bulk data transfer --

ConnectionError: ERROR [problem: ConnectionProblem] = Filing.ConnectionError;
ConnectionProblem: TYPE = Filing.ConnectionProblem;

-- problem with a file handle --

HandleError: ERROR [problem: HandleProblem] = Filing.HandleError;
HandleProblem: TYPE = Filing.HandleProblem;

-- problem during insertion in directory (or changing attributes) --

InsertionError: ERROR [problem: InsertionProblem] = Filing.InsertionError;
InsertionProblem: TYPE = Filing.InsertionProblem;

-- problem during random access operation --

RangeError: ERROR [problem: ArgumentProblem] = Filing.RangeError;

-- problem during logon or logoff --

ServiceError: ERROR [problem: ServiceProblem] = Filing.ServiceError;
ServiceProblem: TYPE = Filing.ServiceProblem;
```

```
-- problem with a session --

SessionError: ERROR [problem: SessionProblem] = Filing.SessionError;
SessionProblem: TYPE = Filing.SessionProblem;

-- problem obtaining space for file content or attributes --

SpaceError: ERROR [problem: SpaceProblem] = Filing.SpaceError;
SpaceProblem: TYPE = Filing.SpaceProblem;

-- problem during bulk data transfer --

TransferError: ERROR [problem: TransferProblem] = Filing.TransferError;
TransferProblem: TYPE = Filing.TransferProblem;

-- some undefined (and implementation-dependent) problem occurred --

UndefinedError: ERROR [problem: UndefinedProblem] = Filing.UndefinedError;
UndefinedProblem: TYPE = Filing.UndefinedProblem;

-- INTERPRETED ATTRIBUTE DEFINITIONS --

accessList: AttributeType = Filing.accessList;
AccessList: TYPE = Filing.AccessList;

checksum: AttributeType = Filing.checksum;
Checksum: TYPE = Filing.Checksum;

childrenUniquelyNamed: AttributeType = Filing.childrenUniquelyNamed;
ChildrenUniquelyNamed: TYPE = Filing.ChildrenUniquelyNamed;

createdBy: AttributeType = Filing.createdBy;
CreatedBy: TYPE = Filing.CreatedBy;

createdOn: AttributeType = Filing.createdOn;
CreatedOn: TYPE = Filing.CreatedOn;

dataSize: AttributeType = Filing.dataSize;
DataSize: TYPE = Filing.DataSize;

defaultAccessList: AttributeType = Filing.defaultAccessList;
DefaultAccessList: TYPE = Filing.DefaultAccessList;

fileID: AttributeType = Filing.fileID;
FileID: TYPE = Filing.FileID;
nullFileID: FileID = [0, 0, 0, 0, 0];

isDirectory: AttributeType = Filing.isDirectory;
IsDirectory: TYPE = Filing.IsDirectory;

isTemporary: AttributeType = Filing.isTemporary;
IsTemporary: TYPE = Filing.IsTemporary;
```

```
modifiedBy: AttributeType = Filing.modifiedBy;
ModifiedBy: TYPE = Filing.ModifiedBy;

modifiedOn: AttributeType = Filing.modifiedOn;
ModifiedOn: TYPE = Filing.ModifiedOn;

name: AttributeType = Filing.name;
Name: TYPE = Filing.Name;

numberOfChildren: AttributeType = Filing.numberOfChildren;
NumberOfChildren: TYPE = Filing.NumberOfChildren;

ordering: AttributeType = Filing.ordering;
Ordering: TYPE = Filing.Ordering;

pathname: AttributeType = Filing.pathname;
Pathname: TYPE = Filing.Pathname;

parentID: AttributeType = Filing.parentID;
ParentID: TYPE = Filing.ParentID;

position: AttributeType = Filing.position;
Position: TYPE = Filing.Position;

readBy: AttributeType = Filing.readBy;
ReadBy: TYPE = Filing.ReadBy;

readOn: AttributeType = Filing.readOn;
ReadOn: TYPE = Filing.ReadOn;

storedSize: AttributeType = Filing.storedSize;
StoredSize: TYPE = Filing.StoredSize;

subtreeSize: AttributeType = Filing.subtreeSize;
SubtreeSize: TYPE = Filing.SubtreeSize;

subtreeSizeLimit: AttributeType = Filing.subtreeSizeLimit;
SubtreeSizeLimit: TYPE = Filing.SubtreeSizeLimit;

type: AttributeType = Filing.type;
Type: TYPE = Filing.Type;

version: AttributeType = Filing.version;
Version: TYPE = Filing.Version;
lowestVersion: Version = 0;
highestVersion: Version = 177777B;

-- BULK DATA FORMATS --

-- Attribute Series Format, used in List --

StreamOfAttributeSequence: TYPE = CHOICE OF {
    nextSegment (0) => RECORD [
        segment: SEQUENCE OF AttributeSequence,
        restOfStream: StreamOfAttributeSequence],
    lastSegment (1) = > SEQUENCE OF AttributeSequence};

-- Line-oriented ASCII text file format, used in file interchange --

StreamOfAsciiText: TYPE = CHOICE OF {
    nextLine (0) = > RECORD [
        line: AsciiString,
        restOfText: StreamOfAsciiText],
    lastLine (1) = > AsciiString};

AsciiString: TYPE = RECORD [
    lastByteSignificant: BOOLEAN,
    bytes: SEQUENCE OF UNSPECIFIED];

-- FILE TYPES --

-- Clients are encouraged to use these predefined types to identify files that have the specified
characteristics, to promote information sharing --

tUnspecified: Type = 0; -- nothing is known about the content or attributes of a file of this
    type; it is useful for files that have a private format --

tDirectory: Type = 1; -- this file is a directory, and it has no content (only children) --

tAsciiText: Type = 7; -- this file is not a directory, and its content is comprised of line-oriented
ASCII data --

END;     -- of FilingSubset--
```

F. PATHNAME SYNTAX

In many applications it is necessary or useful to identify files by their user-sensible names and not file identifiers. A *pathname* specifies a hierarchical access path to a file by encoding the **name** and **version** attributes of its ancestors. A *qualified pathname* is a pathname prefixed by a designation of its network location.

In order to promote information sharing, it is strongly recommended that the following syntax be used in any user interface involving qualified pathnames (pathnames which include a service designation).

```
QualifiedPathname: = Service Pathname

Service: = (ClearinghouseName)

ClearinghouseName: = Clearinghouse.Name

Pathname : = NameVersionPairList

NameVersionPairList : = NameVersionPair | NameVersionPair/NameVersionPairList

NameVersionPair : = Name | Name!Version

Name : = [-- string with reserved characters escaped not exceeding 100 bytes in
          unescaped form --]

Version : = [-- string of digits with numeric value in the range (0..65535) --]
            | '+ -- i.e. highestVersion --
            | '-  -- i.e. lowestVersion --
```

Note that the syntax defined for pathnames is consistent with the definition of the **pathname** attribute in §4.2.2.5.

<u>Example</u>:

(Development:Office Systems:Xerox)Filing/Protocol/Pathname Syntax Definition!+

This pathname identifies the highest version of the file having the name "Pathname Syntax Definition" within the "Protocol" subdirectory of the "Filing" directory. The named file is stored with the "Development:Office Systems:Xerox" file service.