Xerox System Integration Standard

FILING PROTOCOL

XEROX

Notice

This *Xerox System Integration Standard* describes the Filing Protocol.

This document is being provided for informational purposes only. Xerox makes no warranties or representations of any kind relative to this document or its use, including the implied warranties of merchantability and fitness for a particular purpose. Xerox does not assume any responsibility or liability for any errors or inaccuracies that may be contained in the document, or in connection with the use of this document in any way.

The information contained herein is subject to change without any obligation of notice on the part of Xerox.

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

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 Systems Institute (XSI) Office
475 Oakmead Parkway, Bldg. 5
Sunnyvale, California 94086

TABLE OF CONTENTS

5. Remote errors 75

Appendices

.

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.

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.

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 = > RECORD [],
    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 3.0 **Primary and secondary credentials combinations**

Table 3.0 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 section 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 section 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.

<u>Examples</u>:

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;

<u>Arguments</u>: **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),**
    -- *full access* -- **fullAccess(177777B)}**;

AccessSequence: TYPE = SEQUENCE 5 OF **AccessType**;

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.
fullAccess	A shorthand access type denoting combined **read, write, owner, add,** and **remove** access permissions. This value may not be used in combination with any other access type.

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 section 4.2.7). A client's effective access may be empty. The access type **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 is omitted.

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** is the file handle of interest; **types** is a sequence of the types of control items that are desired; **session** is the client's session handle.

<u>Results</u>: **controls** is a sequence of control items corresponding 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 = [377777777777B];

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 section 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.

Results: **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.

Access: Add access is required to **directory** (if it is not the null handle).

Example:

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;
```

Arguments: **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.

Access: Read access is required to **file**.

Example:

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 **children-UniquelyNamed** 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	illegal		
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		

Effects of Filing Operations on Attributes

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				

Effects of Filing Operations on Attributes

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	

Effects of Filing Operations on Attributes

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	

Effects of Filing Operations on Attributes

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	<u>set</u>	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		

Effects of Filing Operations on Attributes

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		

Effects of Filing Operations on Attributes

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	

Effects of Filing Operations on Attributes

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		

Effects of Filing Operations on Attributes

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		

Effects of Filing Operations on Attributes

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	

Effects of Filing Operations on Attributes

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	

Effects of Filing Operations on Attributes

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		

Effects of Filing Operations on Attributes

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	

Effects of Filing Operations on Attributes

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 section 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 section 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 section 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 section 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: ["Xerox", "Office Systems", "John Q. Public"], access: [read, write]],
    [key: ["Xerox", "Office Systems", "UserGroup1"], access: [read]],
    [key: ["Xerox", "Office Systems", "UserAlias1"], 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* --

<u>Examples</u>:

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 --*

<u>Examples</u>:

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 section 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]

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 System Integration 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 System Integration 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 System Integration Standard. Stamford, Connecticut; May 1986; XNSS 058605.
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 System Integration 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 System Integration 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 System Integration 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 System Integration 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 System Integration 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.

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 Systems Institute (XSI) Office
475 Oakmead Parkway, Bldg. 5
Sunnyvale, California 94086

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 = [377777777777B];
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),
     -- full access -- fullAccess(177777B)};
AccessSequence: TYPE = SEQUENCE 5 OF AccessType;
```

-- 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 ■ > RECORD [],
        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 files 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 (section 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 (section 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" (section 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** (section 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** (section 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** (section 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** (section 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** (section 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 (section 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** (section 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** (section 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 (section 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** (section 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" (section 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 (section 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 (section 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 (section 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.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 requirement is to provide a minimal but useful level of service within the

context of the Filing Protocol. Specific 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 who 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).

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 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.

Note: Clients should not assume that an arbitrary FilingSubset service implementation will support multiple Courier connections for a single session; a service may not allow a session obtained on one connection to be used on any other connection.

Note: In order to foster compatibility 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.

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 are 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 are 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 it 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 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]**.

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 Filing Protocol pathname syntax):

FILING SUBSET

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.

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

114   FILING PROTOCOL

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 compatibility, 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.

Table E.1 **Relationship of FilingSubset and Filing attribute classes**

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	

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.4.1.2 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, childrenUniquelyNamed, defaultAccessList, isTemporary, ordering, subtreeSizeLimit** and **version**.

A FilingSubset implementation of the **Store** procedure must always permit the specification of implied attributes. Specification of an unsupported (non-default) value, however, 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 supported, 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--*

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: = ObjectString:DomainString:OrganizationString

ObjectName: = **Clearinghouse.ObjectName**

DomainName: = **Clearinghouse.DomainName**

OrganizationName: = **Clearinghouse.OrganizationName**

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 section 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.