Xerox System Integration Guide

# FILINGSUBSET IMPLEMENTOR'S GUIDE

XNSG 098609
September 1986

XEROX

**Notice**

This *Xerox System Integration Guide* 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 warrant that the use of the information herein will produce results in an intended manner.

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.

Printed in U.S.A. Publication number: 610P50681

# TABLE OF CONTENTS

1. Introduction 1

1.1 Purpose 1

1.2 Document organization 1

1.3 Document conventions 2

1.3.1 Notation 2

1.3.2 Notation for Courier examples 2

1.3.3 Notation for C examples 3

2. XNS protocol dependencies 5

2.1 Internet Transport Protocols 5

2.1.1 Relationship of transport to FilingSubset session 5

2.2 Courier and Bulk Data Protocols 6

2.3 Clearinghouse Protocol 6

2.3.1 Implementation with a Clearinghouse service 6

2.3.2 Implementation without a Clearinghouse service 7

2.4 Authentication Protocol 8

2.4.1 Implementation with an Authentication Service 8

2.4.2 Implementation without an Authentication Service 9

2.5 Time Protocol 10

3. Client implementation 11

3.1 Opening a session 11

3.3.1 Maintaining an open session 12

3.2 Closing a session 13

3.3 Enumerating a file(s) 13

3.4 Storing a file(s) 14

3.4.1 Overwriting an existing remote file 14

3.4.2 Bulk data transfer 15

3.5 Retrieving a file(s) 15

3.5.1 Overwriting an existing local file 16

3.5.2 Bulk data transfer 16

3.6 Deleting a file(s) 16

3.7 Creating a directory 17

4. Service implementation 19
4.1 Common data structures 19
4.1.1 Session handle 20
4.1.2 File handle 20
4.2 Common support 21
4.2.1 Session validation 21
4.2.2 Use of the continuance timer 21
4.2.3 File handle validation 22
4.3 Procedures 22
4.3.1 Logon 22
4.3.2 Logoff 24
4.3.3 Continue 24
4.3.4 Open 24
4.3.5 Close 26
4.3.6 List 26
4.3.7 Store 27
4.3.8 Retrieve 30
4.3.9 Delete 31
5. UNIX system interface 33
5.1 Attribute support 33
5.1.1 Mandatory attributes 34
5.1.2 Implied attributes 39
5.1.3 Optional attributes 39
5.2 Client procedure support 40
5.2.1 Continuance timer support 40
5.2.2 Determining mandatory attribute values 41
5.3 Service procedure support 42
5.3.1 Logon 43
5.3.2 Continue 45
5.3.3 Open 45
5.3.4 List 47
5.3.5 Store 48
5.3.6 Retrieve 51
5.3.7 Delete 52
6. VMS system interface 55
6.1 Attribute support 55
6.1.1 Mandatory attributes 56
6.1.2 Implied attributes 64

6.1.3 Optional attributes 64
6.2 Client procedure support 65
6.2.1 Continuance timer support 65
6.2.2 Determining mandatory attribute values 66
6.3 Service procedure support 68
6.3.1 Logon 69
6.3.2 Continue 69
6.3.3 Open 70
6.3.4 List 72
6.3.5 Store 76
6.3.6 Retrieve 79
6.3.7 Delete 79

## Appendices

A. References 81

# LIST OF TABLES

## Tables

4.1 Primary and secondary credentials combinations 23

5.1 UNIX supported values for implied attributes 39

6.1 VMS supported values for implied attributes 65

# 1. INTRODUCTION

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

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

## 1.1 Purpose

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

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

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

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

## 1.2 Document organization

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

## 1.3 Document conventions

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

### 1.3.1 Notation

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

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

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

### 1.3.2 Notation for Courier examples

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

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

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

then a call to that procedure would be denoted by:

```
Add [first: 7, second: 5]
```

and the call would yield the result:

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

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

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

**OVERFLOW**

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

**Overflow: ERROR [carry: CARDINAL] = 99;**

then an example of its use might be:

**Overflow [carry: 1]**

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

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

### 1.3.3 Notation for C examples

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

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

# 2. XNS PROTOCOL DEPENDENCIES

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

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

## 2.1 Internet Transport Protocols

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

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

### 2.1.1 Relationship of transport connection to FilingSubset session

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

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

## 2.2 Courier and Bulk Data Protocols

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

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

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

## 2.3 Clearinghouse Protocol

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

### 2.3.1 Implementation with a Clearinghouse service

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

#### 2.3.1.1 Location of a Clearinghouse server

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

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

#### 2.3.1.2 Location and description of file services

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

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

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

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

## 2.3.2 Implementation without a Clearinghouse service

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

### 2.3.2.1 Location of a Clearinghouse server

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

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

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

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

### 2.3.2.2 Location and description of file services

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

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

## 2.4 Authentication Protocol

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

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

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

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

### 2.4.1 Implementation with an Authentication Service

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

### 2.4.1.1 Primary credentials

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

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

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

### 2.4.1.2 Secondary credentials

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

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

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

## 2.4.2 Implementation without an Authentication Service

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

### 2.4.2.1 Primary credentials

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

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

### 2.4.2.2 Secondary credentials

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

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

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

## 2.5 Time Protocol

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

# 3. CLIENT IMPLEMENTATION

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

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

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

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

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

## 3.1 Opening a session

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

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

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

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

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

## 3.1.1 Maintaining an open session

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

Once the **Logon** has completed, the client issues a **Continue** to the service to determine the service specific continuance value. The value returned is specified in seconds, so the client should decrease this by some factor (i.e., $\frac{1}{3}$) and establish a timeout mechanism which will issue another **Continue** at the expiration of the interval. This allows the client to issue the next **Continue** well before the service timeout interval.

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

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

## 3.2 Closing a session

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

## 3.3 Enumerating a file(s)

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

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

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

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

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

## 3.4 Storing a file(s)

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

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

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

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

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

### 3.4.1 Overwriting an existing remote file

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

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

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

### 3.4.2 Bulk data transfer

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

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

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

## 3.5 Retrieving a file(s)

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

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

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

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

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

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

### 3.5.1 Overwriting an existing local file

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

### 3.5.2 Bulk data transfer

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

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

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

## 3.6 Deleting a file(s)

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

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

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

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

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

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

## 3.7 Creating a directory

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

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

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

# 4. SERVICE IMPLEMENTATION

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

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

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

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

## 4.1 Common data structures

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

### 4.1.1 Session handle

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

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

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

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

### 4.1.2 File handle

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

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

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

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

## 4.2 Common support

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

### 4.2.1 Session validation

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

#### 4.2.1.1 Verifier validation

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

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

#### 4.2.1.2 Session consistency validation

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

### 4.2.2 Use of the continuance timer

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

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

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

### 4.2.3 File handle validation

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

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

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

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

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

## 4.3 Procedures

### 4.3.1 Logon

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

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

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

| Secondary<br>Primary | none | simple | strong |
|---|---|---|---|
| nullPrimaryCredentials | legal | legal | illegal |
| simple | legal | legal | illegal |
| strong | legal | illegal | legal |

Table 4.1 Primary and secondary credentials combinations

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

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

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

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

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

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

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

### 4.3.2 Logoff

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

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

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

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

### 4.3.3 Continue

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

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

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

### 4.3.4 Open

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## 4.3.5 Close

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

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

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

## 4.3.6 List

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## 4.3.7 Store

**Store: PROCEDURE [directory: Handle, attributes: AttributeSequence,**
**controls: ControlSequence, content: BulkData.Source, session: Session]**
**RETURNS [file: Handle]**

REPORTS [AccessError, AttributeTypeError, AttributeValueError,
AuthenticationError, ConnectionError, ControlTypeError, ControlValueError,
HandleError, InsertionError, SessionError, SpaceError, TransferError,
UndefinedError] = Filing.Store;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## 4.3.8 Retrieve

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

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

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

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

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

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

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

FilingSubset services are not required to permit the retrieval of directory files. A service which does not allow this reports the error **AccessError [problem: accessRightsInsufficient]**. The **isDirectory** entry in the file context block is used to determine if the file is indeed a directory.

## 4.3.9 Delete

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

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

The session handle is verified, the continuance mechanism rearmed, and the file handle verified, as described in section 4.2. The file specified by the file handle will then be deleted. Different actions may be taken, depending upon whether the file is a directory as determined by examining the **isDirectory** entry in the file context block. Upon successful deletion of the file, the associated file handle is removed from the open file queue in the session context block and released.

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

# 5. UNIX SYSTEM INTERFACE

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

This section describes those interface procedures required by the client and service implementations presented in sections 3 and 4. These are by no means the only method for providing the facility desired; they have been chosen either because they have actually been tested, or are more likely to be portable between various versions of UNIX. In those cases where differences arise between implementations on UNIX 4.2BSD, UNIX 4.3BSD, and UNIX System V, these differences are noted.

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

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

## 5.1 Attribute Support

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

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

- clients to retrieve and retain the attributes when dealing with remote services

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

## 5.1.1 Mandatory attributes

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

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

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

### 5.1.1.1 createdOn

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

*[Retention]*

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

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

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

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

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

set_create_time(file_context_block)
file_handle    *file_context_block;
{

    time_t     time_buffer[2];
    time_t     time();

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

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

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

*[Retrieval]*

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

### 5.1.1.2 dataSize

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

*[Retention]*

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

*[Retrieval]*

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

### 5.1.1.3 isDirectory

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

*[Retention]*

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

*[Retrieval]*

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

### 5.1.1.4 modifiedOn

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

*[Retention]*

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

*[Retrieval]*

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

### 5.1.1.5 pathname

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

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

*[Retention]*

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

*[Retrieval]*

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

### 5.1.1.6 type

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

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

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

*[Retention]*

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

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

*[Retrieval]*

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

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

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

```
#include   <stdio.h>

#define    CHARS_TO_READ  2048

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

    returns:
        Cardinal containing Filing defined file type
*/

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

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

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

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

## 5.1.2 Implied attributes

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

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

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

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

Table 5.1 **UNIX supported values for implied attributes**

## 5.1.3 Optional attributes

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

## 5.2 Client procedure support

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

### 5.2.1 Continuance timer support

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

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

```
#include   <signal.h>

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

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


/*
    routine:
        set_continuance_timer

    called after a successful Logon
*/

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

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

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

```
/*
    routine:
        reset_continuance_timer

    called after any FilingSubset procedure call
*/

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

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

```
/*
    routine:
        cancel_continuance_timer

    called after a successful Logoff
*/

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

## 5.2.2 Determining mandatory attribute values

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

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

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

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

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

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

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

    datasize= file_stat.st_size;                                 /* dataSize */

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

return(-1);
}
```

## 5.3 Service procedure support

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

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

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

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

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

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

## 5.3.1 Logon

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

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

The verifyandposition_user routine is defined to perform these functions.

```
#include   <pwd.h>

#define    SERVICE_ROOT    "/"

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

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

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

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

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

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

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

    return(-1)
}
```

## 5.3.2 Continue

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

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

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

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

`set_continuance_timer` initially establishes the timeout mechanism.

```
#include    <signal.h>

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

/*
    routine:
        set_continuance_timer
*/

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

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

## 5.3.3 Open

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

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

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

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

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

        file_context_block entries filled in
*/

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

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

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

            case ENOTDIR:                         /* directory doesn't exist */
            case ENOENT:                          /* file doesn't exist */
                error_value.problem= Filing_fileNotFound;
                return(error_value);

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

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

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

return(-1);
}
```

## 5.3.4 List

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

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

The routine `list_directory` is defined to perform this function. [Note: A slightly altered version of the `get_attributes` routine presented in section 5.2.2 can be used to determine the mandatory attributes for a file.]

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

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

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

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

    strcpy(command,"/bin/ls -1d ");                  /* form appropriate command */
    strcat(command,file_spec);

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

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

## 5.3.5 Store

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

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

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

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

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

        file_context_block->file_desc filled in
*/

Filing_Error create_file(file_context_block)
file_handle    *file_context_block;
{

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

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

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

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

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

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

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

    return(-1);
}
```

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

```
{
    int count;

    ....

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

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

    ....
}
```

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

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

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

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

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

    return(-1);
}
```

## 5.3.6 Retrieve

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

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

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

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

        file_context_block->file_desc filled in
*/

Filing_Error open_file(file_context_block)
file_handle    *file_context_block;
{

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

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

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

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

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

    return(-1);
}
```

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

## 5.3.7 Delete

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

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

```
#include   <errno.h>

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

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

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

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

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

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

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

    return(-1);
}
```

# 6. VMS SYSTEM INTERFACE

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

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

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

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

## 6.1 Attribute Support

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

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

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

### 6.1.1 Mandatory attributes

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

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

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

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

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

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

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

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

```
{
    double  time, gmt_offset;

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

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

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

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

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

The routine `convert_xns_time` illustrates this:

```
#include    ssdef

#define     XNS_EARLIEST_TIME  2114294400

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

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

double  convert_xns_time(xns_time)
LongCardinal    xns_time;
{

    double      vms_time;
```

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

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

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

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

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

        return(vms_time);
    }
```

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

```
    #include   ssdef

    #define XNS_EARLIEST_TIME     2114294400

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

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

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

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

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

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

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

}
```

### 6.1.1.1 createdOn

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

*[Retention]*

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

```
#include   rms
#include   ssdef

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

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

```
set_create_time(file_context_block)
file_handle    *file_context_block;
{

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

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

*[Retrieval]*

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

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

### 6.1.1.2 dataSize

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

*[Retention]*

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

*[Retrieval]*

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

```
#include fatdef

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

LongCardinal compute_datasize(file_attributes)
struct fat         vms_value;
{

    int block_count;

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

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

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

### 6.1.1.3 isDirectory

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

*[Retention]*

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

*[Retrieval]*

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

### 6.1.1.4 modifiedOn

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

*[Retention]*

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

*[Retrieval]*

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

### 6.1.1.5 pathname

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

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

*[Retention]*

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

*[Retrieval]*

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

### 6.1.1.6 type

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

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

*[Retention]*

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

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

*[Retrieval]*

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

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

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

```
#include    fatdef

/*
    routine:
        get_type
    input:
        record_format          - VMS record format (FAB$B_RFM)
        record_atttributes     - VMS record attributes (FAB$B_RAT)
```

```
   returns:
      Cardinal containing XNS type value
*/

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

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

## 6.1.2 Implied attributes

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

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

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

## 6.1.3 Optional attributes

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

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

Table 6.1 **VMS supported values for implied attributes**

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

## 6.2 Client procedure support

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

### 6.2.1 Continuance timer support

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

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

```
#include   signal

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

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


/*
    routine:
        set_continuance_timer
        called after a successful Logon
    */

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

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

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

```
    /*
        routine:
            reset_continuance_timer

        called after any FilingSubset procedure call
    */

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

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

```
    /*
        routine:
            cancel_continuance_timer

        called after a successful Logoff
    */

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

## 6.2.2 Determining mandatory attribute values

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

The stat routine returns the various file dates in a form similar to UNIX. The conversion mechanism described in section 6.1.1 is not required to convert this value to XNS format.

Instead, the conversion mechansim for use with the stat routine described in section 6.1.1.1 is used. The XNS time is computed by adding the returned value to the constant 2177452800, which represents the base time (00:00:00 GMT Jan. 1, 1970).

```
#include stat

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

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

/*
    routine
        get_attributes
    input:
        pathname   - service-specific pathname of file
    returns:
        -1  - success
         1  - error
*/

get_attributes(pathname)
char   *pathname;
{
    struct stat    file_stat;

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

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

    datasize= file_stat.st_size;                             /* dataSize */

    if ( file_stat.st_mode & S_IFDIR ) {                     /* directory file */
        isdirectory= TRUE;
        type= tDirectory;
    } else {                                                 /* non-directory */
        isdirectory= FALSE;
        type= get_type(file_stat.st_fab_rfm,file_stat.st_fab_rat);
    }

    return(-1);
}
```

## 6.3 Service procedure support

A FilingSubset service implemented on the VMS operating system may use various VMS, RMS, and VAX C routines to access the local file system and provide VMS-specific procedure support. This section presents detailed examples of this interaction.

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

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

- a copy of the **pathname** attribute value as specified on the **Open** or **Store**
- a cardinal identifying the file type requested by the client on the **Open**
- a cardinal specifying the file type as determined by the service
- a cardinal specifying the **dataSize** value for the file
- a boolean specifying the **isDirectory** value for the file
- a long cardinal specifying the **createdOn** value for the file in XNS format
- a long cardinal specifying the **modifiedOn** value for the file in XNS format
- FAB, XABDAT, RAB, and NAM file structures used when accessing a file
- an appropriate buffer for use with the RAB structure

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

```
typedef file_handle {
    char           *pathname;          /* pointer to pathname value */
    Cardinal       type;               /* client requested type (from Open) */
    Cardinal       truetype;           /* file system file type */
    Cardinal       datasize;           /* dataSize value */
    Boolean        isdirectory;        /* isDirectory */
    LongCardinal   createon;           /* createdOn value */
    LongCardinal   modifiedon;         /* modifiedOn value */
    struct fab     file_fab;           /* file access block (FAB) */
    struct rab     file_rab;           /* record access block (RAB) */
    struct xabdat  file_xab;           /* extended attribute block (XABDAT) */
    struct nam     file_nam;           /* name block (NAM) */
    char           file_buffer[32767]; /* input/output buffer 32767= max size */
};
```

## 6.3.1 Logon

The **Logon** procedure is responsible for validating the user attempting the connection and, if successful, altering the process ownership to that of the user. This alteration of ownership ensures that the process is subject to the normal access/protection mechanisms employed by the VMS operating system when subsequent procedure calls request access to files on the service. The user name and password entries of the secondary credentials supplied on the **Logon** are validated against the standard VMS UAF file. Once this has been completed, the UIC and privileges of the process are changed to that of the respective user, as determined from the authorization file entry for the user.

The process is also positioned to the appropriate root directory for the service, which corresponds to a VMS disk/directory pair (generally the VMS root, [000000], on a specific disk). This provides a VMS disk and directory which can be associated with **nullHandle** as the root for the service. The examples in this section use the external variable `service_root_device` to specify the default device for the service. The Logon procedure will set this variable to the appropriate value.

VMS does not export routines to perform these services and the nature of these routines is such that they are proprietary to VMS. Because of this, no routines are presented here. Instead, it is assumed that implementors of a VMS service will have access to VMS internal documentation which describes the VMS mechanisms for performing the required functions.

## 6.3.2 Continue

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

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

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

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

`set_continuance_timer` initially establishes the timeout mechanism.

```
#include   signal

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

```
/*
    routine:
        set_continuance_timer
*/

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

The `reset_continuance_timer` and `cancel_continuance_timer` routines are identical to the client routines specified in section 6.2.1.

## 6.3.3 Open

The **Open** procedure opens a file for subsequent access by the client. The file is identified by the value specified for the **pathname** attribute. VMS supports multiple versions, so the **version** values **lowestVersion** and **highestVersion** will indicate different files if more than one version exists. If the pathname does not contain a version specification, the string ";-0" or ";0" can be catenated to the **pathname** value to indicate the lowest version or highest version of a file, respectively. When an explicit **version** value is specified along with a **pathname** value that contains a version, the explicit **version** value will take precedence over the **pathname** value.

The **Open** routine first performs a call to `stat_file` to determine values for the **dataSize**, **isDirectory**, and **type** attributes for the desired file. This allows the subsequent file transfer procedures to access necessary information, simply by examining the file context block.

```
#include   stat

/*
    routine:
        stat_file
    input:
        pointer to file handle
    returns:
        -1  - success
        1   - failure (specific errors will be determined on the subsequent file
              open)

        file_context_block filled in
*/

stat_file(file_context_block)
file_handle    *file_context_block;
{
    struct stat    file_stat;
    /* stat does not return detailed errors so specific errors will be returned
    when the actual open is attempted */
    if ( stat(file_context_block->pathname,&file_stat) == -1 )
        return(1);

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

    if ( file_stat.st_mode & S_IFDIR ) {
        file_context_block->isdirectory= TRUE;                    /* directory */
        file_context_block->truetype= Filing_tDirectory;
    } else {
        file_context_block->isdirectory= FALSE;                   /* non-directory */
        file_context_block->truetype= get_type(file_stat.st_fab_rfm,
                                                  file_stat.st_fab_rat);
    }

    return(-1);
}
```

The routine open_file is subsequently called to open the file. This routine will be called regardless of any errors that are returned from stat_file, since specific error conditions cannot be determined until the open is atempted. The only possible errors are: **accessRightsInsufficient** if the file cannot be accessed, **fileNotFound** if the file or some component of the pathname does not exist, and **accessRightsIndeterminate** if any other error occurs.

```
#include    rms

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

        file_context_block entries filled in
*/

Filing_Error    open_file(file_context_block)
file_handle     *file_context_block;
{

    int error;
    Filing_Error  error_value;                    /* Filing error, problem pair */

    error_value.error= Filing_AccessError;        /* set to Filing AccessError */

    file_context_block->file_fab= cc$rms_fab;                     /* initialize FAB */
    file_context_block->file_fab.fab$l_fna= cbptr->pathname;
    file_context_block->file_fab.fab$b_fns= strlen(cbptr->pathname);
    file_conterxt_block->file_fab.fab$b_fac= FAB$M_GET | FAB$M_BRO;
    file_context_block->file_fab.fab$b_shr= FAB$M_NIL;
    error= sys$open(&file_context_block->file_fab);      /* open file */
    if ( error != RMS$_NORMAL ) {
        if ( error == RMS$_FNF )                         /* no such file */
            error_value.problem= Filing_fileNotFound;
        else if ( error == RMS$_PRV )                    /* user has no access */
            error_value.problem= Filing_accessRightsInsufficient;
        else                                             /* all other errors */
            error_value.problem= Filing_accessRightsIndeterminate;
        return (error_value);
    }

    file_context_block->file_rab= cc$rms_rab;                /* initialize RAB */
    file_context_block->file_rab.rab$l_fab= &file_context_block->file_fab;
    file_context_block->file_rab.rab$l_ubf= file_context_block->file_buffer;
    file_context_block->file_rab.rab$w_usz= MAX_RECORD_SIZE;

    error= sys$connect(&file_context_block->file_rab);   /* connect rab to fab */

    if ( error != RMS$_NORMAL ) {
        error_value.problem= Filing_accessRightsIndeterminate;
        return (error_value);
    }

return(-1);
}
```

## 6.3.4 List

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

This function is easily accommodated through the use of the IO$_ACCESS QIO to the Files-11 ACP, which lists a directory and returns the files matching some file name criteria along with various file characteristics requested. The file names are returned in ascending order by name, which is the Filing Protocol **defaultOrdering** .

The routine list_directory is defined to perform this function. This routine performs a VMS SYS$ASSIGN to the service disk device, as specified by service_root_device. The routine get_directory_id is then called to parse the specified pathname to return the VMS file identifier for the appropriate directory. list_directory then repetitively performs an IO$ACCESS QIO to retrieve the next filename matching the **pathname** specification taken from the **filter** of type **matches**. Appropriate VMS file characteristics are requested so that the values for the FilingSubset defined mandatory attributes can be returned to the client. ATR$C_UCHAR is used to determine the value for the **isDirectory** attribute. The **type** and **dataSize** values are determined from the values returned from ATR$C_RECATTR. The **createdOn** and **modifiedOn** values come from ATR$C_CREDATE and ATR$C_REVDATE, respectively. The resulting filename buffer specified on the QIO will return the VMS specific **pathname** value.

The error **AccessError accessRightsInsufficient** is returned if an error is encountered during the SYS$ASSIGN or returned from get_directory_id. If an error occurs when accessing an individual file, that file is simply omitted from the list returned.

```
#include rms
#include ssdef
#include descrip
#include atrdef               /* assumes include files which define public */
#include fibdef               /* structures ATR, FIB, FAT, FCH and IOSB */
#include fatdef               /* these are not necessarily included in VAX C */
#include fchdef
#include iosb

#define    EVENT_FLAG     2       /* event flag for QIO use */

extern char    *service_root_device;     /* VMS device known as service root */

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

Filing_Error   list_directory(file_spec)
char   *file_spec;            /* pathname attribute from filter of type matches */

{
    int            error;
    struct fib     fib= 0;
    struct atr     attributes[5];
    char           file_name[NAM$C_MAXRSS];
    char           result_name[NAM$C_MAXRSS];
    long           length;
    double         creation_date;
    double         revision_date;
    struct fat     record_attributes;
    long           file_characteristics;
    short          channel;
    struct iosb    io_status;
    Filing_Error   error_value;

           /* following are used to hold mandatory attributes until added to
           outgoing bulk data stream */
    LongCardinal   createdon;
    LongCardinal   modifiedon;
    Cardinal       datasize;
    Boolean        isdirectory
    Cardinal       type;
    String         pathname;
```

```
struct dsc$descriptor_s result_name_descriptor;
struct dsc$descriptor_s device_descriptor;
struct dsc$descriptor_s fib_descriptor;
struct dsc$descriptor_s file_descriptor;

error_value.error= Filing_AccessError;        /* set to Filing AccessError */

fib.fib$w_nmctl= FIB$M_WILD;                  /* turn on wildcard mechanism */
                                                  /* fill in directory id */
if ( get_directory_id(file_spec,&fib) == 1 ) {
   error_value.problem= Filing_accessRightsInsufficient;
   return(error_value);
}

                              /* VMS descriptor setup */
                                          /* file name returned from QIO */
result_name_descriptor.dsc$a_pointer= result_name;       /* pathname value */
result_name_descriptor.dsc$w_length= NAM$C_MAXRSS+1;
result_name_descriptor.dsc$b_class= DSC$K_CLASS_S;
result_name_descriptor.dsc$b_dtype= DSC$K_DTYPE_T;

                                          /* device known as service root */
device_descriptor.dsc$a_pointer= service_root_device;
device_descriptor.dsc$w_length= strlen(service_root_device);
device_descriptor.dsc$b_class= DSC$K_CLASS_S;
device_descriptor.dsc$b_dtype= DSC$K_DTYPE_T;

                                                  /* file identifier block */
fib_descriptor.dsc$a_pointer= &fib;
fib_descriptor.dsc$w_length= FIB$K_SMALLSIZE;
fib_descriptor.dsc$b_class= DSC$K_CLASS_S;
fib_descriptor.dsc$b_dtype= DSC$K_DTYPE_T;

                                                  /* input pathname value */
file_descriptor.dsc$a_pointer= file_spec;
file_descriptor.dsc$w_length= strlen(file_spec);
file_descriptor.dsc$b_class= DSC$K_CLASS_S;
file_descriptor.dsc$b_dtype= DSC$K_DTYPE_T;

                      /* set up VMS attribute structures to be returned */
attributes[0].atr$w_size= ATR$S_UCHAR;        /* file characteristics */
attributes[0].atr$w_type= ATR$C_UCHAR;        /* isDirectory value */
attributes[0].atr$l_addr= &file_characteristics;

attributes[1].atr$w_size= ATR$S_RECATTR;      /* record attributes */
attributes[1].atr$w_type= ATR$C_RECATTR;      /* type and dataSize values */
attributes[1].atr$l_addr= &record_attributes;

attributes[2].ATR$W_SIZE= ATR$S_CREDATE;      /* creation date */
attributes[2].atr$w_type= ATR$C_CREDATE;      /* createdOn value */
attributes[2].atr$l_addr= &creation_date;
```

```
    attributes[3].atr$w_size= ATR$S_REVDATE;          /* revision date */
    attributes[3].atr$w_type= ATR$C_REVDATE;          /* modifiedOn value */
    attributes[3].atr$l_addr= &revision_date;

    attributes[4].atr$w_size= attributes[4].atr$w_type= 0;

    if ( (error= sys$assign(&device_descriptor,&channel,0,0)) !=  SS$_NORMAL) {
        error_value.problem= Filing_accessRightsInsufficient;
        return(error_value);
    }

    while (TRUE) {                /* get each file matching file_spec */
        fib.fib$w_fid[0]= fib.fib$w_fid[1]= fib.fib$w_fid[2]= 0;

                                 /* returns next file and characteristics */
        sys$qiow(EVENT_FLAG,channel,IO$_ACCESS|IO$M_ACCESS,&io_status,0,0,
            &fib_descriptor,&file_descriptor,&length,&result_name_descriptor,
            attributes,0);

                                     /* break out when no file returned */
        if ( io_status.s_status != SS$_NORMAL ) {
            if ( fib.fib$l_wcc && (io_status.s_status != SS$_NOMOREFILES) )
                continue;        /* any other error simply omit file from list */
            else   break;
        }
                                     /* determine mandatory attribute values */
        strncpy(pathname,result_name,length);     /* pathname from result_name */
        *(pathname+length)= '/0';

                              /* convert createdOn, modifiedOn to XNS format */
        convert_vms_time(&creation_date,createdon);
        convert_vms_time(&revision_date,modifiedon);

                              /* compute dataSize value */
        datasize= compute_datasize(record_attributes);

                              /* set isDirectory and type as appropriate */
        if ( file_characteristics & FCH$M_DIRECTORY ) {
            isdirectory= TRUE;
            type= tDirectory;
        } else {
            isdirectory= FALSE;
            type= get_type(record_attributes.fat$b_rtype,
                                     record_attributes.fat$b_rattrib);
        }
/* insert implementation specific routines here:
        - make an attribute sequence from createdon, datasize, isdirectory,
             modifiedon, pathname and type variables
            (if other non-mandatory attributes arerequested, appropriate values
            must also be returned)
        - write the attribute sequence to the bulk data stream
*/
    return(-1);
}
```

```
/*
    routine:
        get_directory_id
    input:
        file_spec   - pointer to pathname value
        file_fib    - pointer to fib structure to fill in directory id info
    returns:
        -1  - success
        1   - error occurred
*/

get_directory_id(file_spec,file_fib)
char            *file_spec;
struct FIB      *file_fib;

{
    struct fab      file_fab;
    struct nam      file_nam;

    file_fab= cc$rms_fab;                           /* initialize fab */
    file_fab.fab$l_fna= file_spec;
    file_fab.fab$b_fns= strlen(file_spec);

    file_nam= cc$rms_nam;                           /* initialize nam */
    file_fab.fab$l_nam= &file_nam;

                                        /* use sys$parse to obtain directory id */
    if ( sys$parse(&file_fab) != SS$_NORMAL )
        return(1);

    file_fib->fib$w_did[0]= file_nam.nam$w_did[0];
    file_fib->fib$w_did[1]= file_nam.nam$w_did[1];
    file_fib->fib$w_did[2]= file_nam.nam$w_did[2];

    return(-1);
}
```

### 6.3.5 Store

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

After the **Store** routine has validated the argument and attribute values, a file handle is allocated. The `create_file` routine is then called to actually create the file with the specified attributes and default values for unspecified mandatory attributes. A FilingSubset service will always create a new version of a file. If a specified pathname includes a specific version and that version already exists, the service will return **InsertionError fileNotUnique**. Appropriate values for **AccessProblem** are returned, if the file cannot be created for any other reason.

```
#include   rms

#define     MAX_RECORD_SIZE    32767                  /* maximum VMS record size */
/*
    routine:
        create_file
    input:
        pointer to file handle
    returns:
        -1  - success
        else Filing Error, Problem

        file_context_block structures filled in
*/

Filing_Error create_file(file_context_block)
file_handle    *file_context_block;
{

    int             error;
    Filing_Error    error_value;                      /* Filing error, problem pair */

    error_value.error= Filing_AccessError;            /* set default Filing error */

    file_context_block->file_fab= cc$rms_fab;             /* initialize FAB */
    file_context_block->file_fab.fab$l_fna= file_context_block->pathname;
    file_context_block->file_fab.fab$b_fns= strlen(file_context_block->pathname);
    file_context_block->file_fab.fab$l_alq= file_context_block->dataSize/512 + 1;
    file_context_block->file_fab.fab$l_fop= FAB$M_MXV;
    file_context_block->file_fab.fab$w_mrs= MAX_RECORD_SIZE;

                                                  /* pick VMS type from type value */
    if ( file_context_block->type == Filing_tAsciiText ) {
        file_context_block->file_fab.fab$b_rfm= FAB$C_VAR;
        file_context_block->file_fab.fab$b_rat= FAB$M_CR;
    } else if ( file_context_block->type == Filing_tUnspecified ) {
        file_context_block->file_fab.fab$b_rfm= FAB$C_UDF;
    }
                                                          /* initialize XAB */
    file_context_block->file_fab.fab$l_xab= &file_context_block->file_xab;
    file_context_block->file_xab= cc$rms_xabdat;
                                             /* save createdOn, per Section 6.1.1.1 */
    set_create_time(file_context_block);

    error= sys$create(&file_context_block->file_fab);     /* create file */
    if ( error != RMS$_NORMAL ) {                         /* check for errors */
        if ( error == RMS$_PRV )                          /* no privilege */
            error_value.problem= Filing_accessRightsInsufficient;
        else if ( error == RMS$_FEX ) {                   /* file exists */
            error_value.error= Filing_InsertionError;
            error_value.problem= Filing_fileNotUnique;
        } else                                      /* all others */
            error_value.problem= Filing_accessRightsIndeterminate;
        return(error_value);
    }

    file_context_block->file_rab= cc$rms_rab;       /* initialize RAB */
    file_context_block->file_rab.rab$l_fab= &file_context_block->file_fab;
    file_context_block->file_rab.rab$rbf= file_context_block->file_buffer;
    file_context_block->file_rab.rab$w_rsz= MAX_RECORD_SIZE;

                                                    /* connect RAB to FAB */
    if (error= sys$connect(file_context_block->file_rab)) != RMS$_NORMAL )
        error_value.problem= Filing_accessRightsIndeterminate;
        return(error_value);
    }

    return(-1);
}
```

After the file is successfully created by `create_file`, the bulk data stream will be read and written to the file and the file is closed. Files of type **tAsciiText** will have to be decoded to determine the correct record size before writing the record, as illustrated below:

```
{
    int count;

    ....

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

    file_context_block->rab.rab$b_rsz= count;       /* set count in rab */
                                                    /* write characters */
    sys$put(&file_context_block->fab);

    ....

}
```

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

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

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

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

                                              /* create directory */
    if (mkdir(file_context_block->pathname,0) == -1) {
        error_value.problem= Filing_accessRightsInsufficient;
        return(error_value);
    }

    return(-1);
}
```

## 6.3.6 Retrieve

The **Retrieve** procedure transfers a file from a service to the calling client. FilingSubset services are not required to allow the retrieval of directory files. The VMS implementation does not allow the retrieval of directory files, since the format of these files is VMS-specific. If the file is not a directory, then the file is opened and the content transferred via a bulk data stream to the client.

The **Retrieve** procedure assumes that the file was previously opened by an **Open** procedure and that the appropriate `file_context_block` fields have been initialized. The content of the file is then read and written to the bulk data stream.

Files of type **tAsciiText** are transferred as a **StreamofAsciiText**. The content of the file as read from the file must be encoded into this format for transmission to the client.

## 6.3.7 Delete

The **Delete** procedure is used by clients to delete files. If the specified file is a directory, a FilingSubset service is not required to support the deletion of that file and all its descendants. VMS does not provide a simple mechanism for supporting the deletion of non-empty directories; therefore, the service returns **AccessProblem accessRightsInsufficient** if the directory file cannot be deleted.

The VMS delete routine is used to delete the specified file. The delete routine will return an error if the file is a non-empty directory. Appropriate Filing errors are reported as type **AccessProblem**.

The following procedure illustrates the file deletion mechanism:

```
#include   errno

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

*/

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

    error_value.error= Filing_AccessError;        /* set to Filing AccessError */
                      /* attempt delete of directory, success only if empty */
    if (file_context_block->isdirectory ) {
        if ( delete(file_context_block->pathname) == -1 ) {
            error_value.problem= Filing_accessRightsInsufficient;
            return(error_value);
        }
    } else {                                  /* use delete for non-directories */
        if ( delete(file_context_block->pathname) == -1 ) {
            switch (errno) {
                case EACCES:                          /* user has no access */
                case EPERM:                           /* user has no access */
                    error_value.problem= Filing_accessRightsInsufficient;
                    return(error_value);

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

                default:                              /* all other errors */
                    error_value.problem= Filing_accessRightsIndeterminate;
                    return(error_value);
            }
        }
    }
    return(-1);
}
```

The following documents describe those protocols and data structures referenced within this guide.

[1] Xerox Corporation. *Authentication Protocol.* Xerox Network Systems Standard. Stamford, Connecticut; May 1986; XNSS 098605.
This reference defines the Authentication Protocol upon which the Filing and FilingSubset Protocols rely for authentication.

[2] Xerox Corporation. *Bulk Data Transfer.* Xerox Network Systems Standard. Stamford, Connecticut; April 1984; XNSS 038112 (XSIS 038112); Addendum 1a. Augments [6].
This reference defines the Bulk Data Transfer Protocol upon which the Filing and FilingSubset Protocols rely for bulk data transfer.

[3] Xerox Corporation. *Character Code Standard.* Xerox Network Systems Standard. Stamford, Connecticut; May 1986; XNSS 058605.
This reference defines the character set and the string format which provide the basis for Courier's string data type.

[4] Xerox Corporation. *Clearinghouse Protocol.* Xerox Network Systems Standard. Stamford, Connecticut; April 1984; XNSS 078404 (XSIS 078404).
This reference defines the protocol which FilingSubset implementations use to provide various services. It also defines the structure of user names which appear as various file attributes.

[5] Xerox Corporation. *Clearinghouse Entry Formats.* Xerox Network Systems Standard. Stamford, Connecticut; April 1984; XNSS 168404 (XSIS 168404).
This document defines Clearinghouse property types and the structure of their entries in terms of Courier data types.

[6] Xerox Corporation. *Courier: The Remote Procedure Call Protocol.* Xerox Network Systems Standard. Stamford, Connecticut; December 1981; XNSS 038112 (XSIS 038112).
This reference defines the Courier language, in terms of which the Filing and FilingSubset Protocols are defined.

[7] Xerox Corporation. *Filing Protocol.* Xerox Network Systems Standard. Stamford, Connecticut; May 1986; XNSS 108605.
This reference defines the Filing and the FilingSubset Protocols.

[8] Xerox Corporation. *Internet Transport Protocols.* Xerox Network Systems Standard. Stamford, Connecticut; December 1981; XNSS 028112 (XSIS 028112).
This reference defines the Sequenced Packet Protocol upon which Courier relies for data transport.

[9] Xerox Corporation. *Secondary Credentials Formats*. Xerox Network Systems Standard. Stamford, Connecticut; May 1986; XNSS 258605.
This reference documents specific type assignments and data formats for secondary credentials. Implementations of FilingSubset on hybrid hosts may require secondary authentication information.

[10] Xerox Corporation. *Time Protocol.* Xerox Network Systems Standard. Stamford, Connecticut; April 1984; XNSS 088404 (XSIS 088404).
This reference defines the Time Standard upon which the Filing and FilingSubset Protocols rely for the definition of the format for time and date quantities.