XEROX

Filing
Programmer's Manual

November 1984

PRELIMINARY

Xerox Corporation
Office Systems Division
3450 Hillview Avenue
Palo Alto, California 94304

Table of contents

3 File/session operations (CONTINUED)

Tables

1

Introduction

The *Filing Programmer's Manual* [12] is a reference for programmers who are familiar with the Mesa programming language. It defines and describes the interfaces and structure of *Filing*, a software package that allows programmer access to assorted local and remote network services in the Xerox 8000 Network Systems environment.

This manual is primarily intended for the designers and implementors of *client programs* of Filing. It provides sufficient information to allow programmers to understand the available Filing facilities, and to write procedure calls in the Mesa language to invoke the facilities. In particular, for each Filing facility, this manual lists the procedure names, parameters, results, data type of each argument, and possible signals (errors, etc.) which can be generated. This information is captured in the Mesa **DEFINITIONS** modules which are part of each release.

Differences between the descriptions provided and the released versions of Filing are noted in the documentation which accompanies each release. This document describes the version of Filing released in Services 8.0.

1.1 Filing and the services architecture

A *service* is an entity (software or hardware) that accepts and responds to submitted requests for some type of service. Ordinarily, it accepts these requests from the communications network; the requests are encoded according to protocols at various levels. A service need not be implemented in Mesa on a Mesa processor, as long as it can accept and respond to appropriately-encoded requests. Likewise, the *client* making the requests need not be a Mesa program as long as the requests conform to the appropriate protocols.

Because the interaction between clients and services at the network level is somewhat involved, software packages are supported that allow a Mesa client to interact with a service using ordinary Mesa procedure calls. Although such a software package has traditionally been called a *service client*, it will be referred to here as an *agent* to avoid confusion with the package's client. The agent takes care of the details of encoding requests and decoding replies in a piece of software known as the *stub*. In addition, the agent may perform some local processing that is related to, but separate from, the interaction with the remote service. This local processing may be extensive or minimal. For example, for filing applications, the local processing allows interaction with files

stored on the local disk, and transfer of information between the local disk and remote services.

Figure 1.1 shows a network with a client system and a server system attached. The client system contains several agents, one for each service that the client needs to talk to, and some common facilities associated with more than one agent. Conceptually, the collection of agents together with these common facilities make up the Filing release. (**Note**: Currently, only the agent for filing applications is part of the Filing release.)

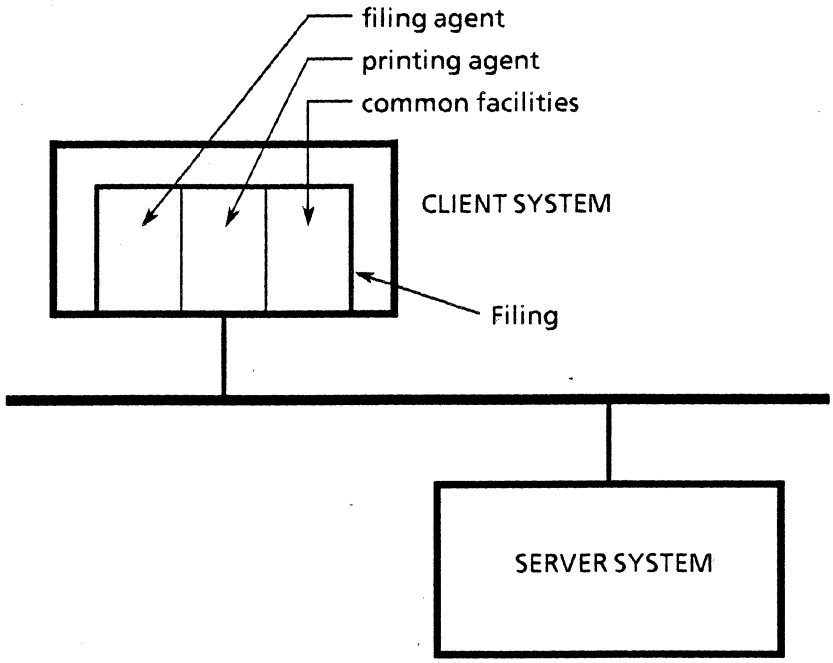

Figure 1.1 Client and server

1.2 Using Filing

In order to make use of the facilities described in this document, the programmer's configuration must include one or more of the software packages released in Filing—the specific packages required depend on the interfaces used. The documentation accompanying each release lists the interfaces exported by each package, the interfaces required by each package, any special considerations involved in using the package, and where the package is located.

1.3 Organization of the Filing Programmer's Manual

The rest of this manual describes the standard interfaces to Filing in terms of the Mesa data types and procedures used by client programs. In the implementation, these types and procedures are embodied in one or more Mesa interfaces (**DEFINITIONS** modules) made available to programmers of client software. The manual describes mechanisms that manipulate files on local disk volumes, and that communicate with file systems on remote system elements (such as file servers). Facilities are provided to initialize and maintain volumes, to create, delete, and manipulate files and directories, and to access the attributes and contents of files.

2

Overview

This section describes the facilities of NSFiling, the file system component of the 8000 series product line. It is a manual and a reference guide for use by programmers and clients who wish to interact with the product file system.

File system software structures data on the disks of the 8000 series product line. It supports multiple clients and mediates conflicting requests automatically. The file system provides facilities for the creation, deletion, and maintenance of filing volumes and files (local and remote). Each permanent file on a volume resides in a tree-structured hierarchy maintained by the file system.

2.1 Clients, the file system, and file service

The *file system* is a software entity that accepts and responds to locally-submitted filing requests. A *file service* is a remote file system that handles filing requests. A *client* is an entity that submits requests to the file system (or service). A client may or may not be operating on behalf of a human being. All interaction between a client and the file system is initiated by the client. The file system never spontaneously interacts with a client.

Requests submitted by the filing client are distinguished as local or remote. A request is *remote* if one of the operands (a file) is not local to the hardware on which the file system executes. Fulfilling a remote filing request requires interaction with a file service. In most cases, this interaction is accomplished with no additional effort by the client. The file system initiates the interaction with the remote file service on behalf of the client and reports results in a manner similar to a local request. Because of this similarity, the descriptions given in this manual apply equally to both local requests and remote requests.

2.2 Users, authentication, and sessions

A client always interacts with the file system on behalf of a *user*. The user may be a human being, or some other entity such as a file system or another service. In any event, the user has a *user name* that distinguishes him from other users.

Before making use of file system facilities, a client must *log on*. To do this, the client presents its *identity* to the file system. The client presents the user name to obtain this identity from the Authentication Service. The system responds by establishing a *session*

and returning a *session handle* which is used to identify the client (and the state of the interaction) in future requests. The *Authentication Protocol* [2] describes the format and the nature of the interaction with the Authentication Service .

The client may choose to log on directly to a particular file service, and use the session to interact solely with that service. This type of session is called a *directed* session. For convenience, the file system also allows the client to log on without specifying a particular file service with which he wishes to interact, and to defer specification of a service to the time when he submits a filing request for interaction with that service. This kind of session is called a *distributed* session, and may be used to interact with a variety of file services without requiring the client to log on to each one explicitly. The file system logs on, on behalf of the client, to each service with which interaction is desired, using the identity supplied at the initial Logon.

Once established, a session encapsulates the state of the client with respect to its interaction with the file system. For example, the session keeps track of files that are open, locks that are held, and the identity of the user on whose behalf the client is operating. The session handle is included in all subsequent requests in order to identify the client and its state. When interaction is complete, the client *logs off*. This terminates the session, freeing any allocated resources and invalidating the session handle. The client must log on again before any further interaction may occur.

For the convenience of the client, a session may be designated as the default session. The *default session* is a session distinguished by the client for use in operations requiring a session where none is specified in the request. In one mode of use, the client may log on, distinguish the session as the default session, and avoid the use of session handles until it is necessary to log off.

Sessions may vary greatly in duration. In some patterns of use a session is established to perform a single operation and then terminated. In others, a session may last a very long time even though it is largely inactive. The local file system never terminates a session; a remote file service reserves the right to terminate a session any time a procedure call is not in progress. This might occur if a session remains inactive for a long period, or if the system element supporting the file service has to be shut down.

Only clients who interact directly with a remote file service by using a directed session need to be concerned with the possible termination of the session. A client who logs on without directing the session to a particular file service is freed from this concern.

There may be several sessions simultaneously in existence for the same user regardless of whether they were established by the same client.

2.3 Volumes and services

In the file system sense, a *volume* is a logical group of disk pages which contains a wholly self-contained tree of files and the internal data structures needed to describe it. Such a volume is not necessarily related to a physical volume, such as a disk pack; a physical volume may contain several file system volumes, or a file system volume may span several physical volumes.

A single file system may manage a number of volumes on a given system element. To facilitate user identification of the volume containing a file of interest, the volume is given

a name. This name conforms to the specifications of Clearinghouse names and is often registered in the Clearinghouse so that network users may identify the volume. In order to make transparent the concept of multiple filing volumes located at a single system element, each volume is referred to by clients as an individual file *service*. A client identifies the service of interest by specifying its name and, if known, its system element address. Usually, specification of the name of the service is sufficient, since the file system will perform any interaction with the Clearinghouse which is necessary to obtain the system element address of the service.

A volume must be opened to become available as a service and closed before it may be dismounted. The **NSVolumeControl** interface contains operations to open, close and initialize local volumes (see §8.2). More than one volume may be open at a time, thus making multiple services available at the same system element. Operations that deal with several files may operate between services on the same system element, just as they may operate between services on different system elements.

One service (local or remote) may be distinguished as the *default service*; this service is assumed in operations in which no service is specified or implied. For example, if a parent directory is specified in **NSFile.Create**, the service on which the directory resides is implied; however, if no such directory is specified and no service is specified in the attribute list, the default service is assumed. The default service may be changed at any time via **NSFile.SetDefaultService** (see §3.1.6).

2.4 Files, content, and attributes

The file system stores and operates on *files*. A file is a body of data that has been grouped and provided to the file system for the purpose of short- or long-term storage. Every file is *temporary* or *permanent*. A permanent file resides in a directory and exists until it is explicitly deleted. A temporary file does not reside in a directory; it exists only until it is closed by all sessions that have opened it.

A file consists of two types of information, content and attributes. The *content* of a file is the data actually contained within the segments of the file. Usually, the content is the file's reason for existence. The content of a file is obtained or modified only by explicit action.

The content of a file is organized into a set of segments. A *segment* is an independently-addressed, growable region of a file. A file may potentially have many segments. Every file has at least one segment. This segment is distinguished by the file system as the *default segment*. The client manages segments and their content with the use of the **NSSegment** interface (see section 4).

The content of a file may also be accessed and modified using a positionable stream mechanism provided by the **NSFileStream** interface (see section 5). [In Services 8.0, only the content of local files may be accessed using **NSSegment** and **NSFileStream**. The content of remote files may be accessed only in bulk, using the bulk data transfer operations (see §3.8).]

Attributes are data items that identify the file, describe its content, or are in some other way associated with the file. Attributes vary widely in purpose, structure, and behavior. Some attributes have a particular meaning to the file system; specifying such an attribute results in a defined behavior in the file system. These attributes are said to be *interpreted*.

All other attributes are *uninterpreted*. Such attributes, if specified, are associated with the file, and are returned unchanged when requested.

Every attribute is identified by its *attribute type*. The interpretation and behavior of all interpreted attributes is defined by the file system. A client may also define attributes that are useful in a particular application. All such client-defined attributes are uninterpreted.

A number of procedures accept arbitrary attributes. However, not all attributes are allowed in all contexts. Information on where an attribute is allowed, the behavior when it is specified, and the default behavior when it is unspecified, is given in §6.8.

Attributes may be obtained or modified by explicit action. In addition, attributes are obtained when a directory is listed and interpreted attributes are implicitly modified by many procedures. Attributes are described in detail in Attributes (see section 6).

2.5 Directories

Every file is either a *directory* or a *non-directory*. A directory is a special type of file which can reference other files. A directory also has all of the characteristics of a non-directory in that it can have content and attributes. However, a directory cannot be temporary.

Within a filing volume, files exist in a hierarchical structure. Every permanent file resides at some level in this hierarchy. The files directly referenced by a directory are its *children*. The *descendants* of a directory include its children and the children of its descendants. The directory which directly references a file is that file's *parent*. The *ancestors* of a file include its parent and the parents of its ancestors. Each volume has a single *root* file. This file is unique in that it has no parent and is an ancestor of all other permanent files on the volume.

2.6 Handles and controls

To manipulate a file, a client must *open* that file. The file is then said to be *open within the session*, and remains open until the session ends or the file is explicitly *closed*. Opening a file marks it as "in use" (so that other clients cannot delete it, for example), and indicates that the file is to be used in some way in the near future. Closing a file clears this "in use" mark and indicates that access to the file is no longer needed.

When a file is created or when an existing file is opened, the file system returns a *handle*. The structure of a handle is private to the implementation. This handle is presented by the client in subsequent operations to identify the file to the file system, and remains valid either until the session ends or the file is closed using the handle. A handle is relative to a session and so cannot be used in conjunction with any session other than the one used to obtain it.

A client may wish to explicitly specify certain characteristics of his intended use of a file handle. These characteristics are called *controls*. For example, a client may obtain a share lock to stipulate that no other clients are allowed to modify the file while it is in use. Or, a client may specify that he be notified immediately if the file is in use in a way that conflicts with his own use, rather than waiting for access to the file. Controls persist only as long as a handle exists; they are lost when the handle is used to close the file.

If a file is opened several times, several handles result. Each handle is distinct and the file remains open within the session until the session ends or all handles have been presented in requests to close the file. Controls applied to a file are associated with a particular handle. If several handles for the same file exist, a change to the controls of one handle does not affect the others. Also, locks obtained on multiple handles to a file within the same session do not conflict with one another. However, the effective lock for a file in a given session is the most restrictive one obtained for that file within the session.

2.7 Creating, deleting, and accessing files

A number of procedures are provided for creating new files, deleting files that are no longer needed, and modifying files in various ways.

A file can be created without storing its content. A file can also be created and filled with data transferred to the file system by the client. Finally, a file can be created that is a copy of an existing file.

An existing file may be deleted. The attributes of a file may be accessed or modified, and the content of a file may be accessed or replaced. In addition, a file may be moved to another directory.

Since directories are also files, all of these procedures may be applied to directories as well as non-directories. When directories are copied, moved, or deleted, all descendants are copied, moved, or deleted as well.

2.8 Enumerating and locating files in directories

Several file system procedures enumerate files in a directory, and perform some action when files are encountered that satisfy client-specified criteria . The procedures differ in the action taken. If the client *lists* files in a directory, the attributes of each file that satisfies the criteria are furnished to the client. If the client attempts to *find* a file, the first file that satisfies the criteria is opened and a handle is returned.

The arguments that describe how enumeration is to proceed and the criteria to be satisfied are *scopes*. Scopes include the direction of enumeration (first-to-last or last-to-first), a condition on the attributes of the files, the maximum number of files that may satisfy the condition, the depth or number of levels of the file system hierarchy to be searched, and the desired order of enumeration.

2.9 Bulk data transfer

The file system supports stream-based access in bulk to the content of files. Usually, these facilities are used to transfer files between system elements, but they may be used in any situation in which sequential access to the content of a file is desired. Each of the stream-based filing operations relies on the bulk data transfer mechanism supported by **NSDataStream** (see *Common Facilities Programmer's Manual*, section 3). Random access to local files is provided by a separate positionable stream mechanism, **NSFileStream** (see section 5).

2.10 Serializing and deserializing files

A subtree of files, consisting of a file and all its descendants, can be a useful entity with which to work; file system operations are designed to make it easy to operate on such subtrees. However, there are times when it is useful to encapsulate all of the information within such a subtree so that the information can be stored or manipulated outside the file system.

A *serialized file* is a series of eight-bit bytes that encapsulates a file's content, its attributes and its descendants (including their content and attributes). The file system provides a procedure that *serializes* a file, producing such a series of bytes, and another procedure that *deserializes* the series of bytes, reconstructing the file's content, attributes, and descendants.

3

File/session operations

NSFile: DEFINITIONS = ...;

NSFile contains operations that manipulate files on local file services and that communicate with remote file services. Facilities are provided to create, delete and manipulate files and directories, and to access the attributes and content of files.

3.1 Sessions

A *session* encapsulates the state of interaction between a client and the file system on a particular system element. A session begins when a client *logs on* to make use of Filing, and is completed when the client *logs off*. A client may establish more than one session at any given time.

Two types of sessions are supported, *directed* and *distributed* sessions. A session is directed if the client specifies an explicit file service at Logon (thereby directing the session to that file service), otherwise it is distributed. Directed sessions are mainly used to initiate file system operations directly on a remote file service, perhaps even without the presence of a local file system. A directed session may also be used to access a local file service. Distributed sessions are used to initiate operations on any number of file services, both local and remote, with the same session handle.

When the client establishes a distributed session and initiates an operation on a particular file service, the file system automatically establishes a directed session to that file service (if one is not already active) on behalf of the client. This session is called an *auxiliary session*, and is transparent to the client. Creation of an auxiliary session is required because a session applies only to the file service for which it was created. To accomplish the directed Logon, the file system must possess sufficient information about the client's identity; if not enough information is available, the operation on the specified file service is not allowed.

3.1.1 Session handles

A session handle is used to identify and refer to the state of interaction between a client and the file system. A session handle is included as a parameter in almost all other calls to file system procedures. The value of a session handle remains the same throughout the life of the session. At any given instant in time, a session handle may be involved in at most one filing operation, so that consecutive requests to the file system using the same session

handle are always executed consecutively and not concurrently. The exceptions to this rule are the list and bulk data transfer operations, whose call-back procedures are permitted to call other Filing operations with the same session handle.

```
NSFile.Session: TYPE [2];
```

The constant **nullSession** is provided for defaulting of the session argument in those operations requiring a session. When specified by the client, this constant implies use of the default session (see below).

```
NSFile.nullSession: NSFile.Session = [LONG[NIL]];
```

3.1.2 Establishing a session

NSFile.Logon and **NSFile.LogonDirect** are used to initiate a session. The client supplies the file system with his identity which has been obtained from the Authentication Service. The file system creates a session and returns a handle identifying the new session.

Note: During **LogonDirect**, the file system validates the supplied identity with the authentication service. This validation is not done immediately when using **Logon**, but is done the first time the file system establishes an auxiliary session directed to a particular file service.

LogonDirect is used to initiate a session with a particular file service.

```
Identity: TYPE = Auth.IdentityHandle;

NSFile.LogonDirect: PROCEDURE [identity: Identity, service: Service]
    RETURNS [Session];
```

Arguments: **identity** provides authentication information about the client who wishes to establish a session; **service** identifies the file service with which a session is to be established.

Results: The session handle for the new session is returned.

Errors: **NSFile.Error** is raised with the following error types: **authentication**, **clearingHouse** and **service**; **Courier.Error** may also be raised.

The service specified during **LogonDirect** identifies the particular file service to which filing requests are to be directed. If the system element address of the service is not supplied by the client, the file system uses the supplied name of the service to obtain the system element address from the clearinghouse.

```
NSFile.Service: TYPE = LONG POINTER TO ServiceRecord;

NSFile.ServiceRecord: TYPE = RECORD [
    name: NSName.NameRecord,
    systemElement: SystemElement ← nullSystemElement];

NSFile.SystemElement: TYPE = System.NetworkAddress;
```

```
NSFile.nullSystemElement: NSFile.SystemElement = System.nullNetworkAddress;

NSFile.localSystemElement: READONLY NSFile.SystemElement;
```

NSFile.nullSystemElement is used in a **ServiceRecord** when the client wants the file system to determine the network address of the service. **NSFile.localSystemElement** is the network address for the system element on which the local file system executes.

Logon is used to initiate a session which may be used to access files on a variety of file services.

```
NSFile.Logon: PROCEDURE [identity: Identity] RETURNS [Session];
```

Arguments: **identity** provides authentication information about the client who wishes to establish a session.

Results: The session handle for the new session is returned.

Errors: **NSFile.Error** is raised with the following error types: **authentication** and **service**; **Courier.Error** may also be raised.

3.1.3 Logoff

NSFile.Logoff is used to end a session. The file system verifies that the request is valid, closes all file handles still open in the session, destroys the session, releases any allocated resources, and invalidates the session handle.

```
NSFile.Logoff: PROCEDURE [session: Session ← nullSession];
```

Arguments: **session** refers to the session that is to be ended; if no session is specified, the default session is assumed.

Results: The session ends; **session** is no longer a valid session handle.

Errors: **NSFile.Error** is raised with the following error types: **authentication**, **service**, and **session**; **Courier.Error** may also be raised.

3.1.4 Probe

NSFile.Probe registers interest in a session directed at a remote file service. A client who wishes a session to remain in existence through some period of inactivity may call **Probe** to prevent Filing from terminating the session due to inactivity. This operation has no effect on distributed sessions or sessions directed at a local file service.

```
NSFile.Probe: PROCEDURE [session: NSFile.Session] RETURNS [probeWithin: CARDINAL];
```

Arguments: **session** refers to the session that is to be probed.

Results: **probeWithin** is expressed in seconds. Under normal conditions, the **session** will not be terminated unless it remains inactive more than

this number of seconds. If **session** is a distributed session, the value LAST[CARDINAL] is returned to indicate the session will not time out.

Errors: NSFile.Error is raised with the following error types: **authentication** and **session**; Courier.Error may also be raised.

3.1.5 The default session

For the convenience of local clients, the file system maintains a *default session.* The default session is any session distinguished by the client via NSFile.SetDefaultSession. In calls to file system operations requiring a session handle, if no session is specified, the default session is used; if no session has been distinguished as the default session, NSFile.Error[[session[sessionInvalid]]] is reported.

```
NSFile.SetDefaultSession: PROCEDURE [session: NSFile.Session];
```

Arguments: **session** refers to the session that is to be distinguished as the default session.

Results: The session becomes the default session.

Errors: None.

The operation **GetDefaultSession** returns the session handle for the current default session; if no session has been distinguished as the default, NSFile.nullSession is returned.

```
NSFile.GetDefaultSession: PROCEDURE RETURNS [NSFile.Session];
```

Arguments: None.

Results: The returned **session** identifies the default session.

Errors: None.

3.1.6 The default service

Since **Logon** does not establish a session with a particular service, the service to be accessed must be specified in each subsequent operation with that session. The client may also distinguish one service as the default service, to be used if a service is not explicitly specified in a call to a Filing operation. The default service is global to a system element and is not session-specific.

The current value of the default service is given by NSFile.defaultService.

```
NSFile.defaultService: NSFile.Service;
```

NSFile.SetDefaultService sets the default service. It overwrites any previous value of the default service.

```
NSFile.SetDefaultService: PROCEDURE [service: Service];
```

Arguments: **service** specifies the default service to be used in the absence of an explicit service specification.

Results: **NSFile.defaultService** is set to **service**. Subsequent calls to filing operations which lack an explicit service specification will be directed to this service (for any distributed session).

Errors: None.

The service passed to **SetDefaultService** should always be completely resolved, i.e., it should have both the service name and system element address filled in. If the system element address is left **nullSystemElement**, then the file system will assume the service is local and will substitute **localSystemElement**.

3.2 Naming, opening, and closing files

In order to operate on a particular file, the client must specify the file's location and *open* it. This establishes the client's intent to make use of the file in some way and guarantees the file's existence until the client *closes* the file, relinquishing access to it. A variety of means are supported to specify the location and identity of a file.

3.2.1 Naming

A file is most commonly located or *named* by specifying a *reference* to its location. A **Reference** to a file includes the name and network location of the file service containing it (**service**), and a designation of the file within the specified (or implied) service (**fileID**).

```
NSFile.Reference: TYPE = RECORD[
    fileID: ID, service: Service];

NSFile.Service: TYPE = LONG POINTER TO ServiceRecord;
NSFile.ServiceRecord: TYPE = RECORD [
    name: NSName.NameRecord,
    systemElement: SystemElement ← nullSystemElement];

NSFile.nullService: Service = LONG [NIL];

NSFile.nullReference: Reference = [nullID, nullService];
```

Supplying the **nullService** within a reference implies the **defaultService**. A **Reference** may be used to name a file only when using a distributed session since, for a directed session, the service is implicit. If **fileID** is null, the root file of the specified (or implied) service is implied. Note that file identifiers are relative to a particular service and are *not* guaranteed to be unique across services.

The file system will look up a service name in the Clearinghouse if its system element address is not specified. To minimize repetitive lookups, it maintains a cache of service records in which the system element is resolved. In order to allow clients the ability to compare services (and thus references) for equality, the file system also caches pointers to the cached service records. Thus a client who obtains all of its service pointers from the file system's cache need not compare the individual fields of two service records to determine if they are equal, but can compare the service pointers directly for equality. To receive a

pointer to a service from the file system's cache, the client uses the operation **NSFile.MakeReference**. The **service** field of the returned reference refers to the appropriate cached value, and can be compared for equality against any other service pointer which has been obtained from the file system. The corresponding service record is the property of the file system and the client should make no attempt to free it.

```
NSFile.MakeReference: PROCEDURE [fileID: ID, service: Service ← nullService]
RETURNS [reference: Reference];
```

Arguments: **service** is the file service on which the file resides; **fileID** is the designation of the file on the service.

Results: A reference for the file is returned. The **service** field of the reference points into the file system's service cache, and storage for the corresponding service record should *not* be freed by the client.

Errors: None.

The second means of naming a file involves the **name** and **version** attributes of the file. Every permanent file on a volume is uniquely identified within its parent directory by its **name/version** attribute pair. In those contexts where a directory-relative specification of a file is allowed, the **name** and **version** attribute combination may be used.

Finally, a *pathname* may be used to identify a file in those file system operations which accept a pathname argument. The pathname gives a hierarchical list of directories in the path to the file. The pathname may be relative to a specified starting directory or to the root file of the service. The syntax of pathnames and the separator characters are defined in Section 7.

3.2.2 Opening

In order to examine or manipulate a file, a client must first open the file, thereby obtaining a handle for it. While a handle exists, no other handle can be used to delete or move that file.

```
NSFile.Open: PROCEDURE [
    attributes: AttributeList, directory: Handle ← nullHandle,
    controls: Controls ← [], session: Session ← nullSession]
    RETURNS [file: Handle];
```

Arguments: **attributes** identifies the file as described below; **directory** specifies a starting directory in which to look for the file (it may be the null handle); **controls** specifies the controls to be applied to the new handle; **session** is the client's session handle.

Results: **file** is the file handle for the file being opened. It remains valid until **session** ends or the file is closed using the handle.

Errors: **NSFile.Error** may be raised with the following types: **access**, **attributeType**, **attributeValue**, **authentication**, **clearingHouse**, **handle**, **service**, **session**, and **undefined**; **Courier.Error** may also be raised.

A file is opened by specifying a list of attributes that identify the file, plus an optional working directory. Only certain interpreted attributes may be specified, and of these, at most one of **name**, **pathname**, or **fileID** may be specified. If extended attributes are specified, a remote file system may treat them unpredictably; therefore, they should be avoided.

When opening a file, the client may specify controls for the resulting handle (see §3.3).

The interpreted attributes that may be specified on **Open** are:

service This attribute specifies the file service on which the file resides. This attribute may only be specified when using a distributed session, since for a directed session the service is implicit. If omitted or a value of **nullService** is specified and **directory** is null, the file is assumed to reside on the **defaultService**. If this attribute is specified and **directory** is non-null, then the directory must be on the specified service.

fileID This attribute is a fixed-size identifier for the file which is unique on the specified (or implied) service. A **nullID** may not be specified.

parentID This attribute specifies the **fileID** for the parent of the desired file. This attribute is optional; if specified and **directory** is non-null, then the specified **parentID** must match the directory's **fileID**. If a null **parentID** is specified, then the specified file must have no parent (it is either a temporary file or a service root file).

name The string name of the file is specified using the **name** attribute; a lookup is performed either in the specified directory or the specified **parentID**; **parentID** must name a directory on the specified or implied service. If **version** is not specified, the highest-version file with this name is opened.

version The **version** attribute indicates the version number of the file. This may be specified only if either **name** or **pathname** is also specified. If omitted, the file with the highest version is opened. If specified with a **pathname**, a version number in the last component of the pathname will supersede this value.

pathname The **pathname** attribute is a string giving a hierarchical list of directories in the path to the file. If specified and **directory** (or **parentID**) is null, then the pathname is assumed to be relative to the root file of the specified (or implied) service. If specified and **directory** (or **parentID**) is non-null, the pathname is assumed to be relative to **directory**, and the first name in the path must be an immediate descendant of **directory**.

If **name**, **pathname**, and **fileID** are all omitted, the root file of the specified (or implied) service is opened.

3.2.3 Simpler forms of Open

Many clients do not need the full generality of **Open**; for such clients, the simpler operations **OpenByReference**, **OpenByName**, and **OpenChild** are provided. Each reports the same classes of errors as **Open**.

```
NSFile.OpenByReference: PROCEDURE [
    reference: Reference, controls: Controls ← [],
    session: Session ← nullSession]
    RETURNS [file: Handle];
```

A **Reference** (see §3.2.1) is a convenient data structure containing all the information needed to uniquely identify a file. A file's reference can be obtained by calling **GetReference** (see §6.5). If **service** is null, the file is expected to reside on the **defaultService**. If **fileID** is null, the root file of the specified (or implied) service is opened. This operation may only be used with a distributed session, since with a directed session the service is implicit.

```
NSFile.OpenByName: PROCEDURE [
    directory: Handle, path: String, controls: Controls ← [],
    session: Session ← nullSession]
    RETURNS [Handle];
```

OpenByName opens a descendant of the specified directory with the given pathname. If the directory is null, the pathname is assumed relative to the root file of the service implied by the session. The pathname may contain the name of the service on which the file resides, in which case the rest of the pathname is assumed to be relative to the root file of that service, and the specified directory *must* be null. For details on the syntax of pathnames and qualified pathnames (i.e., pathnames which contain a service name)(see Section 7).

```
NSFile.OpenChild: PROCEDURE [
    directory: Handle, id: ID, controls: Controls ← [], session: Session ← nullSession]
    RETURNS [Handle];
```

OpenChild opens the child of the specified directory with the given **fileID**.

3.2.4 Close

```
NSFile.Close: PROCEDURE [file: Handle, session: Session ← nullSession];
```

Arguments: **file** is the handle to be closed; **session** is the client's session handle.

Results: **file** is no longer a valid handle.

Errors: **NSFile.Error** can be raised with the following types: **authentication**, **handle**, **session**, and **undefined**; **Courier.Error** may also be raised.

A client should close a file when the client no longer needs to operate on it. Closing a file releases any lock associated with that handle, and may allow the file to be moved or deleted. If the file is temporary, it is deleted when the last handle to it is closed.

All file handles in a session are automatically closed when the session is logged off.

3.3 Handles and controls

A file *handle* is a means of identifying an open file to file system operations. It is returned by operations which open or create files and is normally used to complete access to a file during **Close**.

```
NSFile.Handle: TYPE = [2];

NSFile.nullHandle: Handle = [LONG[NIL]];
```

The special constant **nullHandle** is a reserved value of **Handle** used to imply various options in file system operations in which it may be specified. Consult the individual operations for further details.

```
NSFile.Controls: TYPE = RECORD [
    lock: Lock ← none,
    timeout: Timeout ← defaultTimeout,
    access: NSFile.Access ← fullAccess];

NSFile.ControlType: TYPE = MACHINE DEPENDENT {lock(0), timeout(1), access(2)};
```

Every file handle is subject to parameters known as **controls** which specify the nature of file access using that handle. Three types of controls are currently defined: locks, timeouts, and access. Controls are always specified when a file is opened, and can subsequently be changed with the **ChangeControls** operation.

3.3.1 Locks

A lock on a file is a restriction on the use of the file by other sessions. A client might specify a lock to prevent certain types of access to the file while operating on it.

```
NSFile.Lock: TYPE = MACHINE DEPENDENT {none(0), share(1), exclusive(2)};
```

A **none** lock prevents other sessions from deleting or moving the file, as well as preventing the same session from deleting or moving the file except with this handle.

A **share** lock prevents other sessions from acquiring an **exclusive** lock, as well as providing the protections of the **none** lock above.

An **exclusive** lock prevents other sessions from acquiring a **share** or **exclusive** lock, as well as providing the protections of the **none** lock above.

Share and exclusive locks only restrict file access through other sessions, not through other handles in the same session. A client may simultaneously hold several handles to the same file, some of which carry share locks while others carry none or exclusive locks.

Access to a file from another session is limited by the most restrictive lock in place on the file.

An open file may never be deleted if there are other open handles to the file. Thus, no matter what kind of lock is in place on an open file, the client is assured that the file will not "disappear" unexpectedly.

A **none** lock should be used when the client simply wants to guarantee the file's continued existence. It provides no special locking of the file. A **none** lock may be acquired at any time regardless of what other locks are held on the file, and only ensures that the file will not be moved or deleted.

A **share** lock should be used when the client is reading, but not modifying a file. A **share** lock prevents other sessions from modifying the file in any way, (including changing its attributes, adding children, etc.) and prevents them from acquiring an **exclusive** lock.

An **exclusive** lock should be used when the client wishes to modify a file. An **exclusive** lock prevents other sessions from reading or modifying a file, and from acquiring either an **exclusive** lock or a **share** lock.

While executing an **NSFile** procedure, the file system internally acquires the locks that it needs to ensure correct and consistent execution of that procedure, and releases them before the procedure returns. The client never needs to explicitly acquire locks unless it wants to prevent modification or examination of a file by clients using other sessions *between* calls to **NSFile** procedures.

In operations defined in **NSSegment** (see Section 4) and **NSFileStream** (see Section 5) which access the content of a file, the file system does not acquire internal locks to preserve the integrity of the file's content. To ensure adequate protection of files when using the operations in these interfaces, the client should acquire the appropriate lock when opening the file.

3.3.2 Timeouts

When a process requests a lock that is unavailable (either explicitly with **Open** and **ChangeControls**, or implicitly within an **NSFile**, **NSFileStream**, or **NSSegment** procedure), the process is delayed until the lock becomes available or the file handle's timeout expires, whichever comes first. If the lock becomes available, it is acquired and execution continues. If the timeout expires, the error **NSFile.Error[[access[fileInUse]]]** is reported.

```
NSFile.Timeout: TYPE = Process.Seconds;
```

A timeout value is expressed in seconds. The timeout associated with a handle applies to any request to acquire a lock on that handle. If a timeout of zero is specified, the file system does not wait. In this case if the requested lock is unavailable, an error is immediately reported. Conversely, a very large timeout may cause the file system to wait a very long time for a lock to become available. Such timeouts should be used with care.

```
NSFile.defaultTimeout: Timeout = LAST[Timeout];
```

If **defaultTimeout** is specified, an implementation-dependent default (which may be set using **NSFileControl.SetDefaultTimeout**, §8.1.3) is applied. When the current timeout value

is requested from the file service, it is this actual timeout value, rather than the constant **defaultTimeout**, which is returned.

3.3.3 Access

```
NSFile.Access: TYPE = PACKED ARRAY AccessType OF BooleanFalseDefault;

NSFile.AccessType: TYPE = MACHINE DEPENDENT {
    read(0), write(1), owner(2), add(3), remove(4)};
```

Access determines what operations are allowed for a particular file handle. An **Access** is a set of bits, each of which enables a particular form of access to a file:

read The client may read the file's contents and attributes. If the file is a directory, the client may also list its children and search for files in the directory.

write The client may change the file's contents and data attributes, and may delete the file. If the file is a directory, the client may also change environment attributes and access lists of the directory's children.

owner The client may change the file's access list.

add If the file is a directory, the client may add children to it (using **Create, Copy, Move, Store**, or **Deserialize**).

remove If the file is a directory, the client may remove children from it (using **Move** or **Delete**).

The access actually available to a client is the logical **AND** of the access last specified in **Open** or **ChangeControls**, and the access allowed by the file's access control list (see §6.3.5). This **AND**ed value is the one returned when an access value is requested via **GetControls**.

Note: The returned value may be more restrictive than the value last specified, but will never be less restrictive.

3.3.4 Retrieving and changing controls

```
NSFile.ControlSelections: TYPE = PACKED ARRAY ControlType OF BooleanFalseDefault;

NSFile.allControlSelections: ControlSelections = ALL[TRUE];

NSFile.noControlSelections: ControlSelections = []; -- ALL[FALSE]
```

A **ControlSelections** is used to specify a set of controls of interest during **GetControls** and **ChangeControls**. The currently effective controls on a file handle are obtained by calling **GetControls**.

```
NSFile.GetControls: PROCEDURE [
    file: Handle, controlSelections: ControlSelections ← allControlSelections,
    session: Session ← nullSession]
    RETURNS [controls: Controls];
```

Arguments: **file** is the file handle of interest; **controlSelections** identifies the types of control items that are desired; **session** is the client's session handle.

Results: A **Controls** record containing the desired control items is returned. Fields which were not requested have undefined values and should not be accessed.

Errors: **NSFile.Error** is raised with the types: **access**, **authentication**, **handle**, **session**, and **undefined**; **Courier.Error** may also be raised.

Only the values of the specific controls requested in **controlSelections** are returned. Since different controls are obtained with varying degrees of difficulty, the client should request only those controls that it needs. In particular, requesting access may cause a time-consuming access list evaluation, potentially requiring communication with a Clearinghouse.

The controls on a file handle may be changed by calling **ChangeControls**.

```
NSFile.ChangeControls: PROCEDURE [
    file: Handle, controlSelections: ControlSelections, controls: Controls,
    session: Session ← nullSession];
```

Arguments: **file** is the file handle whose controls are to be modified; **controlSelections** identifies the types of control items to be changed; **controls** is a record containing the control items to be changed; **session** is the client's session handle.

Results: The controls specified by **controlSelections** are changed to the values supplied.

Errors: **NSFile.Error** is raised with the following types: **access**, **authentication**, **controlValue**, **handle**, **session**, and **undefined**; **Courier.Error** may also be raised.

Only the controls specified in **controlSelections** are actually changed; other fields in **controls** are ignored.

3.4 Creating and deleting files

The operations described in this section allow clients to create and delete files. Permanent or temporary files may be created or deleted; a temporary file remains only until all handles to it within the session creating it are closed or until the session ends.

3.4.1 Create

A file may be created by calling **NSFile.Create**. This procedure creates a new file with unspecified contents. The new file may either be temporary or contained in a directory.

Create is particularly useful for creating directories and for creating local files whose contents will subsequently be initialized by **NSFileStream** (see Section 5) or **NSSegment** (see Section 4) operations. **NSFile.Store** is usually a more appropriate operation for creating remote non-directory files (see §3.8).

```
NSFile.Create: PROCEDURE [
    directory: Handle, attributes: AttributeList ← nullAttributeList,
    controls: Controls ← [], session: Session ← nullSession]
    RETURNS [file: Handle];
```

Arguments: **directory** is a file handle for the directory into which the created file is placed (the null handle may be specified, in which case a temporary file is created on the service specified or implied by **attributes** and **session**); **attributes** specifies the characteristics of the new file; **controls** specifies the controls to be applied to the returned handle; **session** is the client's session handle.

Results: **file** is a file handle for the newly-created file. It remains valid until **session** ends or the file is closed using the handle.

Access: **Create** requires add access to **directory**.

Errors: **NSFile.Error** is raised with the following types: **access, attributeType, attributeValue, authentication, clearingHouse, handle, insertion, service, session, space**, and **undefined**; **Courier.Error** may also be raised.

The file system acquires an exclusive lock on **directory** while creating the file.

3.4.2 Deleting files

NSFile.Delete deletes an existing file. The file is closed and deleted, freeing the resources allocated to the file and removing any association with a directory. If the file is a directory, all descendants are also deleted.

```
NSFile.Delete: PROCEDURE [file: Handle, session: Session ← nullSession];
```

Arguments: **file** is a file handle for the file to be deleted; **session** is the client's session handle.

Results: **file** and any descendants are deleted, freeing their resources. Errors may cause partial deletion to occur (i.e., some, but not all, descendants are deleted).

Access: Remove access to **file**'s parent; write access to **file** (and each descendant).

Errors: **NSFile.Error** is raised with the following types: **access, authentication, handle, session**, and **undefined**.

The file system must be able to acquire an exclusive lock on the parent of **file**. There must be no other handles in order to delete the file in **session** or any other session. Furthermore, none of the file's descendants may be open in **session** or any other session. If the latter condition is violated, **Delete** may fail part way through with **NSFile.Error [[access [fileInUse]]]** or **NSFile.Error[[access[fileOpen]]]**.

Delete requires remove access to the parent of **file**, and write access to **file** and all its descendants. If write access cannot be obtained to a descendant, the delete may fail part way through with **NSFile.Error [[access[accessRightsIndeterminate]]]** or **NSFile.Error[[access [accessRightsInsufficient]]]**.

If **Delete** returns normally, the file handle is invalidated. If it raises any error, the file remains open and the file handle is still valid.

3.5 Finding and listing files within directories

The client may examine the files in a directory using **NSFile.List** or **NSFile.Find**. Scope information describes the files of interest and how they are to be examined. A *qualified* file satisfies the constraints of client scope information. Depending on the specific procedure, either the attributes of qualified files are returned to the client, or the first qualified file is opened.

3.5.1 Scopes

Scope items determine what files in a directory are of interest to the client and how they are to be examined. The client may specify the order of consideration, the direction of listing or searching, which files are to be examined, the depth in the file system hierarchy to be spanned in the search, and (in the case of listing) the maximum number of files. Scope-type parameters are effective only in the procedure to which they are arguments.

```
NSFile.ScopeType: TYPE = MACHINE DEPENDENT{
    count(0), direction(1), filter(2), ordering(3)};

NSFile.Scope: TYPE = RECORD [
    count: CARDINAL ← LAST[CARDINAL],
    direction: Direction ← forward,
    filter: Filter ← nullFilter,
    ordering: Ordering ← nullOrdering,
    depth: CARDINAL ← 1];
```

scope.count specifies the maximum number of files the client wishes to see. The file system attempts to locate **scope.count** number of files satisfying all scope constraints and terminates the search when that number has been found or no further files remain for consideration. A defaulted value of **scope.count** implies that the client wishes to consider all qualified files.

```
NSFile.Direction: TYPE = MACHINE DEPENDENT {forward(0), backward(1)};
```

scope.direction specifies whether enumeration of the directory is to proceed from beginning to end or from end to beginning. The actual order of files within a directory is determined by the **ordering** attribute (see §6.3.6).

If the direction is **forward**, enumeration begins with the first file in the ordering. If the direction is **backward**, enumeration begins with the last file. Direction affects both listing (files are listed in the specified direction) and searching (the first encountered file that matches the specified criteria is returned).

scope.filter specifies a condition that distinguishes files of interest in the directory under consideration. The condition is one of: the constants **TRUE** or **FALSE**; a relation between an attribute and a constant (a *filter condition*); or a logical combination of conditions.

```
NSFile.FilterType: TYPE = MACHINE DEPENDENT{
    -- relations --
    less(0), lessOrEqual(1), equal(2), notEqual(3), greaterOrEqual(4), greater(5),
    -- logical --
    and(6), or(7), not(8),
    -- constants --
    none(9), all(10),
    -- patterns --
    matches(11)};

NSFile.Filter: TYPE = MACHINE DEPENDENT RECORD [
    var: SELECT type: FilterType FROM
        less, lessOrEqual, equal, notEqual, greaterOrEqual, greater = > [
            attribute: Attribute, interpretation: Interpretation ← none],
            -- interpretation ignored if attribute not 'extended'
        matches = > [attribute: Attribute],
        and, or = > [list: LONG DESCRIPTOR FOR ARRAY OF Filter],
        not = > [filter: LONG POINTER TO Filter],
        none, all = > [],
        ENDCASE];

NSFile.Interpretation: TYPE = MACHINE DEPENDENT {
    none(0), boolean(1), cardinal(2), longCardinal(3), integer(4),
    longInteger(5), string(6), time(7)};
```

A filter whose value is **matches []** is satisfied if the corresponding string attribute of a file satisfies the string pattern of the filter. Two wildcard characters are defined: * (asterisk) and # (pound sign). The * character matches zero or more characters within a string attribute; # matches any single character. Wildcard characters meant to be interpreted literally within the pattern must be escaped by quoting them with ' (apostrophe).

A filter whose value is **and** [*filter*$_1$, *filter*$_2$, ..., *filter*$_n$] is satisfied only if all of *filter*$_1$, *filter*$_2$, ..., *filter*$_n$ are satisfied.

A filter whose value is **or** [*filter*$_1$, *filter*$_2$, ..., *filter*$_n$] is satisfied when at least one of *filter*$_1$, *filter*$_2$, ..., *filter*$_n$ is satisfied.

A filter whose value is **not** *filter* is satisfied when *filter* is not satisfied.

A filter whose value is **none** [] is never satisfied, while a filter whose value is **all** [] is always satisfied.

All other filters are relations between a constant attribute value and the corresponding attribute of a file. Each of these filters is satisfied if the file's attribute satisfies the specified relation, when interpreted in an appropriate way and compared to the constant value given in the filter.

Example: Consider only those files having a name of "Monthly Status Report," not placed in the directory by "Upper Management" of the XYZ Company.

```
[and [
    [equal [ [name ["Monthly Status Report"] ] ] ],
    [not [
        [equal [ [filedBy ["Upper Management:Headquarters: XYZ Company"] ] ] ]
    ]]
]]
```

The **interpretation** component of a filter provides the file system with the information it needs to properly compare the attribute in the file with the specified constant value. The file system needs this information only for extended attributes. For attributes that the file system interprets, the standard interpretation is used; in this case any specified interpretation is ignored. Attribute values with a given interpretation are compared as follows:

none	Values are compared word by word, starting with the first. Corresponding sixteen-bit words are compared as though they were of type **CARDINAL**, starting with the first, until an unequal pair is found. The relationship of this unequal pair is considered to be the relationship of the two attributes. If the attributes are equal up to the length of the shorter, the longer attribute is considered to be greater.
boolean	A value of **TRUE** is greater than **FALSE**.
cardinal	Values are compared as unsigned sixteen-bit numbers.
longCardinal	Values are compared as unsigned thirty-two-bit numbers.
integer	Values are compared as signed sixteen-bit numbers.
longInteger	Values are compared as signed thirty-two-bit numbers.

string Values are compared using **NSString.CompareStrings**. Case is ignored.

time Values are compared as points in a linear time span where a later time is considered to be greater than an earlier time. Because of the time encoding, this comparison is not the same as for **longCardinal**.

Note: To find which of two times comes first, apply **System.SecondsSinceEpoch** to each; this gives the number of seconds that each is after the system epoch. Finally, compare the results to determine which is the later time. Refer to *Pilot Programmer's Manual* [26] for further details.

If the value of an attribute is not a valid representation of a value of the stated interpretation, that attribute is considered to be less than any attributes that are valid representations.

Note: In Services 8.0, **scope.interpretation** is not implemented. It is always ignored.

If no filter is specified, **nullFilter** is assumed.

```
NSFile.nullFilter: Filter = [all[]];
```

Not all attributes are supported within filters. Attributes within filters are restricted to: **backedUpOn**, **createdBy**, **createdOn**, **filedBy**, **filedOn**, **modifiedBy**, **modifiedOn**, **name**, **position**, **readBy**, **readOn**, **type**, and **version**. Use of other attribute types causes **NSFile.Error[[scopeValue[unimplemented, filter]]]** to be raised.

Scope.ordering specifies the desired enumeration or search ordering to be used. If the value of **Scope.ordering** is not equal to **NSFile.defaultOrdering** or the current value of the directory's **ordering** attribute, **NSFile.Error [[scopeValue[unimplemented, ordering]]]** is reported. If the current value of the directory's ordering attribute is desired as the order of enumeration, **nullOrdering** should be used.

```
NSFile.nullOrdering: extended Ordering = [extended[key: 0]];
```

Note: The value of **nullOrdering** is not a valid value for the **ordering** attribute of a directory; therefore, it may not be used in any other context.

scope.depth specifies the number of levels in the file system hierarchy which the enumeration should span. If the depth is 1 (the default), then only the immediate descendants of the directory are enumerated. If the depth is 2, then if an immediate descendant of the directory is itself a directory, its immediate descendants will be enumerated as well. And so on for increasing depth values. For convenience, a constant, **NSFile.allDescendants**, is defined which indicates that all the levels of the file system hierarchy should be traversed in the enumeration.

```
NSFile.allDescendants: CARDINAL = LAST[CARDINAL];
```

In Services 8.0, the depth and the filter portions of the scope are independent of each other. In other words, for a **depth** > 1, the descendants of a directory will be enumerated only if the directory itself satisfies the specified filter requirement. Therefore the depth field is most useful in enumerating whole or partial subtrees of files *without* specifying a filter.

3.5.2 Locating files

Find is called to locate and open a particular file in a directory. The file system enumerates the directory's children in the specified direction and order and opens the first file that meets the specified filter criteria; it reports **NSFile.Error[[access[fileNotFound]]]** if no such file can be found.

```
NSFile.Find: PROCEDURE [
    directory: Handle, scope: Scope ← [], controls: Controls ← [],
    session: Session ← nullSession]
    RETURNS [file: Handle];
```

Arguments: The directory to be searched is given by **directory**; **scope** specifies characteristics of the enumeration and the search criteria; **controls** specifies the controls to be applied to the new handle; **session** is the client's session handle.

Results: **file** refers to the file that was found and opened.

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

Errors: **NSFile.Error** is raised with the following types: **access**, **authentication**, **controlType**, **controlValue**, **handle**, **scopeTypeError**, **scopeValue**, **session**, and **undefined**; **Courier.Error** may also be raised.

3.5.3 Listing files

List enumerates the files in a directory, returning selected attributes of each. The file system enumerates the directory in the specified direction and order and invokes the client-supplied procedure for each file that meets the specified criteria.

```
NSFile.List: PROCEDURE [
    directory: Handle, proc: AttributesProc, selections: Selections,
    scope: Scope ← [], clientData: LONG POINTER ← NIL, session: Session ← nullSession];
```

Arguments: The directory to be searched is given by **directory**; **proc** is a procedure specified by the client (see below); **selections** specifies the set of attributes to be returned for each file; **scope** specifies characteristics of the enumeration and the search criteria; **clientData** is a pointer to client data which will be passed to **proc** along with the attributes specified by **selections**; **session** is the client's session handle.

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

Errors: **NSFile.Error** is raised with the following types: **access**, **attributeType**, **authentication**, **connection**, **handle**, **scopeType**, **scopeValue**, **session**; **Courier.Error** may also be raised.

The client is free to modify the contents of **directory** during **List**; however, the effect of such modifications on the behavior of the remaining operation is undefined. For example, if the entire contents of a directory are deleted after the first file is listed, continuation of the list may produce none, some, or all of the deleted entries.

```
NSFile.AttributesProc: TYPE = PROCEDURE [
    attributes: Attributes, clientData: LONG POINTER]
    RETURNS [continue: BOOLEAN ← TRUE];
```

An **AttributesProc** is a client procedure supplied to **List** for the purpose of returning attribute values of files within the given scope. Since different attributes are obtained with varying degrees of difficulty, the client should request only the attributes that are needed. For each file within the scope, **proc** is called with **attributes**, a pointer to an attributes record. This record and its attached structures belong to the file system, and may not be deallocated by the client; any data which is to be preserved must be copied by the client before returning. The **clientData** pointer supplied to **List** by the client is also passed to **proc**. Enumeration may be interrupted by returning **FALSE** from **proc**. The client may raise signals during execution of **proc**; if the signal is not one of the standard filing errors, the client may not log off until the signal is unwound.

3.6 Copying files

Copy creates a file which is a copy of an existing one. If the existing file has descendants, they are copied as well. The file system creates a set of files which are copies of the specified file and all of its descendants, and inserts the new structure into the specified directory. A file cannot be copied into itself or any of its descendants.

```
NSFile.Copy: PROCEDURE [
    file: Handle, destination: Handle, attributes: AttributeList ← nullAttributeList,
    controls: Controls ← [], session: Session ← nullSession]
    RETURNS [newFile: Handle];
```

Arguments: **file** is a file handle for the file to be copied; destination is a file handle for the directory into which the copy is to be placed (the null handle may be specified); **attributes** specifies the characteristics of the new file and overrides those of the original file; **controls** specifies the controls to be applied to the returned handle; **session** is the client's session handle.

Results: **newFile** is a file handle for the newly-created file.

Access: **Copy** requires add access to destination, and read access to file and all of its descendants.

Errors: **NSFile.Error** may be raised with the following types: **access**, **attributeType**, **attributeValue**, **authentication**, **clearingHouse**, **connection**, **handle**, **insertion**, **service**, **session**, **space**, **transfer**, and **undefined**.

If **destination** is **nullHandle**, a temporary copy is made which is deleted by invoking **NSFile.Delete** or when all handles to it are closed. Since temporary directories are not allowed, **NSFile.Error[[handle[nullDisallowed]]]** is raised if **file** is a directory and **destination** is **nullHandle**.

The file system acquires a share lock for each file while copying it, and holds a share lock for each directory while enumerating and copying its children. However, it does *not* obtain a share lock on all descendants of **file** before starting to copy any of them; therefore, if a file is inserted in or deleted from some descendant of **file** while **Copy** is executing, the copy may or may not reflect the change.

3.7 Moving files

Move changes the directory structure of the file system. A specified file is moved into a specified directory. If the file was previously a child of another directory, it is removed from that directory. If the file was temporary, it becomes permanent. If the file has descendants, they are moved as well (i.e., they remain descendants of the file). A file may not be moved into itself or any of its descendants.

```
NSFile.Move: PROCEDURE [
    file: Handle, destination: Handle, attributes: AttributeList ← nullAttributeList,
    session: Session ← nullSession];
```

Arguments: **file** is a file handle for the file to be moved (it must be the session's only file handle for this file unless the file is temporary); **destination** is a file handle for the directory into which the file is to be placed (the null handle cannot be specified); **attributes** modifies the characteristics of the file; **session** is the client's session handle.

Results: **file** is moved to the new location.

Access: The file system must be able to acquire an exclusive lock on **file**, **destination**, and the parent of **file** (if one exists). In addition, if **file** is permanent, there must be no other open handles for **file** in **session** or any other session. The file system does not acquire locks to any descendant of **file**; if such a descendant is open, its chain of ancestors may be changed without warning.

Errors: **NSFile.Error** is raised with the following types: **access, attributeType, attributeValue, authentication, handle, insertion, session, space**, and **undefined**.

Note: Move between different services is equivalent to **Copy** followed by **Delete**; therefore, all notes applying to these operations also apply in this case.

Since only one session handle is specified, the **Move** and **Copy** operations can be used to move or copy files across different services only when a distributed session is used. When a directed session is used, these operations may only be used to move or copy files within the same service. To **Move** or **Copy** across services using directed sessions, one should use the **Serialize** and **Deserialize** operations (followed by **Delete**, when moving a file) to achieve the data transfer (see §3.8.2).

3.8 Bulk data transfer operations

Bulk data transfer operations in **NSFile** provide a means for clients to read or write the entire contents of a file or subtree of files sequentially. Each makes use of the **NSDataStream** abstraction, and therefore allows clients to pair two **NSFile** bulk data transfer operations or an **NSFile** bulk data transfer operation with a non-**NSFile** operation. Adherence to **NSDataStream** conventions permits the transfer of data between otherwise independent software entities.

Each bulk data transfer operation accepts one of the types **NSFile.Sink** or **NSFile.Source** as a parameter. Operations which expect to receive data from the client (**NSFile.Store**, **NSFile.Replace**, **NSFile.Deserialize**) must be supplied an **NSFile.Source** as a parameter. Operations which expect to send data to the client (**NSFile.Retrieve**, **NSFile.Serialize**) must be supplied an **NSFile.Sink** as a parameter. By selecting the appropriate variant (**proc**, **stream** or **none**) of the type required by an operation, the client controls exactly how the data stream is determined.

```
NSFile.Sink: TYPE = NSDataStream.Sink;
NSDataStream.Sink: TYPE = RECORD [
    SELECT type: * FROM
        proc = > [proc: PROCEDURE [NSDataStream.SourceStream]],
        stream = > [stream: NSDataStream.SinkStream],
        none = > [],
    ENDCASE];

NSFile.Source: TYPE = NSDataStream.Source;
NSDataStream.Source: TYPE = RECORD [
    SELECT type: * FROM
        proc = > [proc: PROCEDURE [NSDataStream.SinkStream]],
        stream = > [stream: NSDataStream.SourceStream],
        none = > [],
    ENDCASE];
```

The **proc** variant allows the client to be provided with a data stream. The client provides a procedure which is called at most once with the data stream on which the data is to be sent or received. After all of the data is sent or received, the client deletes the data stream (by invoking **Stream.Delete**) and then returns. At a point prior to deleting the data stream, the client may also elect to abort it using **NSDataStream.Abort**. This indicates that the data was not completely sent or received. Signals and errors may be raised from within the client's procedure and caught by the procedure which called the bulk data transfer operation; however, the client's procedure is still required to delete the data stream in an **UNWIND** catch phrase. If the **NSFile** bulk data transfer operation raises an error prior to sending or receiving the first byte of data, the client's procedure may or may not be called. If it is called, the error is as though it occurred during data transfer and the client is notified of the problem by the error **NSDataStream.Aborted** which is raised on the next (or first) **Stream** operation invoked by the client.

The **stream** variant allows the client to supply a data stream to an operation. This data stream is one typically received from another operation called previously with a **proc** variant of an **NSDataStream.Sink** or **NSDataStream.Source**. This first operation, however, need not be an **NSFile** operation. By using the **proc** variant in one bulk data transfer operation and supplying the resulting data stream in a **stream** variant to another bulk data transfer

operation, the client routes data directly from one operation to another. If one of the operations is a remote operation and the other is local, then the data will be transferred along the **Courier** connection of the remote operation. If both are remote operations to distinct system elements, then data is transferred along a connection joining those system elements (a third-party transfer).

Note: In Services 8.0, direct third party transfers are not implemented; instead, data is transferred via the two already-established **Courier** connections with the data being relayed on the client machine. This has no effect, however, on the client of Filing.

The **none** variant of an **NSFile.Sink** is used to request that the data be discarded. This might be useful for a client who wishes to determine if he has sufficient access to any indicated or implied files without actually performing the transfer. The **none** variant of an **NSFile.Source** indicates that the client has no data to send. It has the same effect as if the client immediately deleted his data stream, but eliminates any overhead incurred in establishing the data stream.

The content of a file which has more than one segment is always sent on a data stream using *segment encoding*. Single segment files in certain situations also use segment encoding. The segment encoding consists of a 512-byte index followed by the content of each of the segments, in ascending order, each padded with zeros to fill an integral number of 512-byte pages. The index consists of a **SegmentIndex**, serialized using standard Courier serialization padded with zeros to fill 512 bytes. **SegmentIndex** is provided here for explanatory purposes although it does not currently appear in any Mesa interface.

```
SegmentIndex: TYPE = LONG DESCRIPTOR FOR ARRAY OF SegmentIndexEntry;

SegmentIndexEntry: TYPE = RECORD [segmentNumber:CARDINAL, length: LONG CARDINAL];
```

Segment encoding is employed by **Retrieve** and **Serialize** for each file that has more than one segment or that is distinguished as a segmented file type (see §8.1.3). Segment encoding is assumed by **Replace**, **Store** and **Deserialize** for each file received which has one of the client-specified segmented file types. During subtree operations, **Serialize** and **Deserialize**, segment encoding is employed or not employed independently for each file in the subtree of files.

3.8.1 Single file operations

Retrieve, **Replace**, and **Store** operate on single files. Data the client sends or receives on the data stream constitute the content portion of the file, represented as an uninterpreted series of 8-bit bytes or as a segmented file using the segment encoding described above. No attribute or descendant information is included. Except when segment encoding is employed, the length of the file is exactly the number of bytes transferred.

Replace replaces the content of an existing file with the data received from the specified source. The previous content is discarded, and the file and segment lengths are set to reflect the data received.

```
NSFile.Replace: PROCEDURE [
    file: Handle, source: Source, attributes: AttributeList ← nullAttributeList,
    session: Session ← nullSession];
```

Arguments: **file** is a file handle for the file whose content is being replaced; **source** specifies the source that is to supply the new content of the file in accordance with **NSDataStream** conventions; **attributes** specifies characteristics of the resulting file; **session** is the client's session handle.

Results: The content of the file is changed to the supplied data.

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

Errors: **NSFile.Error** is raised with the following types: **access, attributeType, attributeValue, authentication, connection, handle, session, space, transfer, undefined**.

Retrieve transfers the content of an existing file to the specified sink.

```
NSFile.Retrieve: PROCEDURE [file: Handle, sink: Sink, session: Session ← nullSession];
```

Arguments: **file** is a file handle for the file whose content is being retrieved; **sink** specifies the sink that is to receive the content of the file in accordance with **NSDataStream** conventions; **session** is the client's session handle.

Results: None.

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

Errors: **NSFile.Error** is raised with the following types: **access, authentication, connection, handle, session,transfer, undefined**.

Store creates a file with a specified content. A new file is created with the specified attributes in the specified directory, and is filled with data received from the specified source.

```
NSFile.Store: PROCEDURE [
    directory: Handle, source: Source, attributes: AttributeList ← nullAttributeList,
    controls: Controls ← [], session: Session ← nullSession]
    RETURNS [file: Handle];
```

Arguments: **directory** is a file handle for the directory into which the new file is to be placed (the null handle may be specified, implying a temporary file); **source** specifies the source that is to supply the content of the file in accordance with **NSDataStream** conventions; **attributes** specifies the characteristics of the new file; **controls** specifies the controls to be applied to the resulting handle; **session** is the client's session handle.

Results: **file** is a file handle for the newly-created file.

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

Errors: NSFile.Error is raised with the following types: access, attributeType, attributeValue, authentication, clearingHouse, connection, controlType, controlValue, handle, insertion, service, session, space, transfer, undefined.

3.8.2 Subtree operations, serialized files

At times, it is useful to compress all of the information contained in a file and its descendants into a series of eight-bit bytes in order to transfer it to another system element, store it on some other medium, or manipulate it in some other way. The format of data in this series of bytes is the serialized file format. Serializing a file produces a series of bytes which contains all of the information in the file and its descendants; deserializing such a series of bytes recreates a file and its descendants.

Procedures in this section transfer a serialized file to a sink or from a source using the **NSDataStream** mechanism. The data may be thought of as a single **Courier** object (i.e., encoded according to the **Courier** conventions) of type **SerializedFile**. However, neither the following definitions nor the corresponding **Courier** descriptions actually appear in any Mesa interface. This is done because it is more desirable to process the information on the data stream sequentially than to do so all at once, thereby avoiding the allocation of a very large Mesa data structure.

```
SerializedFile: TYPE = RECORD [version: LONG CARDINAL, file: SerializedTree];

currentVersion: LONG CARDINAL = 2;

SerializedTree: TYPE = RECORD [
    attributes: NSFile.AttributeList,
    content: RECORD [data: SeriesOfUnspecified, lastByteIsSignificant: BOOLEAN],
    children: LONG DESCRIPTOR FOR ARRAY OF SerializedTree];

SeriesOfUnspecified: TYPE = RECORD [
    SELECT type:* FROM
    nextBlock = > RECORD [
        block: LONG DESCRIPTOR FOR ARRAY OF UNSPECIFIED,
        restOfStream: LONG POINTER TO SeriesOfUnspecified],
    lastBlock = > LONG DESCRIPTOR FOR ARRAY OF UNSPECIFIED];
```

A serialized file begins with a version number to distinguish it from other versions of the serialized file format. The representation of each file within the serialization consists of its attributes, its content, and all of its children. The attribute list contains attributes that apply to the file, in arbitrary order. The sequence of children corresponds to their order within the original directory.

The content of a file is represented as a series of sixteen-bit words followed by an indication of whether the last byte of the last word is significant (that is, whether the length in bytes is even). If not, the last byte has the value zero and should be ignored. The type **SeriesOfUnspecified** partitions the content of a file into variable-sized blocks. The concatenation of the ARRAY OF UNSPECIFIED fields makes up the content of the file. This content, as described above, may either be an uninterpreted series of bytes or a segment encoding.

Note: Block boundaries are insignificant; that is, they carry no information. Blocks within the serialization of a file are unrelated to and should not be confused with the blocks sent or received on a data stream, or the unit of storage on any physical storage medium.

The bulk data transfer operations which operate on a subtree of files are **Serialize** and **Deserialize**.

Serialize encodes all of the information of a file and its descendants into a series of bytes according to the serialized file format above (including its attributes, content, and descendants).

```
NSFile.Serialize: PROCEDURE [file: Handle, sink: Sink, session: Session ← nullSession];
```

Arguments: **file** is a file handle for the file which is being serialized; **sink** specifies the sink that is to receive the serialization in accordance with **NSDataStream** conventions; **session** is the client's session handle.

Results: None.

Access: Read access is required to **file** and all its descendants.

Errors: **NSFile.Error** is raised with the following types: **access**, **authentication**, **connection**, **handle**, **session**, **transfer**, **undefined**.

Deserialize reconstructs a file and its descendants from a serialized representation. A new file is created in the specified directory; its attributes, content and descendants are constructed from the serialized file and a file handle for the file is returned. During deserialization, some attributes (for example, **numberOfChildren**) are ignored because the attribute duplicates information implicit in the rest of the data. It does not replace the existing file. Rather than reporting an error, **Deserialize** ignores attributes in the serialized file that are not allowed to be specified. Attributes that are not specified are given default values.

```
NSFile.Deserialize: PROCEDURE [
    directory: Handle, source: Source, attributes: AttributeList ← nullAttributeList,
    controls: Controls ← [], session: Session ← nullSession]
    RETURNS [file: Handle];
```

Arguments: **directory** is a file handle for the directory into which the file is to be placed (the null handle may be specified only if the subtree being deserialized is a non-directory); **source** specifies the source that is to supply the file in accordance with **NSDataStream** conventions; **attributes** specifies the characteristics of the new file (overriding corresponding attributes specified in the serialized file); **controls** specifies the controls to be applied to the returned handle; **session** is the client's session handle.

Results: **file** is a file handle for the newly-created file.

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

Errors: **NSFile.Error** is raised with the following types: **access**, **attributeType**, **attributeValue**, **authentication**, **clearingHouse**, **connection**, **controlType**, **controlValue**, **handle**, **insertion**, **service**, **session**, **space**, **transfer**, **undefined**.

3.9 Macro operations

For convenience, the file system provides *macro operations* which execute common sequences of **NSFile** operations. There are two categories of such operations: child operations and pathname operations.

3.9.1 Child operations

A client might wish to operate on a child of a directory without the bother of opening the child. Child operations implement a limited set of operations on children, identified uniquely by their **fileID** attribute:

OpenChild opens the child of **directory** with the given **id**, using the specified controls. Calling this operation is equivalent to calling **NSFile.Open** with an attribute list containing the single attribute **fileID**.

```
NSFile.OpenChild: PROCEDURE [
    directory: Handle, id: ID, controls: Controls ← [], session: Session ← nullSession]
    RETURNS [Handle];
```

CopyChild copies the child of **directory** with the given **id**, inserting the copy into **destination**. The **fileID** attribute of the copy is returned.

```
NSFile.CopyChild: PROCEDURE [
    directory: Handle, id: ID, destination: Handle,
    attributes: AttributeList ← nullAttributeList, session: Session ← nullSession]
    RETURNS [ID];
```

MoveChild moves the child of **directory** with the given **id** into **destination**.

```
NSFile.MoveChild: PROCEDURE [
    directory: Handle, id: ID, destination: Handle,
    attributes: AttributeList ← nullAttributeList, session: Session ← nullSession];
```

DeleteChild deletes the child of **directory** with the given **id**.

```
NSFile.DeleteChild: PROCEDURE [directory: Handle, id: ID, session: Session ← nullSession];
```

GetAttributesChild fills in **attributes** with the selected attributes of the child of **directory** with the given **id**.

```
NSFile.GetAttributesChild: PROCEDURE [
    directory: Handle, id: ID, selections: Selections, attributes: Attributes,
    session: Session ← nullSession];
```

ChangeAttributesChild changes the specified attributes of the child of **directory** with the given **id**.

```
NSFile.ChangeAttributesChild: PROCEDURE [
    directory: Handle, id: ID, attributes: AttributeList, session: Session ← nullSession];
```

ReplaceChild replaces the contents of the child of **directory** with the given **id** from the supplied **source**.

```
NSFile.ReplaceChild: PROCEDURE [
    directory: Handle, id: ID, source: Source, attributes: AttributeList ←
nullAttributeList,
    session: Session ← nullSession];
```

RetrieveChild retrieves the contents of the child of **directory** with the given **id** to the supplied **sink**.

```
NSFile.RetrieveChild: PROCEDURE [
    directory: Handle, id: ID, sink: Sink, session: Session ← nullSession];
```

All child operations may raise **NSFile.Error** or **Courier.Error** with any parameter that might result from calling **NSFile.Open** or the underlying operation (**Copy**, **Move**, etc.).

3.9.2 Pathname operations

A client may wish to operate on a file by specifying a pathname rather than identifying it by its **fileID** or explicitly opening the file and its ancestors. Pathname operations implement a limited set of operations on files identified by their pathnames.

The pathname specified may be the absolute pathname for the file (in which case it is relative to the root file of the service implied by the session), or it may be relative to the specified directory. If the pathname does not end with a version number, the file with the highest version number is used (except in **DeleteByName**, in which the file with the lowest version number is deleted).

Remote files may be accessed by specifying the *qualified* pathname for the file, i.e., a pathname containing the name of the service on which the file resides. In this case, the specified directory *must* be null, or **NSFile.Error [handle[nullRequired]]** is raised.

The pathname operations use the operations defined in section 7, to parse qualified pathnames. See this section for complete details on pathname syntax, as well as some examples of both qualified and service-relative pathnames.

OpenByName opens a file with the specified pathname relative to the specified directory, using the specified controls.

```
NSFile.OpenByName: PROCEDURE [
    directory: Handle, path: String, controls: Controls ← [],
    session: Session ← nullSession] RETURNS [Handle];
```

CopyByName copies a file with a specified pathname relative to the specified directory, inserting the copy into **destination**, and closing the copy. The **fileID** attribute of the copy is returned.

```
NSFile.CopyByName: PROCEDURE [
    directory: Handle, path: String, destination: Handle,
    attributes: AttributeList ← nullAttributeList, session: Session ← nullSession]
    RETURNS [ID];
```

MoveByName moves a file with a specified pathname relative to the specified directory into **destination**.

```
NSFile.MoveByName: PROCEDURE [
    directory: Handle, path: String, destination: Handle,
    attributes: AttributeList ← nullAttributeList, session: Session ← nullSession];
```

DeleteByName deletes a file with a specified pathname relative to the specified directory. If the pathname does not include a version number, the file with the lowest version number is deleted.

```
NSFile.DeleteByName: PROCEDURE [
    directory: Handle, path: String, session: Session ← nullSession];
```

GetAttributesByName fills **attributes** with the selected attributes of a file with a specified pathname relative to the specified directory.

```
NSFile.GetAttributesByName: PROCEDURE [
    directory: Handle, path: String, selections: Selections, attributes: Attributes,
    session: Session ← nullSession];
```

ChangeAttributesByName changes the specified attributes of a file with a specified pathname relative to the specified directory.

```
NSFile.ChangeAttributesByName: PROCEDURE [
    directory: Handle, path: String, attributes: AttributeList,
    session: Session ← nullSession];
```

ReplaceByName replaces the contents of a file having a specified pathname relative to a specified directory with data from the supplied source.

```
NSFile.ReplaceByName: PROCEDURE [
    directory: Handle, path: String, source: Source,
    attributes: AttributeList ← nullAttributeList, session: Session ← nullSession];
```

RetrieveByName retrieves the contents of a file having a specified pathname relative to a specified directory to the designated sink.

```
NSFile.RetrieveByName: PROCEDURE [
    directory: Handle, path: String, sink: Sink, session: Session ← nullSession];
```

3.10 Errors

When a Filing operation is unable to complete successfully, it reports this fact by raising one of the Mesa errors, NSFile.Error or Courier.Error. These errors are used to report any condition that makes continued execution of a procedure impossible. For example, the client may have specified incorrect arguments to a procedure, or some required resource may be unavailable.

Note: Courier.Error may be raised by any filing operation when one of the operands is a remote file or a remote file service is implied. Operations on local files only raise NSFile.Error. Consult *Pilot Programmer's Manual* [26] for further details about Courier.Error.

```
NSFile.Error: ERROR [error: ErrorRecord];

NSFile.ErrorRecord: TYPE = RECORD [SELECT errorType: ErrorType FROM
    access => [problem: AccessProblem],
    attributeType, attributeValue => [
        problem: ArgumentProblem, type: AttributeType,
        extendedType: ExtendedAttributeType ← LAST[ExtendedAttributeType],
    authentication => [problem:AuthenticationProblem],
    clearingHouse => [problem:ClearinghouseProblem],
    connection => [problem: ConnectionProblem],
    controlType, controlValue => [problem: ArgumentProblem, type: ControlType],
    handle => [problem: HandleProblem],
    insertion => [problem: InsertionProblem],
    range => [problem: ArgumentProblem],
    scopeType, scopeValue => [problem: ArgumentProblem, type: ScopeType],
    service => [problem: ServiceProblem],
    session => [problem: SessionProblem],
    space => [problem: SpaceProblem],
    transfer => [problem: TransferProblem],
    undefined => [problem: UndefinedProblem],
    ENDCASE];

NSFile.ErrorType: TYPE = {
    access, attributeType, attributeValue, authentication, clearingHouse, connection,
    controlType, controlValue, handle, insertion, range, scopeType, scopeValue,
    service, session, space, transfer, undefined};
```

The argument to NSFile.Error is a variant record, each arm of which defines a subclass (NSFile.ErrorType) of error conditions. The specific problem is described by the fields of the particular variant. For example, an **ErrorType** of **handle** indicates that something is wrong with a file handle specified in the arguments of a procedure. The particular problem with the file handle is specified by the **problem** field which is of type **HandleProblem**.

When an exceptional condition arises during execution of a procedure, the file system makes every effort to undo the effects of the partial execution so that the file system appears to the client as though the procedure had never been called. However, the file system does not guarantee that such effects can always be reversed. Therefore, when an error is raised, the client must be prepared for the possibility that the procedure was partially executed. In any event, no files are lost unless deletion was requested.

3.10.1 Access errors

```
NSFile.ErrorRecord: TYPE = RECORD [SELECT errorType: ErrorType FROM
    access => [problem: AccessProblem], ...];

NSFile.AccessProblem: TYPE = MACHINE DEPENDENT {
    accessRightsInsufficient(0), accessRightsIndeterminate(1), fileChanged(2),
    fileDamaged(3), fileInUse(4), fileNotFound(5), fileOpen(6),
    fileNotLocal(7)};
```

An error of type **access** may be raised by any procedure that requires access to a file. It indicates that access to the file is not possible. The inaccessible file is not necessarily the one whose handle was specified as an argument to the procedure call because some procedures operate on additional files. For example, **Delete** deletes the descendants of a specified file as well as the file itself.

The argument **problem** describes the problem in greater detail.

accessRightsInsufficient — The user does not have the access rights (**NSFile.Access**) needed to satisfy the request, either because the access list does not grant that access or because a handle's controls do not permit that access.

accessRightsIndeterminate — The file system could not determine whether the user has the access rights needed to satisfy the request; e.g., a Clearinghouse containing group-membership information is inaccessible.

fileChanged — While the procedure was executing, the file changed in such a way that execution could not continue; this condition can occur during **List** if the ordering of the directory changes.

fileDamaged — A file was found to be internally damaged in some way, but not badly enough to require shutdown of the file system.

fileInUse — Even after expiration of a timeout, the file system could not acquire a lock needed to satisfy the request. A conflicting lock on a handle to the file exists within another session.

fileNotFound — A file was not found in the context in which it was expected.

fileOpen — During an attempt to move or delete a file, another file handle for the file was found to exist in the same session.

fileNotLocal — An attempt was made to access a non-local file with an operation which is implemented only for local files. **NSFileStream** and **NSSegment** operations may never be called with remote files.

3.10.2 Argument errors

There are argument error classes for several types of Filing procedure arguments: attributes, controls, and scopes. A given argument error may be raised by any procedure that has an argument of the corresponding type. For each argument type, there are two error classes. The type-related error indicates that specifying that attribute (control, scope) type resulted in a problem; the value-related error indicates that the attribute (control, scope) type was legitimate, but the specified value caused a problem.

```
NSFile.ErrorRecord: TYPE = RECORD [SELECT errorType: ErrorType FROM
    ..., attributeType, attributeValue => [
        problem: ArgumentProblem, type: AttributeType,
        extendedType: ExtendedAttributeType ← LAST[ExtendedAttributeType]], ...];
```

An error of type **attributeType** is raised when an attribute type specified in an **NSFile.AttributeList** or **NSFile.ExtendedSelections** causes a problem. An error of type **attributeValue** is raised when an attribute value specified in an **NSFile.AttributeList** causes a problem. The argument **type** indicates the type of the offending attribute or the type of the offending attribute value. If **type** has the value **extended**, then the argument **extendedType** indicates the type of the offending extended attribute or the type of the offending extended attribute value.

```
NSFile.ErrorRecord: TYPE = RECORD [SELECT errorType: NErrorType FROM
    ..., controlType, controlValue => [
        problem: ArgumentProblem, type: ControlType],...];
```

Errors of type **controlType** and **controlValue** are not currently raised by a correctly functioning file system.

```
NSFile.ErrorRecord: TYPE = RECORD [SELECT errorType: ErrorType FROM
    ..., scopeType, scopeValue => [problem: ArgumentProblem, type: ScopeType], ...];

NSFile.ScopeType: TYPE = MACHINE DEPENDENT {count(0), direction(1), filter(2), ordering(3)};
```

Errors of type **scopeType** are not currently raised by a correctly functioning file system.

An **NSFile.Error** of type **scopeValue** (with **problem** equal to **unimplemented**) may be raised if a designated scope's filter or ordering value is not implemented by the file system containing the supplied file handle. The argument **type** indicates the type of the offending scope value.

In each of the above error classes, the argument **problem** describes the problem in greater detail.

```
NSFile.ArgumentProblem: TYPE = {
    illegal(0), disallowed(1), unreasonable(2), unimplemented(3), duplicated(4),
    missing(5)};
```

illegal	The attribute, control, or scope value is never allowed; for instance, a **name** attribute of length zero, a string attribute with an invalid string format, a string attribute whose length is greater than **NSFile.maxNameLength**, an invalid position, or a filter containing an invalid string or position.
disallowed	The attribute type or value is sometimes allowed, but is never allowed by this remote procedure.
unreasonable	The attribute type or value is sometimes allowed by the procedure raising the error, but not in the context in which it was supplied; for example, it may conflict with other arguments.
unimplemented	The value is not supported by this implementation of the file system; this condition can only occur for certain values of the **filter** scope, **ordering** scope, and the **ordering** attribute, but never occurs for types.
duplicated	The attribute type is specified more than once in an **NSFile.AttributeList** or **NSFile.ExtendedSelections**.
missing	The attribute type is missing in a context in which it is required; this condition can occur for certain attribute types in **NSFile.Open**, for example.

3.10.3 Authentication errors

An **NSFile.Error** of type **authentication** may be raised by **NSFile.Logon**, **NSFile.LogonDirect**,or by any procedure that accepts an argument of type **NSFile.Session**. It indicates that there is a problem communicating with the authentication service, that there is a problem with the supplied identity, or a problem with the identity of the remote service.

```
NSFile.ErrorRecord: TYPE = RECORD [SELECT errorType: ErrorType FROM
    ..., authentication = > [problem:AuthenticationProblem], ...];

NSFile.AuthenticationProblem: TYPE = {
    cannotReachAS, credentialsTooWeak, keysUnavailable, other,
    simpleKeyDoesNotExist, strongKeyDoesNotExist, tooBusy};
```

cannotReachAS	The Authentication Service cannot be reached.
credentialsTooWeak	Stronger credentials are required for interaction with the desired service.

keysUnavailable The Clearinghouse serving the domain in which the client identity or service identity is registered is unavailable.

other An unanticipated error in authentication occurred.

simpleKeyDoesNotExist If the supplied client identity was validated by the client prior to **Logon**, then this indicates that the service being accessed is not registered with a simple key. If the client identity was not validated prior to **Logon**, then this could also indicate that the client identity is not registered with a simple key.

strongKeyDoesNotExist If the supplied client identity was validated by the client prior to **Logon**, then this indicates that the service being accessed is not registered with a strong key. If the client identity was not validated prior to **Logon**, then this could also indicate that the client identity is not registered with a strong key.

tooBusy The authentication service is currently too busy to serve the authentication request.

The reader should consult *Authentication Protocol* [2] for further explanation of authentication problem types.

3.10.4 Clearinghouse errors

An **NSFile.Error** of type **clearingHouse** may be raised by **NSFile.LogonDirect**, or by any procedure that accepts a reference or an attribute list which may contain a **service** attribute. It indicates that there is a problem communicating with the Clearinghouse Service, that there is a problem with a supplied Clearinghouse name of a service, or that the Clearinghouse entry for a service was not found.

```
NSFile.ErrorRecord: TYPE = RECORD [SELECT errorType: ErrorType FROM
    ..., clearingHouse => [problem:ClearinghouseProblem], ...];

NSFile.ClearinghouseProblem: TYPE = {
    notAllowed, rejectedTooBusy, allDown, illegalOrgName, illegalDomainName,
    illegalLocalName, noSuchOrg, noSuchDomain, noSuchLocal, other,
    wasUpNowDown};
```

notAllowed A clearinghouse operation was prevented by access controls.

rejectedTooBusy The Clearinghouse service is too busy to service the current Clearinghouse request.

allDown The Clearinghouse server was unavailable and was needed for the operation.

illegalOrgName	The organization portion of the supplied Clearinghouse name has illegal length or illegal characters.
illegalDomainName	The domain portion of the supplied Clearinghouse name has illegal length or illegal characters.
illegalLocalName	The local portion of the supplied Clearinghouse name has illegal length or illegal characters.
noSuchOrg	The specified organization does not exist.
noSuchDomain	The specified domain does not exist in the specified organization.
noSuchLocal	The specified local name does not exist in the specified domain.
other	An unanticipated error in Clearinghouse interaction occurred.
wasUpNowDown	The Clearinghouse became unavailable during the course of the operation.

The reader should consult *Clearinghouse Protocol* [8] for further explanation of Clearinghouse problem types.

3.10.5 Connection errors

An **NSFile.Error** of type **connection** may be raised by any procedure that accepts an argument of type **NSFile.Sink** or **NSFile.Source**. It indicates a problem in establishing the connection for transfer of bulk data in a third-party transfer (see §3.8).

[**Note:** Because direct third-party transfers are not implemented in Services 8.0, connection problems are not reported.]

```
NSFile.ErrorRecord: TYPE = RECORD [SELECT errorType: ErrorType FROM
    ..., connection => [problem: ConnectionProblem], ...];

NSFile.ConnectionProblem: TYPE = MACHINE DEPENDENT {
    -- communication problems
    noRoute(0), noResponse(1), transmissionHardware(2), transportTimeout(3),
    -- resource problems
    tooManyLocalConnections(4), tooManyRemoteConnections(5),
    --remote program implementation problems
    missingCourier(6), missingProgram(7), missingProcedure(8), protocolMismatch(9),
    parameterInconsistency(10), invalidMessage(11), returnTimedOut(12),
    -- miscellaneous
    otherCallProblem(177777B) };
```

The argument **problem** describes the problem in greater detail.

noRoute	No route to the other party could be found.
noResponse	The other party never answered.
transmissionHardware	Some local transmission hardware was inoperable.
transportTimeout	The other party responded but the connection was broken.
tooManyLocalConnections	No additional connection is possible.
tooManyRemoteConnections	The other party rejected the connection attempt.
missingCourier	The other party had no **Courier** implementation.
missingProgram	The other party did not implement the bulk data program.
missingProcedure	The other party did not implement the procedure.
protocolMismatch	The two parties have no **Courier** version in common.
parameterInconsistency	A protocol violation occurred in parameters.
invalidMessage	A protocol violation occurred in message format.
returnTimedOut	The procedure call never returned.
otherCallProblem	Some other protocol violation during a call.

3.10.6 Handle errors

An **NSFile.Error** of type **handle** may be raised by any procedure that takes an argument of type **NSFile.Handle**. It indicates a problem with a supplied file handle.

```
NSFile.ErrorRecord: TYPE = RECORD [SELECT errorType: ErrorType FROM
    ..., handle => [problem: HandleProblem], ...];

NSFile.HandleProblem: TYPE = MACHINE DEPENDENT {
    invalid(0), nullDisallowed(1), directoryRequired(2) obsolete (3), nullRequired(4)};
```

The argument **problem** describes the problem in greater detail.

invalid	An invalid file handle was specified; it may be a handle that was already closed in the current session or it may be a valid file handle in another session.
nullDisallowed	The null handle was specified as a value for an argument that requires a valid handle to a file.

directoryRequired — A handle to a non-directory was specified as a value for an argument to a procedure (e.g., **NSFile.List**, **NSFile.Store**) that requires a handle to a directory.

obsolete — A handle for a non-local file is no longer valid because of a communication failure between the client's machine and the machine containing the file. The client should close the obsolete file handle and reopen it. This error can happen for non-local files only.

nullRequired — A non-null file handle was specified as an argument where a null handle should have been specified. This error will occur when calling a pathname operation (like **NSFile.OpenByName**) with a qualified pathname and a non-null directory handle.

3.10.7 Insertion errors

An **NSFile.Error** of type **insertion** may be raised by any procedure that inserts a file into a directory whether the file being inserted is a new file or is being moved from elsewhere. It indicates that the directory could not accommodate the file. It may also be raised by **NSFile.ChangeAttributes** if the file's new name or version number cannot be accommodated.

```
NSFile.ErrorRecord: TYPE = RECORD [SELECT errorType: ErrorType FROM
    ..., insertion => [problem: InsertionProblem], ...];

NSFile.InsertionProblem: TYPE = {
    positionUnavailable(0), fileNotUnique(1), loopInHierarchy(2)};
```

The argument **problem** describes the problem in greater detail.

positionUnavailable — The directory is ordered by position, and the density of files in the area surrounding the specified position is so great that no point for insertion is available; the directory must be reorganized (as described in §6.3.6).

fileNotUnique — The directory already contains a file with the same name (if the directory's **childrenUniquelyNamed** attribute is **TRUE**) or the same name and version (if the directory's **childrenUniquelyNamed** attribute is **FALSE**). This error will also be raised if the file system finds that it must assign a value of **LAST [CARDINAL]** to a file because there already exists a file of the same name with the highest legal version number (i.e., **LAST [CARDINAL] - 1**).

loopInHierarchy — The directory is the same as, or a descendant of, the file being moved or copied.

3.10.8 Service errors

An NSFile.Error of type **service** may be raised by **LogonDirect**, **Logoff**, or **Open**, and when creating a temporary file using **Create**, **Copy**, **Deserialize**, or **Store**. It indicates that the file system encountered a problem while attempting to create or destroy a session, possibly on a remote file service.

```
NSFile.ErrorRecord: TYPE = RECORD [SELECT errorType: ErrorType FROM
    ..., service => [problem: ServiceProblem], ...];

NSFile.ServiceProblem: TYPE = MACHINE DEPENDENT {
    cannotAuthenticate(0), serviceFull(1), serviceUnavailable(2), sessionInUse(3),
    serviceUnknown(4)};
```

The argument **problem** describes the problem in greater detail.

cannotAuthenticate	The specified file service is unable to determine whether the user's credentials are valid; this could occur if the file service needs to contact some service that is unavailable.
serviceFull	This operation would cause the number of sessions on the specified file service to exceed an implementation-dependent limit.
serviceUnavailable	The remote file service is currently unavailable for use by new clients.
sessionInUse	The client may not log off because another **NSFile** procedure is still executing in the session. This can occur if the client attempts to log off within a call-back procedure (a **source** data stream procedure, a **sink** data stream procedure, **AttributesProc**, or from a separate process).
serviceUnknown	The specified file service is unknown. This error is raised if there is no open volume at the specified system element having the specified service name.

3.10.9 Range errors

An NSFile.Error of type **range** is used to report errors on remote random access operations. Since random access to remote files is not implemented in Services 8.0, this error is not currently raised.

```
NSFile.ErrorRecord: TYPE = RECORD [SELECT errorType: ErrorType FROM
    ..., range => [problem: ArgumentProblem], ...];
```

See §3.10.2 for a description of **ArgumentProblem** types.

3.10.10 Session errors

An NSFile.Error of type **session** may be raised by any procedure which accepts a session handle argument. It indicates that the session handle is invalid.

```
NSFile.ErrorRecord: TYPE = RECORD [SELECT errorType: ErrorType FROM
    ..., session => [problem: SessionProblem], ...];

NSFile.SessionProblem: TYPE = MACHINE DEPENDENT {sessionInvalid(0)};
```

There is currently only one session problem type, **sessionInvalid**. It indicates that the passed session handle is not valid. The client may have already called **Logoff** or the session may have been forcibly terminated by the file system.

3.10.11 Space errors

An NSFile.Error of type **space** may be reported by any procedure that must allocate physical space for the storage of information. It indicates that the request for space could not be satisfied.

```
NSFile.ErrorRecord: TYPE = RECORD [SELECT errorType: ErrorType FROM
    ..., space => [problem: SpaceProblem], ...];

NSFile.SpaceProblem: TYPE = MACHINE DEPENDENT {
    allocationExceeded(0), attributeAreaFull(1), mediumFull(2)};
```

The argument **problem** describes the problem in greater detail.

allocationExceeded	The space required by the procedure caused some ancestor's subtree size limit to be exceeded.
attributeAreaFull	There was not enough space in the attribute area to satisfy the request; the limits described in §6.3.7 would have been exceeded.
mediumFull	There was not enough space on the appropriate file service to satisfy the request.

3.10.12 Transfer errors

An NSFile.Error of type **transfer** may be reported by any procedure that sends data to a sink or receives data from a source. It indicates that a problem occurred during the transfer.

```
NSFile.ErrorRecord: TYPE = RECORD [SELECT errorType: ErrorType FROM
    ..., transfer => [problem: TransferProblem], ...];

NSFile.TransferProblem: TYPE = MACHINE DEPENDENT {
    aborted(0), checksumIncorrect(1), formatIncorrect(2),
    noRendezvous(3), wrongDirection(4)};
```

The argument **problem** describes the problem in greater detail.

aborted	The sink or source's procedure aborted the transfer, or the bulk data transfer was aborted by the party at the other end of the sink or source's stream. If the party aborting the transfer is another **NSFile** operation, it reports an **NSFile.Error** or **Courier.Error** describing the nature of the problem.
checksumIncorrect	After transfer of a file's content to a sink, the checksum computed over the data did not match the file's stored **checksum** attribute, or after transfer of a file's content from a source, the checksum computed over the data did not match the **checksum** attribute specified in the attribute list to the operation.
formatIncorrect	The bulk data received from the source did not have the expected format; for instance, **Deserialize** only accepts files in the serialized file format.
noRendezvous	The identifier from the other party never appeared.
wrongDirection	The other party wanted to transfer the data in the wrong direction.

3.10.13 Undefined errors

An **NSFile.Error** of type **undefined** may be reported by any procedure. It indicates that an implementation-dependent problem occurred that could not be reported by another error. This error is normally reported only when the local or remote file service is malfunctioning. The client has no way of recovering from undefined errors.

```
NSFile.ErrorRecord: TYPE = RECORD [SELECT errorType: ErrorType FROM
    ..., undefined => [problem: UndefinedProblem], ...];
```

The argument **problem** describes the problem in greater detail and is uninterpretable.

```
NSFile.UndefinedProblem: TYPE = CARDINAL;
```

4

Segment/content operations

NSSegment: DEFINITIONS . . . ;

Every file is divided into segments, which are simply disjoint, independently-growable sections of a file. Every segment has a unique identifier associated with it, its **ID**. All files are created with a *default* segment (**defaultID**), which is always present and cannot be renumbered or deleted.

```
NSSegment.ID: TYPE = CARDINAL;
NSSegment.defaultID: ID = 0;
NSSegment.nullID: ID = LAST [ID];
```

NSSegment provides procedures useful in operating on the segments of a file.

Note: **NSSegment** operations may operate on local files only. Any attempt to operate on a remote file will cause **NSFile.Error [access[fileNotLocal]** to be raised.

4.1 Finding and listing segments of a file

FindUnused is called to discover an unused segment in a file.

```
NSSegment.FindUnused: PROCEDURE [
    file: NSFile.Handle, startID: ID ← defaultID, session: Session ← nullSession]
    RETURNS [ID];
```

Arguments: **file** refers to the handle of a file in which an unused segment is to be found; **startID** specifies a minimum value for the identifier of the unused segment (and need *not* describe an existing segment); **session** is the client's session handle.

Results: The returned **ID** refers to the first unused segment whose segment identifier is greater than or equal to **startID**.

Errors: **NSSegment.Error [noSuchSegment]**, and **NSSegment.Error [tooManySegments]** may be raised.

GetNext enumerates the set of segments comprising a file.

```
NSSegment.GetNext: PROCEDURE [
    file: NSFile.Handle, currentSegment: ID, session: Session ← nullSession] RETURNS [ID];
```

Arguments: **file** identifies the file whose segments are to be enumerated; **currentSegment** identifies the segment from which the enumeration is to proceed; **session** is the client's session handle.

Results: The returned **ID** refers to the segment after **currentSegment** in the enumeration.

GetNext is a stateless enumerator. To begin the enumeration, the caller specifies either **nullID** (in which case **defaultID** is included in the enumeration) or **defaultID**. The enumeration is complete when **nullID** is returned.

Errors: **NSSegment.Error [noSuchSegment]** may be raised.

4.2 Adding, deleting, and moving segments

The client may add segments to a file by using **NSSegment.Add**.

```
NSSegment.Add: PROCEDURE [
    file: NSFile.Handle, segment: ID, size: PageCount, session: Session ← nullSession];
```

Arguments: **file** refers to the handle of a file to which a new segment is to be added; **segment** is an identifier for the new segment; **size** is the new segment size in pages; **session** is the client's session handle.

Errors: **NSSegment.Error[invalidSegmentID]**, **NSSegment.Error[segmentAlreadyExists]**, **NSSegment.Error[tooManySegments]**, and **NSFile.Error[[space[mediumFull]]]** may be raised.

The maximum number of non-default segments that may be added to a file is defined by the read-only constant, **maxNumberOfSegments**.

```
NSSegment.maxNumberOfSegments: READONLY CARDINAL;
```

The client may remove any segment (except the default) by calling **NSSegment.Delete**.

```
NSSegment.Delete: PROCEDURE [
    file: NSFile.Handle, segment: ID, session: Session ← nullSession];
```

Arguments: **file** identifies the file from which a segment is to be removed; **segment** is the identifier of the segment to be deleted; **session** is the client's session handle.

Errors: **NSSegment.Error[illegalForDefault]**, and **NSSegment.Error[noSuchSegment]** may be raised.

It is also possible for a client to change the segment identifier associated with a particular segment (except the default). This is done by invoking **NSSegment.Move**.

```
NSSegment.Move: PROCEDURE [
    file: NSFile.Handle, oldSegment, newSegment: ID, session: Session ← nullSession];
```

Arguments: **file** identifies the file which contains the segment of interest; **oldSegment** is the segment's current identifier; **newSegment** is the segment's new identifier; **session** is the client's session handle.

Errors: **NSSegment.Error[illegalForDefault]**, **NSSegment.Error[invalidSegmentID]**, **NSSegment.Error[noSuchSegment]**, and **NSSegment.Error[segmentAlreadyExists]** may be raised.

No error is raised if **oldSegment** and **newSegment** are the same, provided that the segment actually exists and is not the default segment.

NumberOfSegments returns the total number of segments in a file, *including* the default segment. Thus calling **NumberOfSegments** always returns a result greater than or equal to one.

```
NSSegment.NumberOfSegments: PROCEDURE [
    file: NSFile.Handle, session: Session ← nullSession] RETURNS [CARDINAL];
```

Arguments: **file** refers to the file handle of interest; **session** is the client's session handle.

Results: The result represents the total number of segments comprising **file**.

4.3 Accessing and modifying segment sizes

The client may obtain the size of an existing segment with **GetSizeInBytes** or **GetSizeInPages** and change the size of a segment with **SetSizeInBytes** or **SetSizeInPages**.

```
NSSegment.ByteCount: TYPE = LONG CARDINAL;
NSSegment.PageCount: TYPE = LONG CARDINAL;

NSSegment.GetSizeInBytes: PROCEDURE [
    file: NSFile.Handle, segment: ID ← defaultID, session: Session ← nullSession]
    RETURNS [ByteCount];
```

Arguments: **file** is the handle of a file which contains the segment of interest; **segment** is the identifier of a segment whose size in bytes is to be determined; **session** is the client's session handle.

Results: The returned **ByteCount** is the size of the specified segment.

Errors: **NSSegment.Error[noSuchSegment]** may be raised.

```
NSSegment.SetSizeInBytes: PROCEDURE [
    file: NSFile.Handle, bytes: ByteCount, segment: ID ← defaultID,
    session: Session ← nullSession];
```

Arguments: **file** is the handle of a file which contains the segment of interest; **bytes** is the new size which the segment is to have (must be a multiple of 512 if not the default segment); **segment** is the identifier of a segment whose size is to be changed; **session** is the client's session handle.

Errors: **NSSegment.Error[noSuchSegment]** and **NSFile.Error[[space[mediumFull]]]** may be raised.

```
NSSegment.GetSizeInPages: PROCEDURE [
    file: NSFile.Handle, segment: ID ← defaultID, session: Session ← nullSession]
    RETURNS [PageCount];
```

Arguments: **file** is the handle of a file which contains the segment of interest; **segment** is the identifier of the segment whose size is to be determined; **session** is the client's session handle.

Results: The returned **PageCount** is the size of the specified segment in pages.

Errors: **NSSegment.Error[noSuchSegment]** may be raised.

```
NSSegment.SetSizeInPages: PROCEDURE [
    file: NSFile.Handle, pages: PageCount, segment: ID ← defaultID,
    session: Session ← nullSession];
```

Arguments: **file** is the handle of a file which contains the segment of interest; **pages** is the new size which the segment is to have; **segment** is the identifier of the segment whose size is to be changed; **session** is the client's session handle.

Errors: **NSSegment.Error[noSuchSegment]**, and **NSFile.Error[[space[mediumFull]]]** may be raised.

4.4 Mapping

NSSegment provides a number of procedures to map file segments to Pilot spaces and to transfer data between file segments and spaces.

```
NSSegment.Map: PROCEDURE [
    origin: Origin, access: NSFile.Access ← NSFile.readAccess,
    usage: Space.Usage ← Space.unknownUsage,
    life: Space.Life ← file,
    swapUnits: Space.SwapUnitOption ← Space.defaultSwapUnitOption,
    session: Session ← nullSession]
    RETURNS [mapUnit: Space.Interval];

NSSegment.MapAt: PROCEDURE [
    at: Space.Interval, origin: Origin, access: NSFile.Access ← NSFile.readAccess,
    usage: Space.Usage ← Space.unknownUsage,
    life: Space.Life ← file,
    swapUnits: Space.SwapUnitOption ← Space.defaultSwapUnitOption,
```

```
    session: Session ← nullSession]
    RETURNS [mapUnit: Space.Interval];

NSSegment.CopyIn: PROCEDURE [
    pointer:LONG POINTER, origin: Origin, session: Session ← nullSession]
    RETURNS [countRead: PageCount];

NSSegment.CopyOut: PROCEDURE [
    pointer:LONG POINTER, origin: Origin, session: Session ← nullSession]
    RETURNS [countWritten: PageCount];

NSSegment.MakeWritable: PROCEDURE [
    interval: Space.Interval, file: NSFile.Handle, segment: ID ← defaultID,
    session: Session ← nullSession];
```

These procedures have the same semantics as the corresponding procedures of the Pilot **Space** interface, and raise the same errors, with the following exceptions:

- Instead of a **Space.Window**, the **NSSegment** version of **Map**, **MapAt**, **CopyIn**, and **CopyOut** require an **NSSegment.Origin**, defined as follows:

```
NSSegment.Origin: TYPE = RECORD [
    file: NSFile.Handle, base: PageNumber,
    count: PageCount, segment: ID ← defaultID];
```

- **Map** and **MapAt** take an **NSFile.Access** parameter to determine whether writing is to be permitted in the mapped space. If **access[read]** is **FALSE**, **NSFile.Error[[access [accessRightsInsufficient]]]** is raised.

- **MakeWritable** accepts an **NSFile.Handle** and an **NSSegment.ID** as well as a **Space.Interval**.

GetBase obtains the page number of the base of the window that is mapped to a given space (i.e., the number returned equals **origin.base**, if **origin** was the **NSSegment.Origin** mapped to the given space).

```
NSSegment.GetBase: PROCEDURE [
    pointer:LONG POINTER, session: Session ← nullSession] RETURNS [PageNumber];

NSSegment.PageNumber: TYPE = LONG CARDINAL;
```

Arguments: **pointer** is the long pointer to the space interval of interest; **session** is the client's session handle.

Results: the returned **PageNumber** is the base of the window mapped to the space interval specified by **pointer**.

Errors: **Space.Error[invalidParameters]** may be raised.

4.5 Errors

Any **NSSegment** operation may raise the following error. In addition, some operations may raise other signals, which are documented with each operation.

```
NSSegment.Error: ERROR [type: ErrorType];

NSSegment.ErrorType: TYPE = {
    illegalForDefault, improperByteCount, invalidSegmentID,
    noSuchSegment, segmentAlreadyExists, tooManySegments};
```

The argument **type** describes the problem in greater detail:

illegalForDefault	The default segment cannot be moved or deleted.
improperByteCount	A byte count was specified for a non-default segment that was not a multiple of 512.
invalidSegmentID	The specified segment identifier is not valid.
noSuchSegment	No segment with the specified identifier was found.
segmentAlreadyExists	A specified segment identifier is already in use.
tooManySegments	An attempt was made to exceed the maximum number of segments allowed per file.

5

Positionable stream operations

```
NSFileStream: DEFINITIONS . . . ;
```

The **NSFileStream** interface provides a positionable stream mechanism whereby the content of local files may be randomly accessed and modified. Once an **NSFileStream.Handle** is created for a file, the operations provided by the Pilot **Stream** interface may be used to manipulate the stream. In particular, the operation **Stream.SetPosition** may be used to access data at any byte position in the stream. See the *Pilot Programmer's Manual* [26] for a detailed description of the **Stream** facility.

Note: In Services 8.0, the **NSFileStream** facility may be used to access the content of local files only.

5.1 Creating the file stream

To access the content of a file via a positionable file stream, the client must first obtain a **Handle** for the stream using the **Create** operation.

```
NSFileStream.Handle: TYPE = RECORD [Stream.Handle];

NSFileStream.Create: PROCEDURE [
    file: NSFile.Handle, closeOnDelete: BOOLEAN ← TRUE,
    options: Stream.InputOptions ← Stream.defaultInputOptions,
    session: NSFile.Session]
    RETURNS [fileStream: Handle];
```

Arguments: **file** is a handle for the file for which the stream is to be created. If **closeOnDelete** is **TRUE** then the file is closed automatically when the stream is deleted. **options** give the stream input options as defined by the Pilot **Stream** interface, and **session** is the client's session handle.

Results: A stream handle is created for **file** and returned as **fileStream**. The client must call **Stream.Delete** to release the stream when he is through accessing it.

Access: **Create** requires read access to **file**. If data is to be modified via the stream, the client should also have write access to **file**.

Errors: **NSFile.Error** is raised with type **handle** or **session** if the client file handle or session handle are invalid. **NSFile.Error [access[fileNotLocal]]** is raised if **file** is not a local file.

Once the client has obtained a file stream handle for a file, he may operate on the content of the file using any of the operations defined in the Pilot **Stream** facility for which he has the proper access. Operations which modify the contents of the file require both read and write access to the file. Note that the controls the client specified for the file when opening it remain in effect when accessing the content of the file via the file stream, and are used to determine the client's access to the file.

Any operation performed on the file stream which writes data to the stream may cause **NSFile.Error** to be raised with type **space[mediumFull]** if the operation would cause the amount of free space on the volume to be exceeded, or **space[allocationExceeded]** if the **subtreeSizeLimit** of an ancestor of the file would be exceeded by performing the operation.

Note: The **NSFileStream** mechanism may be used to access only the default segment of a file. Content of segments other than the default may be accessed only by the **NSSegment** facility (see section 4).

When the client is through accessing the file stream, he must call **Stream.Delete** to release the resources allocated to the stream. If the client specified that the file should be closed upon deletion of the file stream, then the file is closed and the corresponding file handle is no longer valid. The client should make no attempt to close the file or otherwise operate on it in any way using **NSFile** operations until the file stream for that file is deleted.

5.2 Getting and setting the length of the stream

The **GetLength** operation allows the client to obtain a count of the data bytes in a file stream. Note that this length is not necessarily equal to the size of the underlying file. (If the client is appending data to a file stream, for example, the actual size of the file may be smaller than the number of data bytes in the file stream until the stream is deleted.) The client may insure equivalence between the data in the file and the data in the stream at any time by calling the operation **Stream.SendNow**.

```
NSFileStream.GetLength: PROCEDURE [
    fileStream: Handle] RETURNS [lengthInBytes: LONG CARDINAL];
```

Arguments: **fileStream** is the stream handle whose length is desired.

Results: **lengthInBytes** is the number of data bytes in **fileStream**.

Access: **GetLength** requires read access to the file underlying **fileStream**.

Errors: None.

NSFileStream.SetLength may be used to set the length of data bytes in the file stream. This length may be set to a value smaller than the current length, thereby truncating the stream (and the underlying file). The file may be extended by setting the length of the file stream to a value larger than the current value. This truncation or extension of the file takes place immediately.

```
NSFileStream.SetLength: PROCEDURE [fileStream: Handle,lengthInBytes: LONG CARDINAL];
```

Arguments: **fileStream** is the file stream handle whose length is to be set, **lengthInBytes** is the number of data bytes to which the length should be set.

Results: The number of data bytes in **fileStream** is set to **lengthInBytes**. The size of the underlying file is also set to **lengthInBytes**.

Access: **SetLength** requires both read and write access to the file underlying **fileStream**.

Errors: **NSFile.Error** may be raised with type **space[mediumFull]** if this would cause the amount of free space on the volume to be exceeded, or **space[allocationExceeded]** if the **subtreeSizeLimit** of an ancestor of the file would be exceeded.

5.3 Miscellaneous operations

The **NSFile.Handle** underlying a file stream may be obtained using the procedure **NSFileStream.FileFromStream**. This handle is the same one used to create the file stream and is valid only in the session used to create the stream.

```
NSFileStream.FileFromStream: PROCEDURE [
    fileStream: Handle] RETURNS [file: NSFile.Handle];
```

Arguments: **fileStream** is the stream handle whose underlying **NSFile.Handle** is to be obtained.

Results: **file** is a copy of the handle for the file from which **fileStream** was created.

Errors: None.

The client may determine if he is positioned past the last byte of the data in the file stream by calling the operation **NSFileStream.EndOf**.

```
NSFileStream.EndOf: PROCEDURE [fileStream: Handle] RETURNS [atEnd: BOOLEAN];
```

Arguments: **fileStream** is the handle for the stream to be checked.

Results: **atEnd** is returned as **TRUE** if the file stream is positioned past the last byte of the data, and **FALSE** if not.

Errors: None.

6

Attributes

An *attribute* is a data item that is associated with a file. Any information associated with a file which is not a part of the file's content is contained in the file's attributes. Attributes may help to identify the file so that it can be distinguished from other files, to describe the structure or behavior of the file, to record information about certain events in the life of the file, or to perform any other desired function.

6.1 Attribute model

Every attribute has an *attribute type* which identifies the attribute. Certain types are defined and supported by the file system. Additional types may be defined by the client. Types to be defined in this way must be allocated from ranges assigned by the manager of Filing. A client or application is the *type owner* of attribute types within an assigned range.

Not every attribute is meaningful for all files. For example, directory-related attributes have no meaning for files that are not directories. Such attributes may not be specified when they are inappropriate, and for non-directory files these attributes always have default values when examined.

The file system imposes a limit on the total amount of attribute data which may be stored in a single file. This limit is 32,768 sixteen-bit words.

6.2 Classes of attributes

Since attributes serve a wide variety of purposes, they exhibit a variety of behaviors. However, certain classifications are helpful in pointing out similarities between attributes.

6.2.1 Interpreted vs. uninterpreted

Many attributes have a particular meaning to the file system, and specifying such an attribute results in a defined behavior. These attributes are said to be *interpreted*. All other attributes are *uninterpreted*, or client-defined (also called *extended* attributes).

Most interpreted attributes are normally maintained by the file system: the value of an interpreted attribute may change even when it has not been specified during a procedure call, as a side-effect of that procedure. Various restrictions are imposed on the use of an interpreted attribute in certain procedures. In general a client cannot always expect an interpreted attribute to remain unchanged during arbitrary procedure calls.

Uninterpreted attributes are defined by the client. An uninterpreted attribute should have an established data type defined by the client, but the file system does not know what this data type is and therefore cannot enforce it. When an uninterpreted attribute is specified during a procedure call, it is stored with the file. The values of uninterpreted attributes do not change except when they are changed explicitly by a client. Uninterpreted attributes may be passed to any procedure that accepts attributes. The value of an uninterpreted attribute is always exactly the value to which it was explicitly set by the client.

6.2.2 Environment vs. data

An *environment* attribute describes the relationship of a file to its environment such as its name or parent directory. A *data* attribute describes aspects of the file that are contained entirely within the file. This distinction is useful because it determines many of the differences in attribute behavior. For example, the **name** and **parentID** of a file are environment attributes, while **sizeInBytes** is a data attribute.

Data attributes are tightly bound to a file. They may be thought of as extensions of the file's content. Data attributes are always carried along when a file is moved, copied, or deserialized. They may not be explicitly changed during procedures which change the file's environment but not the file itself. In addition, data attributes may not be used to identify a file when opening it.

Environment attributes are much more loosely bound to a file. Environment attributes may be thought of as part of the file's parent directory. It is common to want the values of these attributes to change when a file's context changes, as in moving, copying, or deserializing. Some environment attributes may be used to identify a file when opening it. For example, **fileID**, **name**, and **version** are environment attributes. An uninterpreted attribute may be considered a data or an environment attribute depending on the client's use of the attribute.

6.2.3 Primary vs. derived

A *primary* attribute is an attribute that carries information for which the attribute is the only source; **name** and **ordering** are primary attributes.

A *derived* attribute carries information that is derived from other characteristics of the file. For example, **numberOfChildren** records the number of children in a directory, and **sizeInBytes** records the length of a file's default segment.

6.3 Attribute descriptions

This section defines each attribute supported by NSFiling. Each definition provides a description of the meaning and purpose of the attribute, the Mesa definition, significant values of the attribute type, a description of those values, and a statement of where the attribute may legally be specified.

```
NSFile.AttributeType: TYPE = MACHINE DEPENDENT{
    -- protocol-documented
    checksum(0), childrenUniquelyNamed(1), createdBy(2), createdOn(3),
    fileID(4), isDirectory(5), isTemporary(6), modifiedBy(7), modifiedOn(8),
    name(9), numberOfChildren(10), ordering(11), parentID(12), position(13),
    readBy(14), readOn(15), sizeInBytes(16), type(17), version(18),
    -- protocol-undocumented
    accessList(19), defaultAccessList(20), pathname(21),
    -- locally interpreted
    service(22), backedUpOn(23), filedBy(24), filedOn(25), sizeInPages(26),
    subtreeSize(27), subtreeSizeLimit(28),
    -- other
    extended(29)};
```

AttributeType enumerates the attributes supported by the file system.

6.3.1 Identity attributes

Identity attributes serve to identify a file. They are the attributes that would typically be used to specify a file when operating on it.

```
NSFile.Attribute: TYPE = RECORD[..., fileID => [value: ID], ...];

NSFile.ID: TYPE [5];
```

fileID — The **fileID** attribute unambiguously and uniquely identifies a file within a service. It is *not* unique over all space and time, but only within a given service.

```
NSFile.nullID: ID = [nullIDRepresentation];

NSFile.nullIDRepresentation: ARRAY [0..SIZE[ID]) OF UNSPECIFIED = [0, 0, 0, 0, 0];
```

The **fileID** attribute names a file within a service independent of its parent directory. The value for a given file is guaranteed to remain constant as long as the file remains on the same service. The **fileID** of a file cannot be explicitly changed by the client. The distinguished value of this attribute, **nullID**, is never assigned to any file.

```
NSFile.Attribute: TYPE = RECORD[..., service => [value: Service], ...];

NSFile.Service: TYPE = LONG POINTER TO ServiceRecord;
```

```
NSFile.ServiceRecord: TYPE = RECORD [
    name: NSName.NameRecord,
    systemElement: SystemElement ← nullSystemElement];

NSFile.SystemElement: TYPE = System.NetworkAddress;
```

service The **service** attribute records the physical location of a file. This value consists of two parts: the name of the service and the network address for the processor where the file physically resides. Both parts are needed to identify the physical location of the file since more than one file service can be located on a given system element.

```
NSFile.nullService: Service ← LONG [NIL];
NSFile.defaultService: READONLY Service;
```

For convenience, the client can set a **defaultService** to be used in operations where an explicit service is not specified or where the **nullService** is specified. The **defaultService** is set using **NSFile.SetDefaultService** (see §3.1.6).

```
NSFile.localSystemElement: READONLY SystemElement;

NSFile.nullSystemElement: SystemElement = System.nullNetworkAddress;
```

The read-only variable, **localSystemElement** contains the value of the **systemElement** portion of the **service** attribute for all files on local services. The special constant value, **nullSystemElement** may be used in those contexts where the system element address of a service is to be filled in by the file system. When **nullSystemElement** is specified, the file system will look up the name of the service in the Clearinghouse to obtain its system element address.

```
NSFile.String: TYPE = NSString.String;

NSFile.Attribute: TYPE = RECORD [ . . ., name = > [value: String], . . . ];
```

name The **name** attribute is the human-sensible name assigned to the file. This name may be used to specify the file during filing operations, or it may merely be a human-sensible description.

The name of a file is not necessarily unique within its parent. However, the name-version pair is always unique within a parent. No **name** attribute may have zero length. It is also recommended that it not contain any of the reserved characters: , (comma), (,), /, !, *, #, ' (apostrophe). Capitalization is ignored when names are compared.

```
NSFile.Attribute: TYPE = RECORD [ . . ., pathname = > [value: String], . . . ];
```

pathname The **pathname** attribute of a file is the concatenation of the **name** and **version** attributes of each of that file's ancestors, beginning with the root file of the service on which the file resides. The name of each file is separated from the version number by the **NSFileName.versionSeparator**, and name-version pairs in the pathname are separated from each other by the

NSFileName.nameVersionPairSeparator. (See section 7 for full details on pathname syntax.)

The **pathname** attribute of a file may be used during by-name operations (such as NSFile.**OpenByName**) as a specification of the file of interest.

```
NSFile.Attribute: TYPE = RECORD [ . . ., version = > [value: CARDINAL], . . . ];
```

version — The **version** attribute distinguishes files having the same **name** attribute within a directory. The name-version pair is always unique across all children of a directory.

This attribute may be specified by the client whenever a file is added to a directory, for example, during NSFile.**Create**. Ordinarily, however, it is omitted, and the new file is assigned a version number by the file system. If there are files in the specified directory with the same name as the new file, the assigned version number is one greater than the highest version number associated with any of those files. If there are no such files, a version number of one is assigned.

When used to identify a file, if **lowestVersion** or **highestVersion** is specified, the file to be accessed is the one within the directory having the specified name and the lowest or highest version number, respectively.

```
NSFile.highestVersion: CARDINAL = LAST[CARDINAL];

NSFile.lowestVersion: CARDINAL = 0;
```

Because an error is reported when the client attempts to create a file with a non-unique name-version pair, a client may not specify either **lowestVersion** or **highestVersion** when creating a file. Within a filter, **lowestVersion** and **highestVersion** may be specified but only when the order of enumeration (in procedures **Find** or **List**) is by the **name** attribute.

Note: In Services 8.0, the constants **lowestVersion** and **highestVersion** are not allowed within filters.

6.3.2 File attributes

File attributes describe basic characteristics of a file. Generally, they are attributes that govern the interpretation of the file, or that describe the file's relationship with its parent directory.

```
NSFile.Attribute: TYPE = RECORD [ . . ., checksum = > [value: LONG CARDINAL], . . . ];
```

checksum — The **checksum** attribute of a file helps to verify the validity of the content of the file. It is intended to detect file damage that may occur while the file is stored by the file system.

The file system computes a checksum whenever the content of a file is transferred. This occurs in **Store**, **Retrieve**, **Replace**, **Serialize**, and **Deserialize**. When the content is transferred, the computed value is saved in the **checksum** attribute. If the client has

specified a value for the attribute, it is compared to the computed value and an error is reported if there is a mismatch.

A checksum is a ones-complement, add-and-cycle sum computed over the sixteen-bit words comprising a file's content. It is calculated by initializing it to zero and for each successive data word, adding the word to the sum (using ones-complement addition), performing a left cycle of the result. If an odd number of bytes is involved, a last byte of zero is assumed for purposes of the checksum computation. If the result is the ones-complement value minus zero (177777B), it is converted to plus zero (0B) to avoid conflict with the **unknownChecksum** value.

```
NSFile.unknownChecksum: CARDINAL = Checksum.nullChecksum;
```

If the checksum is not known because, for example, it was never computed after a file's content was initialized or changed, the value of the **checksum** attribute is set to **unknownChecksum**. The client may also set this value explicitly via **ChangeAttributes** (see §6.6.1). Any computed value of checksum is always considered to match **unknownChecksum**. It is permissible for the client to set the value of the **checksum** attribute to **unknownChecksum** to avoid checksum validation.

```
NSFile.Attribute: TYPE = RECORD [ . . .,type = > [value: Type], . . . ];
```

```
NSFile.Type: TYPE = LONG CARDINAL;
```

type — The **type** attribute of a file describes the nature of the content or attributes of the file in order to communicate to potential users how the file is to be interpreted.

A client or application may define types for files of his own that he wishes to distinguish. Types to be defined in this manner must be allocated from ranges assigned by the manager of Filing. A number of defined types can be found in the **NSAssignedTypes** interface (see §6.4). Clients are encouraged to use these types to identify files that have the specified characteristics in order to promote information sharing.

The file system interprets neither the type nor the content of a file. In particular, the **type** attribute may not be used to determine whether a file is a directory or a non-directory. This information is determined by the **isDirectory** attribute.

```
NSFile.Attribute: TYPE = RECORD [ . . ., isDirectory = > [value: BOOLEAN], . . . ];
```

isDirectory — The **isDirectory** attribute of a file indicates whether the file is a directory or a non-directory. Certain procedures may not be applied to a file that is a non-directory. Directories cannot be temporary files.

```
NSFile.Attribute: TYPE = RECORD [ . . ., isTemporary = > [value: BOOLEAN], . . . ];
```

isTemporary — The **isTemporary** attribute of a file indicates whether the file is temporary or permanent. A temporary file is a file which is not a directory and which has no parent directory. Such a file is deleted when all file handles to it are closed. A permanent file resides in a

directory and is not deleted until there is an explicit request to do so.

```
NSFile.Attribute: TYPE = RECORD [ . . ., parentID = > [value: ID], . . . ];
```

parentID The **parentID** attribute of a file is equal to the **fileID** attribute of the file's parent. For temporary files and the root file, this attribute always has the value **nullID**.

```
NSFile.Attribute: TYPE = RECORD [ . . ., position = > [value: Position], . . . ];

NSFile.Position: TYPE = Words;
```

position The **position** attribute of a file specifies a file's position within its parent directory. It may be used to indicate starting and ending points for listing and locating files in a directory, or to specify the insertion point when creating a file in a directory that is ordered by position.

A position defines a point within the linear span of a directory at which there is at most one file. A position value remains valid even if the file to which it applies is moved or deleted. The position then refers to the point where the file resided. However, a position value is tied to the ordering of the directory into which it points. It cannot be used after the directory has been reordered (by changing its **ordering** attribute) and it cannot be used to specify a position within any other directory.

The value of a position is uninterpretable by the client. Because the internal structure of positions is private, the client may not compare positions, not even for equality.

Positions exist in several flavors. Each is dependent on the ordering on which it is based and may not be applied to any other ordering. For the purpose of this description assume the *sort order* to mean the ordering defined by a directory's **ordering** attribute.

The operation **NSFile.GetAttributes** returns a position applicable to the sort order of a file's parent (a default ordering or an alternate ordering). **NSFile.List** returns positions corresponding to the ordering specified or implied by its **scope.ordering** argument (which may differ from the sort order of the directory being listed). Only positions applicable to the sort order of a directory are allowed in attribute list arguments to other **NSFile** operations; within **scope.filter** (to **List** or **Find**), only positions derived from the ordering specified or implied by **scope.ordering** are allowed.

```
NSFile.firstPosition, lastPosition: READONLY Position;

NSFile.firstPositionRepresentation: ARRAY [0..0] OF UNSPECIFIED = [0];

NSFile.lastPositionRepresentation: ARRAY [0..0] OF UNSPECIFIED = [177777B];
```

Two special values of position identify distinguished points within a directory. The constant **firstPosition** specifies a point before the first file in the directory and **lastPosition** specifies a point after the last file. The first and last files within a directory are determined by the directory's **ordering** attribute.

6.3.3 Activity attributes

Activity attributes record the date and time of significant events in the life of a file and the name of the user on whose behalf an event occurred. The name of a user is derived or implied by information supplied in establishing a session; times are obtained from Pilot and the system hardware.

```
NSFile.Time: TYPE = System.GreenwichMeanTime;

NSFile.nullTime: Time = System.gmtEpoch;

NSFile.nullString: String = NSString.nullString;
```

The special constants **nullTime** and **nullString** are used to denote that a particular event has not yet occurred.

For performance reasons the file system does not necessarily change these times and names exactly when the related event occurs. Rather, it may cache changes for later application or group several changes together. The file system guarantees that if an event occurs during a session then the times and names will be updated appropriately sometime during that session. The file system also guarantees that explicitly-requested changes to times and names, where allowed, occur immediately.

```
NSFile.Attribute: TYPE = RECORD [ . . ., backedUpOn => [value: Time], . . . ];
```

backedUpOn — The **backedUpOn** attribute of a file records the time at which the file was last backed up.

When a new file is created, the **backedUpOn** attribute is set to **nullTime** and is changed only by explicit action (via **ChangeAttributes**). The file system does not maintain this attribute.

```
NSFile.neverBackup: Time = [LAST[CARDINAL]];
```

The client may set the value of the **backedUpOn** attribute of a file to the constant **NSFile.neverBackup**. This indicates that the file should never be backed up when the backup process is run on a file service.

```
NSFile.Attribute: TYPE = RECORD [ . . ., createdBy => [value: String], . . . ];
```

createdBy — The **createdBy** attribute of a file records the name of the user who created the file's content. It is the name of the user who last modified the content of the file.

If the client does not specify this attribute during **Create**, **Store**, or **Replace**, the file system sets it to the name of the current user. However, since the attribute is intended to be the name of the creator of the *content* of the file (rather than the physical file itself), it is strongly recommended that all clients maintain this name with the file and specify it when transferring the file.

```
NSFile.Attribute: TYPE = RECORD [ . . ., createdOn => [value: Time], . . . ];
```

createdOn The **createdOn** attribute of a file records the time of creation of the file's content. This attribute is used to maintain the generation time of the file in order to determine the relative age of similar files.

If the client does not specify this attribute during **Create**, **Store**, or **Replace**, the file system sets it to the current date and time. However, since the attribute is intended to be the time of creation of the *content* of the file (rather than the physical file itself), it is strongly recommended that all clients maintain this time with the file and specify it when transferring the file.

```
NSFile.Attribute: TYPE = RECORD [..., filedBy => [value: String], ...];
```

filedBy The **filedBy** attribute of a file records the name of the user who inserted the file into its parent directory. For temporary files and the root file, this attribute always has the value **nullString**.

```
NSFile.Attribute: TYPE = RECORD [..., filedOn => [value: Time], ...];
```

filedOn The **filedOn** attribute of a file records the time at which the file was inserted into its parent directory. For temporary files and the root file, this attribute always has the value **nullTime**.

```
NSFile.Attribute: TYPE = RECORD [..., modifiedBy => [value: String], ...];
```

modifiedBy The **modifiedBy** attribute of a file records the name of the last user who changed the file's content or attributes.

When a new file is created, the **modifiedBy** attribute is set to the name of the current user. Subsequently, the file system maintains the attribute.

```
NSFile.Attribute: TYPE = RECORD [..., modifiedOn => [value: Time], ...];
```

modifiedOn The **modifiedOn** attribute of a file records the time at which the file's content or attributes were last changed.

When a new file is created, the **modifiedOn** attribute is set to the current time. Subsequently, the file system maintains the attribute.

```
NSFile.Attribute: TYPE = RECORD [..., readBy => [value: String], ...];
```

readBy The **readBy** attribute of a file records the name of the user who last examined the content of the file.

When a new file is created, the **readBy** attribute is set to **nullString** to indicate that the file has never been read. Subsequently, the file system maintains the attribute.

```
NSFile.Attribute: TYPE = RECORD [..., readOn => [value: Time], ...];
```

readOn The **readOn** attribute records the time at which the content of the file was last examined.

When a new file is created, the **readOn** attribute is set to **nullTime** to indicate that the file has never been read. Subsequently, the file system maintains the attribute.

6.3.4 Size attributes

Size attributes record the logical size of a file.

```
NSFile.Attribute: TYPE = RECORD [..., sizeInBytes => [value: LONG CARDINAL], ...];
```

sizeInBytes The **sizeInBytes** attribute records the number of client-visible bytes in a file. This attribute cannot be explicitly changed by the client but is implicitly modified by operations which change the size of the file.

```
NSFile.Attribute: TYPE = RECORD [..., sizeInPages => [value: LONG CARDINAL], ...];
```

sizeInPages The **sizeInPages** attribute records the number of client-visible pages in a file. This attribute cannot be explicitly changed by the client but is implicitly modified by operations which change the size of the file.

These two attributes are not independent sizes; they are merely two ways of looking at the same attribute. In most cases, there is a straightforward arithmetic relationship between them.

Note: If a file contains segments other than the default segment, **sizeInBytes** is the number of bytes in the default segment, while **sizeInPages** is the total number of pages in all segments. Therefore, the two attributes are not arithmetically related in this case.

6.3.5 Access attributes

Access attributes specify the access restrictions of a file. Only users represented within a file's access list are granted access to the file and then only with the specified permissions.

```
NSFile.Attribute: TYPE = RECORD [..., accessList => [value: AccessList], ...];

NSFile.AccessList: TYPE = MACHINE DEPENDENT RECORD [
    entries(0): AccessEntries ← NIL,
    defaulted(3): BOOLEAN ← FALSE];

NSFile.AccessEntries: TYPE = LONG DESCRIPTOR FOR ARRAY OF AccessEntry;

NSFile.AccessEntry: TYPE = MACHINE DEPENDENT RECORD [
    key(0): String, type(4): AccessEntryType, access(5): Access];

NSFile.AccessEntryType: TYPE = {individual, alias, group, other};

NSFile.Access: TYPE = PACKED ARRAY AccessType OF BooleanFalseDefault;
```

```
NSFile.AccessType: TYPE = MACHINE DEPENDENT {
    -- all files -- read(0), write(1), owner(2),
    -- directories -- add(3), remove(4)};
```

accessList The **accessList** attribute specifies who may access a file and in what ways. The access granted a particular session with respect to the file is the union of the permissions specified in all entries containing a key *representing* the session.

```
NSFile.Attribute: TYPE = RECORD [ . . ., defaultAccessList => [value: AccessList], . . . ];
```

defaultAccessList The **defaultAccessList** attribute applies only to directories and specifies the access for files having explicitly defaulted access lists within the directory. For non-directories, this attribute always has the value **[NIL, TRUE]**.

An access list is comprised of a set of typed key/access permission pairs. If a session's user can be identified with the **key** portion of an entry (classified by the given **type**) then the permissions specified by the entry are granted to the session.

When a file is created it receives defaulted values for both its access lists or those specified by the client, if supplied. When a file is inserted into a directory, the file receives access lists as specified by the client; if an access list or default access list is not specified during the insertion, the respective access list remains unchanged. Access lists of descendants of the inserted file are not affected by the insertion.

The access granted a particular session with respect to a file is the union of the permissions specified in all entries containing a key *representing* the session. If the access list for a file has no entries (empty), no access to the file is allowed to anyone (except those that are privileged to bypass access controls). If the **accessList** attribute of a file is explicitly defaulted, access to the file is determined by the **defaultAccessList** attribute of the file's parent directory.

The file system does not determine which keys of access list entries *represent* a session. The client must provide a control procedure for this purpose; in general, an *individual* key matches if the key is equal to the session's name, an *alias* matches if the session's name is implied by it, a *group* matches if the session's name is a group member, and the meaning of *other* is unspecified. A group entry with a null **key** string signifies *world*, which would normally represent all users, although it need not (again depending on the control procedure provided by the client; see §8.1.2 for further details).

When the access list of a file must be determined, the **accessList** attribute stored directly with the file is retrieved. If this value has been defaulted, then the **defaultAccessList** attribute of the file's parent directory is retrieved. If the **defaultAccessList** attribute of the parent is defaulted, the parent's access list is used. The method of determining the access list of the parent is the same as for the original file; this process proceeds recursively until a non-defaulted list is encountered or the root file of the service is reached. If the access list of the root file must be obtained and none is present, the single-entry list **["", group, fullAccess]** is assumed. Note that in the absence of any access lists on files of a volume, **fullAccess** is normally granted.

read — Granting *read* access allows a client to: examine the contents and attributes of a file; list a directory and examine the attributes of its children (**List**); copy the file; search a directory (during **Find**).

write — Granting *write* access allows a client to: modify the content and data attributes of the file; modify the environment attributes of a directory's children; delete the file (*remove* permission for the parent is also required).

add — This permission applies only to directories. It allows the client to insert new files into the directory.

remove — This permission applies only to directories. It allows the client to remove children from the directory (*write* permission for the child is also required).

owner — Granting *owner* access allows a client to modify the access lists of a file (*write* permission to the file's parent directory also allows this).

The ability to modify a file's access attributes is subject to the access granted the client by the access list previously in effect for the file. Note that *owner* access to a file or *write* access to the file's parent is required to modify a file's access attributes.

Changes to access list values, whether by **ChangeAttributes** or **UnifyAccessLists**, take immediate effect for all file handles within the client's session and all new handles acquired by the client's session or other new sessions. Effects of access list changes caused by one session are *not* guaranteed to affect clients of other existing sessions until those sessions end.

```
NSFile.fullAccess: Access = ALL[TRUE];

NSFile.noAccess: Access = ALL[];

NSFile.readAccess: Access = [read: TRUE];
```

The access permission constants **fullAccess**, **noAccess**, and **readAccess** are provided for the convenience of the client in defining access list entries.

6.3.6 Directory attributes

Directory attributes apply only to directory files. They describe useful characteristics of a directory. In non-directories, directory attributes always have default values.

```
NSFile.Attribute: TYPE = RECORD [ . . ., childrenUniquelyNamed = > [value: BOOLEAN], . . .];
```

childrenUniquelyNamed — The **childrenUniquelyNamed** attribute specifies whether the children of a directory are constrained to have distinct **name** attributes.

When this attribute is **TRUE**, no two children of the directory may have the same **name** attribute, and the file system rejects any attempt to add a file with the same **name** attribute as an existing file within the directory. When this attribute is **FALSE**, this

restriction is not enforced. In this case, files having the same **name** attribute are distinguished by their **version** attributes. Comparison of **name** attributes is described in §3.5.1.

The **childrenUniquelyNamed** attribute of a directory may be changed from TRUE to FALSE at any time. The value of **childrenUniquelyNamed** may be changed from FALSE to TRUE as long as no two children of the directory have the same **name** attribute; otherwise, NSFile.Error[[attributeValue[unreasonable, childrenUniquelyNamed]]] is raised.

```
NSFile.Attribute: TYPE = RECORD [ . . ., numberOfChildren => [value: CARDINAL], . . . ];
```

numberOfChildren The **numberOfChildren** attribute maintains a count of the children in a directory. Note that it is not a count of the directory's descendants.

```
NSFile.Attribute: TYPE = RECORD [ . . ., ordering => [value: Ordering], . . . ];

NSFile.Ordering: TYPE = MACHINE DEPENDENT RECORD [
    var(0): SELECT type(0): OrderingType FROM
        key => [
            key(1): AttributeType,
            ascending(3): BOOLEAN ← TRUE,
            dummy1(2): CARDINAL ← 0 -- padding --
            dummy2(4): CARDINAL ← 0],
        extended => [
            key(1): ExtendedAttributeType,
            ascending(3): BOOLEAN ← TRUE,
            interpretation(4): Interpretation ← none],
    ENDCASE];

NSFile.OrderingType: TYPE = MACHINE DEPENDENT {key(0), extended(1)};

NSFile.Interpretation: TYPE = MACHINE DEPENDENT {
    none(0), boolean(1), cardinal(2), longCardinal(3), integer(4),
    longInteger(5), string(6), time(7)};
```

ordering The **ordering** attribute specifies the order of enumeration of files in a directory during filing operations.

Except when ordering by position (described below), the placement of files in a directory is determined by the relative values of a particular attribute. The **key** component of an ordering specifies which attribute is to be the basis of the ordering; **ascending** determines whether ordering is to be in ascending order of the attribute, and **interpretation** (in the case of **extended** orderings) specifies how the file system should interpret the attribute for purposes of comparison.

For **extended** orderings, if a file's attribute value is not a valid representation of the type specified by **interpretation**, then the file is placed *before* those files that have valid values. The comparison rules for various interpretations are described in §3.5.1.

[**Note:** In Services 8.0, orderings based on extended attributes are not supported.]

The behavior of a directory is somewhat different when the specified key is the **position** attribute. In all other cases, the relative placement of files is determined entirely by the value of the specified attribute. When ordering is by position, however, the relative placement of files is explicitly determined by the client. When adding a file to a directory with a position ordering, the client specifies the position at which the file should reside.

```
NSFile.ascendingPositionOrdering: key Ordering = [
    key[key: position, ascending: TRUE]];

NSFile.descendingPositionOrdering: key Ordering = [
    key[key: position, ascending:FALSE]];
```

The two constants, **ascendingPositionOrdering** and **descendingPositionOrdering**, are used to specify an ordering by position. If ordering is by ascending position, a file that is added without specifying its position is placed at the *end* of the directory. If ordering is by descending position, a file that is added without specifying its position is placed at the *beginning* of the directory. Otherwise, there is no difference between these values.

When the **ordering** attribute of a directory is changed to an ordering by position, the relative placement of files in the directory *is not affected.* In other words, when changing to an ordering by position, the files are initially placed according to their placement in the previous ordering. Subsequent additions need not conform to the previous ordering.

After a number of additions at the same point within a directory ordered by position, the density of files may become too great to allow further additions. When this condition occurs, the operation attempting to insert a file raises the error, **[insertion[positionUnavailable]]**. The client should call **ChangeAttributes** specifying an ordering that is the same as the current ordering. This action redistributes the files without changing their relative placement. The current implementation allows for several hundred insertions at the same point of a directory ordered by position before this condition occurs.

```
NSFile.defaultOrdering: key Ordering = [key[key: name, ascending: TRUE]];

NSFile.nullOrdering: extended Ordering = [extended[key: 0]];
```

If the **ordering** attribute is not specified during the creation of a directory, **defaultOrdering** is used. When the **ordering** attribute has this value, or the corresponding value with **ascending** equal to **FALSE**, ordering of the directory is actually based on ascending or descending values of first, the **name** attribute, and second, the **version** attribute, rather than just the **name** attribute alone. The **nullOrdering** constant is used during filtering (see §3.5.1).

```
NSFile.Attribute: TYPE = RECORD [ . . ., subtreeSize = > [value: LONG CARDINAL], . . . ];
```

subtreeSize The **subtreeSize** attribute records the number of client-visible pages allocated to a file and all files it directly or indirectly contains. This total does *not* include internal data structures such as attributes and directory structures.

Note: For non-directory files, the **subtreeSize** attribute is equivalent to the **sizeInPages** attribute.

```
NSFile.Attribute: TYPE = RECORD [ . . ., subtreeSizeLimit = > [value: LONG CARDINAL], . . . ];
```

subtreeSizeLimit — The **subtreeSizeLimit** attribute records the maximum number of client-visible pages which may be allocated to a directory and all files it directly or indirectly contains.

An operation is rejected if it would cause the value of a directory's **subtreeSize** attribute to exceed the limit given by that directory's **subtreeSizeLimit** attribute. The client is permitted to change the value of this attribute so that it is smaller than the current value of the directory's **subtreeSize** attribute.

```
NSFile.nullSubtreeSizeLimit: LONG CARDINAL = LAST[LONG CARDINAL];
```

When a directory is created and no **subtreeSizeLimit** is specified, **nullSubtreeSizeLimit** is assumed. Use of this constant implies that a directory has no cumulative limit on the number of client-visible pages which may be allocated to it and its descendants.

6.3.7 Extended attributes

The extended attribute mechanism allows the client to define, store and retrieve attribute values not directly supported by the interpreted attribute model. Values of extended attributes supplied by the client are associated with a file and are returned upon request. The file system never changes the value of an extended attribute except by explicit request.

```
NSFile.Attribute: TYPE = RECORD [
    . . ., extended = > [type: ExtendedAttributeType, value: Words], . . . ];

NSFile.ExtendedAttributeType: TYPE = LONG CARDINAL;
```

extended — An **extended** attribute type and a value to be associated with the type are defined by the client. The value of an **extended** attribute is always exactly the value to which it was explicitly set by the client.

When an **extended** attribute is specified during a procedure call, it is stored with the file but causes no other action. In particular, other attributes are unaffected except those that indicate file activity (**modifiedBy**, **modifiedOn**) or position within a parent (**position**). The values of **extended** attributes do not change except when they are changed explicitly by a client. **Extended** attributes may be passed to any procedure that expects an attribute list.

The file system imposes a limit of 32,595 words on the total amount of extended attribute data which may be stored with a single file. There is no limit imposed on the number of extended attributes stored with a single file.

6.4 Assigned types

```
NSAssignedTypes: DEFINITIONS = . . . ;
```

This section describes the type ranges and defined types found in the **NSAssignedTypes** interface. An **AssignedType** is a 32-bit numeric quantity used to identify the type of a file or file attribute.

```
NSAssignedTypes.AssignedType: TYPE = LONG CARDINAL;
```

Clients are encouraged to make use of types defined within this interface when possible to promote information sharing.

6.4.1 Type ranges

The **NSAssignedTypes** interface defines ranges of type values for use by particular applications and clients. Each range designates a set of attribute and file types assigned to a given application (numerically these coincide). A client or application is the *type owner* of all types within a range assigned to it. Normally an application will not make use of types within any range not assigned to it unless by agreement with the type owner. Ranges are assigned by the manager of Filing.

```
-- StandardTypes: TYPE = AssignedType [0..4096);
NSAssignedTypes.firstStandardType: AssignedType = 0;
NSAssignedTypes.lastStandardType: AssignedType = 4095;

-- ServicesATypes: TYPE = AssignedType [4096..4352);
NSAssignedTypes.firstServicesAType: AssignedType = 4096;
NSAssignedTypes.lastServicesAType: AssignedType = 4351;

-- StarTypes: TYPE = AssignedType [4352..4608);
NSAssignedTypes.firstStarType: AssignedType = 4352;
NSAssignedTypes.lastStarType: AssignedType = 4607;

-- ServicesBTypes: TYPE = AssignedType [4608..5120);
NSAssignedTypes.firstServicesBType: AssignedType = 4608;
NSAssignedTypes.lastServicesBType: AssignedType = 5119;

-- WS860Types: TYPE = AssignedType [5120..5136);
NSAssignedTypes.firstWS860Type: AssignedType = 5120;
NSAssignedTypes.lastWS860Type: AssignedType = 5135;

-- WSFujiTypes: TYPE = AssignedType [5136..5152);
NSAssignedTypes.firstWSFujiType: AssignedType = 5136;
NSAssignedTypes.lastWSFujiType: AssignedType = 5151;
```

Note: Because the current version of Mesa does not support subranges of **LONG CARDINAL**, these ranges are given in terms of two defined constants: the first (or lowest) value in the assigned range and the last (or highest) value in the assigned range.

6.4.2 Defined types

A number of constant type definitions are defined with **NSAssignedTypes** for the convenience of the client. Each constant is used to identify either the type of a file or the type of a file attribute.

```
NSAssignedTypes.AttributeType: TYPE = NSFile.ExtendedAttributeType;

NSAssignedTypes.FileType: TYPE = NSFile.Type;
```

The following constant definitions coincide with those defined by the *Filing Protocol* [13] for file attributes (and those of **NSFile**). They are included to allow the client to create and decode the serialized file format (see §3.8.2). Most clients will wish to use the Mesa enumerated type, **NSFile.AttributeType**, when interacting with the file system.

```
-- Protocol-documented
NSAssignedTypes.checksum: AttributeType = 0;
NSAssignedTypes.childrenUniquelyNamed: AttributeType = 1;
NSAssignedTypes.createdBy: AttributeType = 2;
NSAssignedTypes.createdOn: AttributeType = 3;
NSAssignedTypes.fileID: AttributeType = 4;
NSAssignedTypes.isDirectory: AttributeType = 5;
NSAssignedTypes.isTemporary: AttributeType = 6;
NSAssignedTypes.modifiedBy: AttributeType = 7;
NSAssignedTypes.modifiedOn: AttributeType = 8;
NSAssignedTypes.name: AttributeType = 9;
NSAssignedTypes.numberOfChildren: AttributeType = 10;
NSAssignedTypes.ordering: AttributeType = 11;
NSAssignedTypes.parentID: AttributeType = 12;
NSAssignedTypes.position: AttributeType = 13;
NSAssignedTypes.readBy: AttributeType = 14;
NSAssignedTypes.readOn: AttributeType = 15;
NSAssignedTypes.sizeInBytes: AttributeType = 16;
NSAssignedTypes.type: AttributeType = 17;
NSAssignedTypes.version: AttributeType = 18;
```

Protocol-undocumented attributes are those attributes intended for eventual inclusion in the filing protocol standard. They are not included in the definition of the filing standard in its present form.

```
-- Protocol-undocumented
NSAssignedTypes.accessList: AttributeType = 19;
NSAssignedTypes.defaultAccessList: AttributeType = 20;
NSAssignedTypes.pathname: AttributeType = 21;
```

Locally-interpreted attributes are those attributes defined for the convenience of clients running on the same system element as the file system. There is no intent to ever include these attributes in the filing protocol standard. As such, these attributes are not available for remote files.

```
-- Locally Interpreted

NSAssignedTypes.service: AttributeType = 22;
NSAssignedTypes.backedUpOn: AttributeType = 23;
NSAssignedTypes.filedBy: AttributeType = 24;
NSAssignedTypes.filedOn: AttributeType = 25;
NSAssignedTypes.sizeInPages: AttributeType = 26;
NSAssignedTypes.subtreeSize: AttributeType = 27;
NSAssignedTypes.subtreeSizeLimit: AttributeType = 28;

-- Standard File Types
NSAssignedTypes.tUnspecified: FileType = firstStandardType;
```

```
NSAssignedTypes.tDirectory: FileType = firstStandardType + 1;
NSAssignedTypes.tText: FileType = firstStandardType + 2;
NSAssignedTypes.tSerialized: FileType = firstStandardType + 3;
NSAssignedTypes.tEmpty: FileType = firstStandardType + 4;
```

The *Filing Protocol* [13] defines a number of commonly-used file types. Clients are encouraged to use these types to identify files that have the specified characteristics in order to promote information sharing. However, *the file system does not enforce the specified semantics.*

Files that have a format private to a single client or for which the format is unknown or uninteresting are conventionally given type **tUnspecified**; files that are directories with no additional semantics (and no content) are conventionally given type **tDirectory**; files that are non-directories containing text conforming to the *Character Encoding Standard* (except that no length information is in the content) are conventionally given type **tText**; files that are non-directories containing the serialization of a file are conventionally given type **tSerialized**; and non-directory files that have no content (only attribute information) are conventionally given type **tEmpty**.

6.5 Retrieving attribute values

GetAttributes returns attributes of a specified file. The file system obtains the requested attributes and returns them to the client. Since different attributes may be obtained with varying degrees of difficulty, the client should request only those attributes that are needed.

```
NSFile.GetAttributes: PROCEDURE [
    file: Handle, selections: Selections, attributes: Attributes,
    session: Session ← nullSession];
```

Arguments: The file whose attributes are desired is given by **file**; **selections** specifies the attributes desired; **attributes** refers to client storage to which attribute values are copied; **session** is the client's session handle.

Results: The attributes record implied by **attributes** is filled with the requested attribute values. Storage for variable-length and extended attributes within this structure is allocated from **Heap.systemZone** and must be freed by invoking **NSFile.ClearAttributes** (see §6.7.1) or **FreeString**, **FreeWords**, etc.

Access: Read access is required to **file** or **file**'s parent directory.

Errors: **NSFile.Error** is raised with the following error types: **access**, **attributeType**, **authentication**, **handle**, **session**, and **undefined**; **Courier.Error** may also be raised.

Conceptually, every file has a value for every attribute. If an attribute is extended and has never been set, then its value is **NIL**. By convention this is taken to mean the attribute is not set and the attribute is said to be null. An extended attribute can be explicitly put in this state by specifying a value of **NIL**.

```
NSFile.Selections: TYPE = RECORD [
    interpreted: InterpretedSelections ← noInterpretedSelections,
    extended: ExtendedSelections ← noExtendedSelections];

NSFile.ExtendedSelections: TYPE = LONG DESCRIPTOR FOR ARRAY OF ExtendedAttributeType;

NSFile.InterpretedSelections: TYPE = PACKED ARRAY AttributeType OF
    BooleanFalseDefault;

NSFile.allSelections: READONLY Selections;

NSFile.allExtendedSelections: READONLY ExtendedSelections;

NSFile.allExtendedSelectionsRepresentation: . . . ;

NSFile.allInterpretedSelections: InterpretedSelections = ALL[TRUE];

NSFile.noExtendedSelections: ExtendedSelections = NIL;

NSFile.noInterpretedSelections: InterpretedSelections = ALL[FALSE];

NSFile.noSelections: READONLY Selections;
```

An **NSFile.Selections** is used to specify the attributes of interest during **GetAttributes**. Requests for interpreted attributes and extended attributes are given by **selections.interpreted** and **selections.extended**, respectively. It is an error to specify an interpreted attribute type as an extended type within a **Selections**. Specifying **NSFile.allSelections** for **selections** requests that all interpreted attributes and all non-**NIL** extended attributes be returned. Extended attributes that are null are not returned in this case. When specifying an explicit list of extended attribute types in **selections**, the order of the extended attribute values within the result (**attributes.extended**) corresponds to the order of the extended types within **selections.extended**. Extended attributes that are null are returned in this case.

Two convenience operations, **GetReference** and **GetType** are provided to retrieve frequently-used attributes. **GetReference** returns a **Reference** containing the two attribute values of a file required to uniquely identify the file (**fileID** and **service**). **GetType** returns the type of a specified file.

Note: The storage allocated to the **ServiceRecord** referenced by the **service** field of the reference returned by **GetReference** is the property of the file system and should not be deallocated by the client.

```
NSFile.GetReference: PROCEDURE [
    file: Handle, session: Session ← nullSession]
    RETURNS [reference:Reference];

NSFile.Reference: TYPE = RECORD [
    fileID: ID, service: Service];
```

Arguments: The file for which a **Reference** is desired is given by **file**; **session** is the client's session handle.

Results: **reference** is returned, containing the required attribute values.

Access: Read access is required to **file** or **file**'s parent directory.

Errors: **NSFile.Error** is raised with the following error types: **access**, **authentication**, **handle**, **session**, and **undefined**; **Courier.Error** may also be raised.

```
NSFile.GetType: PROCEDURE [file: Handle, session: Session ← nullSession] RETURNS [Type];
```

Arguments: The file whose **type** attribute is desired is given by **file**; **session** is the client's session handle.

Results: The value of **file**'s **type** attribute is returned.

Access: Read access is required to **file** or **file**'s parent directory.

Errors: **NSFile.Error** is raised with the following error types: **access**, **authentication**, **handle**, **session**, and **undefined**; **Courier.Error** may also be raised.

6.6 Modifying attribute values

Attributes may be modified implicitly by many procedures. They may also be modified by explicit client action.

6.6.1 ChangeAttributes

ChangeAttributes modifies attributes of a specified file. The changes may have other effects on the file depending on the attribute.

```
NSFile.ChangeAttributes: PROCEDURE [
    file: Handle, attributes: AttributeList, session: Session ← nullSession];
```

Arguments: The file whose attributes are to be changed is given by **file**; **attributes** specifies the attributes to be changed and their new values; **session** is the client's session handle.

Results: The attributes supplied by the client are used to update the attributes of **file**.

Access: Write access is required for **file** if only data attributes are changed; write access to **file**'s parent is required for environment attribute changes. If access list attributes are changed, write access to **file**'s parent directory or owner access to **file** is required as well.

Errors: **NSFile.Error** is raised with the following error types: **access**, **authentication**, **handle**, **insertion**, **session**, and **undefined**; **Courier.Error** may also be raised.

Not all interpreted attributes may be modified by this operation. Some of these attributes are maintained by the file system and cannot be changed by the client. Those which can be

modified are given in §6.8. The client is free to specify any extended attribute for update in this operation.

6.6.2 UnifyAccessLists

Access attributes (**accessList** and **defaultAccessList**) may be modified for a given file or directory using **ChangeAttributes**, but it is sometimes necessary to *unify* the effective access lists of an entire subtree of files (i.e., modify the access lists so that all the files in the subtree have the same effective access controls). **UnifyAccessLists** is used for this purpose.

```
NSFile.UnifyAccessLists: PROCEDURE [
    directory: Handle, session: Session ← nullSession];
```

Arguments: The subtree of files whose access lists are to be unified is given by **directory**; **session** is the client's session handle.

Results: The **accessList** and **defaultAccessList** attributes of each descendant file within the subtree rooted by **directory** are given defaulted values. The cumulative effect is that all files within the subtree obtain the same effective access controls as those in place for **directory**.

Access: Write access is required for **directory**.

Errors: **NSFile.Error** is raised with the following error types: **access**, **authentication**, **handle**, **session**, and **undefined**; **Courier.Error** may also be raised.

Changes to a file's access list attributes, whether by **ChangeAttributes** or **UnifyAccessLists**, take immediate effect for all handles to the file within the client's session and all new handles acquired by the client's session or other sessions. Access list changes within one session are *not* guaranteed to affect clients of other existing sessions until those sessions end.

6.7 Manipulating attribute values

The file system defines the structure and semantics of most file attributes. It also defines the data structures, lists and records which are used to input or receive the attribute values of a file. To assist the client in dealing with attribute values and these structures, a number of operations are provided.

```
NSFile.AttributeList: TYPE = LONG DESCRIPTOR FOR ARRAY OF Attribute;

NSFile.Attribute: TYPE = MACHINE DEPENDENT RECORD [
    var(0): SELECT type(0): AttributeType FROM
        fileID, parentID = > [value(1): ID],
        checksum = > [value(1): CARDINAL],
        type = > [value(1): Type],
        position = > [value(1): Position],
        service = > [value(1): Service],
        ordering = > [value(1): Ordering],
        accessList, defaultAccessList = > [value(1): AccessList],
```

```
        backedUpOn, createdOn, filedOn, modifiedOn, readOn => [value(1): Time],
        createdBy, filedBy, modifiedBy, name, pathname, readBy => [value(1): String],

        childrenUniquelyNamed, isDirectory, isTemporary => [value(1): BOOLEAN],
        version, numberOfChildren => [value(1): CARDINAL],
        sizeInBytes, sizeInPages, subtreeSize, subtreeSizeLimit => [
            value(1): LONG CARDINAL],

        extended => [type(1): ExtendedAttributeType, value(3): Words],
        ENDCASE];

NSFile.nullAttributeList: AttributeList = NIL;
```

Many filing operations allow the client to specify combinations of attribute values to be used during the operation. For example, during **Create**, the client may wish to specify the **name** and **sizeInPages** attributes of the file to be created. Such attributes are always supplied to the file system in the form of an **AttributeList**. It is an error to specify the same attribute twice within an attribute list. Not all attribute combinations are allowed within an attribute list; the set of legal combinations depends on the context. A value of **nullAttributeList** may be specified to indicate that the file system should assign default values for all attributes.

```
NSFile.AttributesRecord: TYPE = RECORD [
    -- identity attributes
    fileID: ID,
    service: Service,
    name, pathname: String,
    version: CARDINAL,

    -- file attributes
    checksum: CARDINAL,
    type: Type,
    isDirectory, isTemporary: BOOLEAN,
    parentID: ID,
    position: Position,

    -- activity attributes
    backedUpOn, createdOn, filedOn, modifiedOn, readOn: Time,
    createdBy, filedBy, modifiedBy, readBy: String,

    -- size attributes
    sizeInBytes, sizeInPages: LONG CARDINAL,

    -- access attributes
    accessList, defaultAccessList: AccessList,

    -- directory attributes
    ordering: Ordering,
    childrenUniquelyNamed: BOOLEAN,
    subtreeSizeLimit, subtreeSize: LONG CARDINAL,
    numberOfChildren: CARDINAL,
```

```
    -- extended attributes
    extended: ExtendedAttributeList];

NSFile.Attributes: TYPE = LONG POINTER TO AttributesRecord;
```

When attribute values are retrieved from the file system, an **AttributeList** can be a cumbersome data structure to work with. For this case, an **AttributesRecord** is used. The client normally supplies the storage occupied by the **AttributesRecord** itself while the file system allocates additional storage for values within the record as necessary (**position** or **name** attributes, for example). It is the client's responsibility to ensure that additional storage allocated in this way is freed properly using **ClearAttributes** (see below).

```
NSFile.ExtendedAttributeList: TYPE = LONG DESCRIPTOR FOR ARRAY OF extended Attribute;
```

Extended attributes are returned to the client via an **ExtendedAttributeList** structure. This data structure is similar to an **AttributeList** except that all entries must be extended attribute values (note that an **AttributeList** may contain extended attribute values as well).

6.7.1 Copying/freeing

Attribute values, lists and records may be copied, manipulated, and subsequently freed by the client. All storage is allocated from the system heap by these operations. It is the client's responsibility to invoke the corresponding free operation after making a copy of an attribute data structure.

```
NSFile.CopyAccessList: PROCEDURE [list: AccessList] RETURNS [AccessList];

NSFile.FreeAccessList: PROCEDURE [list: AccessList];
```

Access list attributes are copied and freed by the operations, **CopyAccessList** and **FreeAccessList**, respectively.

```
NSFile.CopyWords: PROCEDURE [words: Words] RETURNS [Words];

NSFile.FreeWords: PROCEDURE [words: Words];

NSFile.Words: TYPE = LONG DESCRIPTOR FOR ARRAY OF UNSPECIFIED;
```

Variable-length attribute values (such as the **position** attribute) and extended attribute values are copied and freed using **CopyWords** and **FreeWords**.

```
NSFile.CopyAttributes: PROCEDURE [attributes: Attributes] RETURNS [Attributes];

NSFile.FreeAttributes: PROCEDURE [attributes: Attributes];

NSFile.ClearAttributes: PROCEDURE [attributes: Attributes];
```

An **AttributesRecord** and embedded values it contains are copied using **CopyAttributes** (note that this operation allocates a new record). Such a copy of an **AttributesRecord** is freed using **FreeAttributes** (the embedded attribute values *and the record itself* are freed).

ClearAttributes is used to allow the file system to free storage allocated to embedded values within an **AttributesRecord** (such as during **GetAttributes**). The **AttributesRecord** itself is *not* freed by **ClearAttributes**.

```
NSFile.CopyExtendedAttributes: PROCEDURE [extendedAttributes:
    ExtendedAttributeList] RETURNS [ExtendedAttributeList];

NSFile.FreeExtendedAttributes: PROCEDURE [extendedAttributes:
    ExtendedAttributeList];
```

Lists of extended attribute values are copied and freed using **CopyExtendedAttributes** and **FreeExtendedAttributes**, respectively.

```
NSFile.ClearAttributeList: PROCEDURE [attributeList: AttributeList];
```

In constructing an attribute list, the client may wish to include values for extended attributes such as those returned by encoding operations (see below). In this case, **ClearAttributeList** should be applied to the resulting list after the client is done with it to free the encoded values within the list. Alternatively, the client may apply **FreeWords** to each of the embedded values.

```
NSFile.FreeAttributeList: PROCEDURE [list:AttributeList];

NSFile.MergeAttributeLists: PROCEDURE [
    listA, listB: AttributeList, suppressDuplicates: BOOLEAN ← FALSE]
    RETURNS [mergedList: AttributeList];
```

Two attribute lists are combined into a single list by the operation **MergeAttributeLists**. The operation returns a list containing all attributes within **listA** and **listB**. If **suppressDuplicates** is **TRUE**, duplicates within **listB** are ignored; if **suppressDuplicates** is **FALSE**, duplicate attributes within **listB** cause **NSFile.Error [[attributeType[duplicated, ...]]** to be raised. Attribute values within **listA** and **listB** are not validated, nor are detached attribute values (words and strings) copied. The result of **MergeAttributeLists** must be freed by the client by calling **FreeAttributeList**; attached data structures are not freed by the operation.

6.7.2 Encoding/decoding

For extended attributes, the semantics of the stored data are undefined; the client controls the interpretation of the data. Although the meaning of the data is not known by the file system, operations are provided to interpret an extended attribute value as a conventional Mesa data type and to generate an extended attribute value from the representation of a conventional Mesa data type.

```
NSFile.EncodeBoolean: PROCEDURE [b: BOOLEAN] RETURNS [Words];

NSFile.EncodeCardinal: PROCEDURE [c: CARDINAL] RETURNS [Words];

NSFile.EncodeLongCardinal: PROCEDURE [lc: LONG CARDINAL] RETURNS [Words];

NSFile.EncodeInteger: PROCEDURE [i: INTEGER] RETURNS[Words];
```

```
NSFile.EncodeLongInteger: PROCEDURE [li: LONG INTEGER] RETURNS [Words];

NSFile.EncodeString: PROCEDURE [s: String] RETURNS [Words];

NSFile.EncodeReference: PROCEDURE [r: Reference] RETURNS [Words];
```

Each of these operations accepts a typed value as its argument and returns a value of words such that applying the corresponding decoding operation will result in the original value. The result of an encoding operation is normally used to construct an extended attribute value. Storage is allocated from the system heap so the client must use **FreeWords** or **ClearAttributeList** (see above) to free the encoded value.

```
NSFile.DecodeBoolean: PROCEDURE [Words] RETURNS [b: BOOLEAN];

NSFile.DecodeCardinal: PROCEDURE [Words] RETURNS [c: CARDINAL];

NSFile.DecodeLongCardinal: PROCEDURE [Words] RETURNS [lc: LONG CARDINAL];

NSFile.DecodeInteger: PROCEDURE [Words] RETURNS [i: INTEGER];

NSFile.DecodeLongInteger: PROCEDURE [Words] RETURNS [li: LONG INTEGER];

NSFile.DecodeString: PROCEDURE [Words] RETURNS [s: String];

NSFile.DecodeReference: PROCEDURE [Words] RETURNS [r: Reference];
```

Each of the above decoding operations interprets a supplied set of words as the requested data type and returns this as a result. **Courier.Error[parameterInconsistency]** is reported instead if the set of words cannot be interpreted as the given Mesa data type. The result of calling **DecodeString** depends on the continued existence of its **Words** argument.

6.8 Summary of attribute behaviors

Tables on the following pages summarize the behavior of attributes during **NSFile**, **NSFileStream**, and **NSSegment** operations. In all tables, a file's **position** attribute may be changed if the parent is sorted by an activity attribute and that attribute is changed (**readOn** or **modifiedBy**, for example).

Some operations have no effect on a file's attributes. In **NSFile** these include: **ChangeControls**, **Close**, **Find**, **GetAttributes**, **GetControls**, **List**, **Logoff**, **Logon**, **LogonDirect**, **Open**, and **Probe**. In **NSSegment**, **FindUnused**, **GetBase**, **GetNext**, **GetSizeInBytes**, **GetSizeInPages**, and **NumberOfSegments** have no effect on a file's attributes. In **NSFileStream**, **GetLength**, **EndOf**, and **FileFromStream** have no effect on a file's attributes. The activity attributes of a file whose content is accessed via **Stream** operations executed on file stream for the file will be updated in a manner similar to the **NSSegment** operations **CopyIn** (reading data) and **CopyOut** (writing data).

The headings of each table column are interpreted as follows: "In List" specifies the behavior of a given attribute if included in an attribute list argument to the operation; "Behavior" within the same column heading specifies the behavior of the attribute if omitted from the attribute list argument; "Behavior" in other contexts designates the effect of the operation on the attribute.

Summary of Attribute Behaviors

Table 6.1

Attribute	File Behavior
accessList	NC
BackedUpOn	NC
checksum	(1)
childrenUniquelyNamed	NC
createdBy	(2)
createdOn	(3)
defaultAccessList	NC
extended	NC
filedBy	NC
filedOn	NC
fileID	NC
isDirectory	NC
isTemporary	NC
modifiedBy	(2)
modifiedOn	(3)
name	NC
numberOfChildren	NC
ordering	NC
parentID	NC
pathname	NC
position	NC
readBy	NC
readOn	NC
service	NC
sizeInBytes	(4)
sizeInPages	(5)
subtreeSize	(6)
subtreeSizeLimit	NC
type	NC
version	NC

NC = No Change

Add, Delete, SetSizeInBytes, SetSizeInPages (NSSegment)

Notes:
(1) Set to **NSFile.unknownChecksum.**
(2) Session's user.
(3) Current time.
(4) Set (only for default segment).
(5) Set to appropriate value for current content.
(6) Set to new total content in subtree.

Summary of Attribute Behaviors

Table 6.2

Attribute	In List	Behavior	Descendants Behavior
accessList	set	NC	NC
backedUpOn	set	NC	NC
checksum	set	NC	NC
childrenUniquelyNamed	set	NC	NC
createdBy	set	NC	NC
createdOn	set	NC	NC
defaultAccessList	set	NC	NC
extended	set	NC	NC
filedBy	illegal	NC	NC
filedOn	illegal	NC	NC
fileID	illegal	NC	NC
isDirectory	illegal	NC	NC
isTemporary	illegal	NC	NC
modifiedBy	illegal	(1)	NC
modifiedOn	illegal	(2)	NC
name	set	NC	NC
numberOfChildren	illegal	NC	NC
ordering	set	NC	NC
parentID	illegal	NC	NC
pathname	illegal	(3)	(4)
position	(5)	(6)	(7)
readBy	illegal	NC	NC
readOn	illegal	NC	NC
service	illegal	NC	NC
sizeInBytes	illegal	NC	NC
sizeInPages	illegal	NC	NC
subtreeSize	illegal	NC	NC
subtreeSizeLimit	set	NC	NC
type	set	NC	NC
version	set	NC	NC

NC = No Change

ChangeAttributes

Notes:
(1) Session's user.
(2) Current time.
(3) Appropriate to **name** of file and ancestors.
(4) Changes if **name** or **version** changes.
(5) Specified point.
(6) Changes if key of parent's ordering is changed.
(7) Changes if **ordering** attribute is changed.

Summary of Attribute Behaviors

Table 6.3

Attribute	Source File Behavior	Source Descendants Behavior	Destination Parent Behavior	Resulting File In List	Resulting File Behavior	Resulting Descendants Behavior
accessList	NC	NC	NC	set	NC	NC
backedUpOn	NC	NC	NC	set	NC	NC
checksum	NC	NC	NC	illegal	NC	NC
childrenUniquelyNamed	NC	NC	NC	illegal	NC	NC
createdBy	NC	NC	NC	illegal	NC	NC
createdOn	NC	NC	NC	illegal	NC	NC
defaultAccessList	NC	NC	NC	set	NC	NC
extended	NC	NC	NC	set	NC	NC
filedBy	NC	NC	NC	illegal	(1)	(2)
filedOn	NC	NC	NC	illegal	(3)	(4)
fileID	NC	NC	NC	illegal	(5)	(5)
isDirectory	NC	NC	NC	illegal	NC	NC
isTemporary	NC	NC	NC	set (6)	(7)	(8)
modifiedBy	NC	NC	(2)	illegal	(2)	(2)
modifiedOn	NC	NC	(4)	illegal	(4)	(4)
name	NC	NC	NC	set	NC	NC
numberOfChildren	NC	NC	incremented	illegal	NC	NC
ordering	NC	NC	NC	illegal	NC	NC
parentID	NC	NC	NC	illegal	(9)	(10)
pathname	NC	NC	NC	illegal	(11)	(11)
position	NC	NC	NC	(12)	(13)	(14)
readBy	(2)	(2)	NC	illegal	**nullString**	**nullString**
readOn	(4)	(4)	NC	illegal	**nullTime**	**nullTime**
service	NC	NC	NC	(15)	(16)	(16)
sizeInBytes	NC	NC	NC	illegal	NC	NC
sizeInPages	NC	NC	NC	illegal	NC	NC
subtreeSize	NC	NC	(17)	illegal	NC	NC
subtreeSizeLimit	NC	NC	NC	set	NC	NC
type	NC	NC	NC	illegal	NC	NC
version	NC	NC	NC	set	(18)	NC

NC = No Change

Copy

Notes:

(1) Session's user if in a directory; **nullString** otherwise.
(2) Session's user.
(3) Current time if in a directory; **nullTime** otherwise.
(4) Current time.
(5) System-assigned value.
(6) May be **TRUE** only if directory is null.
(7) **TRUE** only if directory is null.
(8) **FALSE**; directories are never temporary.
(9) Set to **fileID** of resulting parent, **nullID** if temporary.
(10) Set to **fileID** of resulting parent.
(11) Appropriate to name of file and ancestors.
(12) Specified point.
(13) Beginning, end, or other, depending on **ordering** attribute of parent.
(14) Same relative point as original file.
(15) Must be consistent with service implied by destination parent.
(16) Same as parent or **defaultService**, if temporary.
(17) New total content in subtree.
(18) Next available version number for **name**.

Summary of Attribute Behaviors

Table 6.4

Attribute	Behavior
accessList	NC
BackedUpOn	NC
checksum	NC
childrenUniquelyNamed	NC
createdBy	NC
createdOn	NC
defaultAccessList	NC
extended	NC
filedBy	NC
filedOn	NC
fileID	NC
isDirectory	NC
isTemporary	NC
modifiedBy	NC
modifiedOn	NC
name	NC
numberOfChildren	NC
ordering	NC
parentID	NC
pathname	NC
position	NC
readBy	(1)
readOn	(2)
service	NC
sizeInBytes	NC
sizeInPages	NC
subtreeSize	NC
subtreeSizeLimit	NC
type	NC
version	NC

NC = No Change

CopyIn (NSSegment)

Notes:
(1) Session's user.
(2) Current time.

Summary of Attribute Behaviors

Table 6.5

Attribute	Behavior
accessList	NC
BackedUpOn	NC
checksum	(1)
childrenUniquelyNamed	NC
createdBy	(2)
createdOn	(3)
defaultAccessList	NC
extended	NC
filedBy	NC
filedOn	NC
fileID	NC
isDirectory	NC
isTemporary	NC
modifiedBy	(2)
modifiedOn	(3)
name	NC
numberOfChildren	NC
ordering	NC
parentID	NC
pathname	NC
position	NC
readBy	NC
readOn	NC
service	NC
sizeInBytes	NC
sizeInPages	NC
subtreeSize	NC
subtreeSizeLimit	NC
type	NC
version	NC

NC = No Change

CopyOut, MakeWritable, Move (NSSegment)

Notes:
(1) Set to **NSFile.unknownChecksum** (unchanged for **Move**).
(2) Session's user.
(3) Current time.

Summary of Attribute Behaviors

Table 6.6

Attribute	Destination Parent Behavior	Resulting File In List	Resulting File Behavior
accessList	NC	set	(1)
backedUpOn	NC	set	**nullTime**
checksum	NC	set	(2)
childrenUniquelyNamed	NC	set	**FALSE**
createdBy	NC	set	(3)
createdOn	NC	set	(4)
defaultAccessList	NC	set	(1)
extended	NC	set	empty
filedBy	NC	illegal	(5)
filedOn	NC	illegal	(6)
fileID	NC	illegal	(7)
isDirectory	NC	set	**FALSE**
isTemporary	NC	set (8)	(9)
modifiedBy	(3)	illegal	(3)
modifiedOn	(4)	illegal	(4)
name	NC	set	(10)
numberOfChildren	NC	illegal	0 (zero)
ordering	NC	set	(11)
parentID	NC	illegal	(12)
pathname	NC	illegal	(13)
position	NC	(14)	(15)
readBy	NC	illegal	**nullString**
readOn	NC	illegal	**nullTime**
service	NC	(16)	(17)
sizeInBytes	NC	set	0 (zero)
sizeInPages	NC	set	0 (zero)
subtreeSize	(18)	illegal	(2)
subtreeSizeLimit	NC	set	(19)
type	NC	set	(20)
version	NC	set	(21)

NC = No Change

Create

Notes:

(1) Set to **[defaulted: TRUE]**.
(2) Set to **nullChecksum**.
(3) Session's user.
(4) Current time.
(5) Session's user if in a directory, **nullString** otherwise.
(6) Current time if in a directory, **nullTime** otherwise.
(7) System-assigned value.
(8) Must be **TRUE** only if directory is null.
(9) **TRUE** only if directory is null.
(10) "Anonymous" or system-dependent.
(11) Set to **NSFile.defaultOrdering**.
(12) Set to **fileID** of resulting parent, **nullID** if temporary.
(13) Appropriate to name of ancestors and file.
(14) Specified point.
(15) Beginning, end, or other, depending on **ordering** attribute of parent.
(16) Must be consistent with service implied by destination parent.
(17) Same as parent or **defaultService**, if temporary.
(18) New total content in subtree.
(19) Set to **NSFile.nullSubtreeSizeLimit**.
(20) **NSAssignedTypes.tUnspecified**.
(21) Next available version number for **name**.

Summary of Attribute Behaviors

Table 6.7

Attribute	Behavior
accessList	NC
BackedUpOn	NC
checksum	NC
childrenUniquelyNamed	NC
createdBy	NC
createdOn	NC
defaultAccessList	NC
extended	NC
filedBy	NC
filedOn	NC
fileID	NC
isDirectory	NC
isTemporary	NC
modifiedBy	NC
modifiedOn	NC
name	NC
numberOfChildren	NC
ordering	NC
parentID	NC
pathname	NC
position	NC
readBy	(1)
readOn	(2)
service	NC
sizeInBytes	NC
sizeInPages	NC
subtreeSize	NC
subtreeSizeLimit	NC
type	NC
version	NC

NC = No Change

Create (NSFileStream)

Notes:
(1) Session's user (if file length non-zero, else no change).
(2) Current time (if file length non-zero, else no change).

Summary of Attribute Behaviors

Table 6.8

Attribute	Parent Behavior
accessList	NC
BackedUpOn	NC
checksum	NC
childrenUniquelyNamed	NC
createdBy	NC
createdOn	NC
defaultAccessList	NC
extended	NC
filedBy	NC
filedOn	NC
fileID	NC
isDirectory	NC
isTemporary	NC
modifiedBy	(1)
modifiedOn	(2)
name	NC
numberOfChildren	decremented
ordering	NC
parentID	NC
pathname	NC
position	NC
readBy	NC
readOn	NC
service	NC
sizeInBytes	NC
sizeInPages	NC
subtreeSize	(3)
subtreeSizeLimit	NC
type	NC
version	NC

NC = No Change

Delete

Notes:
(1) Session's user.
(2) Current time.
(3) New total content in subtree.

Summary of Attribute Behaviors

Table 6.9

Attribute	Destination Parent Behavior	Resulting File In List	Resulting File Behavior	Resulting Descendants Behavior
accessList	NC	set	NC	NC
BackedUpOn	NC	set	NC	NC
checksum	NC	illegal	(1)	(1)
childrenUniquelyNamed	NC	illegal	NC	NC
createdBy	NC	illegal	NC	NC
createdOn	NC	illegal	NC	NC
defaultAccessList	NC	set	NC	NC
extended	NC	set	NC	NC
filedBy	NC	illegal	(2)	(3)
filedOn	NC	illegal	(4)	(5)
fileID	NC	illegal	(6)	(6)
isDirectory	NC	illegal	NC	NC
isTemporary	NC	set (7)	(8)	(9)
modifiedBy	(3)	illegal	(3)	(3)
modifiedOn	(5)	illegal	(5)	(5)
name	NC	set	NC	NC
numberOfChildren	incremented	illegal	NC	NC
ordering	NC	illegal	NC	NC
parentID	NC	illegal	(10)	(10)
pathname	NC	illegal	(11)	(11)
position	NC	(12)	(13)	(13)
readBy	NC	illegal	**nullString**	**nullString**
readOn	NC	illegal	**nullTime**	**nullTime**
service	NC	(15)	(16)	(17)
sizeInBytes	NC	illegal	NC	NC
sizeInPages	NC	illegal	NC	NC
subtreeSize	(18)	illegal	NC	NC
subtreeSizeLimit	NC	set	NC	NC
type	NC	illegal	NC	NC
version	NC	set	(19)	NC

NC = No Change

Deserialize

Notes:

(1) Appropriate to transferred content.
(2) Session's user if in a directory, **nullString** otherwise.
(3) Session's user.
(4) Current time if in a directory, **nullTime** otherwise.
(5) Current time.
(6) System-assigned value.
(7) Must be **FALSE** if directory is not null.
(8) **TRUE** only if directory is null.
(9) **FALSE**; directories may never be temporary.
(10) Set to **fileID** of resulting parent.
(11) Appropriate to name of file and ancestors.
(12) Specified point.
(13) Beginning, end, or other, depending on **ordering** attribute of parent.
(14) Same relative point as original file.
(15) Must be consistent with service implied by destination parent.
(16) Same as parent or **defaultService**, if temporary.
(17) Same as parent.
(18) New total content in subtree.
(19) Next available version number for **name**.

Summary of Attribute Behaviors

Table 6.10

Attribute	Behavior
accessList	NC
BackedUpOn	NC
checksum	(1)
childrenUniquelyNamed	NC
createdBy	(2)
createdOn	(3)
defaultAccessList	NC
extended	NC
filedBy	NC
filedOn	NC
fileID	NC
isDirectory	NC
isTemporary	NC
modifiedBy	(2)
modifiedOn	(3)
name	NC
numberOfChildren	NC
ordering	NC
parentID	NC
pathname	NC
position	NC
readBy	(4)
readOn	(5)
service	NC
sizeInBytes	NC
sizeInPages	NC
subtreeSize	NC
subtreeSizeLimit	NC
type	NC
version	NC

NC = No Change

Map (NSSegment)

Notes:
(1) Set to **NSFile.unknownChecksum** if write access.
(2) Session's user, if write access requested.
(3) Current time, if write access requested.
(4) Session's user.
(5) Current time.

Summary of Attribute Behaviors

Table 6.11

Attribute	Source Parent Behavior	Destination Parent Behavior	File In List	File Behavior	Descendants Behavior
accessList	NC	NC	set	NC	NC
backedUpOn	NC	NC	set	NC	NC
checksum	NC	NC	illegal	NC	NC
childrenUniquelyNamed	NC	NC	illegal	NC	NC
createdBy	NC	NC	illegal	NC	NC
createdOn	NC	NC	illegal	NC	NC
defaultAccessList	NC	NC	set	NC	NC
extended	NC	NC	set	NC	NC
filedBy	NC	NC	illegal	(1)	NC
filedOn	NC	NC	illegal	(2)	NC
fileID	NC	NC	illegal	NC	NC
isDirectory	NC	NC	illegal	NC	NC
isTemporary	NC	NC	set (3)	**FALSE**	NC
modifiedBy	(1)	(1)	illegal	(1)	NC
modifiedOn	(2)	(2)	illegal	(2)	NC
name	NC	NC	set	NC	NC
numberOfChildren	decremented	incremented	illegal	NC	NC
ordering	NC	NC	illegal	NC	NC
parentID	NC	NC	illegal	(4)	NC
pathname	NC	NC	illegal	(5)	(5)
position	NC	NC	(6)	(7)	NC
readBy	NC	NC	illegal	NC	NC
readOn	NC	NC	illegal	NC	NC
service	NC	NC	illegal	(8)	(8)
sizeInBytes	NC	NC	illegal	NC	NC
sizeInPages	NC	NC	illegal	NC	NC
subtreeSize	(9)	(9)	illegal	NC	NC
subtreeSizeLimit	NC	NC	set	set	NC
type	NC	NC	illegal	NC	NC
version	NC	NC	set	(10)	NC

NC = No Change

Move

Notes:
(1) Session's user.
(2) Current time.
(3) Must be **FALSE**.
(4) Set to **fileID** of resulting parent.
(5) Appropriate to name of file and ancestors.
(6) Specified point.
(7) Beginning, end, or other, depending on **ordering** attribute of parent.
(8) Same as destination parent.
(9) New total content of subtree.
(10) Next available version number for **name**.

Summary of Attribute Behaviors

Table 6.12

Attribute	Resulting File In List	Behavior
accessList	illegal	NC
backedUpOn	illegal	NC
checksum	illegal	NC
childrenUniquelyNamed	illegal	NC
createdBy	illegal	NC
createdOn	illegal	NC
defaultAccessList	illegal	NC
extended	ignored	NC
filedBy	illegal	NC
filedOn	illegal	NC
fileID	(1)	NC
isDirectory	illegal	NC
isTemporary	illegal	NC
modifiedBy	illegal	NC
modifiedOn	illegal	NC
name	(1)	NC
numberOfChildren	illegal	NC
ordering	illegal	NC
parentID	(2)	NC
pathname	(1)	NC
position	illegal	NC
readBy	illegal	NC
readOn	illegal	NC
service	(1)	NC
sizeInBytes	illegal	NC
sizeInPages	illegal	NC
subtreeSize	illegal	NC
subtreeSizeLimit	illegal	NC
type	illegal	NC
version	(3)	NC

NC = No Change

Open

Notes:
(1) File with this value is opened.
(2) The **fileID** of directory to search.
(3) File with this value is opened; **name** or **pathname** must also be specified.

Summary of Attribute Behaviors

Table 6.13

Attribute	Parent Behavior	File In List	File Behavior
accessList	NC	illegal	NC
backedUpOn	NC	illegal	NC
checksum	NC	set	(1)
childrenUniquelyNamed	NC	illegal	NC
createdBy	NC	set	(2)
createdOn	NC	set	(3)
defaultAccessList	NC	illegal	NC
extended	NC	ignored	NC
filedBy	NC	illegal	NC
filedOn	NC	illegal	NC
fileID	NC	illegal	NC
isDirectory	NC	illegal	NC
isTemporary	NC	illegal	NC
modifiedBy	NC	illegal	(2)
modifiedOn	NC	illegal	(3)
name	NC	illegal	NC
numberOfChildren	NC	illegal	NC
ordering	NC	illegal	NC
parentID	NC	illegal	NC
pathname	NC	illegal	NC
position	NC	illegal	NC
readBy	NC	illegal	NC
readOn	NC	illegal	NC
service	NC	illegal	NC
sizeInBytes	NC	(4)	(5)
sizeInPages	NC	(4)	(5)
subtreeSize	(6)	illegal	(1)
subtreeSizeLimit	NC	illegal	NC
type	NC	illegal	NC
version	NC	illegal	NC

NC = No Change

Replace

Notes:
(1) Set to appropriate value for current content.
(2) Session's user.
(3) Current time.
(4) Set to value appropriate for transferred content.
(5) Number of bytes transferred.
(6) New total content in subtree.

Summary of Attribute Behaviors

Table 6.14

Attribute	Source File Behavior
accessList	NC
BackedUpOn	NC
checksum	(1)
childrenUniquelyNamed	NC
createdBy	NC
createdOn	NC
defaultAccessList	NC
extended	NC
filedBy	NC
filedOn	NC
fileID	NC
isDirectory	NC
isTemporary	NC
modifiedBy	NC
modifiedOn	NC
name	NC
numberOfChildren	NC
ordering	NC
parentID	NC
pathname	NC
position	NC
readBy	(2)
readOn	(3)
service	NC
sizeInBytes	NC
sizeInPages	NC
subtreeSize	NC
subtreeSizeLimit	NC
type	NC
version	NC

NC = No Change

Retrieve

Notes:
(1) Set if previously unknown.
(2) Session's user.
(3) Current time.

Summary of Attribute Behaviors

Table 6.15

Attribute	Source File Behavior	Source Descendants Behavior
accessList	NC	NC
backedUpOn	NC	NC
checksum	(1)	(1)
childrenUniquelyNamed	NC	NC
createdBy	NC	NC
createdOn	NC	NC
defaultAccessList	NC	NC
extended	NC	NC
filedBy	NC	NC
filedOn	NC	NC
fileID	NC	NC
isDirectory	NC	NC
isTemporary	NC	NC
modifiedBy	NC	NC
modifiedOn	NC	NC
name	NC	NC
numberOfChildren	NC	NC
ordering	NC	NC
parentID	NC	NC
pathname	NC	NC
position	NC	NC
readBy	(2)	(2)
readOn	(3)	(3)
service	NC	NC
sizeInBytes	NC	NC
sizeInPages	NC	NC
subtreeSize	NC	NC
subtreeSizeLimit	NC	NC
type	NC	NC
version	NC	NC

NC = No Change

Serialize

Notes:
(1) Set if previously unknown.
(2) Session's user.
(3) Current time.

Summary of Attribute Behaviors

Table 6.16

Attribute	File Behavior
accessList	NC
BackedUpOn	NC
checksum	(1)
childrenUniquelyNamed	NC
createdBy	(2)
createdOn	(3)
defaultAccessList	NC
extended	NC
filedBy	NC
filedOn	NC
fileID	NC
isDirectory	NC
isTemporary	NC
modifiedBy	(2)
modifiedOn	(3)
name	NC
numberOfChildren	NC
ordering	NC
parentID	NC
pathname	NC
position	NC
readBy	NC
readOn	NC
service	NC
sizeInBytes	(4)
sizeInPages	(5)
subtreeSize	(6)
subtreeSizeLimit	NC
type	NC
version	NC

NC = No Change

SetLength (NSFileStream)

Notes:
(1) Set to **NSFile.unknownChecksum**.
(2) Session's user.
(3) Current time.
(4) Set (affects only default segment).
(5) Set to appropriate value for current content.
(6) Set to new total content in subtree.

Summary of Attribute Behaviors

Table 6.17

Attribute	Destination Parent Behavior	Resulting File In List	Resulting File Behavior
accessList	NC	set	(1)
backedUpOn	NC	set	**nullTime**
checksum	NC	set	(2)
childrenUniquelyNamed	NC	set	**FALSE**
createdBy	NC	set	(3)
createdOn	NC	set	(4)
defaultAccessList	NC	set	(1)
extended	NC	set	empty
filedBy	NC	illegal	(5)
filedOn	NC	illegal	(6)
fileID	NC	illegal	(7)
isDirectory	NC	set	**FALSE**
isTemporary	NC	set (8)	(9)
modifiedBy	(3)	illegal	(3)
modifiedOn	(4)	illegal	(4)
name	NC	set	(10)
numberOfChildren	incremented	illegal	0 (zero)
ordering	NC	set	(11)
parentID	NC	illegal	(12)
pathname	NC	illegal	(13)
position	NC	(14)	(15)
readBy	NC	illegal	**nullString**
readOn	NC	illegal	**nullTime**
service	NC	(16)	(17)
sizeInBytes	NC	(18)	(19)
sizeInPages	NC	(18)	(19)
subtreeSize	(20)	illegal	(2)
subtreeSizeLimit	NC	set	(21)
type	NC	set	(22)
version	NC	set	(23)

NC = No Change

Store

Notes:

(1) Set to **[defaulted: TRUE]**.
(2) Appropriate value for current content.
(3) Session's user.
(4) Current time.
(5) Session's user if in a directory, **nullString** otherwise.
(6) Current time if in a directory, **nullTime** otherwise.
(7) System-assigned value.
(8) Must be **FALSE** if directory is not null.
(9) **TRUE** if directory is null.
(10) "Anonymous" or system-dependent.
(11) Set to **NSFile.defaultOrdering**.
(12) Set to **fileID** of resulting parent.
(13) Appropriate to name of ancestors and file.
(14) Specified point.
(15) Beginning, end, or other, depending on **ordering** attribute of parent.
(16) Must be consistent with service implied by destination parent.
(17) Same as parent or **defaultService**, if temporary.
(18) Number of bytes transferred.
(19) Set to value appropriate to transferred content.
(20) New total content in subtree.
(21) Set to **NSFile.nullSubtreeSizeLimit**.
(22) **NSAssignedTypes.tUnspecified**.
(23) Next available version number for **name**.

Summary of Attribute Behaviors

Table 6.18

Attribute	File Behavior	Descendants Behavior
accessList	NC	(1)
backedUpOn	NC	NC
checksum	NC	NC
childrenUniquelyNamed	NC	NC
createdBy	NC	NC
createdOn	NC	NC
defaultAccessList	NC	(1)
extended	NC	NC
filedBy	NC	NC
filedOn	NC	NC
fileID	NC	NC
isDirectory	NC	NC
isTemporary	NC	NC
modifiedBy	NC	(2)
modifiedOn	NC	(3)
name	NC	NC
numberOfChildren	NC	NC
ordering	NC	NC
parentID	NC	NC
pathname	NC	NC
position	NC	NC
readBy	NC	NC
readOn	NC	NC
service	NC	NC
sizeInBytes	NC	NC
sizeInPages	NC	NC
subtreeSize	NC	NC
subtreeSizeLimit	NC	NC
type	NC	NC
version	NC	NC

NC = No Change

UnifyAccessLists

Notes:
(1) Set to **[defaulted: TRUE]**.
(2) Session's user if modified.
(3) Current time if modified.

7

Pathname parsing operations

```
NSFileName: DEFINITIONS . . . ;
```

The **pathname** attribute of a file gives a list of the names and versions of the file's ancestors, listed in hierarchical order, ending with the name and version of the file itself. It is thus relative to the file service on which the file resides. A *qualified* pathname for a file contains the name of the service which contains the file as well as the file's service-relative pathname.

The **NSFileName** interface provides routines for splitting qualified pathnames into their service name and service-relative components and routines for merging these components into a qualified pathname string. This interface also defines the standard delimiters for pathname components.

7.1 Pathname separators and other special characters

Parsing of pathname strings is based on the separators defined in this section.

7.1.1 Service name separators

Service names are enclosed by the **leftServiceSeparator** and the **rightServiceSeparator** to delimit them from the service-relative portion of the pathname.

```
NSFileName.Character: TYPE = NSString.Character;

NSFileName.leftServiceSeparator: Character = [0, 50B]; -- '(, the left parenthesis
NSFileName.rightServiceSeparator: Character = [0, 51B]; -- '), the right parenthesis
```

Service names conform to the specifications of Clearinghouse names whose components are delimited by **NSName.separatorCharacter**, i.e., (:).

7.1.2 Pathname component separators

```
NSFileName.nameVersionPairSeparator: Character = [0, 57B]; -- '/, the diagonal slash
NSFileName.versionSeparator: Character = [0, 41B]; -- '!, the exclamation point
```

In the service-relative portion of a pathname, the name of a file is separated from the version number of the file by the **versionSeparator**. Name-version pairs in the pathname are delimited from each other by the **nameVersionPairSeparator**. A pathname supplied to the file system need not contain explicit version numbers for each of the files listed; where not specified explicitly an appropriate default for the version number is supplied by the file system.

Following are some examples of pathnames:

(File Service:Unit 1:Acme)Templates/Financial Forms/Expense Report!2
This is a qualified pathname for a file on the file service named "File Service:Unit 1:Acme." The file's name is "Expense Report" and version number is 2, its parent's name is "Financial Forms," and its parent's parent's name is "Templates." Since no version is specified for the files "Templates" and "Financial Forms," their version numbers are defaulted by the file system.

Templates/Financial Forms/Expense Report!2
This is a service-relative pathname. This pathname will name the same file as the pathname in the above example if it is presented to the same service as that named there.

7.1.3 Characters for version number constants

The version number constants **NSFile.lowestVersion** and **NSFile.highestVersion** can be represented explicitly in pathnames using the characters defined below.

NSFileName.lowestVersion: Character = [0,55B]; -- *'- , the minus sign*
NSFileName.highestVersion: Character = [0,53B]; -- *' + , the plus sign*

The following is an example of a pathname which uses these constants:

(File Service:Unit 1:Acme)Templates/Financial Forms!-/Expense Report! +
This is a pathname for a file on the file service "File Service:Unit 1:Acme." The file is the one with the highest version number of files named "Expense Report" contained in the directory whose name is "Financial Forms" and which has the lowest version of files with the same name contained in the directory named "Templates." Since no version is specified for the file "Templates," its version number is defaulted by the file system.

7.1.4 Wildcard characters

NSFileName.matchSingleChar: Character = [0,43B]; -- *'# , the pound sign*
NSFileName.matchMultipleChars: Character = [0,52B]; -- *'* , the asterisk*

Wildcard characters can be used in file names or pathnames for pattern matching in filters. **matchSingleChar** means match a single character and **matchMultipleChars** means match zero or more characters. For a more detailed description of pattern matching in filters, see §3.5.1.

7.1.5 The escape character

```
NSFileName.escapeChar: Character = [0, 47B]; -- '', the apostrophe
```

The name of a file is permitted to contain any of the pathname separator characters or wildcard characters. To indicate that such a character should be interpreted as part of the file's name and not as a special character, this character must be preceded by the **escapeChar** when written in a pathname. If the **escapeChar** is itself a character in a file name, it too must be preceded by the **escapeChar** when written in a pathname.

The following is an example of a pathname which includes an **escapeChar** :

(File Service:Unit 1:Acme)Templates/Vacation'/Holiday Planning Form
This is the pathname for a file on file service "File Service:Unit 1:Acme." The file's name is "Vacation/Holiday Planning Form," and is contained in the directory called "Templates."

7.2 The default domain and organization

The client may set the default domain and organization to be used during parsing when either of these fields are omitted from the service portion of a pathname by calling **SetDefaultDomainAndOrg**. Calling this operation sets **defaultDomain** and **defaultOrg** to the specified values (these are initially set to null strings). These defaults are global to a system element and not session-relative.

```
NSFileName.defaultDomain, NSFileName.defaultOrg : READONLY String;

NSFileName.SetDefaultDomainAndOrg: PROCEDURE [domain, org:String];
```

Arguments: **domain** and **org** indicate the defaults to be used whenever these fields are omitted from the service portion of a qualified pathname.

Results: **NSFileName.defaultDomain** is set to **domain** and **NSFileName.defaultOrg** is set to **org**.

Errors: None.

7.3 Parsing qualified pathnames

A **VirtualPathname** contains both the components of a qualified pathname for a file: the service name and the local, or service-relative, portion of a pathname.

```
NSFileName.VirtualPathname, VPN: TYPE = LONG POINTER TO VPNRecord;

NSFileName. VPNRecord: TYPE = RECORD [
    pathname: String,
    service: Service];

NSFileName.Service: TYPE = NSFile.Service;
NSFileName.String: TYPE = NSString.String;
NSFileName.nullString: String = NSString.nullString;
```

VPNFieldsFromString divides a string containing a well-formed pathname (which may or may not contain a service name) into a **VPNRecord**. Division of the service name (if specified) into an **NSName.Name** is done using **NSName** routines which use **NSName.separatorCharacter**, i.e., (:) to delimit the fields of the service name. The client is responsible for freeing storage allocated for the **pathname** portion of **destination** (the space allocated to the **service** portion of **destination** is managed by the file system and the client should *not* attempt to free it); **ClearVPN** or **FreeVPNFields** can be used to free the **pathname** portion and nullify the **service** portion of **destination**.

```
NSFileName.VPNFieldsFromString: PROCEDURE [
    z: UNCOUNTED ZONE, s: String, destination: VPN];
```

Arguments: **z** is the zone from which the fields of the **VPNRecord** are to be allocated, **s** is the string to be parsed as a **VPN**, and **destination** is the pointer to the client's **VPNRecord** which will contain the results of parsing **s**.

Results: The fields of **destination** are filled with the results of parsing **s** into its **service** and **pathname** component parts. When **s** is missing a service name, or contains a service name which is missing domain and organization parts, the missing parts are filled into **destination** according to the conventions described below.

Errors: **NSFileName.Error [invalidSyntax]**, may be raised.

If the domain and organization fields are both omitted from the service name of a pathname, **VPNFieldsFromString** will fill in those fields in the following manner. If the **NSName.separatorCharacter** is included after the local field of the service name, the domain and organization are set to **defaultDomain** and **defaultOrg**. If the **NSName.separatorCharacter** is omitted after the local field of the service name, the domain and organization are set to **nullString**. When the file system is presented with a service name having null domain and organization parts, it interprets this to mean that the service is local.

The following examples illustrate this distinction:

(File Service:)Templates/Financial Forms/Expense Report

The presence of the colon (:) after the local portion of the file service name indicates that the defaults **defaultDomain** and **defaultOrg** are to be used to fill in the domain and organization fields of the service name.

(Filing)SystemFiles/Fonts/ClassicFont

The absence of the colon (:) after the local portion of the file service name indicates that the service is local, and that null strings should be used to fill in the domain and organization fields of the service name.

If the left and right service separators are included in a pathname, but the service name is not specified, then **VPNFieldsFromString** returns **NSFile.defaultService** as **vpn.service**. If neither a service name nor the service separators are included, then **VPNFieldsFromString** returns **NSFile.nullService** as **vpn.service**.

The following examples illustrate this distinction:

()SystemFiles/Fonts/ClassicFont
This is a qualified pathname which names a file on the default service.

SystemFiles/Fonts/ClassicFont
This pathname is not a qualified pathname, but a service-relative pathname. The service on which the file resides is not indicated by the pathname.

VPNFromString is like **VPNFieldsFromString** except the **VPNRecord** is also allocated from the specified zone. The client is responsible for freeing allocated memory using **FreeVPN**.

```
NSFileName.VPNFromString: PROCEDURE [z: UNCOUNTED ZONE, s: String] RETURNS [vpn: VPN];
```

7.4 Appending VPNs to Strings

CopyVPNToString copies the fields of a **VPN** into a **String** which it allocates from a specified zone, separating the components of the **VPN** by the defined separators. The separator used between fields of service names is the **NSName.separatorCharacter**, i.e., (:).

```
NSFileName.CopyVPNToString: PROCEDURE [
    z: UNCOUNTED ZONE, vpn: VPN, extra: CARDINAL]
    RETURNS [s: String];
```

Arguments: **z** is the zone from which **s** is to be allocated, **vpn** is the **VPN** whose fields are to be copied to **s**. **extra** refers to additional characters to be allocated to **s** beyond those needed to convert **vpn** to a string.

Results: The fields of **vpn** are concatenated together with the appropriate delimiters to form the string **s**; **s** is allocated to be the proper size from **z**.

Errors: None.

AppendVPNToString is like**CopyVPNToString** except that it appends the fields of a **VPN** to a preallocated **String**.

```
NSFileName.AppendVPNToString: PROCEDURE [
    s: String, vpn: VPN, resetLengthFirst: BOOLEAN]
    RETURNS [newS: String];
```

Arguments: **s** is the pre-allocated **String** to which the fields of **vpn** are to be appended, **resetLengthFirst** indicates if the length of **s** should be set to zero before the fields of **vpn** are appended.

Results: The fields of **vpn** are appended to **s**.

Errors: **NSString.StringBoundsFault** can be raised.

7.5 Allocation and deallocation of VPNs

This section defines operations for copying **VPN**s and for freeing **VPN**s which have been allocated by **NSFileName** operations.

7.5.1 Copying VPNs

CopyVPNFields copies a source **VPN** to a destination **VPN**, allocating the fields of destination from a specified zone. The client is responsible for freeing the allocated memory; **ClearVPN** can be used for this purpose.

```
NSFileName.CopyVPNFields: PROCEDURE [
    z: UNCOUNTED ZONE, source, destination: VPN];
```

Arguments: **z** is the zone from which the fields of **destination** are to be allocated, **source** is the **VPN** to be copied, **destination** is the destination of the copy.

Results: **source** is copied to **destination**. The fields of **destination** are allocated from **z**.

Errors: None.

CopyVPN is like **CopyVPNFields** except the copied **VPNRecord** is also allocated from the zone. **FreeVPN** can be used to deallocate the **VPNRecord** and its fields.

```
NSFileName.CopyVPNFields : PROCEDURE [z: UNCOUNTED ZONE, vpn: VPN] RETURNS [VPN];
```

7.5.2 Freeing VPNs

ClearVPN and **FreeVPNFields** free the pathname portion of **vpn** and set **VPNRecord** of **vpn** to **[nullString, NSFile.nullService]**.

```
NSFileName. ClearVPN, FreeVPNFields: PROCEDURE [z: UNCOUNTED ZONE, vpn: VPN];
```

Arguments: **z** is the zone from which the fields of **vpn** have been allocated, **vpn** is the **VPN** whose fields are to be deallocated.

Results: The **pathname** portion of **vpn** is freed and the fields of **vpn** are set to **[nullString, NSFile.nullService]**

Errors: None.

FreeVPN is like **ClearVPN** and **FreeVPNFields** except the **VPNRecord** is also freed.

```
NSFileName. FreeVPN:PROCEDURE [z: UNCOUNTED ZONE, vpn: VPN] ;
```

7.6 Errors

When an **NSFileName** operation is unable to complete successfully, it reports this fact by raising the error, **NSFileName.Error**.

```
NSFileNameError: ERROR [type: ErrorType];

NSSegment.ErrorType: TYPE = {invalidSyntax};
```

The argument **type** describes the problem in greater detail:

invalidSyntax Unable to complete parsing of a pathname due to one of the following errors in syntax:

- The supplied pathname was zero in length.
- The service name specified in the pathname was an invalid **NSName.Name**.
- The **leftServiceSeparator** was encountered at the beginning of a pathname, but the **rightServiceSeparator** was never encountered.
- The service name portion of the pathname had a **local** name, **domain** name, or **org** name which exceeded the maximum length allowed for it as specified in **NSName**.

8

System configuration and administration

8.1 Global file system variables

NSFileControl defines variables and procedures that set certain file system characteristics and defaults for use by other software on the same processor.

```
NSFileControl: DEFINITIONS = ...;
```

The interface offers several facilities. The first, *protocol versions*, provides a way for the system's control module to indicate which versions of the filing protocol are exported by the file service running on the system. The second, *group membership status*, is a facility used by the system's control module to find out whether a session's client is a member of a particular group. Finally, *miscellaneous operations* provide other information global to the file system.

8.1.1 Protocol versions

The file service running on a system may support one or a number of versions of the Filing Protocol. To make these versions of the protocol available or unavailable to network clients, the procedures **ExportProtocol** and **UnexportProtocol** are defined. These procedures accept a **VersionRange** as an argument. All versions of the protocol within the specified range are then made available/unavailable.

```
NSFileControl.Version: TYPE = CARDINAL;

NSFileControl.VersionRange: TYPE = RECORD [low, high: Version];
```

The constant **allVersions** is defined which the client uses to indicate that he wishes all possible versions of the Filing Protocol to be exported/unexported.

```
NSFileControl.allVersions: VersionRange = [low: LAST[Version], high: LAST[Version]];
```

The range of versions of the Filing Protocol which could be exported by calling **ExportProtocol** is indicated by the range **protocolVersions**.

```
NSFileControl.protocolVersions: READONLY VersionRange;
```

```
NSFileControl.ExportProtocol: TYPE = PROCEDURE [
  version: VersionRange ← allVersions];
```

Arguments: **version** specifies the version(s) of the Filing Protocol to be made available to network clients.

Results: The version(s) of the Filing Protocol specified by **version** are made available to network clients. This operation permits this system element to respond to Filing requests.

Errors: **NSFileControl.Error** may be raised with the following types: **duplicateExport** or **versionInvalid**.

Note: Before calling **ExportProtocol**, the file system must be started by calling **NSFileControl.Start** (see §8.1.3).

```
NSFileControl.UnexportProtocol: TYPE = PROCEDURE [
  version: VersionRange ← allVersions];
```

Arguments: **version** specifies the version(s) of the Filing Protocol to be made unavailable to network clients.

Results: The version(s) of the Filing Protocol specified by **version** are made unavailable to network clients.

Errors: **NSFileControl.Error** may be raised with the following types: **noSuchExport** or **versionInvalid**.

8.1.2 Membership status

A **MembershipProc** is a procedure provided by the system's control module to determine the membership status of a session's user with respect to an instance of a class of a specified type. The type, **MembershipStatus**, defines the possible outcomes of the membership evaluation.

```
NSFileControl.MembershipProc: TYPE = PROCEDURE [
  key: NSString.String, type: NSFile.AccessEntryType, session: NSFile.Session]
  RETURNS [status: MembershipStatus];
```

Arguments: **key** identifies an instance of the class for which the client's membership status is requested (e.g., "Filing Implementors:OSBU Nouth:Xerox"); **type** identifies the class of **key** (e.g., **group**); **session** refers to the client's session.

Results: **status** indicates whether the client is a member of the specified class with the given type.

```
NSFileControl.MembershipStatus: TYPE = {member, notAMember, cannotDetermine};
```

During startup, the system's control module may designate the **MembershipProc** to be used in determining a client's membership status. This is done by calling **RegisterMembershipProc**. If the system's control module does not specify a particular **MembershipProc**, then the default, **defaultMembershipProc**, is assumed.

NSFileControl.RegisterMembershipProc: PROCEDURE [membershipProc: MembershipProc];

Arguments: **membershipProc** is the procedure to be used in determining membership status.

Results: None.

Errors: None.

The default **MembershipProc**, **defaultMembershipProc**, returns a status of **member** if **type** is **individual** and **key** exactly matches the full name of the logged on user. It returns **cannotDetermine** if **type** is **group** and returns **notAMember** otherwise.

NSFileControl.defaultMembershipProc: MembershipProc;

8.1.3 Miscellaneous operations

The following items in **NSFileControl** are included for convenience. They pertain to characteristics of all file services on the local system element.

The default timeout is the timeout, in seconds, actually used on this machine when a local or network client specifies an **NSFile.Controls** containing the constant **NSFile.defaultTimeout**.

NSFileControl.defaultTimeout: READONLY NSFile.Timeout;

The default timeout is initially 60 seconds. It can be changed by calling **NSFileControl.SetDefaultTimeout**.

NSFileControl.SetDefaultTimeout: PROC [timeout: NSFile.Timeout];

The default name is the name given to any file created on the local system element by **NSFile.Create** or **NSFile.Store** when no name is specified in the attribute list in the operation.

NSFileControl.defaultName: READONLY NSString.String;

The default name is initially "Anonymous." It can be changed by calling **NSFileControl.SetDefaultName**.

NSFileControl.SetDefaultName: PROC [name: NSString.String];

This procedure copies the passed string, so the storage for **name** may be released after the operation returns.

Files transmitted using the Filing protocol and stored on file servers are understood to have a content which is a single contiguous sequence of bytes, rather than multiple contiguous sequences of bytes (segments). A file which has more than one segment when stored in the local file system must be compressed into a single segment before it is

transferred to a remote file system. When this compressed file is later received from a remote file system, it is decomposed into a multi-segment file only if it has one of the file types which the client has distinguished as a segmented file type. The client may specify a maximum of five distinguished file types by successive calls on the operation NSFileControl.DistinguishSegmentedFileType.

```
NSFileControl.DistinguishSegmentedFileType: PROCEDURE [type: NSFile.Type];
```

The operation NSFileControl.IsSegmentedFileType can be used to determine whether a file type has been distinguished.

```
NSFileControl.IsSegmentedFileType [type: NSFile.Type] RETURNS [BOOLEAN];
```

The procedure NSFileControl.Start is provided for the client who wishes to initialize Filing's exported variables before making use of file system operations. The procedure has no other effect, and redundant calls are ignored.

```
NSFileControl.Start: PROCEDURE;
```

8.1.4 Errors

The protocol version operations (**ExportProtocol** and **UnexportProtocol**) may raise NSFileControl.Error.

```
NSFileControl.Error: ERROR [type: ErrorType];

NSFileControl.ErrorType: TYPE = {
  duplicateExport, versionInvalid, noSuchExport};
```

ErrorType gives more detailed information as to the nature of the problem.

duplicateExport	An attempt was made to export a version of the protocol which is already exported.
versionInvalid	Export of an unknown version of the protocol was requested.
noSuchExport	An attempt was made to unexport a version of the protocol which is already unexported or which was never exported.

8.2 Volumes

```
NSVolumeControl: DEFINITIONS . . . ;
```

The **NSVolumeControl** interface contains variables and procedures used to manage the local file system's volumes.

Local files may be located on one or more disjoint Pilot volumes, each of which represents an independent, complete file system. Each Filing volume has its own root file which resides on that volume. All descendants of a root file reside on the same volume, and all non-temporary Filing files on a volume are descendants of that volume's root file.

Each Filing volume corresponds to a distinct NSFile.Service. The volume must be opened to make the corresponding service available for client use. Every volume is given a name, which is also the name of the corresponding service. Network clients use this name to identify the service, so the name conforms to the specifications for Clearinghouse names. The name of the volume is available to the client of **NSVolumeControl** through the operation **GetName**. The file system requires that the local field of a volume name be unique among the local names of all the open volumes on the same system element.

All of the operations in this interface identify a volume by its Pilot logical volume ID rather than its name. If the name of the volume is known (i.e., the name of the corresponding service is known), the volume ID can be obtained via the operation **GetID**.

The root file's **fileID** can be obtained by calling NSVolumeControl.GetAttributes. Alternatively, a volume's root file can be opened by passing a reference to NSFile.OpenByReference in which the **service** corresponding to the volume is specified and the **fileID** is NSFile.nullID, or by passing an attribute list to NSFile.Open which contains the **service** corresponding to the volume but does not contain a **fileID**, a **name**, or a **pathname**.

8.2.1 Opening and closing volumes

The operation NSVolumeControl.Open makes the files on the specified volume accessible.

```
NSVolumeControl.Open: PROCEDURE [volume: Volume.ID];
```

Arguments: **volume** is the volume to be opened. It is a Pilot volume which may be open or closed.

Errors: If the volume is already open as a Filing volume, NSVolumeControl.Error [alreadyOpen] is raised. If **volume** is not a known Pilot volume, NSVolumeControl.Error [notMounted] is raised. If **volume** is a known Pilot volume but does not appear to be a valid Filing volume, NSVolumeControl.Error [invalidVolume] is raised. If the volume appears to be damaged, NSVolumeControl.Error [needsScavenging] is raised; the client must call NSVolumeControl.Scavenge before trying to open the volume again. If the volume was created by an incompatible version of Filing, NSVolumeControl.Error [incompatibleVolume] is raised; such a volume must be scavenged before it is opened. If there is already an open volume with the same local name as the volume being opened, NSVolumeControl.NameNotUnique is raised.

The operation NSVolumeControl.Close is called when the client no longer wants access to a Filing volume.

```
NSVolumeControl.Close: PROCEDURE [volume: Volume.ID];
```

Arguments: **volume** is the volume to be closed.

Errors: A volume may be closed only if no sessions to its corresponding service exist; if such sessions exist, NSVolumeControl.Error [sessionsExist] is raised. If the volume to be closed is not open, NSVolumeControl.Error [notOpen] is raised.

Note: Closing a Filing volume other than the system volume also causes the underlying Pilot volume to be closed.

8.2.2 The system volume

The system volume is the Pilot volume containing the running boot file. It need not actually be a valid or open Filing volume.

```
NSVolumeControl.systemVolume: READONLY Volume.ID;
```

8.2.3 Initializing volumes

A Pilot volume which does not already contain a Filing file system can be initialized as a Filing volume by calling NSVolumeControl.Initialize.

```
NSVolumeControl.IndexAttributes: TYPE = RECORD [
  size: LONG CARDINAL ← 100,
  pageIncrement: LONG CARDINAL ← 100,
  percentIncrement: Percent ← 20];

NSVolumeControl.Percent: TYPE = [0..100];

NSVolumeControl.Initialize: PROCEDURE [
  volume: Volume.ID, index: IndexAttributes ← [], root: NSFile.AttributeList ← NIL];
```

Arguments: **volume** is the volume to be initialized; **index** specifies certain attributes of the volume's B-tree index file; **root** specifies attributes of the root file to be created.

Errors: NSFile.Error may be raised with arguments identical to those raised by NSFile.Create. In addition, NSVolumeControl.Error [alreadyOpen] is raised if **volume** names an open Filing volume; NSVolumeControl.Error [alreadyInitialized] is raised if the passed Pilot volume already contains a Filing file system; NSVolumeControl.Error [notMounted] is raised if **volume** does not name a known Pilot volume; NSVolumeControl.Error [needsScavenging] is raised if the Pilot volume needs scavenging. NSVolumeControl.NameNotUnique is raised if there is already an open volume having the same local name as that specified for the volume being initialized. NSVolumeControl.Error [nameLengthLimit] is raised if the length of the specified name exceeds NSName.maxLocalLength.

The volume may be an open or closed Pilot volume; it may not be an open Filing volume. There must not already be a Filing file system on the volume.

Initializing a Filing volume allocates a data structure called the B-tree index file, which is used to maintain the volume's directory structure. This file is invisible to clients (except for the space overhead it consumes), but certain attributes of the volume's B-tree index file can be passed in the parameter **index**. Currently, they cannot be subsequently changed by clients after the volume is initialized.

size is the initial size in pages of the index file. **pageIncrement** and **percentIncrement** control the growth of the index file: if the size of the index file must be increased, it is increased either by a fixed amount or by a percentage of its current size, whichever is larger.

The client may specify an attribute list for the root file by passing a non-**NIL** value of **root**. All attributes that may be specified for **NSFile.Create** may be specified here, except **service**. However, the default values of attributes that are not specified are somewhat different:

isDirectory defaults to **TRUE** (it may be specified as **FALSE**, but this is not very useful)

childrenUniquelyNamed defaults to **TRUE**

name defaults to the name of the corresponding Pilot logical volume. The local name of the volume (and thus of its corresponding service) is initialized to this value. This name must be unique among the local names of all open volumes on the same system element. To set the full Clearinghouse name of the volume, the client must call **ChangeName** after initializing the volume.

createdBy defaults to the string "Initialization."

In addition, the **modifiedBy** attribute (which may not be specified) is given the initial value of the string "Initialization"; the **filedOn** and **filedBy** attributes are null.

8.2.4 Volume attributes

NSVolumeControl.GetAttributes is used to obtain the attributes of an open Filing volume.

```
NSVolumeControl.GetAttributes: PROCEDURE [volume: Volume.ID]
  RETURNS [used, available: LONG CARDINAL, index: IndexAttributes, root: NSFile.ID];
```

Arguments: **volume** is the volume of interest.

Results: **used** is the number of Pilot disk pages in use. Since it includes overhead, it will be larger than the **subtreeSize** attribute of the volume's root file. **available** is the number of free pages on the volume; however, there is no guarantee that a file of this size can actually be created. **root** is the **NSFile.ID** for the volume's root file. **index** contains data about the volume's B-tree index file; currently, the index attributes cannot be changed by clients after a volume is initialized.

Errors: **NSVolumeControl.Error [notOpen]** is raised if **volume** is not an open Filing volume.

8.2.5 Volume name

The **NSName.Name** of an open Filing volume can be obtained using **NSVolumeControl.GetName**. Clients of Filing operations use this name to identify the corresponding service.

```
NSVolumeControl.GetName: PROCEDURE [z: UNCOUNTED ZONE, volume: Volume.ID]
  RETURNS [volumeName: NSName.Name];
```

Arguments: **volume** is the volume of interest, **z** is the zone from which storage for the volume name will be allocated.

Results: **volumeName** is the **NSName.Name** of **volume** (and hence of its corresponding service). Storage is allocated for both the name record and the strings from the z, thus **NSName.FreeName** must be used by the client to deallocate that storage.

Errors: **NSVolumeControl.Error [notOpen]** is raised if **volume** is not an open Filing volume.

NSVolumeControl.GetID may be used to obtain the ID of a volume given its **NSName.Name** (i.e., the name of its corresponding service).

```
NSVolumeControl.GetID: PROCEDURE [
  volumeName: NSName.Name, ignoreOrgAndDomain: BOOLEAN ← FALSE]
  RETURNS [ volume: Volume.ID];
```

Arguments: **volumeName** is the name of the volume whose ID is of interest. If **ignoreOrgAndDomain** is **TRUE**, then the organization and domain fields of **volumeName** are ignored in the lookup, and only the local name is considered.

Results: The ID of the volume having the given **NSName.Name** is returned as **volume**.

Errors: **NSVolumeControl.Error [volumeNotFound]** is raised if **volumeName** does not name an open Filing volume.

The **NSName.Name** of an open Filing volume (and hence of its corresponding service) can be changed using **NSVolumeControl.ChangeName**.

```
NSVolumeControl.ChangeName: PROCEDURE [
  volume: Volume.ID, volumeName: NSName.Name];
```

Arguments: **volume** is the volume whose name is to be changed, **volumeName** is the new name of the volume.

Results: The name of **volume** is changed to **volumeName**.

Errors: **NSVolumeControl.Error [notOpen]** is raised if **volume** is not an open Filing volume; **NSVolumeControl.Error [invalidName]** is raised if **volumeName** is not a valid volume name (e.g., it has a null local component, or is an invalid **NSName.Name**); **NSVolumeControl.Error [nameLengthLimit]** is raised if the length of one of the fields of **volumeName** exceeds the corresponding limit. **NSVolumeControl.NameNotUnique** is raised if there is already another open volume with **volumeName**.

8.2.6 Volume scavenging

The operation NSVolumeControl.Scavenge is invoked to repair a Filing volume whose structure has been damaged. It can also be invoked to convert a Filing volume from an older, incompatible format. (The Pilot logical-volume scavenger is invoked automatically as the first part of this operation.)

```
NSVolumeControl.ScavengerOptions: TYPE = RECORD [
  rootType: NSFile.Type,
  index: IndexAttributes,
  orphanDirectoryName: NSString.String,
  orphanDirectoryType: NSFile.Type];
```

Scavenger options is used to specify the options to be used during scavenging of a volume. The field **rootType** specifies the type of the volume's root file; if the scavenger does not find such a file, it creates a new one; **index** provides parameters for the index file (see §8.2.3 for interpretation of **IndexAttributes**); **orphanDirectoryName** and **orphanDirectoryType** specify attributes of an orphan directory, if one is needed. An orphan directory is a directory into which the scavenger places orphan files; that is, permanent files whose parent file cannot be determined. If an orphan directory is needed, it is created in the root of the file system with the specified name and type.

```
NSVolumeControl.Scavenge: PROCEDURE [
  volume: Volume.ID, options: ScavengerOptions, logVolume: Volume.ID]
  RETURNS [logFile: File.File];
```

Arguments: **volume** is the volume to be scavenged. It may not be the system volume and may be open. After scavenging, the volume is closed. **options** specifies the options for the scavenge; **logVolume** indicates the Pilot logical volume on which the scavenger log is created; it must be open if it is not the same as the volume being scavenged.

Results: **logFile** is the permanent Pilot file which is created on **logVolume**; it contains a description of all files and problems found by the scavenger.

Errors: If the log generated by the Pilot scavenger is invalid, NSVolumeControl.Error [badPilotLog] is raised. If **volume** is the system volume, NSVolumeControl.Error [cannotScavengeSystemVolume] is raised. If the log file cannot be written, NSVolumeControl.Error [cannotWriteLog] is raised. If **logVolume** differs from **volume** and is not an open Pilot logical volume, NSVolumeControl.Error [logVolumeNotOpen] is raised. If the Pilot scavenge was completed but the scavenged volume could not be opened, NSVolumeControl.Error [pilotScavengeFailed] is raised. If the Pilot scavenge could not be completed, NSVolumeControl.Error [pilotScavengerError] is raised. If **volume** is not a known Pilot logical volume, NSVolumeControl.Error [unknownPilotVolume] is raised. If the volume is being converted from an older, incompatible format, and there is insufficient space to complete the conversion, NSVolumeControl.Error [insufficientSpaceForConversion] is raised. If the Pilot conversion revealed that the volume was in an inconsistent state prior to the start of conversion, then NSVolumeControl.Error [runPreviousScavenger] is

raised. If the volume in unable to complete conversion to the new format, then NSVolumeControl.**Error [volumeConversionFailed]** is raised.

In addition to putting the scavenger log on **logVolume**, the scavenger also creates the data files it needs for scavenging on **logVolume**. These data files always occupy at least 166 disk pages, and occupy more space if the volume being scavenged contains more than 830 files. For volumes with more than 830 files, the amount of space required by the scavenger's data files is given by the equation:

size of data files (in pages) = 2 × (number of files on volume ÷ 10)

The format of the log file which the scavenger generates is described by the type NSVolumeControl.**Log**. The log is made up of an NSVolumeControl.**Header** followed by one or more NSVolumeControl.**Entrys**. The fields of **header** identify the volume that was scavenged (**volume**), the time that the volume was scavenged (**date**), whether the scavenge was incomplete (**incomplete**), whether the file system was repaired (**repaired**) and the number of files found in the file system (**numberOfFiles**). There is one **Entry** for each file in the file system. The entry indicates the file's ID (**file**), type (**type**) and name (**name**), the number of problems with the file (**numberOfProblems**), and an array describing the problems (**problems**). When no problems exist for a file, **numberOfProblems** is zero and the **name** and **problems** fields of the **Entry** are omitted.

Note: Currently, **repaired** is used to indicate the presence of problems within the log file; if **TRUE**, at least one problem is reported there. Also, the implementation never reports incomplete scavenges.

The size of the scavenger log depends on the number of files on the volume being scavenged, since every file has an entry in the log. The size of the log also increases as the number of files with problems increases. The scavenger log always occupies at least 10 disk pages, regardless of the total number of files or the number of files with problems. Since the size of the scavenger log and the size of the scavenger's data files are both proportional to the number of files on the volume, scavenging a larger capacity volume will generally require more free space on **logVolume** than scavenging a smaller capacity volume.

```
NSVolumeControl.Log: TYPE = MACHINE DEPENDENT RECORD [
  header(0): Header, firstEntry(SIZE[Header]): Entry]; -- other entries follow

NSVolumeControl.HeaderPointer: TYPE = LONG POINTER TO Header;
NSVolumeControl.Header: TYPE = MACHINE DEPENDENT RECORD [
  volume(0): Volume.ID,
  date(5): System.GreenwichMeanTime,
  incomplete(7:0..14), repaired(7:15..15): BOOLEAN,
  numberOfFiles(8): LONG CARDINAL];

NSVolumeControl.EntryPointer: TYPE = LONG POINTER TO Entry;
NSVolumeControl.Entry: TYPE = MACHINE DEPENDENT RECORD [
  file(0): NSFile.ID,
  type(5): NSFile.Type,
  numberOfProblems(7): LONG CARDINAL
  -- name: StringBody,
```

```
  -- problems: ProblemArray
  ];

NSVolumeControl.StringBody: TYPE = MACHINE DEPENDENT RECORD [
  length(0): CARDINAL, bytes(1): PACKED ARRAY [0..0) OF Environment.Byte];

NSVolumeControl.ProblemArray: TYPE = ARRAY [0..0) OF Problem;
NSVolumeControl.ProblemType: TYPE = MACHINE DEPENDENT{
  changedToDirectory(0), duplicatePage(1), duplicateSegmentID(2),
  fileDeleted(3), illegalAttributeValue(4), illegalAttributeValueForNonDirectory(5),
  illegalSegmentID(6), invalidAttributeValue(7), leaderExtensionDeleted(8),
  leaderExtensionMissing(9), leaderExtensionReinserted(10),
  leaderExtensionWrongType(11), loopInHierarchy(13), missingPages(14),
  orphanFile(15), orphanLeaderExtension(16), orphanPage(17),
  orphanSegment(18), segmentDeleted(19), segmentMissing(20),
  segmentReinserted(21), segmentWrongType(22), stringTooLong(24),
  tooManySegments(25), unreadablePages(26), variableAttributesBad(27),
  wrongNumberOfChildren(28), wrongSegmentID(29),
  wrongSizeInBytes(30), wrongSizeInPages(31), newRootCreated(33),
  orphanDirectoryCreated(34), (256)};

NSVolumeControl.ProblemPointer: TYPE = LONG POINTER TO Problem;
NSVolumeControl.Problem: TYPE = MACHINE DEPENDENT RECORD [
  trouble(0): SELECT problemType(0:0..15): ProblemType FROM
    changedToDirectory, duplicatePage, fileDeleted, leaderExtensionDeleted,
      leaderExtensionMissing, leaderExtensionReinserted,
      leaderExtensionWrongType, newRootCreated,
      orphanDirectoryCreated, orphanPage,
      variableAttributesBad = > [],
    duplicateSegmentID, illegalSegmentID = > [
      old(1): NSSegment.ID, changedTo(2): NSSegment.ID],
    illegalAttributeValue, illegalAttributeValueForNonDirectory = > [
      old(1): NSFile.Attribute],
    invalidAttributeValue, stringTooLong = > [type(1): NSFile.AttributeType],
    loopInHierarchy, orphanFile = > [oldParent(1): NSFile.ID],
    missingPages, unreadablePages = > [
      first(1): File.PageNumber, count(3): File.PageCount],
    orphanLeaderExtension = > [id(1): NSFile.ID],
    orphanSegment = > [id(1): NSFile.ID, segment(6): NSSegment.ID],
    segmentDeleted, segmentMissing, segmentReinserted,
      segmentWrongType = > [segment(1): NSSegment.ID],
    tooManySegments = > [oldCount(1): CARDINAL],
    wrongNumberOfChildren = > [old(1): CARDINAL, changedTo(2): CARDINAL],
    wrongSegmentID = > [inEntry(1): NSSegment.ID, inFile(2): NSSegment.ID],
    wrongSizeInBytes, wrongSizeInPages = > [
      old(1): LONG CARDINAL, changedTo(3): LONG CARDINAL],
    ENDCASE];
```

The set of problems detected by the Filing scavenger is given by **ProblemType**. The following describes the nature of each problem and the corrective action taken by the scavenger:

[**Note:** In Services 8.0, the scavenger does not report **loopInHierarchy** problems.]

changedToDirectory — A file (A), has been changed from a non-directory to a directory because another file (B) claimed to be contained within it. After scavenging, file A is a directory and contains file B.

duplicatePage — During scavenging several disk pages were discovered that claimed to be the same page of a file; the scavenger arbitrarily chooses one of these pages as being valid and the others are deleted.

duplicateSegmentID — The contents of the segment directory within a file indicated two segments with the same identifier (which must be unique for all segments of a file); one of the two is modified to the value indicated to make it unique.

fileDeleted — No corrective action was taken to save a file because of other problems encountered, so the file was deleted; in all cases, at least one other problem will accompany this one.

illegalAttributeValue — The value encountered by the scavenger for the given attribute did not represent a semantically legal value (e.g., times greater than today); the value of the attribute is reset to a default (legal) value.

invalidAttributeValue — The value encountered by the scavenger for this attribute did not represent a semantically valid value (e.g., strings with illegal characters); the value of the attribute is reset to a default (valid) value.

illegalAttributeValueForNonDirectory — The reported attribute contained a value not allowed for a file which is not a directory; the value of such an attribute is reset to a default (legal) value.

illegalSegmentID — An entry of the segment directory within a file contained an invalid value for a segment identifier; the scavenger changes the bad value to a valid and unique one.

leaderExtensionDeleted — Because of other problems, the leader extension of a file had to be deleted.

leaderExtensionMissing — A file indicated that it had an extended leader but none was found; the indication of an extended leader is reset for this file.

leaderExtensionReinserted	A leader extension file was detached from its primary file and was reattached by the scavenger.
leaderExtensionWrongType	The leader extension file indicated by the content of a file leader was not of the proper type; the bad leader extension file is deleted and the file leader is changed to indicate that the leader is no longer extended.
loopInHierarchy	A file was encountered which claimed to be a child of one of its descendants; the loop is broken.
missingPages	After reconstructing the mapping of files to the pages representing their content, the indicated pages were not found; each such page is reinitialized with null values.
newRootCreated	No root file of the specified type was found, so a new root file was created.
orphanDirectoryCreated	An orphan directory was created to hold one or more orphan files.
orphanFile	A file was encountered which had no valid parent; such a file is inserted in the orphan folder.
orphanLeaderExtension	A leader extension file was found for which no file could be found; the leader extension file is deleted.
orphanPage	During scavenging, a disk page was encountered which did not appear to belong to any file but appeared to contain data; the contents of the page are lost.
orphanSegment	No file could be found that contained a valid segment entry for the indicated segment and the file designated within the segment was not a valid file; the orphaned segment is deleted.
segmentDeleted	Because of other reported problems, it was necessary to delete the indicated segment.
segmentMissing	The segment directory of a file indicated a segment file which could not be located; the entry for such a segment is deleted from the segment directory.
segmentReinserted	The indicated segment was reinserted into the segment directory of a file.

segmentWrongType	The file designated by the content of a segment directory entry was not of the proper type; the entry is removed and the file is deleted.
stringTooLong	The value of a string attribute exceeded the maximum allowable length for string values; the value is truncated to a length not exceeding the allowable maximum.
tooManySegments	The segment directory of a file contained too many entries; the count of entries is reduced to the maximum allowed and extraneous entries are ignored.
unreadablePages	Certain pages representing the content of a file could not be read from the disk; an attempt is made to rewrite the contents of each such page to allow them to be read, but if this fails the file containing the pages will be lost.
variableAttributesBad	The storage area for variable-length attributes (e.g., string attributes such as **name** or **extended** attributes) was ill-formed and could not be recovered; previous values for these attributes are lost, and they are given default values.
wrongNumberOfChildren	The number of children indicated for a directory disagreed with the actual number found by scavenging; the value of this attribute is set to the correct value.
wrongSegmentID	The segment identifier within a segment directory entry did not agree with that contained within the segment; the identifier within the segment directory entry is changed to agree with the segment.
wrongSizeInBytes	The stored value for the size of the file in bytes did not agree with the actual number of bytes found; the value of this attribute is set to the actual number of bytes found.
wrongSizeInPages	The stored value for the size of the file in pages did not agree with the actual number of pages found; the value of this attribute is set to the actual number of pages found.

Problems relating to files that do not exist or have been deleted are logged under a special **Entry** whose **fileID** is **nullID**. If present, this entry is always the last in the log and its **numberOfProblems** is non-zero.

8.2.7 Errors

```
NSVolumeControl.Error: ERROR [type: ErrorType];

NSVolumeControl.ErrorType: TYPE = {
  alreadyInitialized, alreadyOpen, badPilotLog, cannotScavengeSystemVolume,
  cannotWriteLog, hardwareBroken, incompatibleVolume, invalidVolume,
  insufficientSpaceForConversion, invalidName, logVolumeNotOpen, needsScavenging,
  noFileSystem, notMounted, notOpen, pilotScavengeFailed, pilotScavengerError,
  runPreviousScavenger, sessionsExist, volumeConversionFailed, volumeNotFound};
```

Items of **NSVolumeControl.ErrorType** describe problems that can result from **NSVolumeControl** operations.

alreadyInitialized — The volume specified in **Initialize** already has a Filing file system, so no new file system was created.

alreadyOpen — The volume specified in **NSVolumeControl.Open** was already opened as a Filing volume.

badPilotLog — The log generated by the Pilot scavenger was invalid making it impossible for **Scavenge** to complete.

cannotScavengeSystemVolume — **Scavenge** may not be run on the system volume.

cannotWriteLog — **Scavenge** was unable to write the scavenger log.

incompatibleVolume — The specified volume has a Filing file system from an incompatible release of Filing. If it has a file system from the previous release of Filing, **Scavenge** may be used to convert the file system into a valid file system.

invalidVolume — The specified volume does not appear to have a Filing file system.

insufficientSpaceForConversion — There is not sufficient free space on the volume being scavenged to perform conversion from an old volume format.

invalidName — The specified volume name is not valid (e.g., it has a null local component, or is not a valid **NSName.Name**).

logVolumeNotOpen — The volume designated in **Scavenge** for the log file must be an open Pilot volume if it is not the volume being scavenged.

nameLengthLimit	The length of one of the fields of the specified volume name exceeds the corresponding limit for **NSName.Name**s.
needsScavenging	The specified volume is damaged and must be scavenged before it may be opened.
noFileSystem	No Filing structures were found on the volume being scavenged and no Filing file system was created by **Scavenge**.
notMounted	The specified Pilot logical volume could not be found on any on-line physical volume.
notOpen	The specified Filing volume was not open.
pilotScavengeFailed	The Pilot scavenger completed, but was unable to repair the volume. Consequently, the Filing volume cannot be opened or scavenged.
pilotScavengerError	The Pilot scavenger did not run to completion.
runPreviousScavenger	When converting a volume from an old format, the pilot conversion phase revealed that the volume was in an inconsistent state prior to starting the conversion. The previous version of the Pilot scavenger should be run on the volume before proceeding with conversion to the new format.
sessionsExist	The specified volume cannot be closed because of existing sessions to its corresponding service.
volumeNotFound	There is no open volume having the specified name.

Note: In Services 8.0, **NSVolumeControl.Error[noFileSystem]** is not raised.

If a volume is being opened, or the name of a volume is being initialized or changed, then **NSVolumeControl.NameNotUnique** is raised if there is already an open Filing volume with the same local name.

NSVolumeControl.NameNotUnique: ERROR [name: NSName.Name];

The storage allocated to **name** belongs to the file system and should *not* be deallocated by the client.

Appendix A
References

[1] *Authentication Programmer's Manual*, Version 8.0, November 1984.

[2] *Authentication Protocol*, XSIS 098404, April 1984. Xerox System Integration Standard; Stamford, Connecticut.
[AUTHENTICATION: Describes the Authentication protocol and Courier program.]
[PRINTING: (optional) Defines the hashing algorithm used for the **releaseCode** argument to **Print**. An implementation of the hashing algorithm is contained in the **NSName** interface.]

[3] *Bulk Data Transfer* (Appendix F to *Courier*), XSIS 038112, Addendum 1a, April 1984. Xerox System Integration Standard; Stamford, Connecticut.
[PRINTING: (optional) Defines the protocol used for the transmission of the Interpress file. An implementation of it is provided by the **NSDataStream** interface.]

[4] *Character Code Standard*, XSIS 058404, April 1984. Xerox System Integration Standard; Stamford, Connecticut.
[FILING: (optional) Defines the representation and encoding of characters and sequences of characters. A programmer will be concerned with this document only if the details of internal string format are required.]

[5] *Clearinghouse Entry Formats*, XSIS 168404, April 1984. Xerox System Integration Standard; Stamford, Connecticut.
[CLEARINGHOUSE: (optional) Describes the well-known property ID's and their associated data formats.]

[6] *Clearinghouse Functional Specification*, Version 8.0, July 1984.
[AUTHENTICATION: Describes the Authentication functional specification, in addition to that of the Clearinghouse.]

[7] *Clearinghouse Programmer's Manual*, Version 8.0, November 1984.
[AUTHENTICATION: Describes the Clearinghouse, an example of a client of the Authentication Service.]

[8] *Clearinghouse Protocol*, XSIS 078404, April 1984. Xerox System Integration Standard; Stamford, Connecticut.
[AUTHENTICATION: Describes the Clearinghouse, an example of a client of the Authentication Service.]
[CLEARINGHOUSE: (optional) Describes the Clearinghouse Service remote program protocol. Would be of interest to anyone who must implement their own Clearinghouse stub but is not required reading for clients of the Mesa stub.]

[9] *Common Facilities Programmer's Manual*, Version 8.0, November 1984.

[10] *Courier: The Remote Procedure Call Protocol*, XSIS 038112, December 1981. Xerox System Integration Standard; Stamford, Connecticut.
[CLEARINGHOUSE: Describes the remote procedure call protocol and a machine-independent representation for data.]
[PRINTING: (optional) Defines the transmission protocol for the procedures and arguments supported by this interface.]

[11] *External Communication Programmer's Manual*, Version 8.0, November 1984.

[12] *Filing Programmer's Manual*, Version 8.0, November 1984.

[13] *Filing Protocol*, XSIS 108210, October 1982. Xerox System Integration Standard; Stamford, Connecticut. NOTE: Xerox Private Data.
[FILING: (optional) Defines the protocol used for communication between connected file systems. A programmer should refer to this document only if the details of file or attribute serialization are needed, as when dealing with extended attributes or serialized files.]

[14] *Gateway Software Design Specification*, Version 7.0, April 1981.

[15] *Internet Transport Protocol*, XSIS 028112, December 1981. Xerox System Integration Standard; Stamford, Connecticut.
[CLEARINGHOUSE: Defines the various levels of the Internet protocols below Courier.]

[16] *Interpress 82 Electronic Printing Standard*, XSIS 048201, January 1982. Xerox System Integration Standard; Stamford, Connecticut. NOTE: Xerox Private Data.
[PRINTING: (optional) Defines the language and data encoding contained in the files accepted as part of the **Print** procedure, transmitted via the bulk data transfer protocol.]

[17] *Interpress 82 Reader's Guide*, XSIG 018404, April 1984. Xerox System Integration Guide; Stamford, Connecticut.

[18] *Interpress Electronic Printing Standard*, Version 2.1, XSIS 048404, April 1984. Xerox System Integration Standard; Stamford, Connecticut.

[19] *Interpress Programmer's Manual*, Version 8.0, November 1984.

[20] *Introduction to Interpress*, XSIG 038404, April 1984. Xerox System Integration Guide; Stamford, Connecticut.
[INTERPRESS: A very useful aid to understanding the standard and creating Interpress m]asters.

[21] *Mailing Programmer's Manual*, Version 8.0, November 1984.

[22] *Mesa 6.0 Compiler Update*, October 1980.
[EXTERNAL COMMUNICATION: (mandatory) Contains extensions to Mesa 5.]

[23] *Mesa Language Manual*, Version 11.0, June 1984.
[EXTERNAL COMMUNICATION: (mandatory) Reference manual for the Mesa programming language.]

[24] *OIS Architectural Principles*, January 1976.

[25] *PhoneNet Driver Programmer's Manual*, Version 8.0, November 1984.

[26] *Pilot Programmer's Manual*, Version 11.0, May 1984.
[EXTERNAL COMMUNICATION: (mandatory) Reference manual for the Pilot operating system (see especially section 3 on Streams).]
[FILING: (optional) Reference manual for the Pilot operating system. A programmer who uses segment mapping or stream operations should refer to this document.]

[27] *Print Service 8.0 (OS 5.0) Interpress Product Description.*
[INTERPRESS: Describes the limits of the Interpress implementation provided by PS 8.0 servers.]

[28] *Printing Programmer's Manual*, Version 8.0, November 1984.

[29] *Printing Protocol*, XSIS 118404, April 1984. Xerox System Integration Standard; Stamford, Connecticut.
[PRINTING: (optional) Defines the Courier-based protocol which provides the standard model upon which the implementation of this interface is based.]

[30] *Printing System Interface Standard.*
Internal document, in the process of being released as a standard.
[INTERPRESS: Contains specific standards for font naming and other related issues.]

[31] *Synchronous Point-to-Point Protocol.*
Internal document, in the process of being released as a standard.
Version 3.0, November 1983.

[32] *Time Protocol*, XSIS 088404, April 1984. Xerox System Integration Standard; Stamford, Connecticut.

XEROX

B

Appendix B
Gateway Access Protocol (GAP) Programmer's Manual

November 1984

PRELIMINARY

Xerox Corporation
Office Systems Division
3450 Hillview Avenue
Palo Alto, California 94304

Appendix B consists of six pages that describe the changes between GAP version 3 and GAP version 2, followed by the Specification for GAP version 2.

Gateway Access Protocol version 3 changes

1 Introduction

This document describes the changes to the Gateway Access Protocol (GAP), version 3. The changes since version 2 relate primarily to four areas:

- Asynchronous virtual terminal circuits between system elements
- Access control for gateway resources
- SNA 3270
- New foreign devices types to allow client setting of asynchronous flow control options

Since GAP is a Courier-based protocol, implementations of GAP may support multiple versions. The changes described here allow such implementations. It is recommended that all products supporting either the user or server side of GAP provide backwards compatibility with version 2.

Greeters have been added to the architecture. A greeter is the initial switching point through which external terminals establish GAP connections to network resources, such as interactive network application gateways and server executives. The external terminal user interacts with a greeter to specify the network resource of interest. The greeter establishes the GAP connection and then becomes transparent to the terminal and terminal user.

2 Overview of changes

2.1 TTY service

This new transport type was added to allow asynchronous virtual terminal connections to services on the network. A TTY service is any system element that can present an asynchronous terminal interface, in particular, a simple TTY-like interface. Some examples are the Interactive Terminal Service, network services provided remote system administration, Xerox Development executives, and networked host systems. An ID field is passed to select the asynchronous terminal service on the server.

2.2 Access control and authentication

To allow a GAP service implementation to restrict access to certain RS-232-C ports and IBM3270 cluster ports, access control and authentication information has been added to the Create procedure. The format of this information is defined by the Authentication Protocol Specification. Clients may use any level of authentication. Clients using strong authentication will need to obtain the resource's Clearinghouse name.

2.3 Setting flow control parameters during Create

Some RS-232-C hardware implementations allow a user to change the type of flow control supported when **Create** is called. To enable flow control information to be passed in the **Create** call, two new foreign device types were added, **newTty** and **newTtyHost**. These new types differ from **tty** and **ttyHost** only in the addition of a flow control parameter.

2.4 3270 Read modified support

Controls **readModifiedAll3270** and **readBuffer3270** have been added to support 3270 Read commands. Support for these controls was actually added before GAP version 3; however, not all GAP version 2 services support them.

2.5 SNA 3270 support

These new transport types **sdlcTerminal**, **polledBSCPrinter**, and **sdlcPrinter** have been added. Only **sdlcTerminal** is fully supported by GAP 3.0. **sdlcTerminal** is the transport type used when networked workstations access an SNA 3270 gateway. Data for SNA 3278 terminal emulations running on workstations is encoded on the sequenced packet protocol connection in a manner almost identical to BSC 3270 terminals. The only difference is the treatment of SSCP-LU session exchanges. Data flowing on the SSCP-LU session is marked with SPP packet subtype **sscpData** and character coded, rather than 3270 data stream. [**Note**: Gateways supporting **sdlcTerminal** may also support **polledBSCTerminal** as a backward compatibility measure. Such a gateway would convert data on the SSCP-LU session from character coded to a 3270 datastream.] New controls, **puActive** and **puInactive**, have been added to indicate when the SNA PU-SSCP session is active.

2.6 Additional error reasons

serviceTooBusy, **serviceNotFound**, **userNotAuthenticated**, and **userNotAuthorized** were added to handle conditions arising from the new TTY Service and access control.

2.7 Simplified termination

To terminate the data transfer phase of a session, the two sides of the connection exchange cleanup attention packets and then follow the close protocol described in §7.5 of *Internet Transport Protocol* [15]. Upon completion of the close protocol, the sequenced packet protocol connection is passed back to the Courier implementation for possible reuse by another session.

2.8 Reset and Delete procedures

These procedures are not being used now, but have been left in the protocol for future use.

3 Procedures removed

3.1 GAP callback protocol

The procedures in this protocol were used to reserve ports. This capability has been removed from the protocol since its functionality is now performed by greeters.

3.2 Reserve and IAmStillHere procedures

The functionality of these procedures was also replaced by greeters. These procedures have been removed from the protocol.

3.3 UseMediumForOISCP procedure

No client ever used this procedure remotely, so it has been removed from the protocol .

4 Creating a session in GAP 3

With the removal of the reserving functionality from GAP, **Create** becomes the main procedure. Here is the Courier definition of the **Create** procedure. Other definitions are included only if they have changed since GAP 2. [**UNDERLINED BOLD** is used to indicate changes].

```
Create: PROCEDURE [
  sessionParameterHandle: SessionParamObject,
  transportList: SEQUENCE OF TransportObject,
  createTimeout: WaitTime,
  credentials: Credentials,
  verifier: Verifier]
  RETURNS [session: SessionHandle]
  REPORTS [
    badAddressFormat,
    controllerAlreadyExists, controllerDoesNotExist,
    dialingHardwareProblem,
    illegalTransport, inconsistentParams,
    mediumConnectFailed,
    noCommunicationHardware, noDialingHardware,
    terminalAddressInUse, terminalAddressInvalid,
    tooManyGateStreams, transmissionMediumUnavailable,
    serviceTooBusy, userNotAuthenticated, userNotAuthorized,
    serviceNotFound] = 2;

Credentials: TYPE = ...  -- See Authentication specification

Verifier: TYPE = ...  -- See Authentication specification
```

```
SessionParameterObject: TYPE = CHOICE OF {
   xerox800(0) = > NULL,
   xerox850(1), xerox860(2) = > [pollProc: UNSPECIFIED],
   system6(3), cmcII(4), ibm2770(5), ibm2770Host(6), ibm6670(7), ibm6670Host(8)
      = > [sendBlocksize, receiveBlocksize: CARDINAL],
   ibm3270(9), ibm3270Host(10) = > NULL,
   oldTtyHost(11) ,oldTty(12) = > [
      charLength: CharLength,
      parity: Parity,
      stopBits: StopBits,
      frameTimeout: CARDINAL], -- milliseconds
   other(13) = > NULL,
   unknown(14) = > NULL,
   ibm2780(15), ibm2780Host(16), ibm3780(17), ibm3780Host(18) = > [
         sendBlocksize, receiveBlocksize: CARDINAL],
   siemens9750(19), siemens9750Host(20) = > NULL,
   ttyHost(21) ,tty(22) = > [
      charLength: CharLength,
      parity: Parity,
      stopBits: StopBits,
      frameTimeout: CARDINAL], -- milliseconds,
      flowControl: FlowControl};

FlowControl: TYPE = RECORD[
   type: {none(0), xOnXOff(1)},
   xOn: UNSPECIFIED,
   xOff: UNSPECIFIED];

TransportObject: TYPE = CHOICE OF {
   rs232c(0) = > [
      commParams: CommParamObject,
      preemptOthers, preemptMe: ReserveType,
      phoneNumber: STRING,
      line: CHOICE OF {
         alreadyReserved = > [resource: Resource],
         reserveNeeded = > [lineNumber: CARDINAL]}],
   bsc(1) = > [
      localTerminalID: STRING,
      localSecurityID: STRING,
      lineControl: LineControl,
      authenticateProc: UNSPECIFIED,
      bidReply: BidReply,
      sendLineHoldingEOTs: ExtendedBoolean,
      expectLineHoldingEOTs: ExtendedBoolean],
   teletype(2) = > NULL,
   polledBSCController(3), sdlcController(5), ~~polledSDLCController~~ = > [
      ..not supported via GAP. Used by local service to initialize driver ],
   polledBSCTerminal(4), sdlcTerminal(6), ~~polledSDLCTerminal~~ = > [
      hostControllerName: STRING,
      terminalAddress: TerminalAddress],
   service(7) = > [
      id: LONG CARDINAL],
```

```
    unused(8) => NULL,
    polledBSCPrinter(9), sdlcPrinter(10) => [
        hostControllerName: STRING,
        printerAddress: TerminalAddress]};

BidReply: TYPE = {wack(0), nack(1), default(2)};

ExtendedBoolean: TYPE = {true(0),false(1), default(2)};

DeviceType: TYPE = {undefined(0), terminal(1), printer(2)};


CommParamObject: TYPE = RECORD [ -- Only change is tag position
    accessDetail: CHOICE OF {
        directConn => [
            duplex: {full(0), half(1)},
            lineType: LineType,
            lineSpeed LineSpeed],
        dialConn => [
            duplex: {full(0), half(1)},
            lineType: LineType,
            lineSpeed LineSpeed,
            dialMode: {manual(0), auto(1)},
            dialerNumber: CARDINAL,
            retryCount: CARDINAL};

LineSpeed: TYPE = {
    bps50(0), bps75(1), bps110(2), bps135p5(3), bps150(4),
    bps300(5), bps600(6), bps1200(7), bps2400(8), bps3600(9),
    bps4800(10), bps7200(11), bps9600(12), bps19200(13),
    bps28800(14), bps38400(15). bps48000(16), bps56000(17),
    bps57600(18)};


LineType: TYPE = { -- Note this type incorrectly defined in some GAP 2.0 documentation
    bitSyncrhnous(0), byteSynchronous(1), asynchronous(2), autoRecognition(3)};
```

The field **phoneNumber** specifies the phone number for a Direct Distance Dial (DDD) network. Some additional values were defined to allow all possible digits to be dialed and all dialer functionality to be used. The phone number is a string of ASCII characters (31 characters maximum) from the set

0 1 2 3 4 5 6 7 8 9 * # < > = A B C D E F **[A-F are new for version 3.0]**

representing the digits to be dialed. The character < represents Tandem Dial, the character > represents Delay, and the character = represents EON (End-Of-Number). The Tandem Dial or Delay digit may appear at any place in the string as required by the telephone exchanges being accessed. Tandem Dial causes the Dialer to await the next Dial Tone before dialing subsequent digits while the Delay digit causes the Dialer to wait six (6) seconds before dialing subsequent digits. (The Delay digit is designed to be used in place of Tandem Dial on Dialers that cannot detect Dial Tone.) The EON digit, if present, must be the last digit in the string. This digit causes the Dialer to transfer control to the Modem. The Modem then has the responsibility for detecting Answer Tone. In the absence of the EON digit, transfer is made automatically upon detection and processing of Answer Tone.

An empty string is specified if dialing is to be performed manually or not at all. The characters A, B, C, D, E, and F allow the client to dial codes above 9 that particular dialers may use to enable special non-standard function. A = 10, B = 11, C = 12, D = 13, E = 14, and F = 15.

5 New generic controls

The following SPP packet subtypes have been added:

sscpData = 342_8

readModifiedAll3270 = 344_8

read3270 = 345_8

6 New status values

The following SPP attention bytes have been added:

puActive = 347_8

puInactive = 350_8

XEROX

Gateway Access Protocol (GAP) Specification

Version 5.0 (Protocol Version 2)
February 1983
Revised July 1984

Xerox Corporation
Office Systems Division
Systems Development Department
Palo Alto, California 94304

Table of contents

Appendix

1

Introduction

This document describes the *Gateway Access Protocol* (GAP). The Gateway Access Protocol provides remote access to the transport service supporting communication with foreign systems. A *foreign system* is any hardware or software entity that does not implement the Xerox Network Systems (NS) Internet Transport Protocols. The Gateway Access Protocol provides *at least* a reliable transport across the communication medium connecting the foreign system to the system element providing the transport service. Additional functions may be provided depending on the foreign system.

1.1 Goals

The goals for the Gateway Access Protocol are:

1) Move information over distances.

Moving information over distances is the traditional role of a communication facility. The transport service must provide a model of transport services that allows transmission of information across many types of transmission media, both virtual and real, configured in a variety of topologies.

2) Support many user and application models of communication.

The list of possible user and application communication models is quite long. Examination of a few applications reveals how they are similar. Electronic mail applications suggest a document transfer communication model. Remote access to a data base system often suggests a transaction-oriented model. Interface to a foreign EDP system could suggest an interactive communication model. In gross terms, the variables that capture the differences of each of these models are the *unit of data transfer* and the *frequency of transmission activity* in each direction. To support many communication models, a protocol must provide flexible control of the unit of data transfer and the frequency of transmission.

The above communication models are independent of the content of the data. The content of data passed between a GAP server and a foreign system is extremely application-dependent and of little interest to the transport service. Thus, the Gateway Access Protocol provides *information transcription*, but **not** *information translation*. Information transcription means transferring information from one system to another, performing necessary blocking and unblocking as required by the limitations of the communicators.

The Gateway Access Protocol does *not* provide information translation, which includes format changes on the information or any changes that would affect presentation of the information to the client.

3) Resolve disparities among the communication methods used by foreign systems.

Two complimentary strategies are used to resolve the differences in foreign system communication methods. First, where possible, the most standard communication conventions are used. If many foreign systems communicate using convention (protocol) A, then convention A is supported. It is assumed that no modification of a foreign system is possible or should be necessary (beyond the amount needed to operate the device locally) in order to communicate with it. Foreign systems will not be altered to conform to NS Internet communication conventions; rather, the GAP server must adapt to the conventions of communication defined by the foreign systems. The Gateway Access Protocol provides adaptation to foreign protocols.

The second strategy is to isolate those communication characteristics of a foreign system that are device-specific. Of those characteristics, the ones which can be altered by a local user of the foreign system may be specified by the Gateway Access Protocol client. Other characteristics will be considered to be constant.

1.2 Definition of terms

auto-recognition — *Auto-recognition* is the ability to identify the foreign correspondent through a combination of hardware and software, thus allowing more flexible use of a single line. This capability is not provided by all GAP servers since the necessary hardware may not exist.

controls — *Controls* are directives passed over transmission media for the establishment, maintenance, and termination of communication channels.

data — *Data* is a sequence of bits transferred between end users of a logical communication channel; sometimes called *text*.

generic controls — *Generic controls* are a set of universal device- and protocol-independent directives that can be mapped into/from real device or protocol controls.

information transcription — *Information transcription* is the transfer of information from one physical system to another, often requiring reblocking.

information translation — *Information translation* is the altering of information contained in one format by expressing it in another format.

foreign system — A *foreign system* is a hardware/software entity that communicates using conventions other than internet communication protocols.

protocol

A *protocol* is a set of conventions, particularly the allowed formats and sequences of communication, between two communicators.

protocol layering

Protocol layering is a technique of hierarchically structuring protocols such that the protocol at layer *n* uses the protocol at layer *n-1* as a transmission service without knowing the details of its operation. It allows convenient partitioning, independence of activities between layers, and the sharing of common services among different served protocols.

session

A *session* is an association between a GAP client and the foreign system, by which the exchange of information is managed.

transmission medium

The *transmission medium* is the lowest level physical transport mechanism, *e.g.*, leased lines, DDD circuit, and the Ethernet; also, a virtual transport mechanism.

transport

A *transport* is an entity that implements one layer of a transport service. The entity usually corresponds to the implementation of one layer of protocol.

transport service

A *transport service* is a set of functions offered via an interface that provides transparent transfer of data between a client and a correspondent at the same level. A transport service may be made up of many levels of transport.

2

Overview

The Gateway Access Protocol (GAP) is built upon the Courier and Sequenced Packet Protocols. The Courier Protocol provides the procedure-like interface used to set up the session with the foreign system. After the session has been established, the Sequenced Packet Protocol provides reliable, full-duplex transmission of data, methods for passing control information (packet subtypes), and an out-of-band signalling mechanism (attention bytes).

[In several places references are made to current limitations or to possible future developments and extensions to the models and features discussed. Such references appear in this font.]

2.1 Sessions

A session is a cooperative association between the GAP client and a foreign system. It is the umbrella of communication management under which information exchange occurs. A client can be either the *active* or *passive* participant in the session. When a client is the active participant, the session begins when the foreign system accepts an attempt to start the session. When a client is passive, a session begins when a foreign system actively tries to start a session with a waiting (listening) GAP client.

To start a session, the following questions must be answered by either the GAP server or its client: What is the type of the foreign system? Where is it? What are its unique communication needs? What transport services are to be used? How are the chosen transport services used?

2.2 Types of foreign systems

Foreign system *types* generally correspond to product names. Associated with each type is a set of static characteristics that describes the behavior of the foreign system. A few of the static characteristics are: variations in the use of a protocol (e.g., timeouts), how the foreign system supports setting of its own communication parameters (e.g., set or exchanged remotely during session establishment or set by the operator), and the code set used (if only one is supported).

Currently, communication with the following foreign system types is supported: Xerox 850 IPS, Xerox 860 IPS, IBM Communicating Magnetic Card (CMC) II Typewriter, IBM Office System 6, IBM 3270 hosts, and Teletype-compatible terminals and hosts. The IBM 2770 and IBM 6670 are supported experimentally, being treated exactly like an IBM Office System 6.

2.3 Transport service

GAP allows communication over a layered transport service. A *transport service* has *n* levels of virtual transports layered above some physical transmission medium transport. A transport service offers a communication facility that is transparent to its clients, that is, the client does not need to know the details of how the transport service provides the communication facility.

The client is responsible for defining the transports to be used in providing the transport service. This includes providing access information and other transport-dependent information. The GAP server is responsible for making the transports and transmission medium cooperate. It also makes the transports conform to any static device-specific conventions, such as timeouts and block sizes.

2.3.1 The transports

A transport is a single layer of a transport service. It usually implements a protocol. A *protocol* is a set of conventions, especially the formats and allowed exchanges, used by communicating correspondents. A transport satisfies the layering requirement by providing an interface to an entity that implements a set of functions. The functions are usually related to data and control exchange and session management. A transport can be viewed as communicating with transport entities in the foreign system. [In the future a transport may communicate with transport entities somewhere in interconnected transmission media.]

For the simplest cases, there are only two transports: a block transport and a physical transmission medium transport. For example, when communicating with an IBM Office System 6, there is a Binary Synchronous (BSC) transport and an RS-232-C transmission medium transport. The BSC transport can be thought of as logically exchanging blocks with a BSC transport in the IBM Office System 6. The RS-232-C transport can be thought of as logically exchanging bits with a similar entity in the IBM Office System 6. The client must define the appropriate transport parameters, as well as the hierarchical relationship among them.

[Further layering of transports will occur for one of two reasons. First, the foreign system might be a sophisticated EDP machine that uses layering of transports for modularity, portability, ease of implementation, etc. The more common layering will result when there is a concatenation of transmission media with intermediate access procedures and/or protocols required.]

The transport service model is complicated by the fact that the lowest-level transmission medium may itself be concatenated with other physical media. An example of this is access to a foreign system which is a host on a packet-switched network. First there is dial up through the Direct Distance Dialing (DDD) network to an access node on a packet-switched network. The access node may require further access information to complete the connection to a foreign system. Finally, communication with a process on the foreign system may require using a BSC transport. When concatenation of transmission media occurs, transports are used to handle each level of media that requires protocol interaction.

2.3.2 The physical transmission medium

In the model of a layered transport service, the physical transmission medium is the lowest level communication facility provided. The GAP server is directly connected to the medium. To describe a transmission medium transport, the client provides transport-specific access information, parameters that are used for resolving contention for the transmission medium interface, and information about how to use the medium. The only transmission medium currently supported by the GAP server is an RS-232-C controller port.

For RS-232-C ports, the access information is a telephone number. Dedicated or leased lines require no transmission medium access information.

RS-232-C port reservation is supported by allowing clients to specify reservation priorities. The reservation parameters allow clients to reserve a communication medium exclusively or to reserve use of the medium for low priority activity which can be preempted for higher priority use.

2.4 Sending/receiving data and controls during a session

Once the transport service has been configured and a session has begun, the client can exchange data and control the interaction with the foreign system. Client data and control information is sent and received by the client over the Courier *system data* stream. Client data and control information is neither sent nor received when there is a Courier remote procedure call outstanding.

Controls are directives or commands that are exchanged by communicating entities to support smooth, orderly, and reliable information exchange. A foreign system may be capable of exchanging a variety of controls. The controls supported by GAP are those that affect the flow of data and the management of the session.

Controls are needed for stopping the output of a verbose sender. They are needed for interrupting the sender so that the receiver can change recording media; likewise, for resuming transmission. For alternating communication, a control allows the sender to inform the receiver that it can now send.

To provide a uniform way of sending and receiving controls, GAP defines a set of universal or *generic* controls to and from which most foreign system-specific controls can be mapped. The client sends and receives generic controls using packet subtypes and the Attention facility of the Sequenced Packet Protocol.

2.5 Terminating the session

GAP allows two kinds of session termination by the client. The client may abruptly terminate the session by deleting the session. This method may result in lost data and possibly abnormal operation of more primitive foreign systems. Alternatively, the client may terminate the session gracefully before deleting the session, which will result in the orderly termination of a session and no lost data.

3

Client interface

The remote procedures and errors defined below comprise a Courier remote program called **GAP**.

```
GAP: PROGRAM 3 VERSION 2;
```

3.1 Reset

Reset frees all sessions and resources that are assigned to the system element making the call.

```
Reset: PROCEDURE = 0;
```

3.2 Creating a session

To create a session, the client calls **Create**. If successful, **Create** returns a handle that is used in subsequent calls to identify that particular session.

```
Create: PROCEDURE [
    sessionParameterHandle: SessionParamObject,
    transportList: SEQUENCE OF TransportObject,
    createTimeout: WaitTime]
    RETURNS [session: SessionHandle]
    REPORTS [
        badAddressFormat,
        controllerAlreadyExists, controllerDoesNotExist,
        dialingHardwareProblem,
        illegalTransport, inconsistentParams,
        mediumConnectFailed,
        noCommunicationHardware, noDialingHardware,
        terminalAddressInUse, terminalAddressInvalid,
        tooManyGateStreams, transmissionMediumUnavailable
    ] = 2;

SessionHandle: TYPE = ARRAY 2 OF UNSPECIFIED;
```

sessionParameterHandle specifies a set of device-specific session characteristics (see §3.2.1). **transportList** is an array descriptor describing each of the layers of the transport (see §3.2.2). **createTimeout** specifies an activation timeout. If **createTimeout** seconds elapse before the session has been created, the error **mediumConnectFailed** is reported.

```
WaitTime: TYPE = CARDINAL; -- in secs

infiniteTime: WaitTime = LAST[CARDINAL];
```

If a new session cannot be created due to lack of some system resource, the error **tooManyGateStreams** is reported. A session is terminated and its session handle invalidated by calling **Delete** (see §3.5).

3.2.1 Session parameters

A **SessionParameterObject** describes a set of device-specific session characteristics.

```
SessionParameterObject: TYPE = CHOICE OF {
    xerox800 => NULL,
    xerox850, xerox860 => [pollProc: UNSPECIFIED],
    system6, cmcII, ibm2770, ibm2770Host, ibm6670, ibm6670Host => [
        sendBlocksize, receiveBlocksize: CARDINAL],
    ibm3270, ibm3270Host => NULL,
    ttyHost, tty => [
        charLength: CharLength,
        parity: Parity,
        stopBits: StopBits,
        frameTimeout: CARDINAL], -- milliseconds
    other => NULL,
    unknown => NULL};
```

The designator of a **SessionParameterObject** specifies the foreign device type. **unknown** and **other** are reserved for testing new devices. The text **Host** in the device type indicates that the GAP client is communicating with a host *as though it were* the foreign device type named rather than communicating *with* the device type named. For example, **ibm3270Host** indicates the client is communicating with a host machine *as though it were* an IBM 3270 terminal while **ibm3270** indicates that the client is communicating *with* an IBM 3270 terminal.

Note: Foreign device types of **ibm2770**, **ibm2770Host**, **ibm6670**, and **ibm6670Host** are treated exactly as though the foreign device type were **system6**. Although we believe this to be correct, actual testing with these devices has not occurred.

If the foreign device is a **xerox850** or **xerox860**, the field **pollProc** specifies a handle to be passed to the client when a file poll is received. The handle is passed to the client by the GAP server calling the procedure **FilePoll** on the client's system element using the GAPCallback protocol (see §3.7). If **pollProc** is equal to **NopPollProc**, then all polls are negatively acknowledged without notifying the client.

```
NopPollProc: UNSPECIFIED = 0B;
```

Warning: Polling for files requires that the client implement the GAPCallback protocol and support multiple Sequenced Packet connections. If this is not possible, the client should always set **pollProc** to **NopPollProc** which will cause the GAP server to negatively acknowledge any poll requests without attempting to call the client.

If the foreign device is a **system6**, **cmcll**, **ibm2770**, **ibm2770Host**, **ibm6670**, or **ibm6670Host**, the field **blockSize** specifies the maximum block size for sending and receiving. If **blockSize** is zero, the maximum block size supported by the device is used.

If the foreign device is a **tty**, **charLength** specifies the length of a character (excluding parity, start and stop bits), **parity** specifies the parity type, and **stopBits** specifies the number of stop bits. **frameTimeout** is used to determine when input data should be returned to the client. When receiving data, if the time between successive characters is more than **frameTimeout** milliseconds, then the data received so far is returned to the client.

```
CharLength: TYPE = {five(0), six(1), seven(2), eight(3)};

Parity: TYPE = {none(0), odd(1), even(2), one(3), zero(4)};

StopBits: TYPE = {one(0), two(1)};
```

3.2.2 Defining the transport

The transport service is described by an ARRAY OF **TransportObject** with element zero of the array specifying the lowest layer, the physical transmission medium transport.

```
TransportObject: TYPE = CHOICE OF {
    rs232c => [
        commParams: CommParamObject,
        preemptOthers, preemptMe: ReserveType,
        phoneNumber: STRING,
        line: CHOICE OF {
            alreadyReserved => [resource: Resource],
            reserveNeeded => [lineNumber: CARDINAL]}],
    bsc => [
        localTerminalID: STRING,
        localSecurityID: STRING,
        lineControl: LineControl,
        authenticateProc: UNSPECIFIED],
    teletype => NULL,
    polledBSCController, polledSDLCController => [
        hostControllerName: STRING,
        controllerAddress: ControllerAddress,
        portsOnController: CARDINAL],
    polledBSCTerminal, polledSDLCTerminal => [
        hostControllerName: STRING,
        terminalAddress: TerminalAddress]};
```

Currently, only the following transport services are supported:

1) a two-level transport service whose first level is an **rs232c** transport and whose second level is either a **bsc**, **teletype**, or **polledBSCController** transport,

2) a one-level **polledBSCTerminal** transport.

Note: All variants other than **rs232c**, **bsc**, **teletype**, **polledBSCController**, and **polledBSCTerminal** are currently unimplemented.

3.2.2.1 RS-232-C transport

The **rs232c TransportObject** describes a transport layer implementing a transducer that supports RS-232-C lines:

```
TransportObject: TYPE = CHOICE OF {
    ....
    rs232c => [
        commParams: CommParamObject,
        preemptOthers, preemptMe: ReserveType,
        phoneNumber: STRING,
        line: CHOICE OF {
            alreadyReserved => [resource: Resource],
            reserveNeeded => [lineNumber: CARDINAL]}],
    ....};
```

commParams holds RS-232-C transmission medium parameters. The error **inconsistentParams** is generated if the parameters in **commParamObject** are invalid.

```
CommParamObject: TYPE = RECORD [
    duplex: {full(0), half(1)},
    lineType: LineType,
    lineSpeed LineSpeed,
    accessDetail: CHOICE OF {
        directConn => NULL,
        dialConn => [
            dialMode: {manual(0), auto(1)},
            dialerNumber: CARDINAL,
            retryCount: CARDINAL};

LineType: TYPE = {
    bitSynchronous(0), byteSynchronous(1), asynchronous(2), autoRecognition(3)};

LineSpeed: TYPE = {
    bps50(0), bps75(1), bps110(2), bps135p5(3), bps150(4),
    bps300(5), bps600(6), bps1200(7), bps2400(8), bps3600(9),
    bps4800(10), bps7200(11), bps9600(12)};
```

The **duplex**, **lineType**, and **lineSpeed** fields are used to create the RS-232-C channel. The **netAccess** and **dialMode** fields relate to the network access mode, and **dialerCount** and

retryCount are used if auto-dialing is specified. Dialing retries are made if a line is busy or there is no answer.

The two fields, **preemptOthers** and **preemptMe**, serve to establish a priority between contending RS-232-C channel clients. The state of the channel will be either *available*, *waiting for a connection*, or *active*. When a channel is available then a reserve attempt will always succeed. Otherwise, the success of the reservation will depend on the relative priorities of the current "owner" of the channel and the client trying to reserve it.

ReserveType: TYPE = **{preemptNever(0), preemptAlways(1), preemptInactive(2)}**;

The following matrix defines the result of reserving the channel given the values of the owner's **preemptMe** and the reserver's **preemptOthers**:

		Owner's preemptMe		
		Never	**If Inactive**	**Always**
Reserver's preempt-Others	**Never**	Fail	Fail	Fail
	If Inactive	Fail	Preempt (if inactive)	Preempt
	Always	Fail	Preempt	Preempt

The field **phoneNumber** specifies the phone number for a Direct Distance Dial (DDD) network. For the local RS-232-C/RS-366 port on an 8000 server, it is a string of ASCII characters (31 characters maximum) from the set

0 1 2 3 4 5 6 7 8 9 * # < > =

representing the digits to be dialed. The character < represents Tandem Dial, the character > represents Delay, and the character = represents EON (End-Of-Number). The Tandem Dial or Delay digit may appear at any place in the string as required by the telephone exchanges being accessed. Tandem Dial causes the Dialer to await the next Dial Tone before dialing subsequent digits while the Delay digit causes the Dialer to wait six (6) seconds before dialing subsequent digits. (The Delay digit is designed to be used in place of Tandem Dial on Dialers that cannot detect Dial Tone.) The EON digit, if present, must be the last digit in the string. This digit causes the Dialer to transfer control to the Modem. The Modem then has the responsibility for detecting Answer Tone. In the absence of the EON digit, transfer is made automatically upon detection and processing of Answer Tone. An empty string is specified if dialing is to be performed manually or not at all.

For a port on a Xerox 873 Communication Interface Unit speaking either a Racal-Vadic or Ventel specific protocol, **phoneNumber** is a string of ASCII characters (29 characters maximum) from the set

0 1 2 3 4 5 6 7 8 9 * # <

The Xerox 873 is responsible for waiting for a Dial Tone between the Tandem Dial digit and the subsequent digit, even if Tandem Dialing is not supported by its dialing hardware. When hardware assist is not available, a delay of six (6) seconds is used. The options Delay and EON are not supported.

line specifies the RS-232-C line number. The **alreadyReserved** choice is used when the client has already reserved the line using **Reserve** (see §3.3).

If no RS-232-C hardware exists or if the client selects an invalid line number, the error **noCommunicationHardware** is reported. If the channel is active and reservation (preemption) fails, the error **transmissionMediumUnavailable** is reported.

3.2.2.2 BSC transport

The **bsc TransportObject** describes a transport level which supports the transfer of data using point-to-point BSC protocol to and from the following types of communicating word and data processing systems: Xerox 800, Xerox 850, Xerox 860, IBM Office System 6, IBM CMC-II, IBM 2770, and IBM 6670. The error **inconsistentParams** is reported if the foreign device is not a device supported by this transport.

```
TransportObject: TYPE = CHOICE OF {
    ....
    bsc => [
        localTerminalID: STRING,
        localSecurityID: STRING,
        lineControl: LineControl,
        authenticateProc: UNSPECIFIED],
    ....};
```

localTerminalID and **localSecurityID** are used for authentication at the CFD (or transport node on the path to the CFD). Specification of **localTerminalID** or **localSecurityID** implies that an exchange of this information should be attempted when the connection is being established. Either or both fields can be elided by providing an empty string.

lineControl is used to determine whether the Gateway client or the CFD is to have priority when contending for control of the BSC connection. The setting of **lineControl** must be opposite of that on the CFD.

```
LineControl: TYPE = {primary(0), secondary(1)};
```

Note: **lineControl** must be set to **secondary** when communicating with a Xerox 850.

The **authenticateProc** field specifies a handle to be passed to the client by the GAP server whenever any request for authentication of terminal or security information is received from the CFD. The handle is passed to the client by the GAP server calling the procedure **Authenticate** on the client's system element using the GAPCallback protocol (see §3.7). If **authenticateProc** is equal to **NopAuthenticateProc**, all terminal and security information is positively acknowledged without calling the client.

```
NopAuthenticateProc: UNSPECIFIED = 0B;
```

Warning: Authenticating terminal and security identification requires that the client implement the GAPCallback protocol and support multiple sequenced packet connections. If this is not possible, the client should always set **authenticateProc** to **NopAuthenticateProc** which will cause the GAP server to positively acknowledge any authentication requests without attempting to call the client.

3.2.2.3 Teletype transport

The **teletype TransportObject** describes a transport which allows communication with teletype-like terminal over asynchronous lines.

```
TransportObject: TYPE = CHOICE OF {
    ....
    teletype => NULL,
    ....};
```

The error **inconsistentParams** is reported if the foreign device specified in the session parameters is not a device supported by this transport. Currently, only **ttyHost** and **tty** are supported.

3.2.2.4 PolledBSCController transport

The **polledBSCController TransportObject** describes a transport implementing a controller which communicates using the polled BSC protocol. Currently, the only device type supported for this transport is **ibm3270Host**.

```
TransportObject: TYPE = CHOICE OF {
    ....
    polledBSCController => [
        hostControllerName: STRING,
        controllerAddress: ControllerAddress,
        portsOnController: CARDINAL],
    ....};

ControllerAddress: TYPE = CARDINAL;
```

hostControllerName specifies a name for the controller. The name is formed by concatenating the following substrings into a single string:

the local name of the port (from the Clearinghouse Service RS232CPort entry)
a colon (:)
the domain name of the port (from the Clearinghouse Service RS232CPort entry)
a colon (:)
the local name of the port (from the Clearinghouse Service RS232CPort entry)
a pound sign (#)
the controller number (from the Clearinghouse Service IBM3270Host entry) expressed in octal
a capital B (B)

If a controller with that name already exists, the remote error **controllerAlreadyExists** is reported. **controllerAddress** specifies the controller's address. **portsOnController** specifies the number of terminals this controller supports.

3.2.2.5 PolledBSCterminal transport

The **polledBSCTerminal TransportObject** describes a transport implementing a terminal which communicates through a previously created **polledBSCController** transport. Currently, the only device type supported for this transport is **ibm3270Host**. This transport level must be the only one in the transport list. All other transport levels are provided by the controller specified in **hostControllerName**.

```
TransportObject: TYPE = CHOICE OF {
    ....
    polledBSCTerminal => [
        hostControllerName: STRING,
        terminalAddress: TerminalAddress],
    ....};

TerminalAddress: TYPE = CARDINAL;

unspecifiedTerminalAddress: TerminalAddress = LAST [CARDINAL];
```

hostControllerName specifies the name of a previously created **polledBSCController** transport. Case (upper/lower) is significant. If no controller with that name exists, the remote error **controllerDoesNotExist** is reported. **terminalAddress** specifies the terminal's address. If **terminalAddress** is **unspecifiedTerminalAddress**, the terminal will be assigned any available address. If the terminal address is already in use or is not in the range specified by **portsOnController** during controller creation, the errors **terminalAddressInUse** and **terminalAddressInvalid** are reported, respectively.

3.2.3 Connection establishment

Each layer of the transport service may have its own connection establishment conventions. The client has no direct knowledge of these conventions or of the actual "handshaking" that occurs during connection establishment. The client need only provide enough addressing information and the authentication procedure(s) necessary to complete the connection(s).

A client may be either the *active* or *passive* correspondent, that is, a client may either initiate a connection or wait for initiation by the CFD. The use of GAP varies slightly depending on which the client chooses. To give examples of the different possible situations that arise during connection establishment, five cases of connections are considered below:

If the client is the caller, one of the following scenarios occurs:

1) Caller using a dedicated (leased) line

In this case the line is always available and the modems are usually powered up. The algorithm allows the delayed powering up of the modem. The client sets the **phoneNumber** field to an empty string in the description of the RS-232-C transport. Since auto-dialing is not required, **Create** returns immediately. The client may await reception of the attention byte **mediumUp** to determine when the modems have been powered up and the line is ready. Data transfer operations will be accepted but will be blocked until

the line is ready. A client may set a timeout for the data transfer operation if indefinite waiting is inappropriate.

2) Caller using manual dial

The algorithm is very similar to 1). The only difference is that the action required to complete the connection is manual dialing.

3) Caller using auto-dial

The client passes a phone number in the **phoneNumber** field of the description of the RS-232-C transport. **Create** returns after the circuit has been successfully established. The error **mediumConnectFailed** is reported if dialing fails because of no answer or a busy phone. If no dialing hardware exists or the dialing hardware is malfunctioning, the error **noDialingHardware** or **dialingHardwareProblem** is reported, respectively.

If the client is a listener, one of the following scenarios occurs:

4) Listener using a dedicated line

The algorithm is very similar to 1). The **phoneNumber** field of the description of the RS-232-C transport layer is an empty string. Notification of the listen being satisfied (the other end has sent data or a control) is the completion of a data transfer operation. To abort a listen, **Delete** is called. A listen may also be performed using **Reserve**.

5) Listener using a dialed line.

Same as case 4).

3.3 Reserve

In some situations, the GAP client may wish to simply reserve many RS-232-C lines and await line activation or auto-recognition. This is accomplished by using **Reserve**. **Reserve** creates the RS-232-C channel and notifies the client when the event **callBackType** occurs by calling the remote procedure **ReserveComplete** on the client's system element using the GAPCallback protocol (see §3.7).

```
Reserve: PROCEDURE [
    transport: TransportObject,
    completionProcedure: UNSPECIFIED,
    callBackType: CallBackType]
    RETURNS [resource: Resource]
    REPORTS [
        bugInGAPCode, gapCommunicationError, gapNotExported,
        illegalTransport, inconsistentParams, noCommunicationHardware,
        tooManyGateStreams, transmissionMediumUnavailable
    ] = 4;

Resource: ARRAY 2 OF UNSPECIFIED;

CallBackType: TYPE = {callOnAutoRecognition, callOnActive, dontCall};
```

If **callBackType** is **dontCall**, the client is never notified.

If **callBackType** is **callOnActive**, the client is notified when the connection becomes active or the connection is aborted. For RS-232-C lines, Gateway software defines *becoming active* as the RS-232-C signal Dataset Ready becoming TRUE.

If **callBackType** is **callOnAutorecognition**, the client is notified when auto-recognition occurs. *Auto-recognition* is the ability to identify the foreign system type through a combination of hardware and software, thus allowing more flexible use of a single line. Auto-recognition is not offered in all implementations, since the necessary hardware may not exist on the system element providing the RS-232-C transport.

Reserve reports a subset of the errors reported by **Create**.

After a **Reserve**, the client must either free the resource by calling **AbortReserve**, use the resource in a subsequent **Create**, or pass the resource to the NS Router using **UseMediumForOISCP**.

If the client desires to abort a **Reserve** either before or after the completion procedure has been called, **AbortReserve** is called:

```
AbortReserve: PROCEDURE [resource: Resource] = 5;
```

The client may use the resource in a **Create** call by using the **alreadyReserved** choice of the **rs232c TransportObject**:

```
TransportObject: TYPE = CHOICE OF {
    ....
    rs232c = > [
        commParams: CommParamObject,
        preemptOthers, preemptMe: ReserveType,
        phoneNumber: STRING,
        line: CHOICE OF {
            ...
            alreadyReserved = > [resource: Resource]}],
    ....};
```

The client may pass the resource to the NS Router by calling **UseMediumForOISCP**. After calling **UseMediumForOISCP**, the client no longer has control over the resource.

```
UseMediumForOISCP: PROCEDURE [transport: TransportObject] = 8;
```

3.4 Transferring sessions

Some applications require that a session be transferred to another client. For example, a client communicating with a teletype may wish to act only as a simple executive, determining which service the teletype desires to use and then transferring the session to another system element that implements the service. GAP allows this using the procedures **Transfer** and **Obtain**.

Transfer is used by the owner of the **session** to indicate that the session handle is to be passed to another client. **Transfer** may be called only after the termination protocols defined in §5.5 have been followed to idle the session.

```
Transfer: PROCEDURE [session: SessionHandle] REPORTS [inconsistentParams] = 6;
```

After **Transfer** returns, the owner of **session** must pass it to the new client using a private protocol. The new client then obtains ownership of the **session** by calling **Obtain**:

```
Obtain: PROCEDURE [session: SessionHandle] REPORTS [inconsistentParams] = 7;
```

After **Obtain** returns, the new client is the owner of the **session** and can send and receive data and controls.

Warning: **Obtain** and **Transfer** are unimplemented in Version 2 and always report the error **unimplemented**.

3.5 Deleting a session

A session is deleted by calling **Delete**:

```
Delete: PROCEDURE [session: SessionHandle] = 3;
```

3.6 Freeing inactive resources

Since the resources allocated by GAP are non-sharable, GAP servers timeout and free inactive resources. In most cases, the GAP server can use Sequenced Packet connection errors as an indication that a resource is inactive. However, when an RS-232-C port is reserved using **Reserve**, this is not a valid indication of inactivity. Instead, any resource allocated by **Reserve** is freed after *n* minutes unless the **IAmStillHere** remote procedure has been called by the system element owning the resource. This rule is in effect until the resource is either freed, passed to the NS Router, or used in a subsequent **Create** procedure.

```
IAmStillHere: PROCEDURE [resource: Resource] = 1;
```

Note: *n* is currently set to be 6 minutes and it is recommended that the client call **IAmStillHere** for each reserved resource every 2 minutes.

Note: Only RS-232-C resources allocated by **Reserve** timeout after *n* minutes. Thus, any client that never calls **Reserve** need never call **IAmStillHere**.

3.7 GAPCallBack protocol

Some of the GAP remote procedures return information that may not be not immediately available. Rather than having a remote procedure call be outstanding for a long period of time, GAP "calls back" to the client using the GAPCallBack protocol. To use the protocol, the client must implement a Courier server and must support multiple Sequenced Packet connections. Clients not wishing to implement either of these may still use the GAP protocol, but must not use any of the features requiring the GAPCallBack protocol. These facilities are:

1) Authentication of security and terminal identifications. A client not implementing the GAPCallBack protocol may still communicate with systems using security and terminal identifications, but cannot authenticate them.

2) Responding positively to file polls from Xerox 850s and Xerox 860s. A client not implementing the GAPCallBack protocol may still communicate with a Xerox 850 or Xerox 860, but any file poll received will be negatively acknowledged.

3) Using **Reserve** except when **callBackType** is specified as **dontCall**.

The remote procedures and errors defined below comprise a Courier remote program called **GAPCallBack**:

```
GAPCallBack: PROGRAM 12 VERSION 2;
```

3.7.1 ReserveComplete

ReserveComplete is called when the event described by **callBackType** occurs or the resource is freed using **AbortReserve**. **callBackType** is specified during the **Reserve** call.

```
ReserveComplete: PROCEDURE [
    resource: Resource,
    callBackProcedure: UNSPECIFIED,
    results: ReserveResults] = 0;

ReserveResults: TYPE = CHOICE CallBackType OF {
    callOnAutoRecognition => [outcome(1): AutoRecognitionOutcome],
    callOnActive => [success: BOOLEAN],
    dontCall => NULL};

AutoRecognitionOutcome: TYPE = CARDINAL;
```

resource identifies the resource. **callBackProcedure** is the handle specified during the **Reserve** call. The client must not make any GAP procedure calls until it has returned Courier results for this procedure. If **callBack** is **callOnAutoRecognition**, **outcome** indicates the results of auto-recognition. If **callBack** is **callOnActive**, **success** is **TRUE** if the line became active and **FALSE** if the resource was freed by the client calling **AbortReserve**.

3.7.2 Poll

Poll is called when a file poll is received from a Xerox 850 or Xerox 860.

```
Poll: PROCEDURE [
    pollProc: UNSPECIFIED, pollString: STRING]
    RETURNS [pollResults: BOOLEAN] = 1;
```

The field **pollProc** is the handle passed during the **Create** call. The field **pollString** is the file name requested by the foreign device. The client indicates via the **pollResults** BOOLEAN whether it is prepared to send that file as soon as possible.

3.7.3 Authenticate

Authenticate is called to allow the client to authenticate any terminal or security information passed from the foreign device during connection establishment.

```
Authenticate: PROCEDURE [
    authenticateProc: UNSPECIFIED, idString: IDString]
    RETURNS [authenticateResults: BOOLEAN] = 2;
```

authenticateProc is the handle specified during the **Create** call. **iDString** indicates whether a foreign terminal or security ID is to be authenticated. The client indicates via the **authenticateResults** BOOLEAN whether it accepts the foreign terminal or security ID. The client must not make any GAP procedure calls until it has returned Courier results for this procedure. For efficient utilization of the transport medium, it is important that this procedure return the result of the authentication as quickly as possible.

```
IDString: TYPE = CHOICE OF {
    remoteTerminalID => STRING,
    securityID => STRING};
```

4

Remote errors

The following remote ERRORs are generated by the GAP remote procedures:

0 unimplemented
1 noCommunicationHardware
2 illegalTransport
3 mediumConnectFailed
4 badAddressFormat
5 noDialingHardware
6 dialingHardwareProblem
7 transmissionMediumUnavailable
8 inconsistentParams
9 tooManyGateStreams
10 bugInGAPCode
11 gapNotExported
12 gapCommunicationError
13 controllerAlreadyExists
14 controllerDoesNotExist
15 terminalAddressInUse
16 terminalAddressInvalid

badAddressFormat **4**

The specified **phoneNumber** contains either invalid characters or too may characters.

bugInGAPCode **10**

A non-recoverable error occurred due to a possible bug in the GAP code.

controllerAlreadyExists **13**

A controller with the name **hostControllerName** already exists. This error is reported when a controller is created.

controllerDoesNotExist **14**

A controller with the name **hostControllerName** does not exist. This error is reported when a terminal is created.

dialingHardwareProblem **6**

During auto-dialing, an error occurred that prevented its successful completion. This usually indicates a hardware failure of the auto-dialer.

gapCommunicationError **12**

A communication error occurred when communicating with the server.

gapNotExported **11**

GAP is not exported by the remote system element at this time.

illegalTransport **2**

The specified transport is illegal or unimplemented.

inconsistentParams **8**

The parameters specified are in error or the particular feature requested is not implemented.

mediumConnectFailed **3**

The physical connection cannot be made with the non-OIS system. This error indicates a dialing failure such as busy or no answer.

noCommunicationHardware **1**

Hardware for the specified communication line does not exist.

noDialingHardware **5**

The client specified auto-dialing but auto-dialing hardware did not exist.

terminalAddressInUse **15**

The terminal address of **terminalAddress** is already in use or all possible terminal addresses are already in use.

terminalAddressInvalid **16**

The terminal address specified in **terminalAddress** is not in the range valid for the controller.

tooManyGateStreams **9**

The procedure failed due to lack of some resource. The client should try again later.

transmissionMediumUnavailable **7**

The communication line cannot be reserved. The client should try again later.

unimplemented **0**

The procedure called is unimplemented.

5

Data transfer with the foreign system

5.1 Data transfer

Once **Create** has returned or the client has obtained the session via **Obtain**, the client uses the NS Sequenced Packet Protocol to communicate with the non-OIS system. Since Courier and the client share the same network data stream, the client must not have any Courier procedure calls outstanding during data transfer. In addition, when terminating or transferring a session, all data transfer operations must be quiesced before calling **Delete** or **Transfer** (see §5.5).

5.2 Controls

The client controls the foreign system and/or the transport through a set of *generic controls*. Generic controls may or may not translate into controls that are meaningful for the current session.

5.2.1 Classes of generic controls

The Sequenced Packet Protocol can be thought of as creating two independent duplex information channels. One channel is used mostly for transmitting data, while the other is used for transmitting attentions. There are three classes of generic controls: *in-band*, *out-of-band*, and *out-of-band with mark*. They differ in their use of the two information channels.

An *in-band* control is sent on the data channel and arrives in order relative to data. It is serialized with respect to the data sequence, because its position in the sequence indicates the relative time it was generated. Since it cannot bypass data, an in-band control will be delayed if there is congestion. An in-band control is identified via the packet subtype field. The transition from **none** to some other packet subtype value and back is the event that indicates the arrival of a control in the data sequence.

An *out-of-band* control arrives on the attention channel independently of the data channel. The attention channel is a separate, expeditious channel that is not affected by congestion of the data stream. The attention byte facility of the Sequenced Packed Protocol is used to represent out-of-band controls.

An *out-of-band with mark* control is composed of both an out-of-band control and an in-band mark. The out-of-band control is used to bypass any congestion in the data stream. The in-band mark is used to locate the position relative to the data at which the control was generated. The mark provides synchronization (*i.e.*, that the aborting condition is synchronized with respect to the sender of the abort). An out-of-band with mark control is represented via the attention byte facility for the out-of-band control and the packet subtype field for the mark.

5.2.2 Generic controls

The following are the packet subtypes which represent generic controls:

300_8	none
301_8	interrupt
302_8	resume
303_8	audibleSignal
304_8	areYouThere
305_8	iAmHere
306_8	abortGetTransaction
307_8	abortPutTransaction
310_8	endOfTransaction
311_8	yourTurnToSend
312_8	disconnect
313_8	transparentDataFollows
314_8	endOfTransparentData
315_8	abortMark
320_8	cleanup
331_8	remoteNotResponding
332_8	remoteNotReceiving
333_8	excessiveRetransmissions
334_8	chain3270 (3270 emulation only)
335_8	unchained3270 (3270 emulation only)
336_8	readModified3270 (3270 emulation only)
337_8	status3270 (3270 emulation only)
340_8	testRequest3270 (3270 emulation only)

abortGetTransaction **306_8** [Out-of-band with mark]

Immediately stop the transaction being received. The beginning of the next transaction is designated by the in-band mark **abortMark**.

abortMark **315_8** [In-band]

Marks a transaction boundary in conjunction with an **abortGetTransaction** or an **abortPutTransaction**.

abortPutTransaction **307_8** [Out-of-band with mark]

Stop the current outgoing transaction immediately and resume at the next transaction boundary, as designated by the in-band mark **abortMark**.

areYouThere 304_8 [Out-of-band]

Elicit a response confirming that the foreign device/process is operational.

audibleSignal 303_8 [Out-of-band]

Send an audible signal. For some word processors, this is a signal to "go to voice".

cleanup 320_8 [In-band, Out-of-band]

No more data will be transferred over this channel during this session. Processes receiving this control should terminate. A client wishing to terminate should generate this control on the out-of-band channel only.

disconnect 312_8 [In-band]

Perform a graceful disconnect on all transport services. This takes effect only after all data has been delivered and either a) a disconnect acknowledgement is received, b) the physical connection is broken, or c) there is no response for some protocol-dependent time. An immediate disconnect is achieved by deleting the stream.

endOfTransaction 310_8 [In-band]

Delimit a transaction. It is generally only with transports (e.g., BSC for IBM Office System6) that mix transaction management with line control.

endOfTransparentData 314_8 [In-band]

The following data comply with the normal restrictions of the transport and do not need special framing. This indicates a return to the restrictions in effect at stream creation time.

excessiveRetransmissions 333_8 [Out-of-band with mark]

A data transfer was attempted many times and not accepted by the foreign system. The outgoing transaction is stopped immediately and resumed at the next transaction boundary, as designated by the in-band mark **abortMark**. The **excessiveRetransmissions** control should never be generated by the GAP client.

iAmHere 305_8 [Out-of-band]

Response to the **areYouThere** control confirming that the foreign device/process is operational.

interrupt 301_8 [Out-of-band]

Temporarily halt both processing and output as quickly as possible. Perhaps, depending on device, wait for a **resume** control before continuing.

none 300_8 [In-band]

This is the normal condition. The transition from **none** to some other packet subtype value and back is the the event that indicates the control in the data sequence.

remoteNotResponding 331_8 [Out-of-band with mark]

The foreign system has stopped communicating. The outgoing transaction is stopped immediately and resumed at the next transaction boundary, as designated by the in-band

mark **abortMark**. The **remoteNotResponding** control should never be generated by the GAP client.

resume 302_8 [Out-of-band]

Resume processing where you were when interrupted.line control, although it can be used on other transports to make sure a particular group of data has been transferred successfully. Reception of this control must be acknowledged by generating the **endOfTransactionAck** control.

transparentDataFollows 313_8 [In-band]

The following data may contain bytes which require special framing. It is used only with transports (e.g., BSC for IBM Office System 6) that have restrictions on the data which is contained in a normal record. These restrictions are in effect when the stream is created.

yourTurnToSend 311_8 [In-band]

Tell the receiver that it may send now. This control is used only in alternating-mode transports (e.g., BSC).

~~**chain3270** 334_8 [In-band]~~

~~The data following is a chain of commands for an IBM 3270 terminal. The packet containing the end of the command chain should have the End-of-message bit set.~~

unchained3270 335_8 [In-band]

The data following is a ~~non-chained~~ sequence of one or more commands for an IBM 3270 terminal. The packet containing the end of the command should have the End-of-message bit set.

readModified3270 336_8 [In-band]

The data following is read modified data from an IBM 3270 terminal. The packet containing the end of the read modified data should have the End-of-message bit set.

status3270 337_8 [In-band]

The data following is status from an IBM 3270 terminal. The packet containing the end of the status should have the End-of-message bit set.

testRequest3270 340_8 [In-band]

The data following is test request data from an IBM 3270 terminal. The packet containing the end of the test request data should have the End-of-message bit set.

5.3 Status

Status about the session is also returned to the client using the attention byte facility of the Sequenced Packet Protocol. The following status values are defined:

321_8	mediumUp
322_8	mediumDown
323_8	ourAccessIDRejected
324_8	weRejectedAccessID
325_8	noGetForData
326_8	unsupportedProtocolFeature
327_8	unexpectedRemoteBehavior
330_8	unexpectedSoftwareFailure
341_8	configurationMismatch3270
370_8	hostPolling3270
371_8	hostNotPolling3270

mediumDown **322_8**

The transmission medium has gone from the *up* to the *down* state. This indicates that the connection to the foreign system has terminated.

mediumUp **321_8**

The transmission medium has gone from the *down* to the *up* state. This indicates that the connection to the foreign system is operational.

noGetForData **325_8**

The foreign system is sending data and the client is not reading data. If possible the Gateway software will use flow control to prevent loss of data.

ourAccessIDRejected **323_8**

The foreign system rejected our security id.

unsupportedProtocolFeature **326_8**

The foreign system used some protocol feature that is currently unsupported (e.g., an RVI received from a Xerox 800).

unexpectedRemoteBehavior **327_8**

The foreign system did something unusual that did not follow documented protocols.

unexpectedSoftwareFailure **330_8**

The software encountered an unrecoverable software error.

weRejectedAccessID **324_8**

Either we rejected the security identification from the foreign system or the foreign system did not send a security identification when it was required.

configurationMismatch3270 341_8 [3270 emulation only]

Gateway Software determined that the parameters describing the IBM 3270 controller did not match those of the host. For example, the number of terminals defined may be different.

hostNotPolling3270 371_8 [3270 emulation only]

The 3270 host has not polled our controller for at least 2 minutes.

hostPolling3270 370_8 [3270 emulation only]

The 3270 host, which had not been polling our controller, is now polling our controller.

5.4 Data errors

Certain transports such as the **TeletypeTransport** cannot recover from data transmission errors in a graceful manner and instead return these errors to the client using packet subtypes. The following packet subtypes fall into this category:

garbledReceiveData 317_8

A framing error occurred on all characters in this packet.

parityError 316_8

A parity error occured on all characters in this packet.

Note: Generally, packets with these two subtypes will contain only a single character.

5.5 Termination

After all the data in a session has been transferred, the connection is passed back to Courier and the session is either deleted or passed to another owner by calling the Courier remote procedures **Delete** or **Transfer**. However, before the connection is passed back to Courier, the close protocol described in §7.5 of *Internet Transport Protocols* [15] should be followed. The close protocol is always initiated from the client side; the server never initiates it.

Appendix A
References

Mandatory references are those documents which should be studied before or in conjunction with this protocol specification. Informational references are those documents which provide additional useful information.

A.1 Mandatory references

Courier: The Remote Procedure Call Protocol, December 1981, XSIS 038112.

Internet Transport Protocols, December 1981, XSIS 028112.

A.2 Informational references

DWP (Xerox 850) Performance Specification, S 782-300, Chapter 3.8 [March 1978].

Xerox 860 IPS Program Specification, S 804.300.15, from OPD Engineering [March 1978].

Office System 6 Programmer's Guide for Communicating with: IBM 6/450, 6/440, 6/430 Information Processors, G544-1003 [June 1977].

IBM 3270 Information Display System Component Description, GA27-2749-9 [August 1979].