

Services Programmer's Guide

XEROX

**610E00180
September 1985**

**Xerox Corporation
Office Systems Division
2100 Geng Road
MS 5827
Palo Alto, California 94303**

**Copyright © 1985, Xerox Corporation. All rights reserved.
XEROX®, 8010, and XDE are trademarks of XEROX CORPORATION.**

Printed in U.S. A.



Preface

The Services 8.0 Programmer's Guide comprises nine separate manuals written to aid in programming in the Xerox Development Environment (XDE). This document describes programming interfaces in XDE workstation products for accessing the Xerox Network Services.

Comments and suggestions on this document and its use are encouraged. The form at the back of the guide has been prepared for this purpose. Please address communications to:

**Xerox Corporation
Office Systems Division
XDE Technical Documentation, M/S 37-18
3450 Hillview Avenue
Palo Alto, California 94304**



Table of contents

Common Facilities Programmer's Manual

1	Introduction	1-1
2	NSDataStream	2-1
3	NSName	3-1
4	NSString	4-1

Authentication Programmer's Manual

1	Introduction	1-1
2	Interfaces	2-1
3	Standard authentication scenario	3-1

Clearinghouse Programmer's Manual

1	Introduction	1-1
2	Concepts	2-1
3	Interface	3-1
A	List of operations	A-1
B	CHCommonLookups.mesa	B-1

Mailing Programmer's Manual

1	Introduction	1-1
2	Mail transport	2-1
3	Inbasket	3-1
4	Mail attributes	4-1
5	Mail stream	5-1

Table of contents

Printing Programmer's Manual

1	Introduction	1-1
2	Interface	2-1
3	NSPrint interface	3-1

Print Service 8.0 Interpress (Client) Programmer's Manual

1	Introduction	1-1
2	Interface	2-1

Phone Net Driver Programmer's Manual

1	Introduction	1-1
2	Interface	2-1
3	Usage example	3-1

External Communication Programmer's Manual

1	Introduction	1-1
2	Overview	2-1
3	Client interface	3-1
4	Performance criteria	4-1
5	Status and exception processing	5-1
6	Reliability and maintainability	6-1
7	Multinational requirements	7-1
A	RS-232-C communication parameters	A-1
B	Foreign device considerations	B-1

Filing Programmer's Manual

1	Introduction	1-1
2	Overview	2-1
3	File/session operations	3-1
4	Segment/content operations	4-1
5	Positionable stream operations	5-1
6	Attributes	6-1
7	Pathname parsing operations	7-1
8	System configuration and administration	8-1

Appendices

A	References	A-1
B	Gateway Access Protocol (GAP) Specification	B-1

XEROX



Services 8.0 Programmer's Guide

**Common Facilities
Programmer's Manual**

November 1984

PRELIMINARY

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



Table of contents

1	Introduction	1-1
1.1	Overview	1-1
1.2	NSDataStream	1-1
1.3	NSName	1-1
1.4	NSString	1-2
2	NSDataStream	2-1
2.1	Clients who actively send or receive data	2-2
2.2	Clients negotiating bulk data transfers between two other parties	2-4
2.3	Implementors of local bulk data transfer operations	2-5
2.4	Implementors of remote bulk data transfer operations	2-6
2.5	NSDataStream operations	2-10
3	NSName	3-1
3.1	Network object naming	3-1
3.1.1	Names and name records	3-1
3.1.2	Basic operations	3-2
3.1.3	Comparison and equivalence	3-4
3.1.4	Conversion	3-5
3.1.5	Errors	3-6
3.2	Parameter serialization	3-7
3.2.1	Serialization of arbitrary structures	3-7
4	NSString	4-1
4.1	Strings, substrings, and Mesa strings	4-1
4.2	Basic operations	4-2
4.3	Scanning, comparison, and equivalence	4-5
4.4	Conversion	4-7
4.5	Serialization	4-9
4.6	Errors	4-9

Table of contents



Introduction

Certain facilities are made available which are useful in conjunction with more than one service:

- a data stream facility
- an object naming and authentication facility
- a common string format facility

1.1 Overview

This document describes facilities which are useful in conjunction with more than one service. The mechanisms introduced are *strings*, a package that manipulates sequences of characters encoded according to the Xerox *Character Code Standard* [4]; *data streams*, a package that allows location-independent transmission of large data items according to the Xerox *Bulk Data Transfer Protocol* [3]; and *names*, a package that manipulates network object names and related data items.

1.2 NSDataStream

Section 2 describes the data stream facilities provided by **NSDataStream**. It begins with an overview of data streams, and continues with a description for clients of the interface, a description for implementors of bulk data transfer operations, and a description for stub writers of interfaces containing bulk data transfer operations.

1.3 NSName

Section 3 describes the **NSName** mechanism, a facility that allows manipulation of the data structures used to name objects in the 8000 NS systems. Since many services deal with objects, and those objects must be identified in requests to those services, this facility is used in many contexts.

1.4 **NSString**

Section 4 describes the **NSString** facility. **NSString** provides a set of operations to manipulate sequences of characters encoded according to the Xerox *Character Code Standard* [4].



NSDataStream

NSDataStream: DEFINITIONS . . . ;

This section describes the data stream facilities provided by **NSDataStream**. It begins with an overview of data streams, and continues with a description for clients of the interface, a description for implementors of bulk data transfer operations, and a description for stub writers of interfaces containing bulk data transfer operations.

Some of the key features of the data stream mechanism are:

- Bulk data transfer occurs during data transfer operations—between the procedure call and the return—rather than after the operation is finished, as it was in the past. This allows operations to return results based on the successful data transfer (e.g., file handles) and allows for better status reporting, using the full generality of Mesa errors to report to the initiator of a data transfer specific problems that occur. When the transfer occurs between two system elements, it always occurs on a connection used by a **Courier** remote procedure call.
- Direct, third-party transfers are supported. A client on one system element can initiate bulk data transfer between two other system elements simply by making two remote procedure calls, one to each system element. The **NSDataStream** mechanism will automatically establish a connection between the two parties.
- Clients of bulk data transfer operations have the option to provide or be provided with a data stream for the data transfer. A data stream may be requested in one bulk data transfer operation and then supplied to another. This allows two independent functions (one providing data, such as retrieving a file, and one accepting data, such as printing a document) to be combined without either having more knowledge of the other than that they support **NSDataStream** conventions.

A *data stream*, referenced via an **NSDataStream.Handle**, is a half duplex, non-positionable stream designed for transfer of bulk data (e.g., files). Data streams have the built-in capabilities for aborting a transfer by the sender or the receiver, and for linking sending and receiving processes independent of geographic location.

NSDataStream.Handle: TYPE = RECORD [Stream.Handle];

Data streams come in two varieties: **SinkStream**, on which data may be sent, and **SourceStream**, on which data may be received. A data stream is compatible with a Pilot stream in that an **NSDataStream.Handle** may be passed as a **Stream.Handle** to **Stream** operations. The converse is not true; arbitrary streams may not be supplied to operations which expect a data stream. Thus, it is improper for a client ever to use a Mesa record constructor to obtain an **NSDataStream.Handle** from an arbitrary **Stream.Handle**. Similarly, it is improper to use a Mesa record constructor to obtain a **SinkStream** or a **SourceStream** from an **NSDataStream.Handle**.

NSDataStream.SinkStream: TYPE = RECORD [Handle];

NSDataStream.SourceStream: TYPE = RECORD [Handle];

Clients of data stream operations fall into one of two categories: those who actively send or receive data, and those who negotiate a transfer between two other parties. Senders and receivers may be further classified into (1) those who send or receive data—typically *structured* data—using enumeration call-back procedures as in **NSFile.List**, and (2) those who send or receive data—typically *unstructured* data—using stream primitives (**PutBlock**, **PutByte**, **GetBlock**, **GetByte**, etc.).

2.1 Clients who actively send or receive data

Clients of the first type, those who actively send or receive data using enumeration procedures, are shielded from all stream aspects of the data transfer. They receive (send) Mesa records as arguments (results) through repeated invocations of the supplied call-back procedure, and are given the option of terminating the enumeration any time by a boolean continuation result.

The following is an example of a client who receives data using enumeration:

NSFile.List[...], ListData, ...! NSFile.Error = > REJECT];

```
ListData: PROCEDURE [attributes: NSFile.Attributes]
  RETURNS [continue: BOOLEAN ← TRUE] =
  BEGIN
    continue ← ProcessAttributes[attributes];
  END; -- of ListData
```

Senders using Stream primitives (**PutBlock**, **PutByte**, etc.) will acquire an **NSDataStream.SinkStream** from a data transfer operation in the appropriate interface (**NSFile**, **Telepress**, etc.), and will generate and transmit blocks of data using those primitives. The **SinkStream** can be acquired by supplying a parameter which is a **proc** variant of an **NSDataStream.Source** to the data transfer operation. The supplied call-back procedure is invoked once, at a time before data transfer begins.

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

Similarly, receivers using **Stream** primitives (**GetBlock**, **GetByte**, etc.) acquire an **NSDataStream.SourceStream** from a data transfer operation in the appropriate interface and receive and process blocks of data. The **SourceStream** is acquired by supplying a parameter which is a **proc** variant of an **NSDataStream.Sink** to the data transfer operation.

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

NSDataStream.Abort: PROCEDURE [stream: Handle];

NSDataStream.Aborted: ERROR;

Within the call-back procedure which makes up the **proc** variant of a **Sink** or **Source**, the client should do the following:

- 1) Use the stream primitives. A **SinkStream** should be used to send data, and a **SourceStream** should be used to receive data. The receive procedures associated with a **SinkStream** and the send procedures associated with a **SourceStream** are not implemented. The client should not change the subsequence type of the stream, nor expect to be notified of a subsequence type change. The streams are not positionable.
- 2) Abort the data stream on errors. If the sender is unable to provide, or the receiver is unable to process, any or all of the data, the data stream may be aborted. This is done using the procedure **NSDataStream.Abort**, and has the effect that the next stream primitive (including **Stream.Delete**) employed by the other end of the data stream will result in the error **NSDataStream.Aborted**. Both the sender and the receiver may abort a data transfer, and both must be prepared to accept the **Aborted** error on all stream operations. Both the party that aborts a data stream and the party that receives the abort must call **Stream.Delete** (step 3).
- 3) Delete the data stream. **Stream.Delete** must be called by the sender to indicate the end of data or to acknowledge a receiver abort, and by the receiver to acknowledge the **endOfStream** completion status or a sender abort. If **Stream.Delete** raises the **Aborted** error, it need not be retried.
- 4) Return from the call-back procedure or raise an error. If an exceptional condition arises in the client's procedure, an error may be signaled and caught by a catch phrase in a procedure further up the call chain. The client's procedure is still required to delete the stream and, prior to the deletion, may choose to abort the stream as well. The stream must be deleted by the client's call-back procedure before it returns or raises an error.

The following is an example of a client who sends data using stream primitives:

```
NSFile.Store[...,[proc[SendData]],...! NSFile.Error = > ...];

SendData: PROCEDURE [sinkDS: NSDataStream.SinkStream] =
BEGIN
UNTIL finished DO
  Stream.PutBlock[sinkDS, ...! NSDataStream.Aborted = > EXIT]
  ENDLOOP;
Stream.Delete[sinkDS ! NSDataStream.Aborted = > CONTINUE]
END; -- of SendData
```

The following is an example of a client who receives data using stream primitives:

```
NSFile.Retrieve[...,[proc[GetData]],...! NSFile.Error = > ...];

GetData: PROCEDURE [sourceDS: NSDataStream.SourceStream] =
BEGIN
UNTIL finished DO
  [...] ← Stream.GetBlock[sourceDS, ...! NSDataStream.Aborted = > EXIT];
  ENDLOOP;
Stream.Delete[sourceDS ! NSDataStream.Aborted = > CONTINUE]
END; -- of GetData
```

2.2 Clients negotiating bulk data transfers between two other parties

Clients negotiating transfers between two other parties do so in the same manner, regardless of the location of the two parties. Both parties may be on the same local or remote system element (as in a file conversion); one may be local and the other remote (as in a file retrieval); or they may be on distinct remote system elements (as in a file service to print service file copy).

In each case, the client calls one bulk data transfer operation requesting a data stream by specifying a call-back procedure, and then uses that data stream as an argument in another bulk data transfer operation invoked from within the call-back procedure. Neither bulk data transfer operation returns while the data transfer is in progress, and each is able to return results after the transfer has completed, or to raise a Mesa error if the transfer cannot be completed. By supplying the data stream to the second bulk data transfer operation, the client no longer has any obligation (or privilege) to delete, abort, or operate on the data stream in any way. The stream is deleted by the bulk data transfer operation, even if that operation terminates with an error.

When one party in the bulk data transfer encounters a problem, it will abort the data stream it is using. The client discovers this in two ways. The party that encountered the problem will raise a Mesa error to indicate the exact nature of the problem. The party receiving the abort indication will catch the **NSDataStream.Aborted** error and may choose to raise an error specific to the operation or return normally, with the understanding that the other party will do the real error notification. **NSFile**, for example, raises the error **NSFile.Error[[transfer [aborted]]]** to indicate that a transfer was aborted. The client must take special action (e.g., **CONTINUE**) when the second operation raises an error such as this, since the first operation must be allowed to raise the more descriptive error.

The following is an example of a client who negotiates a transfer:

```
NSFile.Retrieve[..., [proc [SendData]], ...! NSFile.Error = > ...];  
  
SendData: PROCEDURE [sourceDS: NSDataStream.SourceStream] =  
BEGIN  
  NSFile.Store[..., [stream [sourceDS]], ...!  
    NSFile.Error = > IF error = [transfer [aborted]] THEN CONTINUE]  
  END; -- of SendData
```

2.3 Implementors of local bulk data transfer operations

Implementors of local bulk data transfer operations operate on data streams in much the same way that clients do. They may use stream primitives to send or receive data, or to pass a data stream to another bulk data transfer operation. The primary difference between these local implementors and their clients is the manner in which the data stream is acquired.

```
NSDataStream.Couple: TYPE = [sink: SinkStream, source: SourceStream];
```

Bulk data transfer operations should define a parameter which is an **NSDataStream.Source** or an **NSDataStream.Sink**. This allows a client to provide a data stream, choose to be provided with one in a specified call-back procedure, or provide a null data stream indicating that no transfer should occur. Every bulk data transfer operation should implement all three options. When the data stream is provided, that data stream should be used. When the data stream is requested, a data stream **Couple** should be created. A **Couple** consists of two matched data streams, a **SinkStream** and a **SourceStream**. The data streams are matched in that data sent on the **SinkStream** may be received from the **SourceStream**. One of these data streams should be used by the implementor, and the other should be provided to the client in the specified call-back procedure.

```
NSDataStream.OperateOnSink: PROCEDURE [sink: Sink, operation: PROCEDURE [SinkStream]];
```

```
NSDataStream.OperateOnSource: PROCEDURE [  
  source: Source, operation: PROCEDURE [SourceStream]];
```

The implementor may make use of the operations **OperateOnSink** and **OperateOnSource** to acquire a data stream on which to operate and to perform the actions described above. These operations will act differently for each of the variants of **Sink** or **Source**. For a **stream** variant, the stream is supplied directly to **operation**. For a **proc** variant, a **Couple** is created; one half of the couple is supplied to a *forked operation*, while the other is supplied to the client's procedure. The **operation**, therefore, may not raise any errors or signals (because it is a forked process) and must instead return normally and raise any errors after **OperateOnSink** or **OperateOnSource** has returned. For a **none** variant, a **NIL** **SinkStream** or **SourceStream** is supplied to **operation**. The implementor should recognize this value and, without passing it to any **Stream** or **NSDataStream** operation, should treat a **NIL** **SinkStream** as a request to discard the data and a **NIL** **SourceStream** as though it gives an immediate end of stream indication.

The following is an example implementation of a local bulk data transfer operation:

```

NSFile.Store: PROCEDURE [..., source: Source, ...] =
  BEGIN
    outcome: Status ← normal;
    StoreProc: PROCEDURE [sourceDS: NSDataStream.SourceStream] =
      BEGIN
        IF sourceDS = [NIL] THEN RETURN;
        UNTIL finished DO
          [...] ← Stream.GetBlock[sourceDS, ...! NSDataStream.Aborted = >
            {outcome ← aborted; EXIT}];
          IF problemEncounteredProcessingData THEN {
            NSDataStream.Abort[sourceDS]; outcome ← error; EXIT}
          ENDLOOP;
          Stream.Delete[sourceDS];
          NSDataStream.Aborted = > {outcome ← aborted; CONTINUE}
        END;
        NSDataStream.OperateOnSource [source, StoreProc];
        IF outcome ≠ normal THEN ERROR ... -- errors are raised after OperateOnSource returns
      END; -- of Store
    
```

2.4 Implementors of remote bulk data transfer operations (stub writers)

Implementors of remote bulk data transfer operations provide their clients the same flexibility as local implementors. The operations may have **Source** or **Sink** parameters for unstructured data, allowing the client to select how the data stream is determined, or may have enumeration call-back procedure parameters which are called repetitively with structured data.

In the **Source** or **Sink** approach, **OperateOnSink** and **OperateOnSource** are used to determine the data stream in the client stub in a similar manner to a local operation. The difference lies in how the data stream transcends physical machine boundaries to be supplied as a parameter to a bulk data transfer operation local to a server.

This is done by having the client stub check in the data stream using **NSDataStream.Register**. In exchange, the client stub receives a **Ticket** which can be passed as an argument in a remote procedure call using **DescribeTicket** as the **Courier.Description**.

```

NSDataStream.Ticket: TYPE [11];

NSDataStream.DescribeTicket: Courier.Description;

NSDataStream.Register: PROCEDURE [
  stream: Handle, forUseAt: Courier.SystemElement, cH: Courier.Handle,
  useImmediateTicket: BOOLEAN ← TRUE] RETURNS [Ticket];
  
```

The server stub, upon receiving the ticket, uses the ticket to reclaim the data stream using **OpenSink** or **OpenSource**. This ticket system is very much like the baggage check system of an airline. Small data structures can be passed as parameters to a remote operation or returned as results, just as small possessions can be carried onto the plane and stored beneath the seat. Large data structures are passed via streams which are checked in, in exchange for a ticket, and then later exchanged for a data stream (but only at the

destination system element). This resembles large baggage, which is checked in exchange for a claim check redeemable only at the destination.

NSDataStream.OpenSink: PROCEDURE [ticket: Ticket, cH: Courier.Handle] RETURNS [SinkStream];

NSDataStream.OpenSource: PROCEDURE [ticket: Ticket, cH: Courier.Handle]
RETURNS [SourceStream];

In reality, the ticket mechanism deletes one data stream and creates a filter over a network stream at the destination. The network stream used is either one employed by Courier for the remote operation itself, or it is one established between two system elements using an operation from the **BulkDataTransfer** remote program. If the client is to be a party in the transfer rather than an idle third party, the client stub may decide for each **Sink** or **Source** parameter (there may be several for a single procedure) whether the transfer should occur on the **Courier** connection of the operation, or on another network stream. At most one of the **Sink** or **Source** parameters can make use of a single network stream. The client stub indicates which network stream to use by specifying a boolean argument, **useImmediateTicket**, to the **Register** operation. This argument is ignored if the client is not one of the parties in the bulk data transfer using the **Sink** or **Source**. If the **useImmediateTicket** boolean is **TRUE**, the client stub should also supply the procedure **NSDataStream.AnnounceStream** as the **streamCheckoutProc** argument to **Courier.Call**. This will allow Courier to provide **NSDataStream** with its network stream at a time after the arguments have been transmitted, when the client can expect to make use of the network stream.

One should notice that third-party transfers, such as transfers between two servers as controlled by a workstation, are supported by this design without additional effort by the client or stub writer. Each **Register** operation waits until intentions have been stated for the other half of the data stream couple, either by a matching **Register** operation or by an explicit or implicit **AssertLocal** operation (see §2.5). When intentions of both halves have been stated, it is known what two system elements are to participate in the bulk data transfer and the appropriate connection can be established. Thus, in the case of two remote procedure calls with matching data streams, a network stream is established between two other system elements. One system element can, therefore, receive data directly from another without knowing the other's protocol.

The following is an example client stub implementation of a remote bulk data transfer operation:

```
NSFile.Store: PROCEDURE [..., source: Source, ..., session: Session]
RETURNS [file: Handle] =
BEGIN
  outcome: Status ← normal;
  StoreProc: PROCEDURE [sourceDS: NSDataStream.SourceStream] =
    BEGIN ENABLE ANY = > {outcome ← error; CONTINUE}; -- catch possible errors
    arguments.source ← NSDataStream.Register[
      sourceDS, DetermineSystemElement[session], cH, TRUE];
    [] ← Courier.Call [
      cH, ..., [arguments, StoreArgumentsDescription], ...,
      FALSE, NSDataStream.AnnounceStream! Courier.Error = >
        NSDataStream.CancelTicket[arguments.source, cH]];
    file ← results.file
```

```

    END;
    NSDataStream.OperateOnSource [source, StoreProc];
    IF outcome # normal THEN ERROR ...
    END; -- of Store

    StoreArgumentsDescription: Courier.Description =
    BEGIN OPEN notes;
    parameters: LONG POINTER TO StoreArguments = noteSize[
        size: SIZE[StoreArguments]];
    ...
    noteParameters[@parameters.source, NSDataStream.DescribeTicket];
    ...
    END;

```

The following is an example server stub implementation of a remote bulk data transfer operation:

```

Dispatch: Courier.Dispatcher -- [cH: Courier.Handle, procedureNumber: CARDINAL,
arguments: Courier.Arguments, results: Courier.Results]-- =
    BEGIN
    arguments[...];
    ...
    SELECT procedureNumber FROM
    ...
    store = >
    BEGIN
    OPEN arg: LOOPHOLE[argumentList, POINTER TO StoreArguments],
    res: LOOPHOLE[resultList, POINTER TO StoreResults];
    sourceDS: NSDataStream.SourceStream ←
        NSDataStream.OpenSource[arg.source, cH ! NSDataStream.Error => NSFfile.Error[...]]
    res.file ← NSFfile.Store[..., [stream [sourceDS]], ...]
    END;
    ...
    ENDCASE;
    ...
    results[...];
    END;

```

The other method of bulk data transfer provides the client with an enumeration operation. This method is essentially the same as the Source/Sink method, with the addition of a layer of software over both the client and server side. The software layer serializes a Mesa record at the server and reestablishes the Mesa record from the transmitted data for the client. All the client rules regarding direct use of stream primitives apply. In particular, the data stream must be deleted by both the client and server stubs, even in the event of an error raised by the client's enumeration procedure.

The following is an example client stub implementation of a remote bulk data transfer operation using an enumeration procedure:

```
NSFile.List: PROCEDURE [
    ..., proc: AttributesProc, ..., session: Session] =
BEGIN
    ListProc: PROCEDURE [sourceDS: NSDataStream.SourceStream] =
        BEGIN
        UNTIL finished DO
            [...] ← Stream.GetBlock[sourceDS, ...! NSDataStream.Aborted = > EXIT];
            IF NOT proc [! UNWIND = >
                {NSDataStream.Abort[sourceDS]; Stream.Delete[sourceDS]}] THEN EXIT
            ENDLOOP;
            Stream.Delete[sourceDS ! NSDataStream.Aborted = > CONTINUE]
        END; -- of ListProc
    ListByStream[..., sink: [proc [ListProc]], ..., session: Session!
        NSDataStream.Aborted = > CONTINUE]
    END; -- of List

    ListArgumentsDescription: Courier.Description =
        BEGIN OPEN notes;
        parameters: LONG POINTER TO ListArguments = noteSize[size: SIZE[ListArguments]];
        ...
        noteParameters[@parameters.sink, NSDataStream.DescribeTicket];
        ...
    END;

    ListByStream: PROCEDURE [..., sink: Sink, ..., session: Session] =
        BEGIN
        outcome: Status ← normal;
        ListByStreamProc: PROCEDURE [sinkDS: NSDataStream.SinkStream] =
            BEGIN ENABLE ANY = > {outcome ← error; CONTINUE}; -- catch possible errors
            arguments.sink ← NSDataStream.Register[
                sinkDS, DetermineSystemElement[session], cH, TRUE];
            [] ← Courier.Call [
                cH, ..., [arguments, ListArgumentsDescription], ...,
                FALSE, NSDataStream.AnnounceStream! Courier.Error = >
                    NSDataStream.CancelTicket[arguments.sink, cH]];
        END;
        NSDataStream.OperateOnSink [sink, ListByStreamProc];
        IF outcome # normal THEN ERROR ...
    END; -- of ListByStream
```

The following is an example server stub implementation of a remote bulk data transfer operation using enumeration:

```

Dispatch: Courier.Dispatcher -- [cH: Courier.Handle, procedureNumber: CARDINAL,
arguments: Courier.Arguments, results: Courier.Results] -- =
BEGIN
arguments[...];
...
SELECT procedureNumber FROM
...
list = >
BEGIN
OPEN arg: LOOPHOLE[argumentList, POINTER TO ListArguments],
res: LOOPHOLE[resultList, POINTER TO ListResults];
sinkDS: NSDataStream.SinkStream ← NSDataStream.OpenSink[arg.sink, cH !
NSDataStream.Error = > NSFile.Error[...]];
ListByStream[..., sink: sinkDS, ...]
END;
...
ENDCASE;
...
results[...];
END;

ListByStream: PROCEDURE [..., sink: SinkDataStream, ...] =
BEGIN
ListProc: PROCEDURE [...] RETURNS [continue: BOOLEAN ← TRUE] =
BEGIN ENABLE UNWIND = > NSDataStream.Abort[sink];
...
Stream.PutBlock[sink, ...! NSDataStream.Aborted = >
{continue ← FALSE; CONTINUE}]
END;
NSFile.List[..., ListProc, ...! UNWIND = >
Stream.Delete[sink! NSDataStream.Aborted = > CONTINUE]];
Stream.Delete[sink! NSDataStream.Aborted = > CONTINUE]
END; -- of ListByStream

```

2.5 NSDataStream operations

The **Abort** operation aborts a data stream. If the data stream is a **SinkStream**, this indicates that the sender is unable to supply the remainder of the data and suggests to the receiver that the data is incomplete. The receiver may choose to discard all data already received. If the data stream is a **SourceStream**, the receiver is unable to accept and process any more data. This instructs the sender to stop sending data immediately. Repeated aborts of the same data stream are ignored.

The process operating on the other half of the data stream is notified of an aborted data stream on the next **Stream** operation (**PutBlock**, **PutByte**, **GetBlock**, **GetByte**, **Delete**, etc.). The error occurs on **Stream.Delete** in the situation where the receiver aborts the data stream after all of the data is received. In this situation, the next operation by the sender will be to delete the data stream, which raises the **Aborted** error. The **Delete** operation will have completed, however; so the sender is no longer required to delete the data stream

again. In all other situations, it is necessary to delete a data stream after aborting it or being notified of an abort.

NSDataStream.Abort: PROCEDURE [stream: Handle];

Arguments: **stream** is a data stream which may either be a **SinkStream** or a **SourceStream**.

Results: The data stream is aborted.

Errors: None.

The **AssertLocal** operation is called by the holder of a data stream when it is known that **Stream** operations (**PutBlock**, **GetBlock**, etc.) will be performed on the data stream. It should not be called if the data stream is to be passed to an operation on another system element. **AssertLocal** differs from sending or receiving an empty block only in that it returns immediately if the data stream is not yet established as a local or network stream. Performing any **Stream** operation implies **AssertLocal**, and thus a client is not required to use this operation. It is primarily useful in situations where extensive computation is likely to occur in preparation for sending or receiving data. Invoking **AssertLocal** allows the establishment of the data stream as a local or network stream to proceed in parallel with the client's preparatory computation. Repeated calls to **AssertLocal** are ignored.

NSDataStream.AssertLocal: PROCEDURE [stream: Handle];

Arguments: **stream** is a data stream, either a **SinkStream** or a **SourceStream**.

Results: Subsequent use of the data stream may only occur on the local system element.

Errors: None.

CreateCouple creates a pair of coupled data streams such that data sent on **couple.sink** can be retrieved from **couple.source**. Each data stream must eventually be deleted with **Stream.Delete** or exchanged for a ticket using **Register** (see below).

NSDataStream.CreateCouple: PROCEDURE RETURNS [Couple];

Arguments: None.

Results: A couple of data streams.

Errors: **NSDataStream.Error** [tooManyLocalConnections].

The **OperateOnSink** and **OperateOnSource** operations are called by client stubs and local implementations of bulk data transfer operations. They invoke the specified **operation** with a data stream argument derived from the specified **Sink** or **Source**. If the **Sink** or **Source** is a **proc** variant, the procedure is called and **operation** is forked with a matching data stream. If the **Sink** or **Source** was a **none** variant, a **NIL** data stream is supplied to **operation**.

NSDataStream.OperateOnSink: PROCEDURE [sink: Sink, operation: PROCEDURE [SinkStream]];

NSDataStream.OperateOnSource: PROCEDURE [
source: Source, operation: PROCEDURE [SourceStream]];

Arguments: **sink** or **source** is an argument to a bulk data transfer operation; **operation** is a procedure to be called or forked, depending on the nature of **sink** or **source**.

Results: None.

Errors: None.

Register is called by client stub implementations. It asserts that a data stream will be used on a specified system element. The ticket obtained may be passed as an argument to a remote procedure call, where the server stub may exchange it for a network data stream using **OpenSink** or **OpenSource**. **Register** assumes all rights to the data stream; the client need not delete the data stream and may not use the data stream after applying **Register**. Tickets which are not redeemed should be passed to **CancelTicket** by the client stub. A **NIL** stream may be registered to suppress the data transfer in what would have been a bulk data transfer operation.

NSDataStream.Ticket: TYPE [11];

NSDataStream.Register: PROCEDURE [
stream: Handle, forUseAt: Courier.SystemElement, cH: Courier.Handle,
useImmediateTicket: BOOLEAN ← TRUE] RETURNS [Ticket];

Arguments: **stream** is a data stream which has not previously been supplied to any **NSDataStream** or **Stream** operation; **forUseAt** indicates the system element at which the returned ticket will be redeemed; **cH** is the **Courier** handle for the connection on which the remote operation will occur; **useImmediateTicket** indicates whether the **Courier** connection associated with **cH** should be used—it is ignored if the matching data stream is not asserted local.

Results: The resulting ticket may be used by the client stub as an argument to a remote procedure.

Errors: **NSDataStream.Error[tooManyTickets]**, **Courier.Error**.

The **OpenSink** and **OpenSource** operations are called by server stubs, and establish network data streams in exchange for tickets provided to client stubs by the **Register** operation. Streams returned by these operations must be deleted with **Stream.Delete** when data transfer is complete.

NSDataStream.OpenSink: PROCEDURE [ticket: Ticket, cH: Courier.Handle] RETURNS [SinkStream];

NSDataStream.OpenSource: PROCEDURE [ticket: Ticket, cH: Courier.Handle]
RETURNS [SourceStream];

Arguments: **ticket** is a ticket received by the client stub; **cH** is the **Courier** handle received by the server stub's **Dispatcher** and is only used if the ticket is an immediate ticket.

Results: A data stream which is a filter over a network stream.

Errors: **NSDataStream.Error [localEndIncorrect/tooManyLocalConnections].**

The **CancelTicket** operation is called by client stubs when it becomes evident that a ticket returned from **Register** will not be redeemed by a server stub. This will be the case if a problem occurs after the **Register** operation but before the remote procedure call is initiated, or if there is a problem initiating a remote procedure call. Once the server stub's **Dispatcher** is called, the server stub is expected to redeem the ticket, even in the event of an error.

NSDataStream.CancelTicket: PROCEDURE [ticket: Ticket, cH: Courier.Handle];

Arguments: **ticket** is a ticket received from **Register** and **cH** is the **Courier** handle supplied to **Register**.

Results: The ticket may no longer be redeemed.

Errors: None.

The **AnnounceStream** operation is called by the client stub to indicate the appropriate time to use the network stream previously in use by **Courier**. This must occur after arguments of a remote procedure are transmitted and before the results are returned. The most common use of this procedure is to pass it to **Courier.Call** as the **streamCheckoutProc**. **Courier** will then call its **streamCheckoutProc** at the proper time. **AnnounceStream** will have no effect if no immediate ticket was issued for the specified **Courier** handle. This operation will not return until the data stream is deleted.

NSDataStream.AnnounceStream: PROCEDURE [cH: Courier.Handle];

Arguments: **cH** is the **Courier** handle for the remote operation in progress.

Results: None.

Errors: None.

NSName

NSName: DEFINITIONS = ...;

This section describes the **NSName** mechanism, a facility that allows manipulation of the data structures used to name objects in the 8000 NS systems. Since many services deal with objects, and those objects must be identified in requests to those services, this facility is used in many contexts.

NSName provides two facilities. The first, *network object naming*, defines types used to name objects, and operations to manipulate names and convert them to other forms. The second, *parameter serialization*, consists of procedures which help represent general data structures according to the remote procedure calling protocol.

3.1 Network object naming

The network architecture defines a number of objects. File services, users, and distribution lists are all examples of objects. All objects are named in a consistent way so that they can be referenced in messages between systems. A *name* consists of three parts: an *organization*, which is the highest level in the naming hierarchy; a *domain*, which is a subdivision of an organization; and a *local name*, which actually identifies the object. Each part is unique relative to the next-higher part.

3.1.1 Names and name records

A name is represented most often as a record containing three strings, which correspond to the three parts of the name, or by a pointer to that record.

NSName.Name: TYPE = LONG POINTER TO NameRecord;

NSName.NameRecord: TYPE = RECORD [org: Organization, domain: Domain, local: Local];

NSName.Organization: TYPE = NSString.String ← NSString.nullString;

NSName.Domain: TYPE = NSString.String ← NSString.nullString;

NSName.Local: TYPE = NSString.String ← NSString.nullString;

NSName.nullNameRecord: NSName.NameRecord = [];

The components of a name are restricted in length. Clients must not create any name that does not respect these limits, though not all procedures in this interface enforce them.

```
NSName.maxOrgLength: CARDINAL = 20;
NSName.maxDomainLength: CARDINAL = 20;
NSName.maxLocalLength: CARDINAL = 40;
```

Sometimes it is useful for allocation purposes to provide name storage which is local to a procedure. This is facilitated by the following **TYPE**:

```
NSName.NameStore: TYPE = RECORD [
  record; NSName.NameRecord,
  org: PACKED ARRAY [0..maxOrgLength] OF Environment.Byte
  domain: PACKED ARRAY [0..maxDomainLength] OF Environment.Byte
  local: PACKED ARRAY [0..maxLocalLength] OF Environment.Byte
```

In some applications of names, a special meaning of "wild card" is attached to the asterisk character. Although it is defined in this interface for convenience, it has no special meaning to the operations within **NSName**.

```
NSName.wildCard: CHARACTER = '*';
NSName.wildCardCharacter: NSString.Character = [0, [wildCard]];
NSName.wildCardString: NSString.String;
```

Although most names appear as the **Name** or **NameRecord** type, there are some cases in which it is more convenient to deal with a single string. In this case, the three parts of the name are included in the string in the order *local name*, *domain*, and *organization*. Each is distinguished by the separator character defined below. The total length of the string is limited by the component maximum lengths (previously defined) and the overhead of character set changes and separators. An example of such a name is "DragonSeed:OSD West:Xerox," where "DragonSeed" is the local name, "OSD West" is the domain, and "Xerox" is the organization.

```
NSName.separator: CHARACTER = ':';
NSName.separatorCharacter: NSString.Character = [0, [separator]];

NSName.hierarchicalLevels: CARDINAL = 3;
NSName.characterSetChangeOverhead: CARDINAL = 2;
NSName.maxFullNameLength: CARDINAL =
  maxLocalNameLength + maxDomainNameLength + maxOrgNameLength +
  (characterSetChangeOverhead + 1) * (hierarchicalLevels - 1);
```

3.1.2 Basic operations

An empty name is allocated by calling **MakeNameFields**, which allocates the component strings of an existing name record, or **MakeName**, which allocates both the record and the strings.

```
NSName.MakeNameFields: PROCEDURE [
  z: UNCOUNTED ZONE, destination: Name, orgSize: CARDINAL ← maxOrgLength,
  domainSize: CARDINAL ← maxDomainLength, localSize: CARDINAL ← maxLocalLength];
```

Arguments: **destination** refers to the name record in which strings are to be allocated; **orgSize**, **domainSize**, and **localSize** specify the lengths of the allocated strings (in bytes); all storage is allocated from **z**.

Results: None.

Errors: None.

NSName.MakeName: PROCEDURE [
 z: UNCOUNTED ZONE, orgSize, domainSize, localSize: CARDINAL] RETURNS [**Name**];

Arguments: **orgSize**, **domainSize** and **localSize** specify the lengths of the allocated strings (in bytes); all storage is allocated from **z**.

Results: The allocated **Name** is returned.

Errors: None.

A name may also be created by copying an existing one. **CopyNameFields** copies the source into an already-allocated destination, allocating any strings that are not already allocated, while **CopyName** creates a new name that is a copy of the source.

NSName.CopyNameFields: PROCEDURE [**z: UNCOUNTED ZONE, source, destination: Name**];

Arguments: **source** is the name to be copied; **destination** is the name intended to hold the copy; **z** is used to allocate any of the components of **destination** that are not already allocated.

Results: None.

Errors: **NameTooSmall** is raised if an already-allocated component of **destination** is too small.

NSName.CopyName: PROCEDURE [**z: UNCOUNTED ZONE, name: Name**] RETURNS [**Name**];

Arguments: **name** is the name to be copied; all storage is allocated from **z**.

Results: The allocated **Name** is returned.

Errors: None.

Storage allocated by the preceding operations must be freed by the client. **FreeNameFields** (which may also be called as **ClearName**) frees only the component strings of a name, and is therefore suitable for freeing names allocated by **MakeNameFields** or **CopyNameFields**, while **FreeName** frees both the component strings of a name and its name record, and is therefore suitable for freeing names allocated by **MakeName** or **CopyName**.

NSName.FreeNameFields, ClearName: PROCEDURE [**z: UNCOUNTED ZONE, name: Name**];

Arguments: **name** is the name to be freed; storage is assumed to be allocated from **z**.

Results: None.

Errors: None.

NSName.FreeName: PROCEDURE [z: UNCOUNTED ZONE, name: Name];

Arguments: name is the name to be freed; storage is assumed to be allocated from z.

Results: None.

Errors: None.

Local storage can be initialized for the components of an **NSName.Name** using **InitNameRecord**, which operates on objects of the type **NameStore**. The advantage is that after the termination of a procedure call, any storage related to the **NameStore** object is automatically freed.

NSName.InitNameStore: PROCEDURE [store: LONG POINTER TO NameStore];

Arguments: store is a pointer to a **NameStore** object which gets initialized. The store.record field is set to store.org, store.domain, and store.local. The store.org, store.domain, and store.local lengths are set to zero.

Results: None.

Errors: None.

3.1.3 Comparison and equivalence

Names may be compared for equality or order. The sort order defined for strings is used, ignoring case. The org component of a name is the most significant and local is least significant.

NSName.CompareNames: PROCEDURE [

**n1, n2: Name, ignoreOrg, ignoreDomain, ignoreLocal: BOOLEAN ← FALSE]
RETURNS [NSString.Relation];**

Arguments: n1 and n2 are the names to be compared; if ignoreOrg, ignoreDomain, or ignoreLocal is TRUE, the corresponding component is skipped during the comparison.

Results: The appropriate **NSString.Relation** (less, equal, or greater) is returned.

Errors: None.

NSName.EquivalentNames: PROCEDURE [n1, n2: Name] RETURNS [BOOLEAN] = INLINE ... ;

Arguments: n1 and n2 are the names to be compared.

Results: TRUE is returned if the two names are equivalent, ignoring case.

Errors: None.

3.1.4 Conversion

It is sometimes useful to interconvert the single-string form of a name and the three-part form. **AppendNameToString** converts a three-part name to a single-string name by appending the organization, domain, and local name (in that order) to the string, separated by the separator character.

```
NSName.AppendNameToString: PROCEDURE [
  S: NSString.String, name: Name, resetLengthFirst: BOOLEAN ← FALSE]
  RETURNS [newS: NSString.String];
```

Arguments: **s** is the destination string; **name** is the name to be appended; if **resetLengthFirst** is **TRUE**, then the length of **s** is set to zero, effectively clearing any previous contents.

Results: **newS** is the resultant string.

Errors: **NSString.StringBoundsFault** is raised if **s** has insufficient length.

Single-string names may be converted to three-part names by means of several procedures, depending on the type of allocation desired by the client. In all cases, the string is divided at the separator characters, and any missing components (which are assumed to be the trailing ones) are completed from the name **clientDefaults**. For example, a string containing no separator characters is assumed to have a local name, but no domain or organization; therefore these are taken from **clientDefaults**.

NameFieldsFromString takes an existing name and fills in the components, allocating any which are not already allocated, in a manner similar to **CopyNameFields**. **NameFromString** allocates a new name record and components. **SubdivideName** does no allocation, but instead passes the converted name to a client-supplied procedure, which must copy the information it needs before returning. Storage allocated in the first two operations must be freed as described in §3.1.2.

```
NSName.NameFieldsFromString: PROCEDURE [
  Z: UNCOUNTED ZONE, S: NSString.String, destination: Name, clientDefaults: Name ← NIL];
```

Arguments: **s** is the string to be converted; the resultant components are copied into **destination**; unspecified components in **s** are filled in from **clientDefaults**; **z** is used to allocate any of the components of **destination** that are not already allocated.

Results: None.

Errors: **NameTooSmall** is raised if an already-allocated component of **destination** is too small. **Error** may be raised with the argument **tooManySeparators**.

```
NSName.NameFromString: PROCEDURE [
  Z: UNCOUNTED ZONE, S: NSString.String, clientDefaults: Name ← NIL]
  RETURNS [Name];
```

Arguments: *s* is the string to be converted; unspecified components in *s* are taken from *clientDefaults*; the new name is allocated using *z*.

Results: The newly-created **Name** is returned.

Errors: Error may be raised with the argument **tooManySeparators**.

NSName.SubdivideName: PROCEDURE [
s: NSString.String, callBack: PROCEDURE [Name], clientDefaults: Name ← NIL];

Arguments: *s* is the string to be converted; unspecified components in *s* are taken from *clientDefaults*; the new name is passed to the client's procedure *callBack*, and is not valid after *callBack* returns.

Results: None.

Errors: Error may be raised with the argument **tooManySeparators**.

3.1.5 Errors

Two errors are defined by **NSName**. **NameTooSmall** reports the condition that an already-allocated name had a component that was of insufficient length to accommodate the characters to be inserted. The required string lengths, in bytes, are given by the arguments. The operation may be continued by resuming the signal, supplying a new name with sufficient space.

NSName.NameTooSmall: SIGNAL[
oldName: Name, orgLenNeeded, domainLenNeeded, localLenNeeded: CARDINAL]
RETURNS [newName: Name];

All other exceptional conditions are reported via **Error**.

NSName.Error: ERROR [type: ErrorType];

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

NSName.ErrorType: TYPE = {tooManySeparators};

tooManySeparators More than two separators were found in the string.

3.2 Parameter serialization

This section contains operations to serialize various data structures according to the Courier remote procedure calling protocol.

3.2.1 Serialization of arbitrary structures

The following operations allow arbitrary data structures to be serialized or deserialized. **EncodeParameters** takes an **UNCOUNTED ZONE** and a **Courier.Parameters** (containing a pointer to the data structure and a description of that structure) and returns an allocated array of words, which contains the Courier representation of the data structure. This array should be freed using **FreeEncodedParameters**. **DecodeParameters** takes zones for short and long pointers, an array of words, and a **Courier.Parameters** (containing a pointer to an uninitialized data structure and a description of that structure), and fills in the structure from the array of words. The size of an encoding may be determined without actually creating the encoding by calling **SizeOfSerializedData**. Refer to the Courier section of *Pilot Programmer's Manual* [26] for more information on the use of these types and operations.

NSName.EncodeParameters: PROCEDURE [**z**: UNCOUNTED ZONE, **parameters**: Courier.Parameters]
RETURNS [LONG DESCRIPTOR FOR ARRAY OF UNSPECIFIED];

Arguments: **parameters** refers to and describes the data structure to be encoded; the encoding will be allocated from **z**.

Results: A descriptor for an array containing the encoding is returned.

Errors: **Courier.Error** may be raised; refer to *Pilot Programmer's Manual* [26].

NSName.DecodeParameters: PROCEDURE [
z: UNCOUNTED ZONE, **mdsZone**: MDSZone,
encoding: LONG DESCRIPTOR FOR ARRAY OF UNSPECIFIED, **parameters**: Courier.Parameters];

Arguments: **encoding** contains the Courier representation of the data structure; **parameters** refers to and describes the data structure to be filled from the encoding; the MDS and non-MDS nodes in the decoded structure will be allocated from **mdsZone** and **z**, respectively.

Results: None.

Errors: **Courier.Error** may be raised; refer to *Pilot Programmer's Manual* [26].

FreeEncodedParameters: PROCEDURE [
z: UNCOUNTED ZONE, **encoding**: LONG DESCRIPTOR FOR ARRAY OF UNSPECIFIED];

Arguments: **encoding** contains the structure to be freed, which is assumed to be allocated from **z**.

Results: None.

Errors: None.

NSName.SizeOfSerializedData: PROCEDURE [parameters: Courier.Parameters]
RETURNS [sizeInWords: CARDINAL];

Arguments: **parameters** refers to and describes the data structure.

Results: **sizeInWords** is the number of words that would be occupied by the encoded form of the data structure.

Errors: **Courier.Error** may be raised; refer to *Pilot Programmer's Manual* [26].



NSString

NSString: DEFINITIONS = . . . ;

NSString provides a set of operations to manipulate sequences of characters encoded according to the Xerox *Character Code Standard* [4].

4.1 Strings, substrings, and Mesa strings

A *network.string* (an **NSString.String**) is a run-encoded sequence of characters represented as a series of bytes. The current length of a string is given by its **length**; the maximum permitted length in bytes is expressed by **maxlength**, while actual storage for the string body is referenced by **bytes**.

```
NSString.String: TYPE = RECORD [  
  bytes: LONG POINTER TO PACKED ARRAY OF Environment.Byte,  
  length: CARDINAL ← 0,  
  maxlength: CARDINAL ← 0];
```

Because network strings are defined as record structures, any operation which would change one of the record fields must return a **String** as a result. Normal use requires that the result of an **NSString** operation be assigned to one of the arguments to capture these changes. For this reason, a majority of **NSString** operations return a **String** as a result.

A *substring* describes a portion of a *string*. It is comprised of a *base string*, an offset from the beginning of the base string and a designation of the substring length. The string upon which a substring is defined is given by **base**; **offset** defines the beginning of the substring as an offset in logical characters from the beginning of **base.bytes**; and **length** specifies the number of logical characters following **offset** to be included in the substring.

NSString.SubString: TYPE = LONG POINTER TO SubStringDescriptor;

NSString.SubStringDescriptor: TYPE = RECORD [base: String, offset, length: CARDINAL];

The type, **MesaString**, is defined to distinguish conventional Mesa strings from those supported by **NSString**. Although similar in makeup, a **String** may not be constructed directly from the representation of a Mesa string. A number of operations within the

interface support the use of Mesa strings with network strings and conversion between the two types.

NSString.MesaString: TYPE = LONG STRING;

The constant **nullString** defines the value of an empty string.

NSString.nullString: String = String[NIL, 0];

The type, **Character**, defines a representation for encoded characters. It is used to permit clients access to the representation of logical characters within network strings (see below).

NSString.Character: TYPE = MACHINE DEPENDENT RECORD [chset, code: Environment.Byte];

NSString.Characters: TYPE = LONG DESCRIPTOR FOR ARRAY OF Character;

4.2 Basic operations

Basic operations to create, copy, and free strings are supplied by the procedures **MakeString**, **FreeString**, and **CopyString**, respectively.

MakeString is used to initialize a **String** and its string body, allocating storage for the body from a client-specified zone.

NSString.MakeString: PROC [z: UNCOUNTED ZONE, bytes: CARDINAL] RETURNS [String];

Arguments: z specifies a client-designated zone from which the string body of the result is to be allocated; bytes indicates the desired string body length.

Results: The returned **String** has a maxlen at least as great as bytes, a length of zero, and a string body of sufficient length to hold maxlen bytes.

Errors: **Heap.Error[insufficientSpace]** is raised if not enough storage is provided by the designated heap.

FreeString is used to deallocate storage of a string such as allocated by **MakeString**.

NSString.FreeString: PROC [z: UNCOUNTED ZONE, s: String];

Arguments: z specifies the zone from which the string body of s was allocated; s designates the string to be freed.

Results: Storage allocated to the string body of s is returned to z.

Errors: None.

CopyString produces a copy of a specified string, allocating the string body for its result from a specified zone.

NSString.CopyString: PROC [z: UNCOUNTED ZONE, s: String] RETURNS [String];

Arguments: *z* specifies the zone from which the string body of the copy is to be allocated; *s* designates the string to be copied.

Results: A copy of *s* is returned.

Errors: **Heap.Error[insufficientSpace]** is raised if not enough storage is provided by the designated heap.

LogicalLength returns the number of logical characters in a specified string. Note that because of encoding, this result is not directly related to the number of bytes in the body of the argument string.

NSString.LogicalLength: **PROC [s: String] RETURNS [CARDINAL];**

Arguments: *s* specifies the string of interest.

Results: A count of the logical characters in *s* is returned.

Errors: **NSString.InvalidString** is raised if *s* is not a properly encoded network string.

WordsForString returns the number of words required to represent a given number of string bytes.

NSString.WordsForString: **PROC [bytes: CARDINAL] RETURNS [CARDINAL];**

Arguments: *bytes* specifies the number of string bytes.

Results: A count of words required to represent *bytes* bytes is returned.

Errors: None.

AppendCharacter, **AppendString**, and **AppendSubString** respectively attempt to append a specified character, string, or substring to a specified string. Each operation returns an updated string as a result (the string body of the argument is updated).

NSString.AppendCharacter: **PROC [to: String, from: Character] RETURNS [String];**

NSString.AppendString: **PROC [to: String, from: String] RETURNS [String];**

NSString.AppendSubString: **PROC [to: String, from: SubString] RETURNS [String];**

Arguments: *to* specifies the string to which a character, string, or substring is to be appended; *from* specifies the character, string, or substring to be appended.

Results: Each operation returns an updated **String** (with appropriately revised **length**) as a result.

Errors: **NSString.InvalidString** is raised if *to* is not a properly encoded network string; **NSString.StringBoundsFault** is raised if *to* is not sufficiently long to hold the appended result.

AppendToMesaString attempts to append the characters of a network string to a conventional Mesa string.

NSString.AppendToMesaString: PROC [to: MesaString, from: String];

Arguments: **to** specifies the Mesa string to which the network string **from** is to be appended.

Results: **to** is updated appropriately.

Errors: **NSString.InvalidString** is raised if **from** is not a properly encoded network string; **String.StringBoundsFault** is raised if **to** is not sufficiently long to hold the appended result.

ExpandString produces the sequence of logical characters from the encoded bytes of a network string.

NSString.ExpandString: PROCEDURE [z: UNCOUNTED ZONE, s: String] RETURNS [Characters];

Arguments: **z** specifies the zone from which the result is to be allocated; **s** is the string whose characters are desired.

Results: A descriptor for the set of characters comprising **s** is returned.

Errors: **NSString.InvalidString** is raised if **s** is not a properly encoded network string; **Heap.Error[insufficientSpace]** is raised if **z** cannot supply the necessary storage.

FreeCharacters is used to free storage allocated by calling **ExpandString**.

NSString.FreeCharacters: PROCEDURE [z: UNCOUNTED ZONE, c: Characters];

Arguments: **z** specifies the zone from which storage for **c** was allocated.

Results: Storage allocated to **c** is returned to **z**.

Errors: None.

TruncateString returns the longest valid string having a length less than or equal to the lesser of a specified maximum and the length of an argument string.

NSString.TruncateString: PROC [s: String, bytes: CARDINAL] RETURNS [String];

Arguments: **s** is the string to be truncated; **bytes** specifies a limit to the maximum length of the result in bytes (the result cannot exceed the length of **s** either).

Results: A truncated string is returned as a result; note that the result refers to the same storage as that addressed by **s**.

Errors: **NSString.InvalidString** is raised if **s** is not a properly encoded network string.

4.3 Scanning, comparison, and equivalence

Because network strings are encoded, special means are provided to search for a designated character within a network string, to compare network strings, and to test them for equivalence.

Operations which establish the relationship of two network string values with respect to each other return a result of type **Relation**. The values of **Relation** have the obvious interpretation.

NSString.Relation: TYPE = {less, equal, greater};

ScanForCharacter searches a specified string for a designated character from a given starting point.

**NSString.ScanForCharacter: PROC [c: Character, s: String, start: CARDINAL ← 0]
RETURNS [CARDINAL];**

Arguments: **c** is the character being sought; **s** is the string being searched; **start** specifies the logical character of **s** with which the search should begin.

Results: The returned value is the logical character index of the first occurrence of **c** after the starting point. Failure to find the character is indicated by returning **LAST[CARDINAL]**.

Errors: **NSString.InvalidString** is raised if **s** is not a valid string.

CompareStrings, **CompareSubStrings**, and **CompareStringsAndStems** are used to compare network string values. Each returns a relation as a result indicating the sorted relationship of their string or substring arguments, with the case of characters optionally ignored during the comparison.

**NSString.CompareStrings: PROC [s1, s2: String, ignoreCase: BOOLEAN ← TRUE]
RETURNS [Relation];**

**NSString.CompareSubStrings: PROC [s1, s2: SubString, ignoreCase: BOOLEAN ← TRUE]
RETURNS [Relation];**

**NSString.CompareStringsAndStems: PROC [s1, s2: String, ignoreCase: BOOLEAN ← TRUE]
RETURNS [relation: Relation, equalStems: BOOLEAN];**

Arguments: **s1** and **s2** are the strings (or substrings) to be compared; **ignoreCase** specifies if the case of characters is to be ignored during the comparison.

Results: The sorted relationship of **s1** and **s2** is returned; **equalStems** is **TRUE** if both are equal up to the length of the shorter.

Errors: **NSString.InvalidString** is raised if **s1** or **s2** are not valid strings.

CompareStringsTruncated is used to compare network strings when one or both values are truncated, optionally ignoring the case of individual characters during the comparison.

```
NSString.CompareStringsTruncated: PROC [
    s1, s2: String, trunc1, trunc2: BOOLEAN ← FALSE, ignoreCase: BOOLEAN ← TRUE]
    RETURNS [Relation];
```

Arguments: **s1** and **s2** are the optionally truncated strings to be compared; **trunc1** and **trunc2** indicate the respective truncated state of the strings to be assumed during the comparison; **ignoreCase** specifies if the case of characters is to be ignored during the comparison.

Results: **relation** specifies the sorted relationship of the two strings taking into account assumptions regarding truncation and case. A truncated string is compared as if every character after the last provided is a wildcard character (matches all other characters).

Errors: **NSString.InvalidString** is raised if **s1** or **s2** are not valid strings.

A portion of a network string is deleted via the operation **DeleteSubString**.

```
NSString.DeleteSubString: PROC [s: SubString] RETURNS [String];
```

Arguments: **s** describes the portion of the string to be deleted.

Results: The substring specified by **s** is deleted from its parent string.

Errors: **NSString.InvalidString** is raised if the string referred to by **s** is not a valid string.

EqualCharacter is used to compare a designated character to a specific logical character of a network string.

```
NSString.EqualCharacter: PROC [c: Character, s: String, index: CARDINAL]
    RETURNS [BOOLEAN];
```

Arguments: **c** is the character to be compared; **s** is the string containing the character with which **c** is compared; **index** identifies the logical character of **s** to be compared.

Results: **TRUE** is returned if **c** is equal to the specified logical character of **s**, **FALSE** otherwise.

Errors: None.

The following operations provide convenient, abbreviated interfaces to corresponding string comparison operations (defined above). Each operation may raise the same errors and returns comparable results as the string comparison operations.

```
NSString.EqualString, EqualStrings: PROC [s1, s2: String] RETURNS [BOOLEAN];
```

```
NSString.EqualSubString, EqualSubStrings: PROC [s1, s2: SubString] RETURNS [BOOLEAN];
```

```
NSString.EquivalentString, EquivalentStrings: PROC [s1, s2: String] RETURNS [BOOLEAN];  
NSString.EquivalentSubString, EquivalentSubStrings: PROC [s1, s2: SubString]  
RETURNS [BOOLEAN];
```

4.4 Conversion

A set of routines is provided by **NSString** to convert numbers to network strings, network strings to numbers, Mesa strings to network strings, and to manipulate the case of individual characters.

Each of the following routines appends the string representation of a specified numeric argument to a designated network string. The result of each operation is an updated network string (referring to storage of the argument string).

```
NSString.AppendDecimal: PROC [s: String, n: INTEGER] RETURNS [String];
```

```
NSString.AppendOctal: PROC [s: String, n: UNSPECIFIED] RETURNS [String];
```

```
NSString.AppendLongNumber: PROC [s: String, n: LONG UNSPECIFIED, radix: CARDINAL ← 10]  
RETURNS [String];
```

```
NSString.AppendLongDecimal: PROC [s: String, n: LONG INTEGER] RETURNS [String];
```

```
NSString.AppendNumber: PROC [s: String, n: UNSPECIFIED, radix: CARDINAL ← 10]  
RETURNS [String];
```

Arguments: **s** is the network string to which a number is to be appended; **n** is the numeric quantity to be appended; **radix** is the desired radix of the result.

Results: The result is an updated **String** referring to the storage of the argument string.

Errors: **NSString.InvalidString** is raised if **s** is not a properly encoded string; **NSString.StringBoundsFault** is raised if **s** is not long enough to hold the result.

Each of the following operations attempts to interpret a network string value as a specific numeric type, returning the converted value as a result.

```
NSString.StringToDecimal: PROC [s: String] RETURNS [INTEGER];
```

```
NSString.StringToOctal: PROC [s: String] RETURNS [UNSPECIFIED];
```

```
NSString.StringToLongNumber: PROC [s: String, radix: CARDINAL ← 10]  
RETURNS [LONG UNSPECIFIED];
```

```
NSString.StringToNumber: PROC [s: String, radix: CARDINAL ← 10] RETURNS [UNSPECIFIED];
```

Arguments: *s* is the network string whose value is to be numerically interpreted; *radix* is the radix to be used in the conversion.

Results: The characters of *s* are interpreted with the given radix and the numeric value is returned.

Errors: **NSString.InvalidNumber** is raised if *s* cannot be interpreted as a string of the desired radix; **NSString.InvalidString** is raised if *s* is not a properly encoded network string.

StringFromMesaString is provided to allow network strings to be generated from conventional Mesa strings.

NSString.StringFromMesaString: PROC [s: MesaString] RETURNS [String];

Arguments: *s* is a conventional Mesa string to be converted to a network string.

Results: The resulting **String** contains the same bytes as *s*; data of the Mesa string is not copied, so the validity of the result depends on the continued existence of the Mesa string.

Errors: None.

UpperCase and **LowerCase** provide the client a convenient means to obtain the uppercase and lowercase representation of a character encoding, respectively.

NSString.UpperCase, LowerCase: PROC [c: Character] RETURNS [Character];

Arguments: *c* is the character whose corresponding uppercase or lowercase representation is desired.

Results: The uppercase or lowercase representation of *c* is returned.

Errors: None.

ValidAsMesaString produces a boolean result indicating the validity of interpreting the contents of its argument string as a Mesa string.

NSString.ValidAsMesaString: PROC [s: String] RETURNS [BOOLEAN];

Arguments: *s* is the string whose validity as a Mesa string is to be tested.

Results: **TRUE** is returned if *s* can be validly interpreted as a Mesa string (all characters are valid Mesa characters).

Errors: **NSString.invalidString** is raised if *s* is not a properly encoded network string.

WellFormed produces a boolean result indicating the validity of a given string as a network string.

NSString.WellFormed: PROC[s: String] RETURNS [BOOLEAN];

Arguments: s is the string whose validity as a network string is to be tested.

Results: TRUE is returned if s is a properly encoded network string, FALSE otherwise.

Errors: None.

4.5 Serialization

Certain clients, such as protocol implementors, have the need to serialize and deserialize network strings. For this reason, the **Courier** description **DescribeString** is provided.

NSString.DescribeString: Courier.Description;

When using **DescribeString** to deserialize a string, the **maxlength** field may not be set to the correct value. It is only guaranteed to have a value greater than or equal to the **length** of the resulting **String**. The **maxlength** field may, of course, be set by the client after deserialization to match the **length** field.

4.6 Errors

NSString operations which interpret network strings as numbers may raise the error **InvalidNumber** if the characters of the string cannot validly be interpreted in the desired format.

NSString.InvalidNumber: ERROR;

Any **NSString** procedure which accepts a network string argument may raise the error **InvalidString** if the string is not a properly encoded network string.

NSString.InvalidString: ERROR;

StringBoundsFault is raised during append operations when the destination string body is too short to hold the appended result. If the client wishes to continue the operation in such a case, he must provide a new, larger string, whose contents are identical to those of the old string prior to the call which raised the signal.

NSString.StringBoundsFault: SIGNAL [old: String, increaseBy: CARDINAL]
RETURNS [new: String];

XEROX



Services 8.0 Programmer's Guide

Authentication Programmer's Manual

November 1984

PRELIMINARY

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



Table of contents

1	Introduction	1-1
1.1	Definition of terms	1-1
1.2	Encryption and security	1-2
1.3	Strong and simple authentication	1-2
1.4	Strong authentication algorithm	1-3
1.5	Simple authentication algorithm	1-4
1.6	Passwords and keys	1-4
1.7	Authentication's clients	1-4
2	Interfaces	2-1
2.1	Credential and verifier declarations	2-1
2.2	Other types and constants	2-2
2.3	Errors	2-2
2.4	Identities	2-3
2.5	Initiator	2-5
2.6	Recipient	2-6
2.7	Key and password administration	2-8
2.7.1	Access controls	2-8
2.7.2	Strong keys	2-8
2.7.3	Simple keys	2-9
2.8	Other utilities	2-10
2.9	AuthSession.mesa	2-12
3	Standard authentication scenario	3-1
3.1	Identities	3-1
3.2	Initiator	3-1
3.3	Recipient	3-1
3.4	Levels	3-2
3.5	Sample code	3-2
3.5.1	Initiator	3-2
3.5.2	Recipient	3-4

Table of contents

Introduction

This document describes the stub interfaces of the *Authentication Service* (AS). It is intended as a reference for the designers and implementors of client programs. It provides sufficient information to allow programmers to understand and use the facilities available through the public interface **Auth.mesa**, as well as the friends' level interface **AuthSession.mesa**.

Section 1, *Introduction*, is an overview of what the Authentication Service is all about. Section 2, *Nuts and bolts*, is a description of the authentication stub interfaces. Section 3, *Standard authentication scenario*, describes the intended use of the interface functions. More detailed information about the Authentication protocol and the description of the Authentication Courier program can be found in *Authentication Protocol* [2]. For information on the Authentication functional specification, see the *Clearinghouse Functional Specification* [6].

1.1 Definition of terms

<i>Authentication Service</i>	or simply AS. The distributed service supplied by a set of cooperating authentication servers.
<i>authentication server</i>	a server machine running Authentication Service software; one <i>instance</i> of the Authentication Service, or, the software running on such a machine.
<i>authentication stub</i>	a piece of software running in the client's machine which acts as an agent for accessing the Authentication Service. The stub may interact with one or more authentication servers to perform a given function for the client. The stub supplies all the Mesa interfaces described in this document.
<i>authentication client</i>	a piece of software which calls the functions provided by the authentication stub. Authentication clients can be, and often are, stubs or servers of other services.

1.2 Encryption and security

The function of Authentication is to certify that the two parties of a conversation are who they claim to be. In order to do this we must securely distribute information about the participants in the conversation. Because the communication paths of a distributed system are easy to tap, any information which is to be securely distributed must be encrypted. However, the cost of software encryption is prohibitively high. [Not including the key preprocessing overhead, our optimized implementation of the Data Encryption Standard (DES) takes eleven milliseconds to encrypt one eight byte block.] Until hardware support of encryption is available, it is not feasible to encrypt all data flowing over the Internet. A major constraint on the design of the authentication scheme is that it not rely on the encryption of large amounts of data.

1.3 Strong and simple authentication

There are several classes of devices which may be attached to the Xerox office information system. First, there are the 8000 series of workstations and network servers. The software implemented on these machines has available to it a powerful processor, a large amount of memory, and a high speed rigid disk. The second class of devices includes smaller workstations, such as the Xerox 860, which have a micro-processor, less memory, and floppy disks. Finally, there are devices such as simple terminals, with no processing power available at all. The authentication scheme must allow all of these devices to participate in activities on the Ethernet. It is not permissible, for example, to require that a "smart" typewriter implement a complex protocol or encryption algorithm in order to talk to the *Interactive Terminal Service*. On the other hand, we must not allow the existence of simple machines to thwart our attempt to provide a reasonable level of security for the users of more powerful machines. This problem is addressed by defining two levels of authentication, referred to as *strong* and *simple* Authentication.

Each user has two different keys; a *strong* and a *simple* key. The strong key is used on a machine which implements the strong authentication scheme. The simple key is used when logging in through some device which is incapable of providing strong authentication. A service provides some subset of its full set of privileges to a user logged in with a simple key. For example, she might be able to read and send mail but not delete it from the mail server. [The precise subset of privileges provided to a user with simple authentication is determined by the implementors of each service.] Essentially, a service will trust only so far a user logged in with a simple key, because simple keys can be stolen more easily than strong ones.

A machine which implements strong authentication will never reveal that key by transmitting it over the network. A machine which implements simple authentication does not make the same guarantee. Therefore a user who inadvertently types her strong password instead of her simple password may be revealing the strong key to an eavesdropper (e.g., when a user at a dumb terminal dials into the network through a *Communications Interface Unit*). It is the user's responsibility to guard the strong password and avoid typing it when the simple one is required. In a proper implementation, a service will not accept the strong key when the simple one is needed.

1.4 Strong authentication algorithm

All users are registered with the Authentication Service, along with their strong and simple keys. A given user's keys are known only to that user and the Authentication Service.

There are three parties involved in the authentication protocol: the *initiator*, who initiates the proceedings; the *recipient*, the party with whom the initiator wishes to communicate; and the *Authentication Service* (AS). Typically, the initiator is a stub for some service, and the recipient is a server for that service.

When the initiator wishes to communicate with some recipient, she obtains an object from the AS which she uses to identify herself to that recipient, much as a traveler uses her passport to identify herself to authorities at the border of each country she wishes to enter. This object is called the client's *credentials*. (Note: Unlike a traveler, whose one passport is good in many different countries, a client must have a set of credentials for *each* recipient with whom she wishes to interact.) The client presents her credentials with every call to the recipient. The credentials contain data encrypted in such a way that it is comprehensible to only that particular recipient.

Contained within the credentials is the identity of the initiator and a special *conversation key*. The conversation key is generated by the AS and returned to the initiator in a secure fashion at the time the credentials are obtained. If encryption hardware were available, the conversation key would be used to encrypt all information flowing between the initiator and the recipient. Since such hardware is *not* available at this time, every message which flows between the initiator and the recipient includes a unique sequence number (actually, the system time) encrypted with the conversation key. This encrypted value is called the *verifier*. (The overhead required to encrypt the small, fixed length sequence number is much lower than that required to encrypt the entire message.) In addition, whatever encryption of information is cost effective will be done with this key. Notice that anyone can steal a set of credentials but only the proper service can decrypt them to obtain the conversation key. The conversation key is thus known only to the initiator and the recipient.

Without the conversation key, it is impossible to generate a valid verifier. Since a verifier may not be reused, the initiator and the recipient always know that a message containing a valid verifier came from the other. Also notice that the recipient can decrypt the credentials and the verifier without recourse to the AS. The recipient's portion of the strong Authentication protocol is entirely local to the recipient.

Credentials expire. This prevents an intruder who somehow obtains a set of credentials and the conversation key from using those credentials indefinitely. The lifetime of a set of credentials is determined by the Authentication Service; typically, that time is between a few hours and a few days. A set of credentials may be used in any number of calls to the recipient until it expires. A verifier may be used in exactly one call to the recipient; a fresh verifier must be computed for each message to be sent. A verifier will expire shortly after it has been created.

1.5 Simple authentication algorithm

For simple Authentication, credentials and verifiers are passed in the same manner that they are for strong Authentication, with two major differences: nothing is encrypted and the credentials and verifier are created without the aid of the Authentication Service. For simple Authentication, the credentials are the name of the initiator, and the verifier is her simple key. (Note that the credentials/verifier pair is constant both for different messages to a single recipient and for different recipients.) To validate the credentials and verifier, the recipient contacts the AS, which checks whether the verifier (simple key) matches the credentials (initiator's name).

1.6 Passwords and keys

As far as the Authentication protocols are concerned, there is no such thing as a *password*. National Bureau of Standards Data Encryption Standard (DES) encryption keys are the only things which are used by the Authentication machinery. This key is a bit cumbersome for human beings to remember and type, however. Therefore, we supply a function which converts a character string into a DES key. This allows users to deal with mnemonic passwords. This function is defined in *Authentication Protocol* [2].

Passwords should be chosen carefully. Unfortunately, the passwords that are easiest to remember are often easiest to guess or to crack. A password needs to be long enough to foil an exhaustive search for it. A password with eleven or twelve characters is probably long enough. Remember that an intelligent password search will search a space of likely passwords. Two such spaces, for example, are all combinations of one to five letters, or all words in a dictionary. The password should also be difficult to guess. Your name, your boyfriend's name, and your street address all make very bad passwords.

1.7 Authentication's clients

The Authentication Service provides a standardized protocol permitting secure communication. Taking advantage of this protocol is the responsibility of the actual parties of a conversation. The Authentication Service provides credentials and verifiers, and procedures for creating and checking them. The enforcement of the Authentication protocol is done by the actual communicators. It is they who must acquire credentials and verifiers, send them with every message, and check them on every receipt.



Interfaces

This chapter describes the Authentication stub interfaces, **Auth.mesa** and **AuthSession.mesa**. Most of this section describes **Auth.mesa**; the interface to **AuthSession.mesa** is described at the end.

Although the internal algorithms for strong and simple authentication differ in several respects, the authentication stub supports both styles with the same interface.

Auth: Definitions = ...;

2.1 Credential and verifier declarations

```
Creditentials: TYPE = MACHINE DEPENDENT RECORD [
    flavor: PRIVATE Flavor,
    value: LONG DESCRIPTOR FOR ARRAY OF UNSPECIFIED];
```

For historical reasons the **flavor** field is private. The operation **GetFlavor** may be used to extract this field.

```
Flavor: TYPE = MACHINE DEPENDENT{
    simple (0), -- "Trust me!" authentication.
    strong (1), -- Good authentication.
    unknown (LAST[CARDINAL])};
```

Verifier: TYPE = LONG DESCRIPTOR FOR ARRAY OF UNSPECIFIED;

nullCredentials: Credentials = [simple, NIL];

A **nullCredentials** value represents the absence of credentials. It is *not* the same as a set of simple credentials for the null initiator name.

nullVerifier: Verifier = NIL;

A **nullVerifier** value represents the absence of a verifier. It is *not* the result of encoding a **nullHashedPassword**.

2.2 Other types and constants

HashedPassword: TYPE = CARDINAL;

A **HashedPassword** is the key derived from a simple password.

nullHashedPassword: HashedPassword = 0;

Key: TYPE = PACKED ARRAY [0..3] OF UNSPECIFIED;

A **Key** is the key derived from a strong password. This is a DES encryption key. These keys are four words long, and contain 56 data bits and eight parity bits.

nullKey: Key = ALL[0];

A **nullKey** value represents the absence of a key. A **nullKey** has incorrect parity, and so is not a legal key.

nullHostNumber: System.HostNumber = System.nullHostNumber;

2.3 Errors

AuthenticationError: ERROR [reason: AuthenticationProblem];

An **AuthenticationError** indicates a problem with the credentials or verifier. These are raised by **Authenticate** and its variants.

```
AuthenticationProblem: TYPE = MACHINE DEPENDENT {
    credentialsInvalid(0),
    verifierInvalid(1),
    verifierExpired(2),           -- The verifier expired in transit.
    verifierReused(3),           -- An intruder could be re-using this verifier.
    credentialsExpired(4),        -- The credentials have expired.
    inappropriateCredentials(5),  -- You passed strong and it wanted simple or vice versa.
    (LAST[CARDINAL])};
```

For simple credentials: **credentialsInvalid** indicates that the initiator's name is not registered in the Clearinghouse or that the credentials are improperly formed. **verifierInvalid** indicates that the simple key stored with the AS is not the same as that in the verifier. **inappropriateCredentials** indicates that simple credentials are not allowed in this context. The other problems do not apply to simple credentials.

For strong credentials: **credentialsInvalid** indicates that the credentials could not be successfully decrypted (which could indicate that the recipient has incorrectly registered her key with the AS). **verifierInvalid** indicates that the verifier is complete trash or hopelessly out of date. (Currently, a verifier is hopelessly out of date if its date is two or more days in the past or ten or more minutes in the future.) **verifierExpired** indicates that the verifier is older than the acceptable clock discrepancy. **verifierReused** could indicate that an intruder is attempting to reuse a verifier but more likely indicates that the initiator is using a given verifier more than once. (If it occurs, look for places in your lowest level communications code where an operation is retried without computing a fresh verifier.) **credentialsExpired** indicates that the credentials are too old and fresh ones

should be obtained from the AS. **inappropriateCredentials** indicates that strong credentials are not allowed in this context. Note that **AuthenticationError** is raised in the recipient, not the initiator. The initiator must be notified by the recipient.

CallError: ERROR [reason: CallProblem, whichArg: WhichArg];

```
CallProblem: TYPE = MACHINE DEPENDENT {
    tooBusy(0),
    cannotReachAS(1),
    keysUnavailable(2),
    strongKeyDoesNotExist(3),
    simpleKeyDoesNotExist(4),
    badKey(5),
    -- The following problems may occur during CreateStrongKey
    -- and CreateSimpleKey operations:
    accessRightsInsufficient(6),
    strongKeyAlreadyRegistered(7),
    simpleKeyAlreadyRegistered(8),
    domainForNewKeyUnavailable(9),
    domainForNewKeyUnknown(10),
    badNameForNewKey(11),
    databaseFull(12),
    -- The following problem is a catch-all:
    other(13),
    (LAST[CARDINAL])};
```

```
WhichArg: TYPE = MACHINE DEPENDENT {
    initiator(1),
    recipient(2),
    (LAST[CARDINAL])};
```

A **CallError** indicates a problem with a call to the Authentication Service. **whichArg** indicates which argument caused the error in cases where there might be some ambiguity. For example, if **reason** is **keysUnavailable** and **whichArg** is **recipient**, this indicates that the recipient's keys (as opposed to the initiator's keys) were not available. The AS stores its keys in the Clearinghouse, so many of these errors reflect problems with the Clearinghouse.

OrphanConversation: ERROR;

Raised only by **Refresh**. See §2.5 for the circumstances under which this error is raised.

2.4 Identities

An **IdentityHandle** (or "identity" for short) contains the client's name, password, strong key, and simple key. **Note:** A server generally has neither a password nor a simple key. An identity for a server will thus have null values for these fields. An identity is used anywhere the client's name and/or password are required, such as when she initiates a conversation or examines a set of credentials received from someone else. An **IdentityHandle** also contains a list of all the active conversations created using this

identity and a cache of inactive conversations which may be recycled. Identities are monitored records and thus may be shared by multiple processes.

```
IdentityHandle: TYPE = LONG POINTER TO IdentityObject;
IdentityObject: TYPE;
```

```
Makeldentity: PROCEDURE [
    myName: NSName.Name,
    password: NSString.String,
    z: UNCOUNTED ZONE,
    style: Flavor ← strong,
    dontCheck: BOOLEAN ← FALSE]
    RETURNS [identity: IdentityHandle];
```

```
MakeStrongIdentityUsingKey: PROCEDURE [
    myName: NSName.Name,
    myKey: Key,
    z: UNCOUNTED ZONE,
    dontCheck: BOOLEAN ← FALSE]
    RETURNS [identity: IdentityHandle];
```

Makeldentity creates an **IdentityObject**, and returns an **IdentityHandle** for it. All conversations initiated using this identity will use the flavor of credentials indicated by **style**. The **password** provided here should be the one appropriate for the given style. If **dontCheck** is **TRUE** then **myName** and **password** are not checked for validity at this time. This is useful in contexts where it is necessary to create an **IdentityHandle** even if the Authentication Service is unavailable. For example, services and workstations should not fail to boot due to the lack of an authentication server. If **dontCheck** is **FALSE**, the AS is contacted during the evaluation of this procedure call,, and **CallError** may be raised. Two of the most common **CallErrors** are **strongKeyDoesNotExist**, which indicates that the client's strong key is not registered in the Clearinghouse, and **badKey**, which indicates that the key registered in the Clearinghouse is not the same as the one derived from the password passed to **Makeldentity** (e.g., the user typed his password wrong). Clients should create the strongest identity appropriate for the application. If the identity is to be passed to one of the credentials checking operations (e.g., **Authenticate**, **ExtractCredentialsDetails**, etc.) then its style *must* be **strong**. The identity should be freed by **Freeldentity**.

MakeStrongIdentityUsingKey is similar to **Makeldentity**, but it takes a key instead of a password. It is needed so that servers, which have keys but no passwords, may make identities for themselves. This operation may only be used to make strong identities.

```
Freeldentity: PROCEDURE [
    identityPtr: LONG POINTER TO IdentityHandle,
    z: UNCOUNTED ZONE];
```

Frees the storage associated with an **IdentityHandle**. All conversations with **identityPtr** ↑ as their owning identity will become *orphan* conversations. **Freeldentity** is a noop if **identityPtr** ↑ is **NIL**. **identityPtr** ↑ is smashed to **NIL**.

2.5 Initiator

A **ConversationHandle** (or "conversation" for short) contains information relevant to a conversation with a specific recipient: the recipient's name, the conversation key, the credentials, and the last verifier generated in this conversation. Since credentials can expire, it is possible to cause new credentials to be injected into an established conversation using **Refresh**.

Verifiers must arrive at their destination in the exact order that they were produced. (This is because the verifier replay prevention machinery uses the fact that verifiers have an ordering sequence; the recipient assumes that all verifiers prior to the current one have been previously exposed to the network.) Because the scheduling of Mesa processes is not predictable, it is imperative that multiple processes do not share a conversation. Otherwise, verifiers could arrive out of order and be rejected.

ConversationHandle: TYPE = LONG POINTER TO **ConversationObject**;
ConversationObject: TYPE;

```
Initiate: PROCEDURE [
    identity: IdentityHandle,
    recipientsName: NSName.Name,
    recipientsHostNumber: System.HostNumber ← nullHostNumber,
    z: UNCOUNTED ZONE]
    RETURNS [conversation: ConversationHandle];
```

This operation creates a **ConversationHandle**. If the identity style is **strong**, then a set of credentials may need to be fetched from the AS. (The cache of conversations associated with the identity often makes this unnecessary.) If the identity style is **strong** and the client wishes to supply the host number of the recipient, he may do so at this time, and not have to supply the host address later when he makes calls to **CheckOutNextVerifier**. This operation may raise **CallError**. The conversation should be freed by **Terminate**. **identity** is the *owning identity* of the conversation created.

```
Terminate: PROCEDURE [
    conversationPtr: LONG POINTER TO ConversationHandle,
    z: UNCOUNTED ZONE];
```

Terminate frees storage associated with this authentication conversation. There should be no checked out credentials or verifiers when **Terminate** is called, although there is no way for the current implementation to enforce that. **Terminate** is a noop if **conversationPtr** ↑ is **NIL**. **conversationPtr** ↑ is smashed to **NIL**.

Refresh: PROCEDURE [conversation: ConversationHandle];

This causes new credentials to be retrieved from the Authentication Service and stored in the **conversationHandle**. If the conversation is an orphan, then **OrphanConversation** will be raised. **CallError** may also be raised. **Refresh** will fail if the AS is unavailable, or the password for the conversation's owning identity has been changed. **Refresh** can be done within a stub, transparently to its clients.

```
CheckOutCredsAndNextVerifier: PROCEDURE [
    conversation: ConversationHandle,
    recipientsHostNumber: System.HostNumber ← nullHostNumber]
    RETURNS [creds: Credentials, verifier: Verifier];
```

```
CheckOutCredentials: PROCEDURE [
    conversation: ConversationHandle]
    RETURNS [creds: Credentials];
```

```
CheckOutNextVerifier: PROCEDURE [
    conversation: ConversationHandle,
    recipientsHostNumber: System.HostNumber ← nullHostNumber]
    RETURNS [verifier: Verifier];
```

CheckOutCredsAndNextVerifier returns the conversation credentials and a fresh verifier. **CheckOutCredentials** just returns the conversation credentials. **CheckOutNextVerifier** returns only a fresh verifier. The credentials for a particular conversation are invariant.

Warning: To avoid excess storage allocation, copying and freeing, **CheckOutCredsAndNextVerifier**, **CheckOutCredentials** and **CheckOutNextVerifier** return pointers to data structures owned by the conversation; these pointers will become invalid when the conversation is terminated. The credentials and verifier returned by these functions should *not* be freed.

```
ReplyVerifierChecks: PROCEDURE [
    conversation: ConversationHandle, verifierToCheck: Verifier]
    RETURNS [verifierOK: BOOLEAN];
```

This operation is invoked on the initiator's side of a conversation. It confirms that **verifierToCheck** is the proper response to the last verifier created within this conversation. This operation always returns **TRUE** if **conversation** is not a strong conversation.

2.6 Recipient

```
Authenticate: PROCEDURE [
    recipient: IdentityHandle,
    credentialsToCheck: Credentials,
    verifierToCheck: Verifier,
    z: UNCOUNTED ZONE ← NIL]
    RETURNS [initiator: NSName.Name];
```

```
AuthenticateWithExpiredCredentials: PROCEDURE [
    recipient: IdentityHandle,
    credentialsToCheck: Credentials,
    verifierToCheck: Verifier,
    z: UNCOUNTED ZONE ← NIL]
    RETURNS [initiator: NSName.Name];
```

```
AuthenticateAndReply: PROCEDURE [
    recipient: IdentityHandle,
    credentialsToCheck: Credentials,
    verifierToCheck: Verifier,
```

z: UNCOUNTED ZONE]

RETURNS [initiator : NSName.Name, replyVerifier: Verifier];

Authenticating strong and simple credentials are slightly different operations. For strong authentication, the recipient must decrypt the credentials using her strong key. This reveals the initiator's name and the conversation key. The conversation key is then used to decrypt the verifier. Strong **Authenticate** is done entirely locally. For simple authentication, the recipient asks the AS whether the simple key in the verifier belongs to the initiator specified by the credentials. The recipient must contact the AS to do simple authentication. Consequently, **CallError** can be raised by simple **Authenticate**.

Authenticate checks the validity of the given credentials and verifier. **AuthenticationError** is raised if there is anything amiss; if **Authenticate** returns normally then the credentials were acceptable. **z** is an optional heap. If **z** is supplied and the credentials were acceptable, the initiator's name is extracted from the credentials and returned, using space allocated from **z**. If **z** is defaulted to **NIL**, then it is assumed that the caller is not interested in the initiator's name and no storage is allocated (and the initiator returned is **NIL**). **recipient** is the identity of the receiver of the credentials (i.e., the service receiving the credentials). **recipient** *must* be a strong-style identity.

AuthenticateWithExpiredCredentials is similar to **Authenticate** but will tolerate credentials which have expired. This is specifically for use in session-based protocols (e.g., Filing) in which the session may continue to live after the expiration date of the credentials and it is deemed an acceptable security risk to keep the session alive in this case. The **AuthSession.AuthenticateWithExpiredCredentials** operation is preferred over this operation for performance reasons: the **AuthSession** version does not bother to recheck simple credentials, since they were checked at the beginning of the session and rechecking them is expensive and unnecessary.

AuthenticateAndReply is similar to **Authenticate** but a reply verifier is computed. Note that **z** is *not* optional; it is used to allocate storage for **replyVerifier**, which the client must return using **FreeVerifier**. **AuthSession.NextReplyVerifier** operation is preferred in session-based protocols for the performance reasons noted above.

GetFlavor: PROCEDURE [creds: Credentials]

RETURNS [flavor: Flavor];

This operation returns the flavor of credentials. Access control decisions should be based partially on the credential's flavor.

FreeVerifier: PROCEDURE [

verifierPtr: LONG POINTER TO Verifier, z: UNCOUNTED ZONE];

This operation frees the verifier pointed to by **verifierPtr** and smashes **nullVerifier** into **verifierPtr**. It will tolerate **nullVerifiers**. Use **FreeVerifier** to free verifiers returned by **AuthenticateAndReply**.

Warning: Do *not* use **FreeVerifier** to free verifiers returned by **CheckOutCredsAndNextVerifier**, or **CheckOutNextVerifier**.

2.7 Key and password administration

2.7.1 Access controls

The AS stores keys in the Clearinghouse's database. Therefore, these operations are subject to the Clearinghouse's access control restrictions, reflect Clearinghouse problems, etc. A strong identity must be passed to all of these routines, as they modify the Clearinghouse database. Both strong and simple keys can only be created or deleted by an administrator for the domain of **name**. **CreateStrongKey**, **DeleteStrongKey**, **CreateSimpleKey** and **DeleteSimpleKey** are the procedures subject to this restriction. **ChangeMyPasswords**, **ChangeStrongKey** and **ChangeSimpleKey** in contrast, modify the keys of the identity **identity**. Note also that **Refresh** will fail after any of these three operations.

2.7.2 Strong keys

```
ChangeMyPasswords: PROCEDURE [
    identity: IdentityHandle,
    newPassword: NSString.String,
    z: UNCOUNTED ZONE,
    changeStrong, changeSimple: BOOLEAN ← TRUE];
```

This operation changes the client's strong and/or simple keys in the AS database. It may raise **CallError**. The **identity** is altered to reflect the new value of the keys. To be really secure, passwords should be *at least* twelve characters long. The zone **z** is not used for anything and may be **NIL**.

```
CreateStrongKey: PROCEDURE[
    identity: IdentityHandle,
    name: NSName.Name,
    newStrongKey: Key];
```

This operation adds the new strong key **newStrongKey** to the AS database. It may raise **CallError**. **name** must already exist in the Clearinghouse.

```
ChangeStrongKey: PROCEDURE [
    identity: IdentityHandle,
    newStrongKey: Key];
```

This operation replaces the strong key for **identity** in the AS database with **newStrongKey**. It may raise **CallError**. The identity is changed to reflect the new value of the key. **AuthenticationError[inappropriateCredentials]** is raised if **identity** is not a strong identity. Conversations and identities created after this operation will need to use the new key. Existing conversations are not affected, except that **Refresh** will fail.

```
DeleteStrongKey: PROCEDURE [
    identity: IdentityHandle,
    name: NSName.Name];
```

This operation deletes the strong key for **name**. It may raise **CallError**. After this operation, **name** has no strong key, and so can't create new conversations or identities. Existing conversations will continue to work, except that **Refresh** will fail.

PasswordStringToKey: PROCEDURE [password: NSString.String]
RETURNS [key: Key];

This operation computes a DES key from a password string according to the algorithm described in the *Authentication Protocol* [2]. Case is ignored for characters in character set zero.

GetRandomKey: PROCEDURE RETURNS [key: Key];

This operation makes a random strong key. It is useful for making up keys for servers, which have keys but no passwords.

2.7.3 Simple keys

CreateSimpleKey: PROCEDURE [
identity: IdentityHandle,
name: NSName.Name,
newSimpleKey: HashedPassword];

This operation adds the new simple key **newSimpleKey** to the AS database. It may raise **CallError**. **name** must already exist in the Clearinghouse.

ChangeSimpleKey: PROCEDURE [
identity: IdentityHandle,
newSimpleKey: HashedPassword];

This operation replaces the simple key of **identity** in the AS database with **newSimpleKey**. It may raise **CallError**. **name** must already exist in the Clearinghouse. The identity is changed to reflect the new value of the key. **Authenticate** will fail for conversations created before this operation.

DeleteSimpleKey: PROCEDURE [
identity: IdentityHandle, name: NSName.Name];

This operation deletes the simple key for **name**. It may raise **CallError**. **Authenticate** will fail for conversations created before this operation.

HashSimplePassword: PROCEDURE [password: NSString.String]
RETURNS [hashedPassword: HashedPassword];

This operation turns a password string into a **hashedPassword** according to the password-hashing algorithm described in the *Authentication Protocol* [2]. Case is ignored for characters in character set zero.

2.8 Other utilities

DescribeCredentials: Courier.Description;
DescribeVerifier: Courier.Description;

These operations are supplied for the implementors of protocols which use authentication.
Note that **Key** and **HashedPassword** do not require description routines.

ConversationProc: TYPE = PROCEDURE [
 thisConversation: ConversationHandle]
 RETURNS [stop: BOOLEAN \leftarrow FALSE];

A **ConversationProc** must be supplied to the **EnumerateConversations** operation.

EnumerateConversations: PROCEDURE [
 identity: IdentityHandle,
 eachConv: ConversationProc];

This operation is used to enumerate all the active conversations attached to an **IdentityHandle**. **eachConv** is called once for each conversation belonging to the identity. It is permissible to **Terminate** conversations from within the callback proc.

CheckSimpleCredentials: PROCEDURE [
 creds: Credentials,
 verifier: Verifier]
 RETURNS [ok: BOOLEAN];

This operation calls the AS to check the given simple credentials. It may raise **CallError**.
CheckSimpleCredentials is the guts of simple **Authenticate**.

FetchStrongCredentials: PROCEDURE [
 initiator, recipient: NSName.Name,
 initiatorsStrongKey: Key,
 z: UNCOUNTED ZONE]
 RETURNS [creds: Credentials, conversationKey: Key];

This operation calls the Authentication Service directly to get a bare set of credentials.
(**Note:** It is not clear what use this will be without some means of creating a conversation containing it.) The client is responsible for freeing **creds**. **FetchStrongCredentials** is the guts of strong **Initiate**.

FreeCredentials: PROCEDURE [
 credsPtr: LONG POINTER TO Credentials,
 z: UNCOUNTED ZONE];

This operation frees the credentials pointed to by **credsPtr** and smashes **nullCredentials** into **credsPtr** \uparrow . It will tolerate **nullCredentials**.

Warning: Do not use **FreeCredentials** to free credentials returned by **CheckOutCredsAndNextVerifier**, or **CheckOutCredentials**.

GetConversationDetails: PROCEDURE [conversation: ConversationHandle]
 RETURNS [

```
recipient: NSName.Name,  
recipientsHostNumber: System.HostNumber,  
creds: Credentials,  
conversationKey: Key,  
owner: IdentityHandle];
```

This operation extracts information buried in the conversation. If the conversation style is simple, **conversationKey** will be the **nullKey**. Because the return values **recipient** and **owner** point to internal data structures which are owned by the conversation, this operation should be used with care. In particular, the **recipient** should be copied before it is passed to any other operation in this interface.

```
GetIdentityDetails: PROCEDURE [identity: IdentityHandle]  
RETURNS [  
    name: NSName.Name,  
    password: NSString.String,  
    style: Flavor];
```

This operation extracts information buried in the identity. Because the return values **name** and **password** point to internal data structures which are owned by the identity, this operation should be used with care.

```
ExtractHashedPassword: PROCEDURE [simpleVerifier: Verifier]  
RETURNS [hashedPassword: HashedPassword];
```

This operation extracts the initiator's hashed password from the verifier. It should only be passed simple verifiers. **AuthenticationError[verifierInvalid]** is raised if **simpleVerifier** is a strong verifier.

```
ExtractCredentialsDetails: PROCEDURE [  
    recipientsKey: Key,  
    credentialsToCheck: Credentials,  
    z: UNCOUNTED ZONE ← NIL]  
RETURNS [  
    flavor: Flavor,  
    conversationKey: Key,  
    -- conversationKey is uninteresting if  
    -- credentialsToCheck aren't strong.  
    expirationTime: System.GreenwichMeanTime,  
    -- expirationTime is uninteresting if  
    -- credentialsToCheck aren't strong.  
    initiator: NSName.Name,  
    badCredentials: BOOLEAN];
```

This operation extracts the salient data from a set of strong or simple credentials. If **z** is supplied then the initiator's name is extracted from the credentials and returned, using storage allocated from **z**. It is up to the client to return this storage. If **z** is not supplied, no storage is allocated and the initiator returned is **NIL**. If **nullCredentials** are passed to **ExtractCredentialsDetails** then the initiator returned is **NIL** and no storage is allocated. If **badCredentials** is **TRUE** then none of the other returned values are meaningful and no storage is allocated.

```
CopyCredentials: PROCEDURE [
  credentials: Credentials,
  z: UNCOUNTED ZONE]
RETURNS [newCopy: Credentials];
```

This operation makes a copy of the given credentials allocating space from **z**. The client should return the storage when she's done using the **FreeCredentials** operation.

```
CopyIdentity: PROCEDURE [
  identity: IdentityHandle,
  z: UNCOUNTED ZONE]
RETURNS [newCopy: IdentityHandle];
```

This operation makes a copy of the given identity allocating space from **z**. The client should return the storage using the **Freeldentity** operation.

```
EqualCredentials: PROCEDURE [creds1, creds2: Credentials]
RETURNS [equal: BOOLEAN];
```

This operation efficiently compares credentials (of all flavors) for equality.

2.9 AuthSession.mesa

AuthSession contains authentication operations for session-based services. A session-based protocol is one in which information about the transaction in progress is preserved from one call on the service to another. There is an *initial* call of the session, and a number of *subsequent* calls. This makes possible a number of efficiencies. In particular, when doing simple authentication with such a session, the simple credentials can be checked only at the initial call, and not on subsequent calls. This is valuable, as authenticating simple credentials requires contacting the Authentication Service, and so is expensive.

```
InitialAuthenticate: PROCEDURE [
  recipient: Auth.IdentityHandle,
  credentialsToCheck: Auth.Credentials
  verifierToCheck: Auth.Verifier
  z: UNCOUNTED ZONE]
RETURNS [initiator: NSName.Name, replyVerifier: Auth.Verifier];
```

This operation authenticates the credentials and computes a reply verifier. **InitialAuthenticate** is meant to be used at the start of a session. It checks both strong and simple credentials. The client is responsible for freeing **initiator** and **replyVerifier**. **z** must be a valid heap.

```
AuthenticateWithExpiredCredentials: PROCEDURE [
  recipient: Auth.IdentityHandle,
  credentialsToCheck: Auth.Credentials,
  verifierToCheck: Auth.Verifier,
  z: UNCOUNTED ZONE ← NIL]
RETURNS [replyVerifier: Auth.Verifier];
```

This operation is identical to **Auth.AuthenticateWithExpiredCredentials** except that simple credentials are not checked—they were presumably checked at the beginning of the session with **InitialAuthenticate**.

```
NextReplyVerifier: PROCEDURE [
    recipient: Auth.IdentityHandle,
    credentialsToCheck: Auth.Credentials,
    verifierToCheck: Auth.Verifier,
    z: UNCOUNTED ZONE ← NIL]
    RETURNS [replyVerifier: Auth.Verifier];
```

This operation is identical to **Auth.AuthenticateAndReply** except that simple credentials are not checked.

Standard authentication scenario

3.1 Identities

Both the initiator and the recipient have an **IdentityHandle** which contains their name, password, and other information. Before either initiating a conversation or checking a set of credentials received from someone else, the client must assert her identity. This is usually done with **MakIdentity** in the initiator which is a stub, and **MakeStrongIdentityUsingKey** in the recipient which is a server. Identities are monitored records, and so can be shared among multiple processes. **FreelIdentity** is used to free the identity.

3.2 Initiator

The initiator must create a conversation with **Initiate** for each recipient with whom she wishes to interact. The conversation contains the credentials and such things as the last verifier created for this conversation (to avoid handing out duplicate verifiers) and the conversation key. In every message the initiator sends the recipient, she includes a set of credentials and a verifier. These are obtained from **CheckOutCredsAndNextVerifier**. The authenticity of the recipient's response is checked with **ReplyVerifierChecks**. **Terminate** is used to free the conversation obtained through **Initiate**. It is possible for a set of credentials to expire in the middle of a conversation. In this case, the initiator may invoke the **Refresh** operation to cause a new set of credentials to be obtained from the Authentication service (avoiding the need to create an entirely new conversation). Conversations may *not* be shared by more than one process.

3.3 Recipient

Authentication is the process of evaluating a given set of credentials and verifier to determine:

- 1) Are the credentials valid?
- 2) Is the verifier any good?
- 3) Have the credentials expired? (strong only)
- 4) Has the verifier been used before? (strong only), and
- 5) If we accept these credentials as genuine, to whom do they belong?

In order to evaluate a set of credentials, one must have the identity of the recipient specified by the initiator when she created the conversation. In particular, evaluation of a set of strong credentials requires knowledge of the recipient's strong key. For this reason, the various authentication operations all require the caller to pass the identity of the recipient (e.g. the identity of the service evaluating the credentials.)

The recipient uses **Authenticate** or one of its variants to authenticate the credentials and verifier contained in messages from the initiator. For a session-based protocol, the recipient uses **AuthSession.InitialAuthenticate** for the initial call, and **AuthSession.NextReplyVerifier** on subsequent calls. The level of privilege a service grants should be based on the flavor of the credentials, as given by **GetFlavor**.

3.4 Levels

Typically, **CheckOutCredsandNextVerifier**, **ReplyVerifierChecks**, and **Refresh** are used by the initiating stub, transparently to the clients of this stub. **Authenticate** also is done at a low level of the recipient. **Makeldentity** and **Freeldentity** can be done at a very high level, as the identity can be shared.

3.5 Sample code

The Clearinghouse Service is a good example of a use of the authentication protocol. Information on the clearinghouse can be found in *Clearinghouse Protocol* [8], and the *Clearinghouse Programmer's Manual* [7].

This example is not of a session-based use of authentication. A session-based use is very similar. The initiator would make a distinction between the initial call to the recipient and subsequent ones. The recipient would use **InitialAuthenticate** to authenticate the credentials and verifier in the original call, and **NextReplyVerifier** to authenticate subsequent calls.

3.5.1 Initiator

```
BEGIN -- scope for catching Auth.CallError
  -- MakeIdentity, Initiate and Refresh can raise Auth.CallError
ENABLE Auth.CallError => {
    ReportAuthCallError[reason, whichArg];
    Auth.Freeldentity[@me, zj];
    Auth.Terminate[@conversation, z];
    EXIT };

z: UNCOUNTED ZONE <-> ... ;
  -- create the identity handle
userNames: NSName.Name <-> ... ;
userPasswords: NSString.String <-> ... ;
me: Auth.IdentityHandle <-> Auth.Makeldentity[userNames, userPasswords, z, strong];
  -- create the conversation
recipient: NSName.Name <-> ... ;
recipientsHostNumber: System.HostNumber <-> ... ;
conversation: Auth.ConversationHandle
  <-> Auth.Initiate[me, recipient, recipientsHostNumber, z];
  -- the credentials, verifier and reply verifier
```

```

creds: Auth.Credentials;
verifier, replyVerifier: Auth.Verifier;
    -- rc is the reason for AuthenticationErrors raised in the recipient
    -- (and reported back to the initiator)
rc: ...;
    -- counters for retrying after AuthError
    -- We tolerate this stuff happening once, as that's probably ok.
verifierInvalidRetrys: CARDINAL ← 1;
verifierExpiredRetrys: CARDINAL ← 1;
verifierReusedRetrys: CARDINAL ← 1;
credentialsExpiredRetrys: CARDINAL ← 1;
    -- this must be done for each remote call. Note that creds never changes.

DO
    [creds, verifier] ← Auth.CheckOutCredsAndNextVerifier[conversation];
    [replyVerifier, rc] ← RemoteCall[creds, verifier, ... ];
        -- check for AuthenticationProblem reported by recipient
    IF rc.OK THEN EXIT;
    SELECT rc.code FROM
        verifierInvalid = > {
            IF verifierInvalidRetries = 0 THEN EXIT;
            verifierInvalidRetries ← verifierInvalidRetries - 1 };
        verifierExpired = > {
            IF verifierExpiredRetries = 0 THEN EXIT;
            verifierExpiredRetries ← verifierExpiredRetries - 1 };
        verifierReused = > {
            IF verifierReusedRetries = 0 THEN EXIT;
            verifierReusedRetries ← verifierReusedRetries - 1 };
        credentialsExpired = > {
            IF credentialsExpiredRetries = 0 THEN EXIT;
            Auth.Refresh[conversation ! Auth.OrphanConversation = > EXIT];
            credentialsExpiredRetries ← credentialsExpiredRetries - 1 };
        ENDCASE = > EXIT;
    ENDLOOP;
    IF ~Auth.ReplyVerifierChecks[conversation, replyVerifier] THEN
        ERROR MyAuthError[verifierInvalid];
        -- free the conversation. This is done when the initiator no longer wishes to talk to the
        -- recipient.
    Auth.Terminate[conversation, z];
        -- free the identity. This is done when the initiator no longer wishes to talk to anybody.
    Auth.Freelidentity[@me, z];
END;

```

3.5.2 Recipient

```

-- create the identity handle (dontCheck is TRUE)
serverName: NSName.Name ← ... ;
serverKey: Auth.Key ← ... ;
me:Auth.IdentityHandle ← Auth.MakeStrongIdentityUsingKey
    ← [serverName, serverKey, z, TRUE];
    -- the credentials, verifier and reply verifier
creds: Auth.Credentials;
verifier, replyVerifier: Auth.Verifier;
    -- rc is the reason for AuthenticationErrors raised in the recipient
    -- (and reported back to the initiator)
    -- the name of the initiator (returned by Authenticate)
initiator: NSName.Name;
    -- this is done for every message from an initiator
    -- receive creds and verifier as part of message
[creds, verifier] ← RemoteReceive[ ... ];
    -- this server only accepts strong credentials
IF Auth.GetFlavor[creds] # strong THEN {
    rc.ok ← FALSE; rc.code ← inappropriateCredentials;
    rRemoteReply[replyVerifier, rc, ... ] };
    -- Authenticate raises Auth.AuthenticationError if something's wrong here
[initiator, replyVerifier] ← Auth.AuthenticateAndReply[me, creds, verifier, z
    ! Auth.AuthenticationError = > { rc.ok ← FALSE; rc.code ← reason } ];
RemoteReply[replyVerifier, rc, ... ];
    -- the initiator and replyVerifier returned by AuthenticateAndReply must be freed
NSName.FreeName[z, initiator];
Auth.FreeVerifier[@replyVerifier, z];

```

XEROX



Services 8.0 Programmer's Guide

Clearinghouse Programmer's Manual

November 1984

PRELIMINARY

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



Table of contents

1	Introduction	1-1
1.1	Organization of the document	1-1
1.2	Definition of terms	1-2
2	Concepts	2-1
2.1	Names	2-1
2.1.1	Three part names	2-1
2.1.2	How to pick an organization name	2-2
2.1.3	How to pick a domain name	2-2
2.1.4	How to pick a local name	2-2
2.2	Aliases	2-3
2.3	Wildcards	2-3
2.4	Defaults	2-4
2.5	Property lists	2-5
2.5.1	Values	2-5
2.5.2	Groups	2-5
2.6	Group operations	2-6
2.7	The implications of distribution	2-7
2.8	Helpful hints	2-8
2.8.1	Distinguished names, aliases, and capitalization	2-8
2.8.2	Getting started	2-8
3	Interface	3-1
3.1	Name declarations	3-1
3.2	ReturnCodes and errors	3-2
3.3	Common parameters	3-3
3.4	Names and aliases	3-4
3.5	Enumeration	3-5
3.6	Property lists	3-5
3.7	Value properties	3-6
3.8	Group properties	3-7

Table of contents

3	Interface (CONTINUED)	
3.9	Domains and organizations	3-8
3.10	Access control	3-9
3.11	Utilities	3-12

Appendices

A	List of operations	A-1
A.1	Names and aliases	A-1
A.2	Enumeration	A-1
A.3	Property lists	A-1
A.4	Value properties	A-1
A.5	Group properties	A-2
A.6	Domains and organizations	A-2
A.7	Access control	A-2
B	CHCommonLookups.mesa	B-1

Figure

2.1	IsMember/IsMemberClosure example	2-6
-----	---	------------



Introduction

This document describes the concepts and software interfaces of the *Clearinghouse Service*. It is intended as a reference for the designers and implementors of client programs, such as Star Workstation software and the other Services, which run on machines which support Mesa. It provides sufficient information to allow programmers to understand and use the facilities available through the interfaces **CH.mesa** and **MoreCH.mesa**.

The Clearinghouse Service is a *distributed* service; the services it provides are implemented by a set of one or more cooperating instances of the Clearinghouse software running on different server machines in potentially scattered locations. Since the client interacts with the Clearinghouse Service through a piece of software running on his own machine called the *Clearinghouse Stub*, the details of how the Clearinghouse Service is implemented are largely irrelevant.

Throughout this document, when speaking of the distributed Clearinghouse System as a whole, we shall refer to it as either "the Clearinghouse System" or simply "the Clearinghouse" (capital C). Individual instances will be referred to as "a clearinghouse service" or simply "a clearinghouse" (small C).

The Clearinghouse allows *names* to be registered and managed within a global, hierarchical name space. Associated with each name registered in the Clearinghouse may be one or more chunks of data stored in a *property list* indexed by *propertyIDs* from a set of well-known properties. In general, the Clearinghouse does not care about the form or structure of the data it manages. However, in order to support lists of names for mailing and access control, the Clearinghouse supports the special data type *group*. Special functions are provided to make group manipulation and membership checking operations efficient.

1.1 Organization of the document

This document has two major sections. Section 2 describes the concepts, philosophy, and facilities of the Clearinghouse. Section 3 describes the nuts and bolts of the Mesa interfaces (**CH.mesa** and **MoreCH.mesa**) to the Clearinghouse facilities.

1.2 Definition of terms

<i>Clearinghouse System</i>	or simply <i>Clearinghouse</i> (capital C). The distributed service supplied by a set of cooperating clearinghouse servers.
<i>clearinghouse service</i>	or simply <i>clearinghouse</i> (small c). A server machine running Clearinghouse Service software; one <i>instance</i> of the Clearinghouse Service.
<i>clearinghouse stub</i>	a piece of software running in the client's machine which acts as an agent for the Clearinghouse Service. The stub may interact with one or more clearinghouse servers to perform a given function for the client. The stub supplies the Mesa interfaces CH.mesa and MoreCH.mesa , described in this document.
<i>fully qualified name</i>	a name consisting of three parts: the organization name, the domain name, and the local name. A fully qualified name is not necessarily a distinguished name; it could be an alias. Sometimes referred to as a <i>three part name</i> . Generally this term implies that the name does not contain any wildcards. Names are always presented to the clearinghouse in fully qualified form.
<i>distinguished name</i>	the official, full name of an object registered in the Clearinghouse (i.e., as opposed to an alias).
<i>alias</i>	a "nickname" for some distinguished name. When the distinguished name is deleted, the alias is automatically deleted as well.
<i>organization name</i>	the most significant field of a fully qualified name.
<i>domain name</i>	the second most significant field of a fully qualified name.
<i>local name</i>	the least significant field of a fully qualified name.
<i>pattern</i>	a variant of a fully-qualified name which may contain wildcard characters in one or more fields. Depending on the context, the other fields of the name may or may not be significant.
<i>property list</i>	the sequence of (<i>propertyID</i> , <i>valueOrGroup</i>) pairs which is attached to a name. A property list may be empty.
<i>propertyID</i>	or simply "property." This is a well-known number used as a hook on which to hang data in the property list of a name. For example, all Print Services currently stored in the Clearinghouse have an entry associated with the property CHIDs.ps (= 10001).

-
- | | |
|--------------|--|
| <i>value</i> | raw data stored in a property list. The Clearinghouse knows nothing of the structure of values (as opposed to groups). |
| <i>group</i> | structured (as opposed to "raw") data stored in a property list and consisting of a sequence of <i>elements</i> . Elements may be fully qualified names or patterns. Groups are used for access control and mailing lists. |



Concepts

A good understanding of the underlying philosophy of a system is an aid to the clients of that system. This section describes the *whys* of the Clearinghouse and discusses its underlying architecture. To aid readability, the details of use appear in section 3.

2.1 Names

Everyone agrees that names are a good way to talk about things unambiguously. If you had only one print service then you could say: "the printer is broken" and everyone would know what you meant. If you had ten print services, then you would have to say something like: "the printer in the back corner of room 206B is broken." But if your printers all had names, you could simply say: "Old Reliable is down again."

Now suppose you have 10,000 print services in locations scattered across three continents, owned and operated by many different groups and organizations, with new ones being added every day. Because networking makes them all accessible, you would like to be able to talk about any one of these 10,000 print services and have the person or machine you are talking to not be confused. In short, you would like names that are *globally unique*. How can you accomplish this? You might leave it to luck and write your software to handle ambiguous names when they crop up, as they most certainly will. Or you might put someone in an office somewhere to register all names and make sure that there were no duplicates (as, for example, the FCC does for radio and television station call letters). That person would be pretty busy.

2.1.1 Three part names

The Clearinghouse takes a different approach. By dividing the entire name space into a three level hierarchy and establishing different rules for the assignment of names at each level, it allows names to be chosen locally, and yet remain globally unique. To this end, a Clearinghouse name has three components:

Clearinghouse Name ::= <Local name> <Domain name> <Organization Name>

where the <Organization name> part is the "most significant" part. Here are the rules:

2.1.2 How to pick an organization name

Organization names are assigned by a central administration to *organizations*: corporations, universities, governments, etc. Other than using Xerox protocols, these organizations may have little or nothing to do with each other. The intent of the organization name is to avoid name conflicts when multiple computer networks are connected together. It is therefore important that our customers play by the rules and register their organization names with a Central Organization Name Administration (currently provided by the Xerox BSG Software Marketing Group). Within a given organization, everyone should use the same organization name, although very large customers may encompass several organization names. (Within the Xerox family, for example, are subsidiaries like Versatec, and foreign affiliates like Rank Xerox and Fuji Xerox, which all have different organization names.)

2.1.3 How to pick a domain name

Domain names are assigned by some mechanism internal to a given organization. For example, these names could be imposed by an organization-wide domain naming administration, or proposed by individual system administrators. What is essential is that they be unique within the organization to avoid name conflicts among domains.

There is more to assigning domain names than meets the eye, however. It turns out to be very difficult to change a domain name once it gets established because it becomes embedded in a great many things which are not understood by the Clearinghouse: file drawer names, access control lists, desktop names, mail messages, entries in people's private distribution lists, and so on. Changing the domain name after a certain point causes a massive upheaval which effects many people both inside and outside the domain being re-named. It can be weeks or even months before the last tremors die away. Because of this, and because of the tendency of bureaucracies to reorganize themselves frequently, it is unwise to try to make domains reflect organizational boundaries. Experience with the Grapevine mail system inside Xerox suggests that it is much more satisfactory to have domain names reflect geographic boundaries.

2.1.4 How to pick a local name

Local names are assigned by a mechanism internal to a given domain. The Clearinghouse will not let you register two objects of the same type (e.g., two users) under the same name. (The Clearinghouse does not actually prohibit using the same name for multiple objects of different types. For example, the name "Gutenberg:OSBU North:Xerox" may refer to both a Print service and a File service. However, it is generally preferable to avoid duplication since it leads to ambiguity and potential confusion.)

Local names must be unique within their domain—the Clearinghouse guarantees this. But in addition to being unique, you often want names in the Clearinghouse to have associations with things or people in the real world. There is one case worth special mention: the registration of users in the Clearinghouse.

It makes sense for the users' names in the Clearinghouse to be similar or identical to the everyday names of these users. You might start by registering just first names, such as "Dave," "Bob," and "John," in the Clearinghouse. These names are short, easy to type, and unambiguous—as long as your user community remains small. That is the catch: user

communities hardly ever stay small. As soon as someone else named "John" joins the community, you have a problem. You could call the newcomer "JohnT" to distinguish him from "John," but this is not entirely satisfactory because Clearinghouse names are used for addressing mail. The person sending mail to "JohnT," (who might be far away and not intimate with John's co-workers), might simply guess that he should address his mail message to "John" instead of "JohnT." How is he to know that there are two "John"s working there? The result of this is that "John" often gets puzzling messages which were actually intended for "JohnT" while "JohnT" misses important mail. In this situation, it is probably best to change the original John's name also (to "JohnS," perhaps) and eliminate the name "John" entirely, since it has become misleading to the people using the system (though not to the Clearinghouse).

For these reasons, it is better to plan ahead and put more distinguishing information into users names right from the beginning. Full first and last names should be considered the bare minimum, and middle initials (or even full middle names) should generally be registered as well, especially for those users with common last names. If all your users have names like "David T. Kearns" instead of "Dave" you will seldom have name conflicts and, if this convention is followed fairly consistently, even far away mail users stand a good chance of being able to correctly guess how to address mail to someone.

2.2 Aliases

Long names, while descriptive, are cumbersome to type. Imagine typing "Patrick Michael McGillicutty" every time you logged in! For this reason, the Clearinghouse provides *aliases*. An alias is a nickname for some object registered in the Clearinghouse. Of course, the Clearinghouse still remembers the object's real name, henceforth referred to as its *distinguished name*. The object may have many aliases, but only one distinguished name. When the object is deleted, not only does its distinguished name disappear, but so do all its aliases.

Aliases must be unique "names" within a given domain (i.e., you cannot create an alias "Joe" if there is an existing alias or name "Joe"). Most of the Clearinghouse operations which take a name will accept either a distinguished name or an alias; these operations always return the distinguished name of their operand, so the client knows the real name of the object he is dealing with. Since aliases are much more subject to change than distinguished names, it is recommended that distinguished names be used anytime a name is embedded in a "long-lived" data structure (e.g., the access list of a file drawer).

Aliases are also useful for providing indirect action or procedure. For example, "Best Printer" could be an alias for the print service whose print engine was producing the best output on a given day.

Note: An alias *may* point to a distinguished name in another domain.

2.3 Wildcards

The Clearinghouse will do elementary pattern matching on names. The wildcard character is the ASCII/ISO asterisk (*). This is a special character which may not appear in a normal name, but may be used to construct a *pattern*. The wildcard character matches any sequence of zero or more characters. For example:

*	<i>matches</i>	(anything)
cat	<i>matches</i>	cat (only)
c*t	<i>matches</i>	cat, coot, chalet, ...
*c*****t*	<i>matches</i>	cat, coot, chalet, accent, practical...
i*t	<i>matches</i>	it, insert, ...
a*r*k	<i>matches</i>	aardvark, asterisk, ...
*M*D*B*	<i>matches</i>	Jean-Marie R. de La Beaujardiere, ...

The **Lookup** operations and the **Enumerate** operation will accept patterns containing wildcards in the local name field. Various other operations (e.g., **EnumerateDomains**) will accept variations on this theme. Patterns that are group elements may contain wildcards in any combination of fields. The exact handling of patterns depends on the operation being performed, as follows:

The **Enumerate** operation returns all the distinguished names in the given domain that match the pattern. The **EnumerateAliases** operation returns all the aliases in the given domain that match the pattern. (Note: There is a special wildcard-like property type which "matches" all property types, so you can find out about all names in a domain matching a certain pattern, regardless of associated properties.)

The **Lookup** operations return information about only a single name (the first name they find that matches the pattern). This may or may not be the name you wanted, but in any case, the distinguished name that matched is returned. Note that the **Lookup** operations (unlike **Enumerate**) check for aliases as well as distinguished names that match the pattern, and if they find such an alias, will return the corresponding distinguished name. This means that the distinguished name which is returned need not itself match the pattern. For example, if "John Smith" is a user with an alias "Jack," a **LookupValueProperty** of users with the pattern "J*k" may return "John Smith."

The **IsMember** or **IsMemberClosure** operations are used to search a group for a specific name, so these operations do *not* accept patterns as arguments (asterisks in the sought-for name have no special significance; they are treated as normal characters, not as wildcards). The group being searched, on the other hand, *may* contain one or more patterns, and the operation will return **TRUE** if such a pattern matches the sought-for name. (Note: This means that in the **IsMember** operations, the actual name is the argument and the pattern is in the database—just the reverse of the situation with the **Enumerate** and **Lookup** operations.)

2.4 Defaults

In certain circumstances, it is desirable to fill in default values for the domain and/or organization components of a three-part name. (For example, when a user is sending mail to several other users in the same domain.) All such defaulting mechanisms *must* be supplied by the clients of the Clearinghouse. The Clearinghouse itself deals *only* in fully qualified names, and never under any circumstances provides default domain or organization names. Because the Clearinghouse Service is designed to be distributed and replicated (see §2.7) the identity of the specific server providing access to the distributed database at any given time is completely transparent and does not constitute a sound basis for defaulting of partially qualified names.

2.5 Property lists

The Clearinghouse maps names into *property lists*. A property list is a list of zero or more (*propertyID*, *valueOrGroup*) pairs. This allows a number of different attributes to be attached to a name. For example, the name "Hans Christian Andersen:PARC:Xerox" might have associated with it the property list:

```
{(userDescription, <string>), (mailbox, <name>)}
```

which allows a client to look at the description field for Hans and find out the name of the mail server that his mailbox is on.

2.5.1 Values

As far as the Clearinghouse is concerned, the values of userDescription and mailbox are simple sequences of bytes with no internal structure. It is up to the client to interpret this raw data.

Note: Typically, values are Mesa records in Courier serialized form. (The details of Courier serialized data standards are in *Courier: The Remote Procedure Call Protocol* [10].) When extracting data from the Clearinghouse, the interpretation of a value (i.e., what Courier description routine should be used to deserialize that data) is determined by the *propertyID* with which the value is associated. There is no way for the Clearinghouse to enforce this convention; the client reading the data must assume that whoever stored it followed the rules.

A *propertyID* is really a **LONG CARDINAL**. There is a whole set of "well-known" *propertyID*'s which are listed, along with the format (in Courier notation) of the data they are used to store, in *Clearinghouse Entry Formats* [5]. The Mesa interfaces **CHIDs.mesa** and **CHEntries.mesa** together contain the same information.

2.5.2 Groups

The property list of a name may include properties of a different flavor, called *groups*. A value is raw data (to the Clearinghouse), whereas a group has structure. A group contains an arbitrarily large number of *elements*, where each element has the form of a Clearinghouse name. Often, group elements really are names, but they may also be illegal or non-existent names or patterns containing wildcards in one or more fields. The Clearinghouse itself knows the internal structure of groups and supports special group manipulation operations such as **AddMember** and **DeleteMember**. Since the Clearinghouse must guarantee that the data structures used to store groups are maintained consistently, it will not allow clients to manipulate groups as values (i.e., raw data) or vice versa.

2.6 Group operations

The Clearinghouse supports an interesting operation called **IsMember** (and its cousin **IsMemberClosure**). These operations take an element and an existing group, and tell you if the element is in the group or matches a pattern which is an element of the group. This is handy, especially for supporting access control. To illustrate these operations, consider two groups:

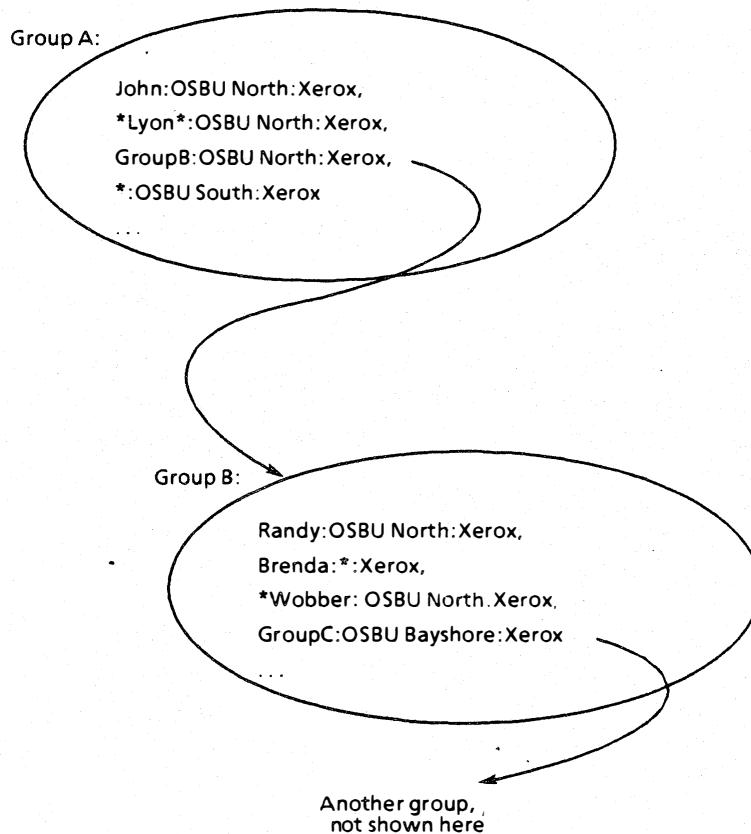


Figure 2.1 **IsMember/IsMemberClosure** example

If we do an **IsMember** operation, only group A will be searched for the element in question. If we do an **IsMemberClosure** operation, the groups searched are (not necessarily in this order): GroupA, any group whose name appears in GroupA (in this case, GroupB), any group whose name appears in any group whose name appears in Group A (in this case, GroupC), and so on. **IsMemberClosure** will not attempt to recur on a pattern (so it will *not* recur on every group in "OSBU South:Xerox"). For example:

```

IsMember["John:OSBU North:Xerox", GroupA] = TRUE
IsMember["Brenda:OSBU North:Xerox", GroupA] = FALSE
IsMemberClosure["Brenda:OSBU North:Xerox", GroupA] = TRUE
IsMember["Mark:OSBU South:Xerox", GroupA] = TRUE
IsMemberClosure["GroupC:OSBU Bayshore:Xerox", GroupA] = TRUE
IsMemberClosure["Brenda:Training:Xerox", GroupA] = TRUE

```

```
IsMemberClosure["Brenda:OSBU North:Xerox," GroupA] = TRUE  
IsMember["J*:OSBU North:Xerox," GroupA] = FALSE
```

2.7 The implications of distribution

As stated previously, the Clearinghouse really is a distributed service. The Clearinghouse Service is provided by a cooperating set of clearinghouse servers. This leads to two important design principles.

- It should not matter which clearinghouse server the stub initially contacts; it will eventually get the data it wants (if that data is available).
- Every clearinghouse server must know about all other clearinghouse servers.

The first principle allows the Clearinghouse Service to continue to be "available" when one or more of the clearinghouse servers are not available (down or incommunicado), albeit crippled because the data stored on the unavailable server(s) will not be accessible. The second principle is sort of a corollary to the first. Why? Because if there is a clearinghouse server X which serves the domain " $d_x:o_x$ " and X is not known to any other clearinghouse, then when a stub talking to one of these other clearinghouses tries to look up a name in " $d_x:o_x$ " it fails and reports the client that the domain " $d_x:o_x$ " does not exist. This is wrong, of course. If the stub happened to talk to the server X, the operation would succeed. However, our first assumption was that it should not matter which server the stub talked to.

The Clearinghouse Service works fairly hard to insure that these two principles are not violated. When a change is made to one copy of a replicated domain, the clearinghouse on which the copy is made mails updates of this event to all other clearinghouses that serve the domain. To handle situations in which these update messages are lost, each clearinghouse periodically runs a background process which compares the various servers' copies of domains to verify that these copies are consistent with one another. Note that there will always be a transition period when some servers do not yet know of a change to a domain, or the addition of a domain or organization. When the dust clears, all copies of domains will be consistent and all will be well, but meanwhile (and this can be a few minutes or up to several days if the internetwork is experiencing persistent communication problems), the effect can be a baffling set of irreproducible symptoms. For instance, sometimes when you look up something it works, and sometimes it fails and you are told that the object or domain does not exist, even though you *know* that it does. The problem may be there one minute and gone the next and may occur on some machines and not on others. If this seems to be happening, *don't panic*. Just call your nearest Clearinghouse administrator (who will probably tell you politely to wait until the problem fixes itself).

2.8 Helpful hints

This section is a collection of potentially useful information that has no other logical home.

2.8.1 Distinguished names, aliases, and capitalization

Every Clearinghouse operation which takes a name will, if the client supplies a place to put it, return the distinguished name of its operand. This information is basically "free," since it gets passed in the protocol no matter what. It can be used to get the distinguished name for what might be an alias, for resolving patterns, and for fixing up capitalization of a name so that it matches the name's "official" capitalization (as stored in the Clearinghouse). For example, say you are writing a program that requires the user to logon. Whatever he types, you look in the Clearinghouse for a user of that name. The **LookupValue** will return the distinguished name of the user, if it can find him. This distinguished name could be used to find the user's mail folder or desktop and could be kept around for use in authenticating the user to other services.

2.8.2 Getting started

The Clearinghouse stub operations are exported through the interfaces **CH.mesa** and **MoreCH.mesa**. **CHPIIDs.mesa** defines the well-known PropertyID's. **CHEntries.mesa** contains the Mesa definitions and Courier descriptions of the values which are associated with the well-known PropertyID's. **CHCommonLookups.mesa** defines procedures that are sometimes useful for doing a few, simple clearinghouse lookups. The configuration **CHStub.bcd** contains nearly everything you need; you also need various name manipulation routines which are exported by **Filing.bcd**.

Note: The Mesa 11.0 development environment is bound with earlier, type-incompatible versions of the Clearinghouse stub and Filing stub. Typically one must load a second version of these stubs before importing the OS 5.0 version of the Clearinghouse interfaces.



Interface

This section describes the Clearinghouse stub interface, **CH.mesa**.

CH: Definitions = ...;

3.1 Name declarations

Name: TYPE = NSName.Name;

NameRecord: TYPE = NSName.NameRecord;

OrgName: TYPE = NSName.Organization;

DomainName: TYPE = NSName.Domain;

LocalName: TYPE = NSName.Local;

wildCard: CHARACTER = NSName.wildCard; -- (*NSString.Character equivalent to ASCII '*'*)

separator: CHARACTER = NSName.separator; -- (*NSString.Character equivalent to ASCII ':'*)

maxOrgNameLength: CARDINAL = NSName.maxOrgLength; -- = 20 bytes

maxDomainNameLength: CARDINAL = NSName.maxDomainLength; -- = 20 bytes

maxLocalNameLength: CARDINAL = NSName.maxLocalLength; -- = 40 bytes

The Clearinghouse naming scheme is a three level hierarchy. The three fields of a name, **localName**, **domainName**, and **orgName**, have maximum lengths of **maxLocalNameLength**, **maxDomainNameLength**, and **maxOrgNameLength** bytes respectively. Clients sometimes encode names as strings in which the three parts of a name are concatenated together separated by **separator** characters. The **wildCard** character is used in constructing patterns.

Note: These lengths are in bytes, not **NSString.Characters**. The maximum length string of **NSString.Characters** allowed in each field is variable, depending upon how that string gets encoded into bytes. For each **NSString.Character**, the worst case requires three bytes and the best requires only one.

Pattern: TYPE = LONG POINTER TO NamePattern;

NamePattern: TYPE = NameRecord;

A **Pattern** is a name in which the use of wildcards is permitted. The Clearinghouse allows wildcards in the "least significant" field of a name in most operations. That means wildcards are generally allowed only in the **localName** field except in the operations

EnumerateDomains and **EnumerateOrganizations**; where wildcards are allowed in the **domain** and **organization** fields, respectively. Wildcards are *never* allowed in operations which modify the Clearinghouse database (**Add**, **Change**, or **Delete** operations).

Note: When a pattern is used in any **Lookup** operation, the "first" match is chosen. This may not be the first alphabetically, and it may be different if the operation is repeated. In searching for a name matching the pattern, **Lookup** considers both distinguished names and aliases. If a matching alias is found, the alias is dereferenced and the operation is performed on the object to which it points. In any case, the **distingName** parameter is always filled in with the full, distinguished name which was chosen (which need not itself match the pattern—i.e., if an alias was matched and dereferenced.)

Element: **TYPE** = LONG POINTER TO **ThreePartName**;
ThreePartName: **TYPE** = **NameRecord**;

An **Element** is similar to a name in that it consists of three string fields with the same length restrictions as names. An **Element**, however, is not necessarily a name which actually exists in the Clearinghouse and it may contain wildcards in any or all fields. Some fields may be left empty. **Elements** are the constituents of groups.

3.2 ReturnCodes and errors

ReturnCode: **TYPE** = MACHINE DEPENDENT RECORD [
 code: **Code**,
 which: **ParameterGrouping**];

A **ReturnCode** is returned by every Clearinghouse operation. The interpretation of the **ReturnCode** is based upon **code**; the **which** field, which is not always significant, provides additional information in certain contexts. These other fields are described first:

ParameterGrouping: **TYPE** = MACHINE DEPENDENT {
 first(1),
 second(2),
 (**LAST[CARDINAL]**)};

In cases where **code** indicates a problem with a name (e.g., **illegalLocalName**), the **which** field of the **ReturnCode** indicates which of the names passed to the operation was bad. For example, if the **AddAlias** operation returned with a code of **illegalLocalName**, then **which** would indicate whether the parameter **name** or the parameter **newAliasName** contained the bad local name.

Code: **TYPE** = MACHINE DEPENDENT {
 done(0),-- *operation succeeded*
 -- *code collection one - "operational problems"*
 notAllowed(1),-- *operation prevented by access controls*
 rejectedTooBusy(2),-- *server is too busy to service this request*
 allDown(3),-- *remote CHServer was down and it was needed for this operation*
 (4),-- *user will never see this code (operationRejectedUseCourier)*
 badProtocol(5),-- *protocol violation (e.g., Name too big in streaming operation)*
 -- *code collection two - "naming problems"*
 illegalPropertyID(10),-- *the specified PropertyID violates the protocol*
 illegalOrgName(11),-- *has illegal length or illegal characters*

```

illegalDomainName(12),-- has illegal length or illegal characters
illegalLocalName(13),-- has illegal length or illegal characters
noSuchOrg(14),-- the specified organization does not exist
noSuchDomain(15),-- the specified domain does not exist in the organization
noSuchLocal(16),-- the specified local does not exist in the domain
-- code collection three - "PropertyID errors"
    propertyIDNotFound(20),-- the name exists, but the PropertyID does not
    wrong.PropertyType(21),-- you wanted a Group, but it was a Value, or vice versa
-- code collection four - "update problems"
    noChange(30),-- the specified operation would not change the database
    outOfDate(31),-- operation ignored - more recent info was in database
    overflowOfName(32),-- the specified name has too much data associated with it
    overflowOfDataBase(33),-- the database has run out of room
-- code collection five - other problems
    (50),-- user will never see this code (wrongServer)
    --authentication problems
    credentialsInvalid(60),
    (61),-- user should never see this code (verifierInvalid)
    (62),-- user should never see this code (verifierExpired)
    (63),-- user should never see this code (verifierReused)
    (64),-- user should never see this code (credentialsExpired)
    credentialsTooWeak(65),
    wasUpNowDown(70),-- the remote service disappeared while streaming data;
        -- the user has an incomplete answer to his query.
    (LAST[CARDINAL]) };

```

Most of these **codes** are self-explanatory. Any call may return **allDown**, which indicates that no servers for the given domain are available, or **rejectedTooBusy**, which indicates that the server is overloaded. **noChange** is returned when the requested modification did not change the database. For example, attempts to delete something which is not there or attempts to add something which already exists will fail and return **noChange**. **outOfDate** indicates that there has been a race to modify some item in a given domain and this call lost. This is because timestamps are used to break ties and distributed clocks are never completely synchronized. **outOfDate** can usually occur only when a domain is replicated, or when adding a domain or organization that was deleted within the past 30 days; otherwise, it probably indicates that the server has a bad clock. The total space for a single property list is bounded (see §3.6); if you try to make a given property list too large, you will get **overflowOfName**.

3.3 Common parameters

```

ConversationHandle: TYPE = RECORD[
    conversation: Auth.ConversationHandle,
    address: LONG POINTER TO System.NetworkAddress ← NIL];

```

All Clearinghouse operations require a **ConversationHandle**. These operations extract credentials and a verifier from **conversation.conversation** and they in turn are used to validate the identity of the client. A handle allowing conversation with any clearinghouse may be obtained by the procedure **MakeConversationHandle**. The field **address** allows calls to be directed to a particular clearinghouse. It is intended to be used to monitor

clearinghouse performance and to detect inconsistencies in replicated databases. Most clients will use the default value for **address**.

Many operations also take the parameter **distingName**. This client-supplied **Name** which is an output parameter to be filled in by the operation with the distinguished name of its operand. If the name supplied is too small, **NSName.NameTooSmall** is raised. If the client does not care about the distinguished name, it may pass **NIL**.

3.4 Names and aliases

```
LookupDistinguishedName: PROCEDURE [
    conversation: ConversationHandle, name: Pattern, distingName: Name]
    RETURNS [rc: ReturnCode];
```

name may contain wildcards in the **localName** field or it may be an alias. **distingName** is filled in with the associated distinguished name.

```
AddDistinguishedName: PROCEDURE [
    conversation: ConversationHandle, name: Name, distingName: Name]
    RETURNS [rc: ReturnCode];
```

Adds **name** to the Clearinghouse as a distinguished name. The initial property list for **name** is nil. Note that **name** must not already exist in its domain, either as a distinguished name or as an alias. Naturally, **name** may not be a pattern.

```
DeleteDistinguishedName: PROCEDURE [
    conversation: ConversationHandle, name: Name, distingName: Name]
    RETURNS [rc: ReturnCode];
```

name may *not* be an alias or pattern. **name** is removed from the Clearinghouse database along with all its aliases and its property list, if these exist. May return [**noChange**, **first**].

```
LookupAliasesOfName: PROCEDURE [
    conversation: ConversationHandle,
    name: Pattern, eachAlias: NameStreamProc, distingName: Name]
    RETURNS [rc: ReturnCode];
```

As usual with lookups, **name** may contain wildcards in the **localName** field or it may be an alias. Calls **eachAlias** for each alias of the distinguished name associated with **name**.

```
AddAlias: PROCEDURE [
    conversation: ConversationHandle,
    name, newAliasName, distingName: Name]
    RETURNS [rc: ReturnCode];
```

name may *not* be an alias or pattern, and must refer to an existing name. **newAliasName** must be a **Name** which does not already exist in this domain. It becomes an alias for the distinguished name associated with **name**.

```
DeleteAlias: PROCEDURE [conversation: ConversationHandle,
    aliasName, distingName: Name]
        RETURNS [rc: ReturnCode];
```

Removes the alias **aliasName** from the Clearinghouse. **distingName** is filled in with the distinguished name for which **aliasName** was an alias.

3.5 Enumeration

```
NameStreamProc: TYPE = PROCEDURE [currentName: Element];
```

A procedure of this type must be supplied by the client in **Enumerate** and **EnumerateAliases** operations. **currentName** is owned by the stub and is only valid within the scope of the **NameStreamProc**. The names are enumerated in alphabetical order.

```
Enumerate: PROCEDURE [conversation: ConversationHandle,
    name: Pattern, pn: PropertyID, eachName: NameStreamProc]
        RETURNS [rc: ReturnCode];
```

name is a pattern, possibly containing wildcards in its **localName** field. Enumeration is only allowed within a single domain; wildcards in the domain and organization fields of **name** are not allowed. For each name in the domain which matches **name** and has **pn** in its property list, **eachName** is called. To enumerate everything in a domain which has a given property, use the pattern "*" in the **localName** field of **name**. To enumerate *all* the names in a domain which match a particular pattern, regardless of type, let **pn** = **unspecified** (see §3.6).

```
EnumerateAliases: PROCEDURE [conversation: ConversationHandle,
    name: Pattern, eachAlias: NameStreamProc]
        RETURNS [rc: ReturnCode];
```

name is a pattern, possibly containing wildcards (e.g., "*") in its **localName** field. For each alias matching **name** in the domain, **eachAlias** is called.

3.6 Property lists

```
PropertyID: TYPE = LONG CARDINAL;
notUsable: PropertyID = LAST[PropertyID];
unspecified: PropertyID = 0;
```

notUsable and **unspecified** are reserved values of **PropertyID**. **unspecified** is the "wildcard" of **PropertyIDs**.

```
Properties: TYPE = LONG DESCRIPTOR FOR ARRAY OF PropertyID;
```

A name may have associated with it at most 250 properties.

```
GetProperties: PROCEDURE [conversation: ConversationHandle,
    name: Pattern, distingName: Name, heap: UNCOUNTED ZONE]
        RETURNS [rc: ReturnCode, properties: Properties];
```

GetProperties returns a list of all the properties associated with **name**. The client must supply a **heap**, from which storage is allocated for the **Properties** returned. The client is responsible for freeing **properties** when she is through with it.

```
DeleteProperty: PROCEDURE [conversation: ConversationHandle,
    name: Name, pn: PropertyID, distingName: Name]
    RETURNS [rc: ReturnCode];
```

The property **pn** and its associated value or group are removed from the property list of **name**. **name** is *not* automatically deleted if this makes the property list empty. This same operation is used for deleting both group and value properties.

3.7 Value properties

```
Buffer: TYPE = LONGPOINTER TO BufferArea;
BufferArea: TYPE = MACHINE DEPENDENT RECORD [
    maxlen(0): CARDINAL [0 .. maxBufferSize],
    length(1): CARDINAL [0 .. maxBufferSize],
    data(2): SEQUENCE COMPUTED CARDINAL OF WORD];
```

A **Buffer** is used to pass values to and from the Clearinghouse. The Clearinghouse doesn't know or care about the internal structure of the data. See Utilities, §3.11, for some handy routines for managing buffers. **length** and **maxlength** are in words.

maxBufferSize: CARDINAL = 500; -- in words

```
BufferTooSmall: SIGNAL [offender: Buffer, lengthNeeded: CARDINAL] --in words
    RETURNS [newBuffer: Buffer];
```

BufferTooSmall will be raised by **LookupValueProperty** if the **Buffer** supplied by the client is too small. **offender** is the offending **Buffer**.

```
LookupValueProperty: PROCEDURE [
    conversation: ConversationHandle,
    name: Pattern, pn: PropertyID, buffer: Buffer, distingName: Name]
    RETURNS [rc: ReturnCode];
```

Retrieves the value property associated with **pn** in the property list of **name**. The value is returned in **buffer.data** and **buffer.length** is filled in. May raise **BufferTooSmall** if the buffer supplied by the client is not large enough to hold the value.

```
AddValueProperty: PROCEDURE [conversation: ConversationHandle,
    name: Name, pn: PropertyID, rhs: Buffer, distingName: Name]
    RETURNS [rc: ReturnCode];
```

Adds a value property to the property list of **name** with the PropertyID **pn** and a value of **rhs** (**rhs** comes from "right hand side"). Returns **rc.code** of **noChange** if **pn** is already a property of **name**. In this case, **ChangeValueProperty** must be used.

```
ChangeValueProperty: PROCEDURE [
    conversation: ConversationHandle,
```

```
name: Name, pn: PropertyID, newRhs: Buffer, distingName:Name]
RETURNS [rc: ReturnCode];
```

Used to change a value property once it exists. The old value of the property is replaced by **newRhs**.

3.8 Group properties

```
NameStreamProc: TYPE = PROCEDURE [currentName: Element];
```

A procedure of this type must be supplied by the client in the **LookupGroupProperty** operation. **currentName** is owned by the stub and is only valid within the scope of the **NameStreamProc**.

```
LookupGroupProperty: PROCEDURE [
    conversation: ConversationHandle,
    name: Pattern, pn: PropertyID, eachElement: NameStreamProc, distingName: Name]
RETURNS [rc: ReturnCode];
```

name must have a group property **pn**. **eachElement** is called once for each element of the group.

```
EnumerateNewGroupElements: TYPE = PROCEDURE [NameStreamProc];
```

A procedure of this type may be supplied by the client in the **AddGroupProperty** operation if he wishes to specify the initial contents of the new group.

```
AddGroupProperty: PROCEDURE [conversation: ConversationHandle,
    name: Name, pn: PropertyID,
    elementEnumerator: EnumerateNewGroupElements ← NIL, distingName: Name]
RETURNS [rc: ReturnCode];
```

Adds a group property to the property list of **name** with the **PropertyID** **pn**. **elementEnumerator** is used to supply the client with a **NameStreamProc** which he must call for each element he wishes to place in the new group. The **NameStreamProc** supplied in the **elementEnumerator** callback is not valid after the callback returns. If the client does not supply an **elementEnumerator**, an empty group will be added.

```
AddGroupMember: PROCEDURE [conversation: ConversationHandle,
    element: Element, name: Name, pn: PropertyID, distingName: Name]
RETURNS [rc: ReturnCode];
```

Adds **element** to the group property **pn** of **name**. The group must already exist.

```
DeleteGroupMember: PROCEDURE [conversation: ConversationHandle,
    element: Element, name: Name, pn: PropertyID, distingName: Name]
RETURNS [rc: ReturnCode];
```

Removes **element** from the group property **pn** of **name**. The group is otherwise untouched. The resulting group may be empty.

```
AddSelf: PROCEDURE [conversation: ConversationHandle,
    name: Name, pn: PropertyID, distingName: Name]
    RETURNS [rc: ReturnCode];
```

Identical to **AddGroupMember** but the element added to the group is the distinguished name of the initiator derived from **conversation**.

```
DeleteSelf: PROCEDURE [conversation: ConversationHandle,
    name: Name, pn: PropertyID, distingName: Name]
    RETURNS [rc: ReturnCode];
```

Identical to **DeleteGroupMember** but the element removed from the group is the distinguished name of the initiator derived from **conversation**.

```
IsMember: PROCEDURE [conversation: ConversationHandle,
    element: Element, name: Pattern, pn: PropertyID, distingName: Name]
    RETURNS [rc: ReturnCode, isMember: BOOLEAN];
```

Returns **isMember** = **TRUE** if **element** is a member of the group property **pn** of **name**. In this operation, the members of the group are treated as patterns; they may contain wildcards which cause them to match **element**.

Note: Wildcard characters which appear in **element** are treated as ordinary characters with no special significance.

```
IsMemberClosure: PROCEDURE [conversation: ConversationHandle,
    element: Element, name: Pattern, pn: PropertyID, distingName: Name,
    pn2: PropertyID ← unspecified]
    RETURNS [rc: ReturnCode, isMember: BOOLEAN];
```

This is a recursive version of **IsMember** and works as follows: **element** is sought in the group property **pn** of **name**. If it is found, **isMember** = **TRUE** is returned immediately. If it is not found, each of the non-pattern elements of the group property **pn** of **name** is treated as a name which has a group property **pn2** which must also be searched for **element**. If this level fails, each of the elements of each of *those* groups (if they really are groups) is searched for **element** (via **pn2**), and so forth until either **element** is found or there are no groups left to search. If **pn2** is defaulted, then **pn2** = **pn**. Because groups are often nested, most clients should use **IsMemberClosure** instead of **IsMember**.

3.9 Domains and organizations

```
NameStreamProc: TYPE = PROCEDURE [currentName: Element];
```

A procedure of this type must be supplied by the client in **EnumerateOrganizations** and **EnumerateDomains** operations. **currentName** is owned by the stub and is only valid within the scope of the **NameStreamProc**.

```
EnumerateOrganizations: PROCEDURE [
    conversation: ConversationHandle,
    orgPattern: Pattern, eachOrg: NameStreamProc]
    RETURNS [rc: ReturnCode];
```

Enumerates all the organizations in the known universe and calls **eachOrg** once for each organization which matches **orgPattern**. When **eachOrg** is called, the **orgName** field of **currentName** contains an organization name and the other two fields are identical to the corresponding fields of **orgPattern**.

Note: The **orgName** field of **orgPattern** may (and probably will) contain wildcards. The **domainName** and **localName** fields of **orgPattern** are ignored.

EnumerateDomains: PROCEDURE [**conversation:** ConversationHandle,
name: Pattern, **eachDomain:** NameStreamProc]
RETURNS [**rc:** ReturnCode];

Enumerates all the domains in the organization specified by the **orgName** field of **name** and calls **eachDomain** once for each domain which matches the **domainName** field of **name**. When **eachDomain** is called, the **domainName** field of **currentName** contains a domain name and the other two fields are identical to the corresponding fields of **name**.

Note: The **orgName** field of **name** must contain a valid organization name (no wildcards allowed). The **domainName** field of **name** may contain wildcards and the **localName** field is ignored.

EnumerateNearbyDomains: PROCEDURE [
conversation: ConversationHandle,
eachDomain: NameStreamProc]
RETURNS [**rc:** ReturnCode];

Returns, via **eachDomain**, a motley assortment of domains which the stub considers to be "nearby." When **eachDomain** is called, the **orgName** field of **currentName** contains an organization name, the **domainName** field contains a domain name, and the **localName** field is identical to the **localName** field of **name**.

Warning: Domains in assorted organizations may be returned.

Warning: This is an unpredictable operation whose results depend on factors which may change from one moment to the next. It is *not* recommended for general use.

3.10 Access control

One may associate a list of names with organizations, domains and properties, for the purpose of specifying who may have various kinds of access to those organizations, domains and properties. Access lists governing the modification of a database and the addition and removal of oneself to or from groups may be constructed, and these kinds of access are enforced by the Clearinghouse. Access lists governing who may read objects of a domain are accommodated by the interfaces, but read access is not enforced; this means that lists of this flavor are ignored by the Clearinghouse. Operations are provided for adding elements to a list, removing elements from a list, determining the contents of a list, and determining if a name is a member of a list.

These lists of names have many of the characteristics of groups. They may contain patterns, names of individuals, names of groups, and any other name that may be included in a standard group. All of the **IsMemberOf*Access** procedures are closure operations, behaving very much like **IsMemberClosure**. The most notable difference between these

lists and groups is that the lists may only be referenced through the procedures described in this section. Access lists may not be referenced by name.

Access control lists may be empty, and if they are, their contents are inferred. Queries of an empty list are automatically redirected at the corresponding list at the next higher level of the name hierarchy. For example, if one looks up the administrators list for some property, and that access list is empty, then the administrators list for the domain will be returned. Likewise, if the administrators list for the domain is empty, then the administrators list for the organization will be returned. There is no higher level than the organization. Therefore, if the list of administrators of an organization is empty, then the list will appear empty, and no one may administer the organization. Although an empty access list appears to have the same content as the corresponding access list at the next higher level of the hierarchy, deleting an element from an empty access list does not alter the contents of the next higher list.

Note: There is, unfortunately, no easy way of determining if an access list is empty and its contents inferred.

Administrators of an organization may create and delete domains of that organization. Administrators of a domain may add and delete objects of that domain. Administrators of a property may modify the value of that property. In addition, administrators may modify the administrator lists they are members of. Self controllers of a group property may add themselves to or remove themselves from that group. Administrators of a group property are always considered self controllers of that property, and may modify the self controllers list.

The procedures and types described in this section are defined in the interface **MoreCH.mesa**. All other referenced types are defined in the interface **CH.mesa**. It is expected that **MoreCH.mesa** will eventually be folded into **CH.mesa**.

The **name** parameters of these procedures may not be patterns.

```
ACLFlavor: TYPE = MACHINE DEPENDENT{
    readers(0), value(1), administrators(2), selfControllers(3), (LAST[CARDINAL]));
```

Parameters of type **ACLFlavor** indicate the kind of access that is of interest. **readers** indicates interest in the right of individuals to read objects of the database. **value** is present because of implementation considerations, and may not be specified by clients. **administrators** indicates interest in the right of individuals to modify objects of the database. **selfControllers** indicates interest in the right of individuals to add themselves to or remove themselves from a particular group property.

```
LookupPropertyAccess: PROCEDURE [
    conversation: ConversationHandle, name: Name, pn: PropertyID, acl: ACLFlavor,
    eachElement: NameStreamProc, distingName: Name]
    RETURNS [rc: ReturnCode];
```

eachElement is called once for each element of the access list of flavor **acl** that is associated with the property **pn** of **name**.

```
IsMemberOfPropertyAccess: PROCEDURE [
    conversation: ConversationHandle, element: Element, name: Name,
```

```
pn: PropertyID, acl: ACLFlavor, distingName: Name,  
pn2: PropertyID ← unspecified]  
RETURNS [rc: ReturnCode, isMember: BOOLEAN];
```

Returns **isMember** = **TRUE** if **element** is a member of the access list of flavor **acl** associated with the property **pn** of **name**. Members of the list are treated as patterns; they may contain wildcards which cause them to match **element**. This is a closure operation, and so the remarks about **IsMemberClosure** apply to this procedure.

```
LookupDomainAccess: PROCEDURE [  
    conversation: ConversationHandle, domain: Name, acl: ACLFlavor,  
    eachElement: NameStreamProc]  
RETURNS [rc: ReturnCode];
```

eachElement is called once for each element of the access list of flavor **acl** that is associated with the domain **domain**. Only the **org** and **domain** fields of **domain** are inspected.

```
IsMemberOfDomainAccess: PROCEDURE [  
    conversation: ConversationHandle, element: Element, domain: Name,  
    acl: ACLFlavor, pn2: PropertyID ← unspecified]  
RETURNS [rc: ReturnCode, isMember: BOOLEAN];
```

Returns **isMember** = **TRUE** if **element** is a member of the access list of flavor **acl** associated with the domain **domain**. Only the **org** and **domain** fields of **domain** are inspected. Members of the list are treated as patterns; they may contain wildcards which cause them to match **element**. This is a closure operation, and so the remarks about **IsMemberClosure** apply to this procedure.

```
LookupOrgAccess: PROCEDURE [  
    conversation: ConversationHandle, org: Name, acl: ACLFlavor,  
    eachElement: NameStreamProc]  
RETURNS [rc: ReturnCode];
```

eachElement is called once for each element of the access list of flavor **acl** that is associated with the organization **org**. Only the **org** field of **org** is inspected.

```
IsMemberOfOrgAccess: PROCEDURE [  
    conversation: ConversationHandle, element: Element, org: Name, acl: ACLFlavor,  
    pn2: PropertyID ← unspecified]  
RETURNS [rc: ReturnCode, isMember: BOOLEAN];
```

Returns **isMember** = **TRUE** if **element** is a member of the access list of flavor **acl** associated with the organization **org**. Only the **org** field of **domain** is inspected. Members of the list are treated as patterns; they may contain wildcards which cause them to match **element**. This is a closure operation, and so the remarks about **IsMemberClosure** apply to this procedure.

```
AddPropertyAccessMember: PROCEDURE [  
    conversation: ConversationHandle, element: Element, name: Name,  
    pn: PropertyID, acl: ACLFlavor, distingName: Name]  
RETURNS [rc: ReturnCode];
```

element is added verbatim to the specified access control list of the property **pn** of **name**. Calls specifying **acl = readers** will return **rc = [badProtocol, first]**..

```
DeletePropertyAccessMember: PROCEDURE [
    conversation: ConversationHandle, element: Element, name: Name,
    pn: PropertyID, acl: ACLFlavor, distingName: Name]
    RETURNS [rc: ReturnCode];
```

This is the inverse of **AddPropertyAccessMember**.

```
AddDomainAccessMember: PROCEDURE [
    conversation: ConversationHandle, element: Element, domain: Name,
    acl: ACLFlavor]
    RETURNS [rc: ReturnCode];
```

element is added verbatim to the specified access control list of **domain**.

```
DeleteDomainAccessMember: PROCEDURE [
    conversation: ConversationHandle, element: Element, domain: Name,
    acl: ACLFlavor]
    RETURNS [rc: ReturnCode];
```

This is the inverse of **AddDomainAccessMember**.

```
AddOrgAccessMember: PROCEDURE [
    conversation: ConversationHandle, element: Element, org: Name, acl: ACLFlavor]
    RETURNS [rc: ReturnCode];
```

element is added verbatim to the specified access control list of **org**.

```
DeleteOrgAccessMember: PROCEDURE [
    conversation: ConversationHandle, element: Element, org: Name, acl: ACLFlavor]
    RETURNS [rc: ReturnCode];
```

This is the inverse of **AddOrgAccessMember**.

3.11 Utilities

```
MakeConversationHandle: PROCEDURE [identity: Auth.Identity, heap: UNCOUNTED ZONE]
    RETURNS [
        conversation: ConversationHandle, ok: BOOLEAN, authCallError: Auth.CallProblem];
```

Creates a **ConversationHandle** that allows conversation with any clearinghouse. **ok = FALSE** if there was a problem while attempting to obtain the conversation handle from the authentication software, and in this case **authCallError** indicates what that problem was.

```
FreeConversationHandle: PROCEDURE [
    conversation: LONG POINTER TO ConversationHandle, heap: UNCOUNTED ZONE];
```

Deallocates the storage for **conversation** ↑ , and sets **conversation.conversation** to NIL.

FreeProperties: PROCEDURE [properties: LONG POINTER TO Properties, heap: UNCOUNTED ZONE];

Deallocates the storage for **properties** ↑ and sets **BASE[properties ↑]** to NIL.

MakeRhs: PROCEDURE [maxlength: CARDINAL[0..maxBufferSize], heap: UNCOUNTED ZONE]
RETURNS [rhs: Buffer];

Allocates a **Buffer** from **heap** with the given **maxlength**, and with **length** set to zero.

SerializeIntoRhs: PROCEDURE [parms: Courier.Parameters, heap: UNCOUNTED ZONE]
RETURNS [rhs: Buffer];

Uses Courier to serialize **parms**, allocating a **Buffer** large enough to hold them from **heap**.

ScopedSerializeIntoRhs: PROCEDURE [
parms: Courier.Parameters, callback: PROCEDURE [Buffer]];

Like **SerializeIntoRhs**, but the client must supply the procedure **callback**, which is called with the resulting **Buffer**. The **Buffer** is valid only inside **callback**; the storage for it is allocated and freed by the stub.

FreeRhs: PROCEDURE [rhs: Buffer, heap: UNCOUNTED ZONE];

MakeRhs and **SerializeIntoRhs** both return **Buffers** allocated from the client-supplied **heap**. This operation may be used to return such **Buffers** to the client's **heap**.

DeserializeFromRhs: PROCEDURE [
parms: Courier.Parameters, heap: UNCOUNTED ZONE, rhs: Buffer]
RETURNS [succeeded: BOOLEAN];

Uses Courier to deserialize **parms** from **rhs**. Uses **heap** to allocate any disjoint data required to store the results of the deserialization. Naturally, the client is responsible for freeing any data allocated in the deserialization process by calling **Courier.Free**.

DeserializeFromBlock: PROCEDURE [
parms: Courier.Parameters, heap: UNCOUNTED ZONE, blk: Environment.Block]
RETURNS [succeeded: BOOLEAN];

Like **DeserializeFromRhs**, but gets the data to be serialized from **blk**.

Appendix A List of operations

This appendix is a simple list of all Clearinghouse Service operations available to the client, arranged into logical categories.

A.1 Names and aliases

LookupDistinguishedName
AddDistinguishedName
DeleteDistinguishedName
LookupAliasesOfName
AddAlias
DeleteAlias

A.2 Enumeration

Enumerate
EnumerateAliases

A.3 Property lists

GetProperties
DeleteProperty

A.4 Value properties

LookupValueProperty
AddValueProperty
ChangeValueProperty

A.5 Group properties

LookupGroupProperty
AddGroupProperty
AddGroupMember
DeleteGroupMember
AddSelf
DeleteSelf
IsMember
IsMemberClosure

A.6 Domains and organizations

EnumerateOrganizations
EnumerateDomains
EnumerateNearbyDomains

A.7 Access control

LookupOrgAccess
LookupDomainAccess
LookupPropertyAccess

IsMemberOfOrgAccess
IsMemberOfDomainAccess
IsMemberOfPropertyAccess

AddOrgAccessMember
AddDomainAccessMember
AddPropertyAccessMember

DeleteOrgAccessMember
DeleteDomainAccessMember
DeletePropertyAccessMember

Appendix B **CHCommonLookups.mesa**

CHCommonLookups.mesa provides utilities that support several common forms of Clearinghouse entry lookups. These utilities are not the most efficient way to do these lookups and are decidedly suboptimal for many applications. In several instances storage is allocated and freed twice and data is copied twice. With the exception of **LookupAddress**, all the utilities share a 500 word buffer used in deserializing the results of Clearinghouse calls. Because of this shared buffer, these utilities cannot run concurrently. Sophisticated users of the Clearinghouse are better off using the **CH.mesa** interface and doing their own storage management.

CHCommonLookups: Definitions = . . . ;

In each of the following procedures, the success of lookups can be determined by examining **rc.Code** and **succeeded**. If **rc.Code** is done and **succeeded** is **TRUE**, then everything went well with the lookup. If **succeeded** is **FALSE**, then something went wrong when deserializing the data. For example, if the value of **pid** is not a string in **LookupStringProperty**, **succeeded** will be **FALSE**. If **rc.Code** is not done, consult the return code to find out what went wrong.

LookupAddress PROCEDURE [
 conversation: CH.ConversationHandle, name: CH.Name]
 RETURNS [**rc: CH.ReturnCode, address: System.NetworkAddress, succeeded: BOOLEAN**];

Looks up the address list associated with **name** and returns the nearest address. This is the only utility that uses a private buffer and can run concurrently with the other utilities.

LookupStringProperty: PROCEDURE [
 conversation: CH.ConversationHandle, name: CH.Name, pid: CH.PropertyID,
 heap:UNCOUNTED ZONE]
 RETURNS [**rc: CH.ReturnCode, stringProperty: NSString.String, succeeded: BOOLEAN**];

Looks up the string that is associated with the property **pid** in the property list for **name** and returns the string. **heap** is used to allocate storage for the returned string.

```
LookupNameProperty: PROCEDURE [
    conversation: CH.ConversationHandle, name: CH.Name, pid: CH.PropertyID,
    heap: UNCOUNTED ZONE]
RETURNS [rc: CH.ReturnCode, nameProperty: NSName.Name, succeeded: BOOLEAN];
```

Looks up the name that is associated with the property **pid** in the property list for **name** and returns that name. **heap** is used to allocate storage for the returned name.

```
LookupAnyValueProperty: PROCEDURE [
    conversation: CH.ConversationHandle, name: CH.Name,
    parameters: Courier.Parameters, pid: CH.PropertyID, heap: UNCOUNTED ZONE]
RETURNS [rc: CH.ReturnCode, succeeded: BOOLEAN];
```

Looks up the value of the property that is associated with the property **pid** in the property list for **name**. **heap** is used to allocate storage for the returned data. If **succeeded** is **TRUE**, then the client is responsible for any storage that was allocated during the lookup. **Courier.Free** will need to be called by the client to free space allocated by Courier.

XEROX



Services 8.0 Programmer's Guide

Mailing Programmer's Manual

November 1984

PRELIMINARY

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



Table of contents

1	Introduction	1-1
2	Mail transport	2-1
2.1	Message envelopes	2-2
2.2	Posting slot access	2-5
2.3	Delivery slot access	2-6
2.3.1	Locating a delivery slot	2-6
2.3.2	Delivery slot operation	2-7
2.4	Transport errors	2-9
2.4.1	Access errors	2-10
2.4.2	Authentication errors	2-11
2.4.3	Connection errors	2-11
2.4.4	Location errors	2-13
2.4.5	Session errors	2-13
2.4.6	Service errors	2-14
2.4.7	Transfer errors	2-14
2.4.8	Undefined errors	2-15
3	Inbasket	3-1
3.1	Standard message format	3-1
3.2	Finding an inbasket server	3-3
3.3	Inbasket sessions	3-3
3.3.1	Creating and deleting sessions	3-3
3.3.2	Inbasket state	3-4
3.3.3	Inbasket caching	3-6
3.4	Inbasket operations	3-6
3.4.1	Locate	3-6
3.4.2	ChangeStatus	3-7
3.4.3	Retrieve	3-7
3.4.4	Delete	3-8
3.4.5	List	3-8

Table of contents

3	Inbasket (CONTINUED)	
3.5	Inbasket errors	3-9
3.5.1	Contents type errors	3-9
3.5.2	Invalid index errors	3-9
4	Mail attributes	4-1
4.1	Message attributes	4-2
4.2	Envelopes	4-4
4.3	Attribute encoding and decoding	4-5
4.3.1	Decoding	4-5
4.3.2	Encoding	4-6
4.4	Encoding and decoding standard format messages	4-7
4.5	Signals and errors	4-8
5	Mail stream	5-1
5.1	Message posting	5-1
5.2	Message retrieval	5-2
5.3	Mail stream errors	5-3

Figures

2.1	Mail transport system	2-1
3.1	Standard format message	3-2

Introduction

This document describes the *Mailing Stub*, a collection of programs which implements the client portion of the Xerox NS Mailing Protocols. The Mailing Stub provides the necessary facilities for Mesa clients to make full use of the 8000 NS Mail Service. In this capacity, it acts as *agent* software for the Mail Service in much the same way that Filing allows its clients to make use of the NS File Service. The Mailing Stub is often similar to Filing in its usage of certain Mesa constructs, and this document makes frequent reference to the *Filing Programmer's Manual* [12].

Collectively, all 8000 NS Mail Services on an internet form a single *mail transport system* capable of supporting the exchange of electronic mail among the users of that internet. A *message* is a unit of electronic mail, consisting of two parts, *envelope* and *content*. The message envelope contains whatever information is necessary to route the message to its eventual destination(s), including a list of intended *recipients*. Message content is simply undiscriminated data. No attempt is made by the mail transport system to interpret it.

The Mail Service relies heavily on the existence of the 8000 NS Clearinghouse Service. Each potential message recipient must be identifiable by a single **NSName.Name** as registered in the Clearinghouse database. (**NSName.Name** is described in detail in §3.2.1 of the *Common Facilities Programmer's Manual* [9].) In addition, a *mailbox* must exist for that recipient on some Mail Service. A mailbox is a logical container which acts as the end repository for messages destined for a given recipient within the mail transport system. Each mailbox is associated with one and only one fully-distinguished recipient name and can be identified either by that name or by some valid alias. To complete the Clearinghouse information necessary for a valid message recipient, the mailbox location must also be registered with the database; this is done by the mail server when a mailbox is added.

The facilities described in this section are logically separated into four groups. **MailTransport** provides a set of user procedures for making use of the basic mail transport mechanism. **Inbasket** makes available a more sophisticated and useful way of examining mail at the Mail Service. **MailAttributes** and **MailStream** allow *standard message* contents to be interpreted and facilitate operations on standard message contents for clients of Filing and Mesa development environment file systems.

Mail transport

This section describes the **MailTransport** facility. This facility provides a set of procedures for posting and accepting delivery of messages at the Mail Service.

MailTransport: DEFINITIONS = ... ;

Communication with the mail transport system takes place through two queues, or *slots*:

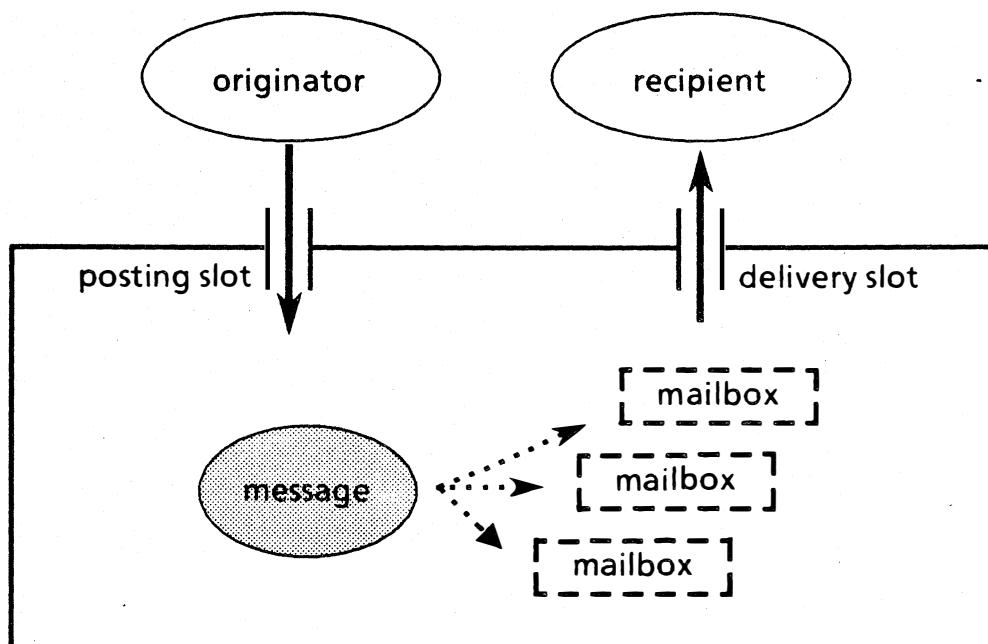


Figure 2.1 Mail transport system

At the *posting slot*, the client provides a message, complete with envelope (recipient) information. When posting is complete, that message has been written to stable storage at the Mail Service, and the client can proceed with other activities. The mail transport system, in its background processing loop, then completes the task of distributing the message to the mailboxes of the specified recipients. Messages are obtained from the transport system at the *delivery slot*. When a message passes through the delivery slot, it becomes the property of the recipient.

Both posting and delivery slots are actually queues. A client accepting mail at the delivery slot has no choice but to take the message currently at the head of the queue. As the message passes through the delivery slot, it is dequeued and flushed from the mail system. This mode of operation is convenient for certain types of clients. The fact that all messages must be processed in order matters little to programs that automatically receive and process mail. Such programs typically deal with known message formats; messages that cannot be interpreted can be discarded. Human users of the Mail Service do not fit this model well, since most user agent programs cannot understand all message formats. Furthermore, it can prove useful for different user agents to be able to examine the same mailbox without taking possession of all the mail it contains. Such clients can make good use of the **Inbasket** facility which allows mail to be examined while remaining at the Mail Service. The **Inbasket** facility is discussed in section 3 of this document.

2.1 Message envelopes

The message envelope contains information needed by the mail transport system to deliver a message to its destination mailboxes. The following definitions will be useful in describing envelope data.

Message recipients and mailboxes are identified by name. All such names must be fully-qualified.

MailTransport.Name: TYPE = **NSName.Name;**

MailTransport.NameRecord: TYPE = **NSName.NameRecord;**

MailTransport.NameList: TYPE = **LONG DESCRIPTOR FOR ARRAY OF NameRecord;**

As a message passes through the posting slot, the mail transport system tags it so that it is uniquely identified. Each message is given two such tags: a *postmark* to identify the place (server name) and time (in seconds) of posting, and a unique 80-bit quantity known as a *message id*.

MailTransport.Postmark: TYPE = **RECORD [server: Name, time: Time];**

MailTransport.Time: TYPE = **System.GreenwichMeanTime;**

MailTransport.MessageID: TYPE [5];

All messages that pass through the mail transport system must carry information on how the message content should be interpreted. This information is stored in the envelope as the *message contents type*. Since the mail transport system never interprets content, there is no guarantee that contents type matches actual message format. It is the responsibility of client programs to be able to handle malformed content encodings.

MailTransport.ContentsType: TYPE = **LONG CARDINAL;**

There are currently five specified contents types, the two most common of which are **ctSerializedFile** and **ctUnspecified**. Messages of type **ctSerializedFile** should be encoded as **Courier** objects of type **NSFile.SerializedFile**. This encoding scheme provides a general-purpose mechanism for representing trees of files, each file containing both attributes and data. It is described in detail in §3.8.2 of the *Filing Programmer's Manual* [12].

```
MailTransport.ctSerializedFile: ContentsType = 0;  
MailTransport.ctUnspecified: ContentsType = 1;
```

It is expected that most clients of the mail transport system will use **ctSerializedFile** as the *standard message* contents type. Later portions of this section describe utilities provided by the Mailing Stub to facilitate the handling of messages of this type. Non-standard contents types are allowed, to promote exchange of data encoded according to private client requirements. This best suits applications in which automated systems wish to exchange data through the mail transport system. It is important that message traffic of this sort be kept disjoint from standard format traffic since the message content will be essentially uninterpretable. Contents type ranges are administered by convention and can be reserved for specific applications upon request. Three such reserved types (which should only be used by their creators) are **ctNull**, **ctClearinghouseUpdate**, and **ctMSInterserver**.

```
MailTransport.ctNull: ContentsType = LAST[ContentsType];  
MailTransport.ctClearinghouseUpdate: ContentsType = 2;  
MailTransport.ctMSInterserver: ContentsType = 3;
```

The data type **MailTransport.Envelope** defines the client-visible portion of the message envelope. This information is kept in the mailbox along with each message and can be examined when the client accepts delivery from the mail transport system.

```
MailTransport.Envelope: TYPE = LONG POINTER TO EnvelopeRecord;
```

```
MailTransport.EnvelopeRecord: TYPE = RECORD [  
    postmark: Postmark,  
    messageID: MessageID,  
    contentsType: ContentsType,  
    contentsSize: LONG CARDINAL,  
    originator: Name,  
    problem: Problem];
```

postmark and **messageID** are identification tags assigned during posting; **contentsType** is the contents type of the message; **contentsSize** is the size in bytes of the message content; **originator** is the authenticated name of the sender; if **problem** is not **NIL**, then this is a returned message which could not be delivered by the mail transport system.

The constant **nullEnvelope** defines null values for all envelope components.

```
MailTransport.nullEnvelope: EnvelopeRecord = ... ;
```

Occasionally, the mail transport system is unable to deliver a message that was successfully posted (e.g., the recipient name is no longer valid). If this occurs, the message is returned to its sender (determined by the name embedded in the **Auth.IdentityHandle** provided to the **Post** call), and a **Problem** is indicated in the message envelope.

```
MailTransport.Problem: TYPE = LONG POINTER TO ProblemRecord;
```

```
MailTransport.ProblemRecord: TYPE = MACHINE DEPENDENT RECORD [  
    undeliverables(0): Undeliverables,  
    returnedEnvelope(3): ReturnedEnvelope];
```

MailTransport.Undeliverables: TYPE = LONG DESCRIPTOR FOR ARRAY OF UndeliveredName;

MailTransport.UndeliveredName: TYPE = MACHINE DEPENDENT RECORD [
 reason(0): UndeliveredNameType,
 name(1): NameRecord];

UndeliveredNameType: TYPE = MACHINE DEPENDENT {
 noSuchRecipient(0), cantValidateNow(1), illegalName(2), refused(3),
 noAccessToDL(4), timeout(5), noDLsAllowed(6), messageTooLong(7)};

There are seven types of **UndeliveredNameType**:

noSuchRecipient The message could not be delivered to the recipient because the recipient does not exist or does not have a mailbox.

cantValidateNow The message could not be delivered to the recipient because there was no Clearinghouse available for name validation. [Not used in OS5.]

illegalName The recipient is not a valid **Name**.

refused The message was refused at the recipient's delivery slot. This appears only in a **ReturnedEnvelope**.

noAccessToDL The sender does not have access to send to this distribution list. [Not used in OS5.]

timeout Indicates that the mail transport system gave up trying to forward the message to a distant Mail Service. For example, a destination Mail Service might be down for an extended period. This appears only in a **ReturnedEnvelope**.

noDLsAllowed Occurs only if **allowDLRecipients = FALSE** while posting and indicates that the recipient name represents a distribution list.

messageTooLong The message could not be delivered to the recipient because the message was too long for the destination Mail Service. This appears only in a **ReturnedEnvelope**.

MailTransport.ReturnedEnvelope: TYPE = [3];

The following procedure is provided for examination of returned envelopes:

MailTransport.DecodeReturnedEnvelope: PROCEDURE [
 encoding: ReturnedEnvelope, envelope: Envelope]
 RETURNS [ok: BOOLEAN];

encoding is the returned envelope to be examined; **envelope** points to a client-allocated **EnvelopeRecord**.

Storage allocated by **DecodeReturnedEnvelope** must be freed using:

MailTransport.ClearEnvelope: PROCEDURE [env: Envelope];

2.2 Posting slot access

MailTransport.Post is the procedure by which mail is passed through the posting slot. This procedure performs several functions. First, a Mail Service capable of accepting mail is located. Second, the recipients of the message are validated by the Mail Service. This validation is successful only if each recipient is correctly registered in the Clearinghouse database. [Note: The Mail Service attempts to validate all recipients. In certain unlikely cases this validation is impossible and invalid names might be accepted. In such cases, the message will ultimately be returned with a **ProblemRecord**.] Finally, the Mail Service forms an envelope using the specified arguments, and both envelope and content are written to stable storage.

```
MailTransport.Post: PROC[
    identity: Auth.IdentityHandle,
    recipients: NameList, postIfInvalidNames, allowDLRecipients: BOOLEAN,
    contentsType: ContentsType, contents: NSDataStream.Source]
RETURNS [undeliverables: Undeliverables];
```

Arguments: **identity** provides authentication information about the client who wishes to post a message (see *Authentication Programmer's Manual* [1] for details on authentication); **recipients** describes a list of fully-qualified recipient names (duplicate names or aliases that resolve to duplicate names are ignored); **postIfInvalidNames** allows messages to be sent even if invalid names exist in **recipients**, otherwise this condition results in an error; **allowDLRecipients** allows messages to be sent to recipient names which represent distribution lists, otherwise this condition results in an error; **contentsType** describes the format of the message content; **contents** specifies the source that is to supply the message content in accordance with **NSDataStream** conventions (see section 2 of *Common Facilities Programmer's Manual* [9]).

Results: The message content provided by **contents** is addressed to **recipients** and posted. If **postIfInvalidNames** is **TRUE**, then **undeliverables** describes the recipients which were found to be invalid; otherwise it is **NIL**.

Errors: If **postIfInvalidNames** or **allowDLRecipients** is **FALSE**, then **MailTransport.InvalidRecipients** may be raised. **MailTransport.Error** may be raised with the following error types: **authentication**, **connection**, **location**, **service**, **transfer**. **Courier.Error** may also be raised.

Note: Clients of Filing will typically use this operation in conjunction with **NSFile.Serialize**, which serves to encode a subtree of files into Filing serialized file (standard message) format. See §3.8.2 of *Filing Programmer's Manual* [12] for more detail on this operation.

The client has the option of allowing message posting to proceed, even if some of the intended recipients are not valid. If the client declines this option, the existence of invalid recipients is reported by the error **InvalidRecipients**.

```
MailTransport.InvalidRecipients: ERROR[nameList: Undeliverables];
```

If the client chose to suppress the **InvalidRecipients** error, a list of invalid recipients is returned as a result of the **MailTransport.Post** call. In this case, the following procedure must be used to free the underlying storage.

```
MailTransport.FreeUndeliverables : PROCEDURE [invalidNames: Undeliverables];
```

2.3 Delivery slot access

The delivery slot protocol provides a method of accessing a mailbox. It allows clients to do two things at the delivery slot: poll for presence of mail, and retrieve existing mail in a FIFO (First In, First Out) manner. Mail retrieval is done within the context of a *session*. A delivery slot session begins with a call to **BeginDelivery** and ends with a call to **EndDelivery**. The **DeliveryHandle** returned by **BeginDelivery** encapsulates state information about the session. Operations on **DeliveryHandle** should be performed sequentially; simultaneous calls on a single handle are not allowed. A **DeliveryHandle** can become invalid at any time. This is most likely if the destination mail server is stopped or if the session is inactive for some length of time.

```
MailTransport.DeliveryHandle: TYPE = [2];
nullHandle: DeliveryHandle = LOOPHOLE[LONG[NIL]] ;
```

2.3.1 Locating a delivery slot

```
MailTransport.Location: TYPE = LONG POINTER TO READONLY LocationRecord;
```

```
MailTransport.LocationRecord: TYPE = RECORD [
    type: MailboxType,
    serverName: Name,
    addr: System.NetworkAddress];
```

```
MailTransport.MailboxType: TYPE = { primary, secondary };
```

```
MailTransport.GetLocation: PROCEDURE [
    identity: Auth.IdentityHandle, mailbox : Name, type: MailboxType ← primary]
RETURNS[Location];
```

Arguments: **identity** provides authentication information about the client who seeks the mailbox location (see *Authentication Programmer's Manual* [1] for details on authentication); **mailbox** is a fully qualified name or alias identifying the mailbox sought; **type** indicates whether the primary or secondary mailbox location is desired.
[Note: Secondary mailboxes are not implemented in OS5.]

Results: The clearinghouse is queried to find the name and address of the server which holds the requested inbasket or delivery slot.

Errors: **MailTransport.Error** may be raised with the type **location**. **Courier.Error** may also be raised.

GetLocation allocates storage which must be freed with a call to **FreeLocation**.

```
MailTransport.FreeLocation: PROCEDURE [loc: Location];
```

2.3.2 Delivery slot operations

A typical delivery slot session would consist of a call to **BeginDelivery**, multiple calls to the triplet **DeliverEnvelope**, **DeliverContent**, **Acknowledge[acknowledge]**, and the concluding call to **EndDelivery**. **Acknowledge** can be called after **DeliverEnvelope**—for instance, in the case of refusal or to abort when the space required by content size is not available—but this is not the normal case. Each call to **DeliverEnvelope** must be followed by a call to **DeliverContent** or **Acknowledge**. An improper sequence of operations will result in the **MailTransport.error handle[wrongState]**.

2.3.2.1 BeginDelivery

BeginDelivery is used to initiate a delivery slot session at the delivery slot specified. This access is exclusive and locks out all other clients who might try to access that same mailbox with this or any other protocol.

```
MailTransport.BeginDelivery: PROCEDURE [  
    identity: Auth.IdentityHandle, deliverySlot: Name, loc:Location ← NIL]  
    RETURNS [handle: DeliveryHandle];
```

Arguments: **identity** provides authentication information about the client who wishes to establish a *session* (see *Authentication Programmer's Manual* [1] for details on authentication); **deliverySlot** is a fully qualified name or alias identifying the mailbox which is to be examined; **loc** identifies the Mail Server which holds the mailbox. If **loc** is defaulted, the clearinghouse will be queried to locate the server holding the primary mailbox in question.

Results: **handle** is the identifier to be used for further reference to this delivery slot session.

Errors: All **MailTransport.Errors** may be raised except **connection** and **transfer**. **Courier.Error** may also be raised.

2.3.2.2 DeliverEnvelope

DeliverEnvelope retrieves the envelope of the message at the top of the delivery slot queue implied by **handle**. This operation is legal only directly after **BeginDelivery** and following any subsequent **Acknowledge**.

```
MailTransport.DeliverEnvelope: PROC [handle: DeliveryHandle, envelope: Envelope]  
    RETURNS [empty: BOOLEAN];
```

Arguments: **handle** must be a valid session handle; **envelope** points to a client-allocated **EnvelopeRecord** where the message envelope will be stored.

Results: If **empty** is **TRUE**, then the delivery slot has no messages queued and **envelope** does not represent a valid message. Storage is allocated for **envelope** fields which must be freed with **MailTransport.ClearEnvelope**.

Errors:

MailTransport.Error may be raised with the following error types: **authentication, handle, service, undefined**. **Courier.Error** may also be raised.

2.3.2.3 DeliverContent

DeliverContent retrieves the contents of the message at the top of the delivery slot queue implied by **handle**. This operation is legal only directly after **DeliverEnvelope**.

MailTransport.DeliverContent: PROC [handle: DeliveryHandle, contents: NSDataStream.Sink];

Arguments:

handle must be a valid session handle; **contents** describes a sink for the incoming data (see section 2 of *Common Facilities Programmer's Manual* [9]).

Results:

None.

Errors:

MailTransport.Error may be raised with the following error types: **authentication, handle, service, transfer, undefined**. **Courier.Error** may also be raised.

2.3.2.4 Acknowledge

The operation **Acknowledge** indicates the desired disposition of the message at the top of the delivery slot queue. An error will result if that message's envelope has not been previously examined within the context of the specified **DeliveryHandle**. This operation is legal following either **DeliverEnvelope** or **DeliverContent**.

**MailTransport.DeliveryAckType: TYPE = MACHINE DEPENDENT {
acknowledge(0), refuse(1), abort(2) };**

abort

The message at the head of the queue will remain at the top of the queue; it will be the first message available to any subsequent **DeliverEnvelope** call.

acknowledge

The message at the head of the queue is considered to have been received and is deleted from the mailbox.

refuse

The message at the head of the queue will be returned to its originator with the **UndeliverableNameType: refused**.

MailTransport.Acknowledge: PROCEDURE [handle: DeliveryHandle, reply:DeliveryAckType];

Arguments:

handle must be a valid session handle; **reply** indicates the desired disposition of the message.

Results:

None.

Errors:

MailTransport.Error may be raised with the following error types: **authentication, handle, service, undefined**. **Courier.Error** may also be raised.

2.3.2.5 EndDelivery

EndDelivery terminates the session initiated by a previous **BeginDelivery**. This operation is valid at any time in the session.

MailTransport.EndDelivery: PROCEDURE [handle: DeliveryHandle] ;

Arguments: **handle** represents the session to be terminated.

Results: None.

Errors: **MailTransport.Error** may be raised with the following error types: **authentication**, **handle**, **service**, **undefined**. **Courier.Error** may also be raised.

2.3.2.6 Poll

Poll allows the client to query the specified delivery slot as to whether mail exists.

MailTransport.Poll: PROCEDURE [
 identity: Auth.IdentityHandle, **deliverySlot**: Name, **loc**: Location ← NIL]
RETURNS [**mailExists**, **isPrimary**: BOOLEAN];

Arguments: **identity** provides authentication information about the client who wishes to poll the mailbox (see *Authentication Programmer's Manual* [1] for details on authentication); **deliverySlot** is a fully qualified name or alias identifying the mailbox which is to be examined; **loc** identifies the Mail Server which holds the mailbox. If **loc** is defaulted, the clearinghouse will be queried to locate the server holding the primary mailbox in question.

Results: If **mailExists** is **TRUE**, then there is mail in the mailbox; **isPrimary** indicates whether the mailbox is a primary or secondary mailbox.
[Note: **isPrimary** will always be **TRUE** as secondaries are not implemented in OS5.]

Errors: **MailTransport.Error** may be raised with the following error types: **access**, **authentication**, **location**, **service**, **undefined**. **Courier.Error** may also be raised.

2.4 Transport errors

When a mail transport operation is unable to complete successfully, it reports this fact by raising one of the Mesa errors, **MailTransport.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 **MailTransport** operation. Consult *Pilot Programmer's Manual* [26] for further details about **Courier.Error**.

```

MailTransport.Error: ERROR [error: ErrorRecord];

MailTransport.ErrorRecord: TYPE = RECORD [SELECT errorType: ErrorType FROM
  access = > [problem: AccessProblem],
  authentication = > [problem: AuthenticationProblem],
  connection = > [problem: ConnectionProblem],
  handle = > [problem: SessionProblem],
  location = > [problem: LocationProblem],
  service = > [problem: ServiceProblem],
  transfer = > [problem: TransferProblem],
  undefined = > [problem: UndefinedProblem],
ENDCASE];

MailTransport.ErrorType: TYPE = {
  access, authentication, connection, handle, location, service, transfer, undefined};


```

The argument to **MailTransport.Error** is a variant record, each arm of which defines a subclass (**MailTransport.ErrorType**) of error conditions. The specific problem is described by the fields of the particular variant.

2.4.1 Access errors

A **MailTransport.Error** of type **access** may be raised by any procedure that requires access to a mailbox. It indicates that access to the mailbox in question is currently not possible.

```

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

```

```

MailTransport.AccessProblem: TYPE = MACHINE DEPENDENT {
  accessRightsInsufficient(0), accessRightsIndeterminate(1), mailboxBusy(2),
  noSuchMailbox(3), mailboxNameIndeterminate(4)};

```

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

accessRightsInsufficient

The user does not have the access rights needed to perform the requested operation.

accessRightsIndeterminate	The mail service could not determine whether the user has the access rights needed to satisfy the request; for example, the clearinghouse service is unavailable to determine membership in a group.
mailboxBusy	The specified mailbox is open in a way which prevents the desired access.
noSuchMailbox	The specified mailbox was not found on the mail service.
mailboxNameIndeterminate	The mail service was not able to contact the clearinghouse service to determine the location of the mailbox.

2.4.2 Authentication errors

A **MailTransport.Error** of type **authentication** may be raised by any procedure.

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

```
NSName.AuthenticationProblem: TYPE = {
  badNameInIdentity, badPwdInIdentity, tooBusy, cannotReachAS,
  cantGetKeyAtAS, credsExpiredPleaseRetry, authFlavorTooWeak, other};
```

badNameInIdentity	There is something wrong with the name in the identity handle (e.g., it might not exist in the clearinghouse).
badPwdInIdentity	The password in the identity handle does not match the name in the identity handle.
tooBusy	The authentication service is too busy to handle the request.
cannotReachAS	Cannot make the necessary contact with the authentication service.
cantGetKeyAtAS	The authentication service cannot get the necessary key (e.g., the clearinghouse with the relevant information could be down or the entry might not exist in the clearinghouse).
credsExpiredPleaseRetry	The mail service's credentials had expired; the operation should be successful if tried a second time.
authFlavorTooWeak	Requested authentication flavor is too weak.
other	Strange or unknown authentication problem.

2.4.3 Connection errors

A **MailTransport.Error** of type **connection** may be reported by any procedure that sends data to a sink or receives data from a source. It indicates a problem in establishing the

connection for transfer of bulk data in a third-party transfer (see §3.8 of *Filing Programmer's Manual* [12]).

[**Note:** Because direct third-party transfers are not implemented in OS5, connection problems are not reported.]

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

```
MailTransport.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.

2.4.4 Location errors

A **MailTransport.Error** of type **location** may be raised by any procedure that requires the Mailing Stub to locate network resources. This can happen either when locating a mail drop for message posting, or when performing a Clearinghouse query to determine the location of a mailbox.

```
MailTransport.ErrorRecord: TYPE = RECORD [SELECT errorType: ErrorType FROM
... , location = > [problem: LocationProblem], ...];
```

```
MailTransport.LocationProblem: TYPE = {
  noCHAvailable, noSuchName, noMailboxForName, noLocationFound, noMailDropUp
};
```

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

noCHAvailable	A Clearinghouse query failed because a needed Clearinghouse Service was not available.
----------------------	--

noSuchName	The mailbox name does not exist (or is no good) in the Clearinghouse.
-------------------	---

noMailboxForName	The Clearinghouse is up but there is no mailbox for the specified name.
-------------------------	---

noLocationFound	The Clearinghouse database contains inconsistent information with respect to the location of the mailbox in question.
------------------------	---

noMailDropUp	No Mail Service was available for message posting.
---------------------	--

2.4.5 Session errors

A **MailTransport.Error** of type **session** indicates that the Mail Service encountered a problem while using a particular **MailTransport.DeliveryHandle** or **Inbasket.Session**.

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

```
MailTransport.SessionProblem: TYPE = MACHINE DEPENDENT {
  handleInvalid(0), wrongState(1) };
```

These errors relate to the state of a delivery slot or inbasket session. The argument **problem** describes the problem in greater detail.

handleInvalid	The specified handle is not valid at the server.
----------------------	--

wrongState	The operation requested is currently illegal within the context of the session.
-------------------	---

2.4.6 Service errors

A **MailTransport.Error** of type **service** indicates that the Mail Service encountered a problem due to the unavailability of some resource.

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

```
MailTransport.ServiceProblem: TYPE = MACHINE DEPENDENT {
    cannotAuthenticate(0), serviceFull(1), serviceUnavailable(2), mediumFull(3) };
```

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

cannotAuthenticate

The Mail Service is unable to determine whether the user's credentials are valid; this could occur if the Mail Service needs to contact some service that is unavailable.

serviceFull

An implementation-dependent limit concerning the activity of the Mail Service has been exceeded.

serviceUnavailable

The Mail Service is unavailable for use by new clients.

mediumFull

Occurs during message posting. The Mail Service disk storage capacity is not sufficient to successfully post this message.

2.4.7 Transfer errors

A **MailTransport.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.

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

```
MailTransport.TransferProblem: TYPE = MACHINE DEPENDENT {
    aborted(0), 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.

noRendezvous

The identifier from the other party never appeared.

wrongDirection

The other party wanted to transfer the data in the wrong direction.

2.4.8 Undefined errors

A **MailTransport.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 a particular Mail Service is malfunctioning. The client has no way of recovering from undefined errors.

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

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

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




Inbasket

This section describes the Mailing Stub **Inbasket** facility, through which clients are able to examine, retrieve, and delete electronic mail that has accumulated at an 8000 NS Mail Service. It is intended to facilitate the task of *user agents*, those programs responsible for displaying mail to human users.

Inbasket: DEFINITIONS = ...;

For the purposes of this section, an *inbasket* is a mailbox as viewed through the **Inbasket** facility. While the delivery slot facilities described in the previous section allow mailboxes to be viewed only as FIFO queues, an inbasket is a mailbox viewed as a container. The contained elements can be accessed at random and the container itself can be viewed simultaneously by several clients. While each client of this facility can choose its own usage pattern, the eventual goal is to allow mail to accumulate at the Mail Service rather than at the local storage of the mail client. This yields two important benefits. A single mailbox can be viewed equally by all Mail Service clients, and there need only be one shared copy of any given message for multiple recipients at a Mail Service. The intent of **Inbasket** is to provide the functionality to make the Mail Service a logical extension of the user agent.

Inbaskets are identified by name. Such names must be fully qualified.

Inbasket.Name: TYPE = NSName.Name;

3.1 Standard message format

As stated in the previous section, a message's content will be considered to be in standard message format if encoded as a **NSFile.SerializedFile**. This encoding format allows a subtree of Filing files to be expressed in a serial fashion, each file consisting of attributes and data. 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. It is expected that most standard format messages will be single files. Nevertheless, the encoding format provides the needed flexibility for clients that wish to send and receive entire subtrees. (Serialized files, attributes, and file content are described in detail in section 3 of *Filing Programmer's Manual* [12].)

The Mailing Stub defines and manages a set of attributes that are generally applicable to Filing files. These attributes correspond to properties normally associated with electronic

mail such as message subject, addresses, and sender. Some of these attributes are Filing extended attributes which are encoded and decoded directly by the mailing stub. Other attributes correspond directly to Filing interpreted attributes. For example, the **mailSubject** attribute maps directly to the Filing **name** attribute. Mail attributes are discussed in section 4.

Although **SerializedFile** format allows entire subtrees to be encoded, the primitives provided by the mailing stub that deal with attributes apply only to the root node of any such subtree. The attributes and contents of that root node will be referred to as the *message attributes* and *message body*. Messages may possess attributes which are not understood by the Mailing Stub. No attempt will be made to interpret such attributes. Similarly, there is no restriction on message body format. It should be remembered, however, that the Mail Service is not a conversion service. In order for useful information to flow, attribute and content format must be understood by both sender and recipient.

The following is a graphical depiction of a standard message format.

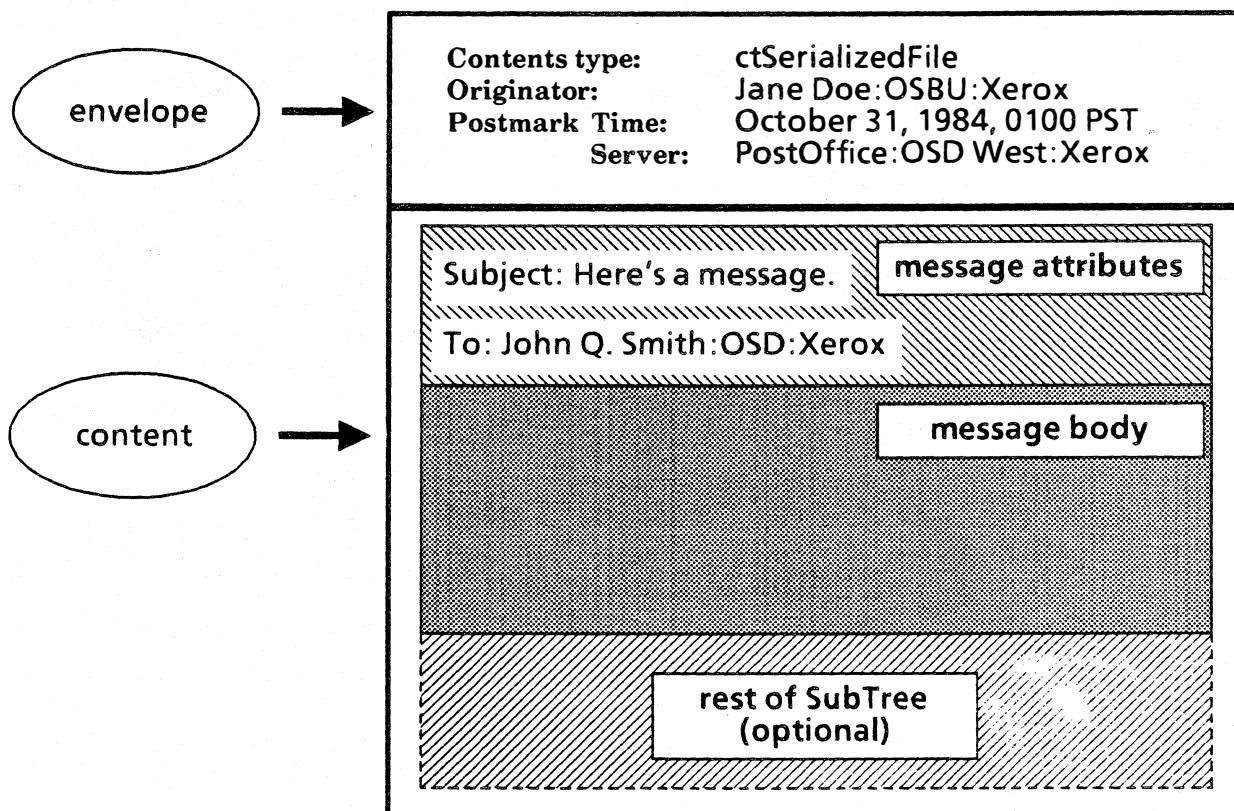


Figure 3.1 Standard message format

3.2 Finding an inbasket server

The inbasket associated with any one Clearinghouse name can legitimately reside on only one Mail Service. The name of this Mail Service and the network address of the server which houses it are registered in the Clearinghouse database. The **MailTransport** procedure **GetLocation** can be used to retrieve this information from the Clearinghouse database.

It is not necessary for every client to determine its inbasket server address. Every Inbasket procedure that requires a **MailTransport.Location** will accept **NIL** and subsequently do the required server location. **MailTransport.GetLocation** is provided to eliminate the unnecessary overhead associated with repetitive Clearinghouse queries. Its use is recommended for clients that need to check inaskets frequently.

Note: Inaskets can reside on any Mail Server and may be moved at will. Any cached information should be rechecked in the event of a failure.

(The association between a name and its inbasket location is denoted by the existence of a **mailboxes** property for that name which the Mail Service adds to the Clearinghouse database. This property is interpreted as the name of the Mail Service holding the inbasket in question.)

3.3 Inbasket sessions

A session encapsulates the state of interaction between an **Inbasket** client and the target **Mail Service**. A session begins when a client *logs on* and is completed when the client *logs off*. A session handle is used to identify and refer to the state information underlying a session. At any time, a session handle may be involved in at most one inbasket operation. A session handle may become invalid at any time. Typically, this will happen only when the target mail service is stopped, or when the session has been inactive for a long period of time.

Inbasket.Session: **TYPE [2];**

The constant **nullSession** is provided for client convenience.

Inbasket.nullSession: **Session = [LONG[NIL]] ;**

3.3.1 Creating and deleting sessions

Logon is used to initiate a session to a given mail service. Inbasket sessions provide access to one and only one inbasket which must be specified when the session is created. The client can optionally acquire exclusive access to the named inbasket or permit the existence of other simultaneous sessions.

```
Inbasket.Logon: PROCEDURE [
    identity: Auth.IdentityHandle, inbasket: Name,
    cacheCheck: CacheVerifier  $\leftarrow$  nullCacheVerifier,
    allowSharing: BOOLEAN  $\leftarrow$  FALSE, loc: MailTransport.Location  $\leftarrow$  NIL
    RETURNS [session: Session, cacheStatus: CacheStatus];
```

Arguments: **identity** provides authentication information about the client who wishes to log on (see *Authentication Programmer's Manual* [1] for details on authentication); **inbasket** is a name or an alias describing the inbasket which is to be examined; **cacheCheck** is a **CacheVerifier** which the client can optionally use to verify the state of a locally cached inbasket (see §3.3.3 for details); if **allowSharing** is **TRUE**, more than one such session involving the named inbasket will be allowed to coexist, otherwise exclusive access is assumed; **loc** identifies the Mail Service to which a session is desired; if **NIL** is specified, the Mailing Stub will query the Clearinghouse to determine the network address of the Mail Service corresponding to **inbasket**.

Results: **session** is a session handle which can be used for further operations on the inbasket just opened; **cacheStatus** is the result of a local cache validity check which is performed using the **cacheCheck** argument.

Errors: **MailTransport.Error** is raised with the following error types: **access**, **authentication**, **location**, and **service**. **Courier.Error** may also be raised.

Logoff is used to terminate a session. The mail system verifies that the request is valid, invalidates the session, and frees any allocated resources.

Inbasket.Logoff: PROCEDURE [session: Session] RETURNS [CacheVerifier];

Arguments: **session** denotes the session to be terminated.

Results: **cacheVerifier** is a token that can be passed to the next **Logon** call for purposes of verifying a local cache.

Errors: **MailTransport.Error** is raised with the error type **handle**.

3.3.2 Inbasket state

An existing session embodies a consistent set of inbasket state information. This information remains consistent over the life of the session, even if the inbasket is shared with another session. This message specific information is totally separate from that kept by the Mail Transport system.

```
Inbasket.State: TYPE = MACHINE DEPENDENT RECORD [
    lastIndex(0): Index,
    newCount(1): CARDINAL,
    isPrimary(2): BOOLEAN,
    isPrimaryUp(3): BOOLEAN];
```

The single most important state data is the range of valid indices by which messages can be referenced. Valid indices for any given session range from [1..**lastIndex**]. The mapping between indices and messages remains constant over the life of a session. This is true even if the entire contents of the inbasket are deleted by another simultaneous session. The deleted messages become invisible with respect to new sessions, but always remain within the *view* of an existing session. **lastIndex** may increase at any time as messages are added to the inbasket, but will never decrease as viewed through a single session.

```
Inbasket.Index: TYPE = CARDINAL;  
Inbasket.IndexRange: TYPE = MACHINE DEPENDENT RECORD [first, last: Index];  
Inbasket.nullIndex: Index = 0;
```

The **newCount** field of **Inbasket.State** describes the number of messages with a message status of **new** (\$4.2 describes the intended values and interpretations of the defined message state values). **Note:** The **isPrimary** and **isPrimaryUp** fields are currently always **TRUE** and will be used in a future release.

The following procedure can be used for checking the inbasket state embodied by a session. It serves the additional purpose of maintaining activity on the session to prevent its deletion due to prolonged inactivity. (Any inbasket operation which takes a session handle as an argument serves to prolong the session, but this is the most efficient.)

```
Inbasket.MailCheck: PROCEDURE [session: Session]  
    RETURNS [state: State, checkAgainWithin: CARDINAL];
```

Arguments: **session** must be a valid session handle.

Results: **state** is the **Inbasket.State** associated with the argument session handle; **checkAgainWithin** indicates the time (in seconds) remaining until session will become invalid due to inactivity.

Errors: **MailTransport.Error** may be raised with the following error types: **authentication, handle, service**. **Courier.Error** may also be raised.

It is also possible to examine the state of an inbasket from outside a session using the procedure **MailPoll**. The state returned is similar to that resulting from **Inbasket.MailCheck**, but since no session is involved, this state is only a temporary hint. Since the inbasket client must subsequently log on to do useful work, there is always a window of time during which the inbasket state might change.

```
Inbasket.MailPoll: PROCEDURE [  
    identity: Auth.IdentityHandle, inbasket: Name, loc: MailTransport.Location ← NIL]  
    RETURNS [State];
```

Arguments: **identity** provides authentication information about the client who wishes to log on (see *Authentication Programmer's Manual* [1] for details on authentication); **inbasket** is a name or an alias describing the inbasket which is to be examined; **loc** identifies the mail service to which a session is desired. If **NIL** is specified the Mailing Stub will query the Clearinghouse to determine the network address of the mail service corresponding to **inbasket**.

Results: The **Inbasket.State** associated with the specified mailbox is returned.

Errors: **MailTransport.Error** may be raised with the following error types: **access, authentication, location, service**. **Courier.Error** may also be raised.

Note: The OS5 mail service does not perform access control checking or authentication on incoming **MailPoll** calls. However, the client should not rely on the continuation of this policy in future releases.

3.3.3 Inbasket caching

[Note: Inbasket caching is not implemented in OS5.]

A client may choose to maintain a cached copy of all or part of an inbasket over more than one session. In order to do this, there must be some way to check the validity of the cached copy with respect to the actual inbasket at the mail service. This is done by means of the **CacheVerifier**.

Inbasket.CacheVerifier: TYPE [4];

Inbasket.nullCacheVerifier: CacheVerifier = ...;

A client keeping a cache will at some time be forced to build it from scratch by enumerating the state of the inbasket (see §3.4.5). Once this is done, the cache is in synch with the state of the real inbasket. It is assumed that the client can continuously update this cache so as to mirror the operations performed during the course of a session. Upon terminating the session, the Mail Service returns a 64-bit *cache verifier* to the client. Since we are assuming that the cache is valid at the termination of the session, this *cache verifier* can be used as an argument to the next session establishment to determine if the state of the real inbasket has changed.

Inbasket.CacheStatus: TYPE = MACHINE DEPENDENT {correct(0), incomplete(1), invalid(2)};

Inbasket.Logon returns a **CacheStatus** as a result along with the session handle. Its value will be based on the state of the inbasket as compared with its state when the argument *cache verifier* was issued. **correct** implies that the inbasket state is the same and that the cache is valid; **incomplete** suggests that new messages have arrived; and **invalid** means that the cache must be rebuilt.

3.4 Inbasket operations

This section describes the operations provided for examining and manipulating the contents of inbaskets within the context of a session.

3.4.1 Locate

Any inbasket can be scanned for the first occurrence of a message of a particular message status. This might be useful, for example, in finding the first unread message.

Inbasket.Locate: PROCEDURE [session: Session, status: MessageStatus] RETURNS [Index];

Arguments: **session** is a valid session handle; **status** is the message status to be searched for.

Results: The index of the first message with the specified message status.

Errors: **MailTransport.Error** may be raised with the following error types: **authentication**, **handle**, **service**. **Courier.Error** may also be raised.

3.4.2 ChangeStatus

The inbasket client can change the message status of any range of messages.

Inbasket.ChangeStatus: PROCEDURE [
 session: Session, **range:** IndexRange, **status:** MailAttributes.MessageStatus];

Arguments: **session** is a valid session handle; **range** is an **IndexRange** specifying the messages to be affected; **status** is the new value for the message status of the affected messages.

Results: None.

Errors: **MailTransport.Error** may be raised with the following error types: **authentication**, **session**, **service**. **Inbasket.InvalidIndex** and **Courier.Error** may also be raised.

3.4.3 Retrieve

The contents of any message can be retrieved at will. The retrieval operation also returns the message transport envelope along with whatever inbasket information is stored with the message.

It is assumed that inbasket clients will typically be prepared to handle only one message contents type during message retrieval. This is a useful assumption if the bulk data sink is to be some operation like **NSFile.Deserialize** which decodes the incoming serialized data on the fly. To make this possible, **Inbasket.Retrieve** requires that the client specify an **expectedContentsType**. An error will be raised if the actual type does not match the expected one. If the client does not care to specify an expected type, the constant **MailTransport.nullContentsType** may be used to suppress the contents mismatch error.

Inbasket.Retrieve: PROCEDURE [
 session: Session, **message:** Index, **expectedContentsType:** MailTransport.ContentsType,
 contents: NSDataStream.Sink, **envelope:** MailAttributes.Envelope];

Arguments: **session** is a valid session handle; **message** is the index of the desired message; **expectedContentsType** defines the message format which the client expects to receive; **contents** describes a sink for the incoming data; **envelope** points to a client-allocated **EnvelopeRecord** where the message envelope will be stored.

Results: The contents of the specified message are retrieved to **contents**. The message envelope is returned within the referent of **envelope**.

Errors: **MailTransport.Error** may be raised with the following error types: **authentication**, **connection**, **handle**, **service**, **transfer**. The **Inbasket** errors **ContentsTypeMismatch** and **InvalidIndex** may be raised, **Courier.Error** may also be raised.

Note: Filing clients expecting to encounter standard format messages will typically use this operation in conjunction with **NSFile.Deserialize**, which serves to decode such messages into a subtree of Filing files. See §3.8.2 of *Filing Programmer's Manual* [12] for more detail on this operation.

Inbasket.Retrieve allocates storage in the course of providing envelope information. This storage must be freed with a call to **MailAttributes.ClearEnvelope**.

3.4.4 Delete

Any range of messages may be deleted.

Inbasket.Delete: PROCEDURE [session: Session, range: IndexRange];

Arguments: **session** is a valid session handle; **range** is an **IndexRange** specifying the messages to be deleted.

Results: None.

Errors: **MailTransport.Error** may be raised with the following error types: **authentication**, **handle**, **service**. **Inbasket.InvalidIndex** and **Courier.Error** may also be raised.

3.4.5 List

Inbasket.List makes it possible for a client to enumerate and examine the properties of messages within an inbasket. This procedure gives special status to standard format messages by interpreting contents so as to allow message attributes to be returned in the enumeration. If a non-standard format message is present in an inbasket, no attempt at interpretation will be made and the message will (correctly) appear to have no attributes. Since all messages have an envelope, *message properties* can then be considered to be a summation of the information contained in both envelope and attributes.

Inbasket.List: PROCEDURE [
 session: Session, **range**: IndexRange,
 selections: MailAttributes.Selections, **proc**: ListProc];

Arguments: **session** is a valid session handle; **range** is an **IndexRange** specifying the messages to be enumerated; **selections** determines the message properties the client is interested in examining; **listProc** is a client specified procedure to be called for each message in the enumeration.

Results: None.

Errors: **MailTransport.Error** may be raised with the following error types: **authentication**, **handle**, **service**. **Inbasket.InvalidIndex** and **Courier.Error** may also be raised.

The client must provide a procedure to be called for each element of the inbasket enumeration. This procedure must be of the following type:

```
Inbasket.ListProc: TYPE = PROCEDURE [msg: Index, props: MailAttributes.MailProperties]
RETURNS [continue: BOOLEAN ← TRUE];
```

msg is the index of the current message; **props** is a pointer to a record which contains the relevant property information. The client can terminate the enumeration by returning with **continue** set to **FALSE**.

3.5 Inbasket errors

When an inbasket operation is unable to complete successfully, it reports this fact by raising one of the Mesa errors, **MailTransport.Error**, **Inbasket.ContentsTypeMismatch**, **Inbasket.InvalidIndex**, or **Courier.Error**. These errors are used to report any condition that makes continued execution of a procedure impossible (e.g., the client may have specified incorrect arguments to a procedure, or some required resource may be unavailable).

3.5.1 Contents type errors

```
Inbasket.ContentsTypeMismatch: ERROR [correctType: MailTransport.ContentsType];
```

The error **ContentsTypeMismatch** may be raised by **Inbasket.Retrieve**. It is raised if the expected contents type specified in the **Inbasket.Retrieve** call does not match the actual contents type of the message. The parameter **correctType** denotes the actual contents type in the message envelope.

3.5.2 Invalid index errors

```
Inbasket.InvalidIndex: ERROR [badIndex: Index];
```

The error **InvalidIndex** may be raised by any procedure which takes an **Inbasket.Index** or **Inbasket.IndexRange** as an argument. The parameter **badIndex** describes the index found to be invalid or out of range. Indices of messages that have been deleted within a session take on special semantics. Reference to a deleted message will cause an error only if the procedure in question takes a single **Index** as an argument. Procedures such as **Inbasket.List**, which operate on an **IndexRange**, will simply skip over deleted messages.



Mail attributes

This section describes **MailAttributes**, a facility which describes the standard message format and allows clients of a Xerox 8000 NS Filing file system to attach message attributes to files which can then later be used as standard format messages.

MailAttributes: DEFINITIONS = ...;

Since message attributes are really Filing attributes, the procedures described in this section act as a translation facility between Filing and Mailing Stub attribute interpretations. For the most part, this involves encoding the Mesa representation of message properties into Filing extended attributes, and vice versa. In certain cases, message properties correspond to Filing interpreted attributes. In these cases a direct mapping is made.

When a message is retrieved from the Mail Service, envelope information is returned as a result of the retrieval. Since the message envelope contains useful information that the client might want to retain, the attribute encoding and decoding procedures provide for encoding and decoding this envelope information which will then be stored along with the attributes for future reference.

MailAttributes makes no direct use of the Filing file system, but is designed to be used in conjunction with those Filing procedures that read and write file attributes. For more information concerning Filing attributes and the available operations to manipulate them, see section 5 of *Filing Programmer's Manual* [12].

The following definitions will be used throughout:

MailAttributes.Name: TYPE = **NSName.Name**;

MailAttributes.NameList: TYPE = **MailTransport.NameList**;

MailAttributes.String: TYPE = **NSString.String**;

MailAttributes.Words: TYPE = **LONG DESCRIPTOR FOR ARRAY OF UNSPECIFIED**;

4.1 Message attributes

The following attributes are supported by the Mailing Stub. Each definition provides a description of the meaning and purpose of the attribute and its Mesa definition. Unless otherwise noted in the definition, each attribute is implemented as a Filing extended (uninterpreted) attribute.

```
Attribute: TYPE = RECORD [
    -- end to end message attributes settable by client
    var: SELECT type: AttributeType FROM
        mailAnswerTo, mailCopies, mailFrom, mailTo => [value: NameList],
        mailInReplyTo, mailNote, mailSubject => [value: String],
        mailBodySize, mailBodyType => [value: LONG CARDINAL],
ENDCASE];
```

AttributeList: TYPE = LONG DESCRIPTOR FOR ARRAY OF Attribute;

```
AttributeType: TYPE = {
    mailAnswerTo, mailCopies, mailFrom, mailInReplyTo,
    mailNote, mailSubject, mailTo, mailBodySize, mailBodyType };
```

Message attributes fall into two categories: *mandatory* and *optional*. The following attributes are mandatory and should be defined for all standard format messages.

MailAttributes.Attribute: TYPE = RECORD [. . . , mailFrom => [value: NameList], . . .];

mailFrom

The **mailFrom** attribute is a list of fully-qualified name(s) which the message originator can use to identify the sender(s) of the message. The sender named by this attribute should not be confused with the **originator** specified in the message **envelope** by the message transport system.

MailAttributes.Attribute: TYPE = RECORD [. . . , mailTo => [value: NameList], . . .];

mailTo

The **mailTo** attribute is a list of fully-qualified names which indicate the primary recipients of the message. Additional message recipients can be indicated in the **mailCopies** attribute. Remember that the mail transport system does not interpret message contents. Therefore, there is no direct relationship between the **mailTo** and **mailCopies** fields and the actual recipients of the message. It is the responsibility of the client to maintain that correspondence.

MailAttributes.Attribute: TYPE = RECORD [. . . , mailSubject => [value: String], . . .];

mailSubject

The **mailSubject** attribute is a **String** which contains the subject of the message. This attribute is synonymous with the Filing interpreted attribute **name** and must therefore satisfy the constraints which apply to that attribute.

MailAttributes.Attribute: TYPE = RECORD [. . . , mailBodySize = > [value:LONG CARDINAL], . . .];

Inbasket.nullBodySize: LONG CARDINAL = LAST [LONG CARDINAL];

mailBodySize

The root node of every standard message consists of message attributes and message body. **mailBodySize** records the number of client-visible bytes in a file. This attribute is synonymous with the Filing interpreted attribute **sizeInBytes**.

MailAttributes.Attribute: TYPE = RECORD [. . . , mailBodyType = > [value: LONG CARDINAL], . . .];

MailAttributes.nullBodyType: LONG CARDINAL = NSAssignedTypes.tUnspecified;

mailBodyType

The root node of every standard message consists of message attributes and message body. **mailBodyType** describes the format of the data in the message body. This attribute is synonymous with the Filing interpreted attribute **type**.

The following attributes are optional. They need not be present in all standard format messages.

Inbasket.Attribute: TYPE = RECORD [. . . , mailAnswerTo = > [value: NameList], . . .];

mailAnswerTo

The **mailAnswerTo** attribute is a list of fully-qualified names which identify recipients to whom replies to the message should be sent. This field can be used by client software to fill in the **to** field of the reply message.

Inbasket.Attribute: TYPE = RECORD [. . . , mailCopies = > [value: NameList], . . .];

mailCopies

The **mailCopies** attribute lists any additional recipients of the message.

Inbasket.Attribute: TYPE = RECORD [. . . , mailInReplyTo = > [value: String], . . .];

mailInReplyTo

The **mailInReplyTo** attribute is a **String** which identifies the message to which this message is a response. This field is often filled in by client software when the user chooses to answer a given message.

Inbasket.Attribute: TYPE = RECORD [. . . , mailNote = > [value: String], . . .];

mailNote

The **mailNote** attribute can contain text in addition to or instead of the message body. This attribute is typically used to hold some comment about the message, such as a note as to its importance or a brief description of its contents.

The following data types are provided for returning decoded message properties to the client. There is a field in an **AttributesRecord** corresponding to each possible message attribute type.

```
MailAttributes.Attributes: TYPE = LONG POINTER TO AttributesRecord;
```

```
MailAttributes.AttributesRecord: TYPE = RECORD [
```

-- these map to filing extended attributes:

answerTo: NameList,

copies: NameList,

from: NameList,

to: NameList,

inReplyTo: String,

note: String,

-- these map to filing interpreted attributes:

subject: String,

BodySize: LONG CARDINAL,

bodyType: LONG CARDINAL];

The following constant describes null values for all message properties:

```
MailAttributes.nullAttributesRecord: AttributesRecord = ... ;
```

4.2 Envelopes

In addition to the message attributes information, the Mail Transport system and Inbasket mechanism have message specific information which is passed to the client in a **MailAttributes.Envelope**.

```
MailAttributes.Envelope: TYPE = LONG POINTER TO EnvelopeRecord;
```

```
MailAttributes.EnvelopeRecord: TYPE = RECORD [
```

transport: MailTransport.EnvelopeRecord, **inbasket:** InbasketEnvelopeRecord];

```
MailAttributes.InbasketEnvelopeRecord: TYPE = MACHINE DEPENDENT RECORD [  
status:(0) MessageStatus];
```

```
MailAttributes.nullEnvelopeRecord: EnvelopeRecord = ... ;
```

Each message has a *message status* which describes whether it has been 'seen' by the intended recipient. The message status is totally under control of the client. Here are the intended interpretations for the defined values.

```
MailAttributes.MessageStatus: TYPE = MACHINE DEPENDENT {  
new(0), known(1), received(2), (256B);
```

new The message is newly delivered and unknown to the recipient.

known The recipient knows of the existence of this message.

received The recipient has seen the contents of this message.

The following procedure is provided for freeing storage allocated for envelope storage:

```
MailAttributes.ClearEnvelope: PROCEDURE [envelope: Envelope];
```

4.3 Attribute encoding and decoding

These types and procedures are provided to interconvert Mailing and Filing attribute types. Allowance is made for the envelope information to be stored along with the attributes in a single encoded format.

MailAttributes.FileAttributeList: TYPE = **NSFile.AttributeList**;

MailAttributes.FileSelections: TYPE = **NSFile.Selections**;

MailAttributes.BooleanFalseDefault: TYPE = **BOOLEAN** ← FALSE;

The structure **MailAttributes.Selections** allows a client to specify a set of message properties.

MailAttributes.Selections: TYPE = RECORD [

envelope: BOOLEAN ← TRUE,

attributes: PACKED ARRAY AttributeType OF BooleanFalseDefault];

envelope indicates a desire to examine the **MailAttributes.Envelope** containing both the mail transport and inbasket envelopes. **attributes** includes a **BOOLEAN** value for each message attribute supported by the Mailing Stub. A value of **TRUE** indicates that the client desires to examine the property so specified.

4.3.1 Decoding

In order to decode the message properties associated with a Filing file, the client must be able to discern those Filing attributes which are used for storing message properties. Given a selection of desired message attributes, the following routine returns the selection of corresponding Filing attributes.

MailAttributes.MapSelections: PROCEDURE [

selections: **MailAttributes.Selections**, **mergeWith:** **FileSelections** ← []]

RETURNS [**FileSelections**];

Arguments: **selections** specifies a desired set of message properties; **mergeWith** specifies a set of Filing attributes that can be merged into the results.

Results: The resulting **NSFile.Selections** describes a set of Filing attributes that correspond to the specified message properties.

Errors: None.

The results returned by **MapSelections** must be freed by the client using:

MailAttributes.FreeFileSelections: PROCEDURE [**selections:** **FileSelections**];

Typically, a client will call **MapSelections** to obtain a selection of Filing attributes corresponding to the message properties to be decoded. The resultant attribute selection can then be used to invoke a Filing operation to actually read the file attributes. Filing returns these attribute values using the Mesa structure **NSFile.AttributesRecord**, from which **DecodeProperties** can extract whatever message properties are present. Filing

attributes that do not correspond to message properties will be ignored. The envelope information is also extracted for the client.

```
MailAttributes.MailProperties: TYPE = LONG POINTER TO MailPropertiesRecord;
MailAttributes.MailPropertiesRecord: TYPE = RECORD[
    env: EnvelopeRecord, attrs: AttributesRecord];
```

```
MailAttributes.DecodeProperties: PROCEDURE [
    fAttrs: NSFile.Attributes, props: MailAttributes.MailProperties];
```

Arguments: **fAttrs** points to a **NSFile.AttributesRecord** which should contain the Filing attributes to be decoded; **attributes** must point to a client-allocated **MailPropertiesRecord** where the results of the decoding will be stored.

Results: The message properties contained in **filingAttr** are decoded and returned within the referent of **props**.

Errors: **MailAttributes.BadEnvelope** and **MailAttributes.MalformedAttribute**.

Caution: Partial results will not be returned for attributes that cannot be deserialized (e.g., malformed attributes). In these cases, a **NIL** value for the attribute in question will be returned.

Attribute decoding allocates storage so as to return results to the client. This storage must be freed with the following procedure:

```
MailAttributes.ClearProperties: PROCEDURE [props: MailProperties];
```

To save storage space, the following procedure may be called to eliminate duplicate domain and organization names in an attributes record:

```
MailAttributes.UnqualifyAttributeName: PROC[attributes: Attributes, defaultName: Name];
```

4.3.2 Encoding

Message attributes and envelope can be encoded into Filing attributes with **EncodeProperties**. The resulting **NSFile.AttributeList** can be subsequently passed to a Filing procedure such as **NSFile.ChangeAttributes** for writing the encoded attributes to a file.

```
MailAttributes.EncodeProperties: PROCEDURE [
    attrList: AttributeList, env: MailAttributes.Envelope ← NIL, defaultName: Name ← NIL]
    RETURNS [FileAttributeList];
```

Arguments: **attrList** describes an array of **MailAttributes.Attribute** to be encoded along with the **env** information; **defaultName** is used to qualify any names in **attrList** that are not fully qualified.

Results: A descriptor for array of **NSFile.Attribute** is returned. This array represents the Filing attribute encoding of the arguments **attrList** and **env**.

Errors: **MailAttributes.BadEnvelope** and **MailAttributes.IllegalAttribute**.

The client might choose to eliminate some set of message properties from permanent storage. The following procedure returns an **NSFile.AttributeSetList** which can be used for this purpose. Only Filing extended attributes are affected.

MailAttributes.EncodeNil: PROCEDURE [selections: Inbasket.Selections]
RETURNS [FileAttributeList];

Arguments: **selections** describes a set of message properties to be eliminated.

Results: A descriptor for array of **NSFile.Attribute** is returned. Each element of this array is a **Nil** extended attribute corresponding to a selected message property.

Errors: None.

The following procedure must be used to free any **FileAttributeList** returned by a **MailAttributes** operation. It should not be used for lists allocated by other facilities.

MailAttributes.FreeFileAttributes: PROCEDURE [list: FileAttributeList];

4.4 Encoding and decoding standard format messages

The following procedures provide a stream filter facilitating creation and interpretation of serialized files which are in standard message format.

MailAttributes.SerialStream: TYPE = Stream.Handle;
MailAttributes.SerialStreamDirection: TYPE = { send, receive };

MailAttributes.MakeSerializer: PROCEDURE [
source: Stream.Handle, direction: SerialStreamDirection]
RETURNS [SerialStream];

Arguments: **source** is the stream which will be encapsulated in the **SerialStream**; **direction** indicates whether data is to be sent or retrieved on the stream.

Results: **source** will be encapsulated in the necessary format to make it a serialized file. The caller must do *SendNow* on the **SerialStream** to terminate the file if **direction** is **send**.

Errors: None.

The following procedures can be optionally used with a **SerialStream** to *get/put* mailing attributes. If they are called, they must be called immediately after **MakeSerializer** and before any other *get/put* on the **SerialStream**.

MailAttributes.ReceiveAttributes: PROC [source: SerialStream, attributes:
MailAttributes.Attributes];

Arguments: **attributes** points to a client-allocated **AttributesRecord** which will be filled in with the attributes on **source**.

Results: The attributes on the stream are deserialized and put in **attributes**, and the client need only *get* his message content to complete the transfer of information.

Errors: **MalformedAttribute** and **NotASerializedFile** may be raised.

MailAttributes.SendAttributes: PROCEDURE [

dest: SerialStream, attributesList: MailAttributes.AttributeList, defaultName: Name ← NIL];

Arguments: The attributes described by **attributesList** will be serialized and *put* to **dest**; **defaultName** will be used to qualify any unqualified names in **attributesList**.

Results: The attributes are serialized, and the client need only *put* his message content and call *SendNow* to complete the transfer of information.

Errors: **IllegalAttribute** and **NotASerializedFile** may be raised.

4.5 Signals and errors

MailAttributes.NotASerializedFile: ERROR;

This error may be raised by all of the Serialized File filter operations mentioned in §4.4 except *puts* to the **SerialStream**.

MailAttributes.BadEnvelope: SIGNAL;

This signal indicates unsuccessful decoding/encoding of the envelope. It is raised by **EncodeProperties** and **DecodeProperties**. If resumed while decoding, a **nullEnvelope** will be returned. It is not a good idea to resume it while encoding as the envelope will be left uninitialized.

MailAttributes.IllegalAttribute: SIGNAL;

This signal indicates unsuccessful encoding of an attribute and is raised by **EncodeProperties**. It makes no sense to resume this signal (doing so is a no-op).

MailAttributes.MalformedAttribute: SIGNAL [type: AttributeType, words: Words];

This signal indicates a serialization error during attribute decoding. It is raised by **DecodeProperties** and **ReceiveAttributes**. If resumed, the attribute will be assigned some appropriate *null* value.

Mail stream

This section describes **MailStream**, a facility which allows clients to post and receive standard format messages using unformatted data streams.

MailStream: DEFINITIONS = ...;

As described in §3.1, standard message format allows subtrees of files to be expressed in a serial fashion, each file consisting of attributes and data. **MailStream** is a conversion utility which builds single node standard format messages, given message attributes and a stream of unformatted data. Conversely, it provides a mechanism to parse the root-level element of an incoming serialized tree into attributes and data. This function is useful within operating environments that do not provide operations for handling serialized file format. It is important to note that only root nodes are handled; the remainder of any incoming tree is ignored.

5.1 Message posting

MailStream.Send is used for posting messages. It calls **MailTransport.Post** and so the arguments and argument syntax are similar. Clients of the **Send** operation provide a **StreamProc** in which their stream manipulations are done. Putting data to the stream provided within the **StreamProc** will cause that data, along with the specified attributes, to be posted as a standard format message. The client is not responsible for the deletion of the stream handle provided within the **StreamProc**. A **TRUE** return from this procedure will cause the entire posting operation to be canceled.

MailStream.StreamProc: TYPE = PROCEDURE [
stream: Stream.Handle] RETURNS [aborted: BOOLEAN];

MailStream.Send: PROCEDURE [
identity: Auth.IdentityHandle, recipients: MailTransport.NameList,
postIfInvalidRecipients, allowDLRecipients: BOOLEAN,
attributes: MailAttributes.AttributeList, sendProc: StreamProc]
RETURNS [invalidNames: MailTransport.Undeliverables];

Arguments: **identity** provides authentication information which is used to validate the sender; **recipients** is a list of those to whom the message is to be delivered; **postIfInvalidRecipients** determines whether the message will be delivered to the valid recipients in the event that any invalid recipients are specified, otherwise this condition results in an error. **allowDLRecipients** allows the message to be sent to recipients which represent distribution lists, otherwise this condition results in an error. The **attributes** are sent along with the message body and will be associated with the message file, but are not inspected by the mail transport system. The client puts the message body to the stream in **postProc**.

Results: A single level standard format message, consisting of the argument **attributes** and the data put to the stream within **postProc**, is addressed to **recipients** and posted. The contents type of the resultant message will be **ctSerializedFile**. If **postIfInvalidRecipients** is **TRUE**, all names in **recipients** which are not valid will be returned in **InvalidNames**.

Errors: If either **postIfInvalidRecipients** or **allowDLRecipients** is **FALSE**, then **MailTransport.InvalidRecipients** may be raised. **MailTransport.Error** may be raised with the following error types: **authentication**, **connection**, **location**, **service**, **transfer**. **Courier.Error** may also be raised.

5.2 Message retrieval

MailStream.Retrieve is used for retrieving messages. It calls **Inbasket.Retrieve** and therefore has a similar argument structure. Clients of this operation also provide a **StreamProc** in which their stream manipulations are done. Getting data from the stream provided will return the deserialized data content of the message body in an unspecified format. The client is not responsible for the deletion of the stream handle provided within the **StreamProc**. Returning **TRUE** from this procedure causes the retrieval to be canceled.

```
MailStream.Retrieve: PROCEDURE [
session: Inbasket.Session, message: Inbasket.Index, retrieveProc: StreamProc];
```

Arguments: **session** is obtained by doing an **Inbasket.Logon**; **message** is a number that specifies which message in the inbasket is to be retrieved. The client gets the message body from the stream in **retrieveProc**.

Results: The message body of the specified message can be read from the stream handle provided within **retrieveProc**.

Errors: **MailStream.FormatError** will be raised if the message is not correctly encoded in standard message format. **Inbasket.ContentsTypeMismatch** indicates that the message being retrieved is not of contents type **ctSerializedFile**. **Inbasket.InvalidIndex** may be raised. The following types of **MailTransport.Error** may be raised: **access**, **connection**, **handle**, **service**, **transfer**. **Courier.Error** may also be raised.

5.3 Mail stream errors

In general, **MailStream** allows all errors raised by underlying transport and inbasket operations to pass through to the client. In addition, **FormatError** may be raised during message retrieval if the message is of contents type **ctSerializedFile**, but is not really in standard message format.

MailStream.FormatError: ERROR;

XEROX



Services 8.0 Programmer's Guide

Printing Programmer's Manual

November 1984

PRELIMINARY

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

(

(

(



Table of contents

1	Introduction	1-1
1.1	Overview	1-1
1.2	Definition of Terms	1-1
2	Interface	2-1
2.1	Basic types	2-1
2.2	Freeing storage	2-2
2.3	Print	2-3
	2.3.1 Print request status	2-4
	2.3.2 Printer status	2-5
	2.3.3 Printer properties	2-6
2.4	Errors	2-7
3	NSPrint interface	3-1

Table of contents



Introduction

This document describes **NSPrint**, the interface to the Mesa implementation of the Printing Protocol.

1.1 Overview

NSPrint: DEFINITIONS =
BEGIN'...

NSPrint provides a Mesa interface to the Courier-based *Printing Protocol* [29], which, in turn, provides a standard method of transmitting an *Interpress* [18] master to a Print Service. This interface defines the procedures and data structures required for full compliance with the standard which, together with *Bulk Data Transfer Protocol* [3], provides all that is necessary to communicate with a Print Service that also supports the Printing Protocol.

1.2 Definition of terms

This section defines some common terms used in printing in general and in this document in particular.

Banner sheet (or *break page*) the sheet produced by the printer to identify the request and to separate one print request from the next. It may be optional or not provided at all on some printers.

Interpress master (also, *master*) the file which contains the imaging instructions for producing the printed results. The encoding conforms to the Interpress standard as defined in *Interpress Electronic Printing Standard* [18].

Media the material (and size) upon which the image is to be printed. The only choice is paper with various sizes which conform to standard sizes, or a specific size in millimeters.

Printer the Print Service which provides the Courier export of **NSPrint**.

Printer Properties

the more-or-less static capabilities and enabled options of the printer and the total inventory of the media that is available, including that accessible only through operator intervention.

Printer Status

the current state of the various subsystems comprising Print Service and the media which is immediately available. The **spooler** is the subsystem which processes the **Print** calls, the **formatter** is the subsystem which converts the Interpress master into a form suitable for marking, and the **printer** is the marking engine.

Print Request or job

the attributes, options and interpress master sent to the printer via the **Print** procedure. This job is uniquely identified by the **RequestID** returned by **Print**.



Interface

The following sections describe all aspects of the **NSPrint** interface.

2.1 Basic types

The following are the definitions of Mesa **TYPEs** used in two or more procedures.

Time: **TYPE** = **LONG CARDINAL**;

Time should contain a value consistent with **System.GreenwichMeanTime** (and [32]).

String: **TYPE** = **NSString.String**;

String.bytes format and characters should conform to the OIS Character set.

RequestID: **TYPE** = **System.UniversalID**;

Defines a document transmitted via **Print**. It is returned at the successful completion of **Print** and is used in calls to **GetPrintRequestStatus** (to the same host).

Media: **TYPE** = **LONG DESCRIPTOR FOR ARRAY OF Medium**;

Medium: **TYPE** = **MACHINE DEPENDENT RECORD** [
var(0): **SELECT type(0): MediumType FROM**
paper = > [paper(1): Paper],
ENDCASE];

MediumType: **TYPE** = **MACHINE DEPENDENT {paper(0)}**;

MediumIndex: **TYPE** = **CARDINAL[0..1]**;
Paper: **TYPE** = **MACHINE DEPENDENT RECORD** [
var(0): **SELECT type(0): PaperType FROM**
unknown = > [], --illegal argument, possible result
knownSize = > [knownSize(1): PaperSize],
otherSize = > [otherSize(1): PaperDimensions],
ENDCASE];

PaperType: TYPE = MACHINE DEPENDENT {unknown(0), knownSize, otherSize(2)};

PaperIndex: TYPE = CARDINAL[0..3];

PaperSize: TYPE = MACHINE DEPENDENT {
dontUse(0) --the protocol defines this enumeration as starting at 1!--,
 usLetter, usLegal, a0, a1, a2, a3, a4, a5, a6, a7, a8, a9,
 isoB0, isoB1, isoB2, isoB3, isoB4, isoB5, isoB6, isoB7, isoB8, isoB9, isoB10,
 jisB0, jisB1, jisB2, jisB3, jisB4, jisB5, jisB6, jisB7, jisB8, jisB9,
 jisB10(34)};

PaperDimensions: TYPE = MACHINE DEPENDENT RECORD [
 length(0), width(1): CARDINAL]; *--units are millimeters*

The **Media** array is used for two purposes: (1) It is used by the client to specify the medium on which a print request is to be rendered; and, (2) it is used by the print service to return status information about the media available for printing.

Media is an array that defines the size(s) of output media. When **Media** is the result of a status procedure, the array contains either the single item **unknown** (indicating that the print service cannot determine the media sizes), or one hundred or less other items indicating the sizes of media on which the print service can print.

Medium is used as an argument to **Print** (via **PrintOptions**). As such, it may not contain the item **unknown**.

The various choices of **knownSize** specify standard medium sizes. The specific sizes assigned to each are given in [29], Table 1.

The **otherSize** variant allows the specification of sizes of media other than those contained in **knownSize**. The components **width** and **length** are specified in millimeters. When **otherSize** occurs as an argument to **Print**, it indicates the size of medium on which the client wishes the master to be printed. If **length** is zero, the client is not specifying a length (for example, for a printer that has a roll of paper); at the discretion of the print service, the length may be as long as the document, or some other length chosen by the print service. When an element of **otherSize** is returned to the client as the result of a status request, a length of zero indicates that the print service can produce a page of variable length and with the specified width.

2.2 Freeing storage

Because certain returned arguments require arbitrary storage to be allocated by the **NSPrint** implementation, the interface provides procedures to free that storage once those arguments have been absorbed by the client. The argument storage is considered short term and is allocated out of **Heap.systemZone**.

FreeString: PROCEDURE [string: LONG POINTER TO String];

FreeMedia: PROCEDURE [media: LONG POINTER TO Media];

FreePrinterProperties: PROCEDURE [printerProperties: LONG POINTER TO PrinterProperties];

FreePrinterStatus: PROCEDURE [printerStatus: LONG POINTER TO PrinterStatus];

FreeRequestStatus: PROCEDURE [requestStatus: LONG POINTER TO RequestStatus];

2.3 Print

The **Print** procedure provides the mechanism for transporting the job parameters and the Interpress master to the printer and returns a **RequestID**. The **RequestID** can be used subsequently in calls to **GetPrintRequestStatus** at the same **systemElement**.

PrintAttributes: TYPE = LONG DESCRIPTOR FOR ARRAY OF PrintAttribute;

```
PrintAttribute: TYPE = MACHINE DEPENDENT RECORD [
  var(0): SELECT type(0): PrintAttributeType FROM
  printObjectName = > [printObjectName(1): String ← [NIL, 0, 0]],
  printObjectCreateDate = > [printObjectCreateDate(1): Time ← 0],
  senderName = > [senderName(1): String ← [NIL, 0, 0]],
  ENDCASE];
```

PrintOptions: TYPE = LONG DESCRIPTOR FOR ARRAY OF PrintOption;

```
PrintOption: TYPE = MACHINE DEPENDENT RECORD [
  var(0): SELECT type(0): PrintOptionType FROM
  printObjectSize = > [printObjectSize(1): LONG CARDINAL ← 0],
  recipientName = > [recipientName(1): String ← [NIL, 0]],
  message = > [message(1): String ← [NIL, 0]],
  copyCount = > [copyCount(1): CARDINAL ← 1],
  pagesToPrint = > [pagesToPrint(1): PagesToPrint ← [0, 0]],
  mediumHint = > [mediumHint(1): Medium ← [paper[[knownSize[usLetter]]]]],
  priorityHint = > [priorityHint(1): PriorityHint ← normal],
  releaseKey = > [releaseKey(1): CARDINAL ← LAST[CARDINAL]],
  staple = > [staple(1): BOOLEAN ← FALSE],
  twoSided = > [twoSided(1): BOOLEAN ← FALSE],
  ENDCASE];
```

PrintAttributes provides the basic information identifying the document to be printed. **printObjectName** is the human-sensible name of the master to be printed. **printObjectCreateDate** is the time of creation of the master. **senderName** is the name of the requester of the print service.

PrintOptions provides the parameters needed for further describing the job and indicating how the job is to be printed. Note that some options (i.e., **priorityHint** and **releaseKey**) may not be implemented on all printers. **recipientName** gives the name of the person for whom the printed document is intended and will default to **PrintAttributes[senderName[]]**. **message** is a human-sensible string associated with the specified print request. **copyCount** specifies the number of copies to be printed. **pagesToPrint** specifies the range of pages to be printed. **beginningPageNumber** specifies the first page of the master to be printed; **endingPageNumber** specifies the last page. **pagesToPrint[[1, 177777B]]** will print all pages within a document; the beginning page must be 1. **mediumHint** indicates the medium on which the printed document is to be rendered and cannot have the value **unknown**. This argument acts as a hint in that an implementation may dispose of a request (reject it or use a different medium) as it sees fit, if the specified medium is not

available. **priorityHint** suggests to the print service the execution priority that should be given to the request. **releaseKey** is a datum that must be presented to the print service in order to release a held request. It is hashed password or other text string (see [1]); a value other than **LAST[CARDINAL]** may result in the document being held at the printer until a matching release key is entered. **staple** specifies whether or not the document is to be stapled together. **twoSided** specifies whether or not the document is to be printed on both sides of the paper.

Unsupported or disabled options can incur an **Error[[invalidPrintParameters[]]]**.

```
Print: PROCEDURE [
    master: NSDataStream.Source,
    printAttributes: PrintAttributes,
    printOptions: PrintOptions,
    systemElement: SystemElement]
RETURNS [printRequestID: RequestID];
```

master is the **NSDataStream.Source** handle for the Interpress master. **systemElement** is the host address of the print service. **RequestID** is returned after the successful call is completed. **Print** can incur an **Error[[busy..courier[]]]**.

2.3.1 Print request status

The **GetPrintRequestStatus** procedure provides the mechanism for obtaining status on an outstanding print request via the **printRequestID** provided by the issuing **systemElement**.

```
RequestStatus: TYPE = LONG DESCRIPTOR FOR ARRAY OF RequestStatusComponent;
```

```
RequestStatusComponent: TYPE = MACHINE DEPENDENT RECORD [
    var(0): SELECT type(0): RequestStatusType FROM
        status = > [status(1): Status],
        statusMessage = > [statusMessage(1): String],
    ENDCASE];
```

```
Status: TYPE = MACHINE DEPENDENT {
    pending(0), inProgress, completed, completedWithWarnings, unknown, rejected,
    aborted, canceled, held(8)};
```

```
GetPrintRequestStatus: PROCEDURE [
    printRequestID: RequestID, systemElement: SystemElement]
RETURNS [status: RequestStatus];
```

systemElement is the host address of the printer which originally issued the **RequestID**. Call **FreeRequestStatus** when the status has been absorbed. **GetPrintRequestStatus** can incur an **Error[systemError..courier[]]**.

RequestStatus indicates that processing of the request is in one of the following states: **pending** – has not begun; **inProgress** – is in progress; **completed** – has completed normally; **completedWithWarning** – has completed, but warnings were generated; **unknown** – is unknown to the print service; **rejected** – was not accepted into the marking phase because of errors in the master; **aborted** – was aborted because of problems discovered during

formatting or marking; **canceled** – was queued for printing and subsequently canceled (by human intervention); and **held** – has been held for processing at a later time.

statusMessage is a human-sensible message typically describing some aspect(s) of the status of the print request. In particular, warnings and error messages would be found in this string. The default value is the empty string.

2.3.2 Printer status

The **GetPrinterStatus** procedure provides the mechanism for obtaining status of the printer.

PrinterStatus: TYPE = LONG DESCRIPTOR FOR ARRAY OF **PrinterStatusComponent**;

PrinterStatusComponent: TYPE = MACHINE DEPENDENT RECORD [

```
var(0): SELECT type(0): PrinterStatusType FROM
    spooler => [spooler(1): Spooler],
    formatter => [formatter(1): Formatter],
    printer => [printer(1): Printer],
    media => [media(1): Media],
    ENDCASE];
```

PrinterStatusType: TYPE = MACHINE DEPENDENT {

```
spooler(0), formatter, printer, media(3)};
```

PrinterStatusIndex: TYPE = CARDINAL[0..4];

Spooler: TYPE = MACHINE DEPENDENT {available(0), busy, disabled, full(3)};

Formatter: TYPE = MACHINE DEPENDENT {available(0), busy, disabled(2)};

Printer: TYPE = MACHINE DEPENDENT {

```
available(0), busy, disabled, needsAttention, needsKeyOperator(4)};
```

GetPrinterStatus: PROCEDURE [systemElement: SystemElement]

```
RETURNS [status: PrinterStatus];
```

systemElement is the host address of the printer. Call **FreePrinterStatus** when the status has been absorbed. **GetPrinterStatus** can incur an **Error[systemError..courier[]]**.

The state of the spooling, formatting, and marking phases of printing are indicated by, respectively, spooler, formatter, and printer. Each of these phases can be in any of the following states: available, indicating that the phase is ready to accept input (the spooler can accept masters, the formatter can begin decomposition, or the printer can start marking); busy, indicating that that phase is currently **busy** and cannot accept input, but that this is a transient condition lasting a comparatively short time (a subsequent status request will probably find that phase available); and disabled, indicating that the phase is unavailable and cannot accept input, and that this condition will probably last a long time.

Additional states are defined for some phases:

[spooler[full]] indicates that the spooling queue is full.

[printer[needsAttention]] indicates that the marking engine is not now marking due to some difficulty that human intervention can relieve. The human need not be specially trained to resolve this type of difficulty. [printer[needsKeyOperator]] indicates that the marking engine is not now marking due to some difficulty that human intervention can relieve. In this case, the human should be trained in the marking engine's operation.

media enumerates those media that are available ("on-line") to the print service at the time of the status request. In this context, available indicates that no human intervention is required in order to print on the indicated media.

2.3.3 Printer properties

The **GetPrinterProperties** procedure provides the mechanism for obtaining the current properties of the printer.

PrinterProperties: TYPE = LONG DESCRIPTOR FOR ARRAY OF **PrinterProperty**;

```
PrinterProperty: TYPE = MACHINE DEPENDENT RECORD [
    var(0): SELECT type(0): PrinterPropertyType FROM
        media => [media(1): Media],
        staple => [staple(1): BOOLEAN],
        twoSided => [twoSided(1): BOOLEAN],
    ENDCASE];
```

GetPrinterProperties: PROCEDURE [systemElement: SystemElement]
RETURNS [properties: PrinterProperties];

systemElement is the host address of the printer. Call **FreePrinterProperties** when the status has been absorbed. **GetPrinterProperties** can incur an **Error[systemError..courier[]]**.

media indicates the media that can be made available by the print service. These media need not be immediately available, but the print service must be able to provide them. There is no default value; the print service must return some value of **Media**.

staple indicates the availability of document stapling. The default value is **FALSE**.

twoSided indicates the availability of two-sided printing. The default value is **FALSE**.

2.4 Errors

```
Error: ERROR [why: ErrorRecord];

ErrorRecord: TYPE = RECORD [
    SELECT errorType: ErrorType FROM
        busy, insufficientSpoolSpace, invalidPrintParameters, masterTooLarge,
        mediumUnavailable, serviceUnavailable, spoolingDisabled, spoolingQueueFull,
        systemError, tooManyClients = > [],
        undefinedError = > [undefined: UndefinedProblem],
        transferError = > [transfer: TransferProblem],
        connectionError = > [connection: ConnectionProblem],
        courier = > [courier: Courier.ErrorCode],
    ENDCASE];

ErrorType: TYPE = MACHINE DEPENDENT {
    busy(0), insufficientSpoolSpace, invalidPrintParameters, masterTooLarge,
    mediumUnavailable, serviceUnavailable, spoolingDisabled, spoolingQueueFull,
    systemError, tooManyClients, undefinedError, connectionError, transferError(12),
    courier};

TransferProblem: TYPE = MACHINE DEPENDENT {
    aborted(0), formatIncorrect(2), noRendezvous, wrongDirection(4)};

ConnectionProblem: TYPE = MACHINE DEPENDENT {
    noRoute(0), noResponse, transmissionHardware, transportTimeout,
    tooManyLocalConnections, tooManyRemoteConnections,
    missingCourier, missingProgram, missingProcedure, protocolMismatch,
    parameterInconsistency, invalidMessage, returnTimedOut(12)
    --otherCallProblem(LAST[CARDINAL])--};

UndefinedProblem: TYPE = CARDINAL;
```

The Print Service will return **Error** when a given procedure cannot be completed. The **ErrorRecord** returned by **Error** will describe the specifics of the problem. The following describes the **ErrorTypes** contained in **ErrorRecord**.

busy – the print service is occupied with some activity that prevents it from accepting a print request.

insufficientSpoolSpace – the print service does not have enough space to store the specified master when the print request is made.

invalidPrintParameters – the call on **Print** is made with inconsistent arguments.

masterTooLarge – the master is too large for the print service to accept.

mediumUnavailable – the medium specified in a print request is unavailable.

serviceUnavailable – the print service is unable to process any Printing requests (because of local conditions) and will probably be unavailable for a long period of time. If the condition is only transient, the print service should report **Busy**.

spoolingDisabled – the call on **Print** is made when the print service is not queuing print requests.

spoolingQueueFull – the print service does not have enough space in its spooling queue to accept a new print request.

systemError – the print service has discovered itself in an inconsistent state.

tooManyClients – the print service cannot open another connection to a client. A later call on the print service may succeed.

undefinedError is intended to be used only in the following two circumstances: (1)while testing systems under development before the entire protocol is implemented, and (2) as a last resort when the implementation is on the verge of failure. *This error should never be reported by an operational Printing implementation.* The argument **undefined** returns a implementation-dependent value.

transferError may be reported by the **Print** procedure to indicate that a problem occurred during bulk data transfer. It will further specify either **aborted** (the bulk data transfer was aborted by the sender), **formatIncorrect** (the bulk data received from the source did not have the expected format), **noRendezvous** (the sender never appeared), or **wrongDirection** (the other party wanted to transfer the data in the wrong direction).

connectionError may be reported by the **Print** procedure to indicate that a problem occurred during Bulk Data transfer. It will further specify either **noRoute** (route to the other party could not be found), **noResponse** (other party never answered), **transmissionHardware** (local transmission hardware is inoperable), **transportTimeout** (other party responded but later failed to respond), **tooManyLocalConnections** (additional connection is possible), **tooManyRemoteConnections** (other party rejected the connection attempt), **missingCourier** (other party has no Courier implementation), **missingProgram** (other party does not implement the Bulk Data program), **missingProcedure** (other party does not implement a Bulk Data procedure), **protocolMismatch** (two parties have no Courier version in common), **parameterInconsistency** (protocol violation occurred in parameters), **invalidMessage** (protocol violation occurred in message format), or **returnTimedOut** (procedure call never returned).

courier – procedure call incurred a Courier error. The specific error is returned as a **Courier.ErrorCode**.

NSPrint interface

--*NSPrint.mesa*

--*Mesa interface to Printing protocol.*

--*Copyright (C) Xerox Corporation 1982. All rights reserved.*

DIRECTORY

Courier USING [ErrorCode],
NSDataStream USING [Source],
NSString USING [String],
System USING [NetworkAddress, UniversalID];

NSPrint: DEFINITIONS =

BEGIN

--TYPES

Time: TYPE = LONG CARDINAL;
String: TYPE = NSString.String;

RequestID: TYPE = System.UniversalID;

SystemElement: TYPE = System.NetworkAddress;

Media: TYPE = LONG DESCRIPTOR FOR ARRAY OF Medium;

Medium: TYPE = MACHINE DEPENDENT RECORD [
var(0): SELECT type(0): MediumType FROM
paper => [paper(1): Paper],
ENDCASE];

MediumType: TYPE = MACHINE DEPENDENT {paper(0)};

MediumIndex: TYPE = CARDINAL[0..1];

Paper: TYPE = MACHINE DEPENDENT RECORD [
var(0): SELECT type(0): PaperType FROM
unknown => [], --illegal argument, possible result
knownSize => [knownSize(1): PaperSize],
otherSize => [otherSize(1): PaperDimensions],
ENDCASE];

PaperType: TYPE = MACHINE DEPENDENT {unknown(0), knownSize, otherSize(2)};

PaperIndex: TYPE = CARDINAL[0..3];

PaperSize: TYPE = MACHINE DEPENDENT {

```

dontUse(0) -- the protocol defines this enumeration as starting at 1! --,
usLetter, usLegal, a0, a1, a2, a3, a4, a5, a6, a7, a8, a9,
isoB0, isoB1, isoB2, isoB3, isoB4, isoB5, isoB6, isoB7, isoB8, isoB9, isoB10,
jisB0, jisB1, jisB2, jisB3, jisB4, jisB5, jisB6, jisB7, jisB8, jisB9,
jisB10(34)};

```

```

PaperDimensions: TYPE = MACHINE DEPENDENT RECORD [
    length(0), width(1): CARDINAL]; --units are millimeters

```

```
PrintAttributes: TYPE = LONG DESCRIPTOR FOR ARRAY OF PrintAttribute;
```

```

PrintAttribute: TYPE = MACHINE DEPENDENT RECORD [
    var(0): SELECT type(0): PrintAttributeType FROM
        printObjectName = > [printObjectName(1): String ← [NIL, 0, 0]],
        printObjectCreateDate = > [printObjectCreateDate(1): Time ← 0],
        senderName = > [senderName(1): String ← [NIL, 0, 0]],
    ENDCASE];

```

```

PrintAttributeType: TYPE = MACHINE DEPENDENT {
    printObjectName(0), printObjectCreateDate, senderName(2)};
PrintAttributesIndex: TYPE = CARDINAL[0..3];

```

```
PrintOptions: TYPE = LONG DESCRIPTOR FOR ARRAY OF PrintOption;
```

```

PrintOption: TYPE = MACHINE DEPENDENT RECORD [
    var(0): SELECT type(0): PrintOptionType FROM
        printObjectSize = > [printObjectSize(1): LONG CARDINAL ← 0],
        recipientName = > [recipientName(1): String ← [NIL, 0]],
        message = > [message(1): String ← [NIL, 0]],
        copyCount = > [copyCount(1): CARDINAL ← 1],
        pagesToPrint = > [pagesToPrint(1): PagesToPrint ← [0, 0]],
        mediumHint = > [mediumHint(1): Medium ← [paper[[knownSize[usLetter]]]]],
        priorityHint = > [priorityHint(1): PriorityHint ← normal],
        releaseKey = > [releaseKey(1): CARDINAL ← LAST[CARDINAL]],
        staple = > [staple(1): BOOLEAN ← FALSE],
        twoSided = > [twoSided(1): BOOLEAN ← FALSE],
    ENDCASE];

```

```

PrintOptionType: TYPE = MACHINE DEPENDENT {
    printObjectSize(0), recipientName, message, copyCount, pagesToPrint,
    mediumHint, priorityHint, releaseKey, staple, twoSided(9)};
PrintOptionsIndex: TYPE = CARDINAL[0..10];

```

```

PagesToPrint: TYPE = MACHINE DEPENDENT RECORD [
    beginningPageNumber(0), endingPageNumber(1): CARDINAL];
PriorityHint: TYPE = MACHINE DEPENDENT {low(0), normal, high(2)};

```

```
PrinterProperties: TYPE = LONG DESCRIPTOR FOR ARRAY OF PrinterProperty;
```

```

PrinterProperty: TYPE = MACHINE DEPENDENT RECORD [
    var(0): SELECT type(0): PrinterPropertyType FROM
        media = > [media(1): Media],
        staple = > [staple(1): BOOLEAN],
        twoSided = > [twoSided(1): BOOLEAN],
    ENDCASE];

```

```

PrinterPropertyType: TYPE = MACHINE DEPENDENT {
    media(0), staple, twoSided(2)};

```

```
PrinterPropertiesIndex: TYPE = CARDINAL[0..3];
```

```
PrinterStatus: TYPE = LONG DESCRIPTOR FOR ARRAY OF PrinterStatusComponent;
PrinterStatusComponent: TYPE = MACHINE DEPENDENT RECORD [
    var(0): SELECT type(0): PrinterStatusType FROM
        spooler = > [spooler(1): Spooler],
        formatter = > [formatter(1): Formatter],
        printer = > [printer(1): Printer],
        media = > [media(1): Media],
    ENDCASE];
PrinterStatusType: TYPE = MACHINE DEPENDENT {
    spooler(0), formatter, printer, media(3)};
PrinterStatusIndex: TYPE = CARDINAL[0..4];
Spooler: TYPE = MACHINE DEPENDENT {available(0), busy, disabled, full(3)};
Formatter: TYPE = MACHINE DEPENDENT {available(0), busy, disabled(2)};
Printer: TYPE = MACHINE DEPENDENT {
    available(0), busy, disabled, needsAttention, needsKeyOperator(4)};

RequestStatus: TYPE = LONG DESCRIPTOR FOR ARRAY OF RequestStatusComponent;
RequestStatusComponent: TYPE = MACHINE DEPENDENT RECORD [
    var(0): SELECT type(0): RequestStatusType FROM
        status = > [status(1): Status],
        statusMessage = > [statusMessage(1): String],
    ENDCASE];
RequestStatusType: TYPE = MACHINE DEPENDENT {status(0), statusMessage(1)};
RequestStatusIndex: TYPE = CARDINAL[0..2];
Status: TYPE = MACHINE DEPENDENT {
    pending(0), inProgress, completed, completedWithWarnings, unknown, rejected,
    aborted, canceled, held(8)};

ConnectionProblem: TYPE = MACHINE DEPENDENT {
    noRoute(0), noResponse, transmissionHardware, transportTimeout,
    tooManyLocalConnections, tooManyRemoteConnections,
    missingCourier, missingProgram, missingProcedure, protocolMismatch,
    parameterInconsistency, invalidMessage, returnTimedOut(12)
    --otherCallProblem(LAST[CARDINAL])--};

ErrorType: TYPE = MACHINE DEPENDENT {
    busy(0), insufficientSpoolSpace, invalidPrintParameters, masterTooLarge,
    mediumUnavailable, serviceUnavailable, spoolingDisabled, spoolingQueueFull,
    systemError, tooManyClients, undefinedError, connectionError, transferError(12),
    courier};

TransferProblem: TYPE = MACHINE DEPENDENT {
    aborted(0), formatIncorrect(2), noRendezvous, wrongDirection(4)};

UndefinedProblem: TYPE = CARDINAL;
```

```
--ERRORS
Error: ERROR [why: ErrorRecord];
ErrorRecord: TYPE = RECORD [
    SELECT errorType: ErrorType FROM
        busy, insufficientSpoolSpace, invalidPrintParameters, masterTooLarge,
        mediumUnavailable, serviceUnavailable, spoolingDisabled, spoolingQueueFull,
        systemError, tooManyClients = > [],
        undefinedError = > [undefined: UndefinedProblem],
        transferError = > [transfer: TransferProblem],
        connectionError = > [connection: ConnectionProblem],
        courier = > [courier: Courier.ErrorCode],
    ENDCASE];

--PROCEDURE MODELS
Print: PROCEDURE [
    master: NSDataStream.Source,
    printAttributes: PrintAttributes,
    printOptions: PrintOptions,
    systemElement: SystemElement]
RETURNS [printRequestID: RequestID];

GetPrinterProperties: PROCEDURE [systemElement: SystemElement]
RETURNS [properties: PrinterProperties];

GetPrinterStatus: PROCEDURE [systemElement: SystemElement]
RETURNS [status: PrinterStatus];

GetPrintRequestStatus: PROCEDURE [
    printRequestID: RequestID, systemElement: SystemElement]
RETURNS [status: RequestStatus];

FreeString: PROCEDURE [string: LONG POINTER TO String];
FreeMedia: PROCEDURE [media: LONG POINTER TO Media];
FreePrinterProperties: PROCEDURE [printerProperties: LONG POINTER TO PrinterProperties];
FreePrinterStatus: PROCEDURE [printerStatus: LONG POINTER TO PrinterStatus];
FreeRequestStatus: PROCEDURE [requestStatus: LONG POINTER TO RequestStatus];

END.
```

XEROX



Services 8.0 Programmer's Guide

Print Service 8.0 Interpress (Client) Programmer's Manual

November 1984

PRELIMINARY

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

(

(

(



Table of contents

1	Introduction	1-1
1.1	Overview	1-1
1.2	Notation and terminology	1-2
2	Interface	2-1
2.1	Basic TYPES	2-1
2.2	The header and bodies	2-2
2.3	Declaring fonts	2-3
2.4	Imager Variable operators	2-4
2.5	Current position operators	2-5
2.6	Frame operators	2-5
2.7	Vector operators	2-5
2.8	Body operators	2-6
2.9	Transformation operators	2-6
2.10	Instancing	2-7
2.11	Stack operators	2-8
2.12	Operator operators	2-8
2.13	Mask operators	2-8
2.14	Pixel arrays	2-10
2.15	Sampled masks	2-10
2.16	Support procedures	2-10

Table of contents



Introduction

This document describes **Interpress**, the interface to the Mesa implementation of an aid to producing Interpress masters.

1.1 Overview

This document describes the public types and procedures of the **Interpress** client interface, the implementation for which provides a useful aid in generating *Interpress* masters (per *Interpress Electronic Printing Standard* [18]). It does not provide a syntax or composition service—it is up to the client program to make calls on **Interpress** in the proper sequence. Thus, clients of this interface are expected to be familiar with the *Interpress Electronic Printing Standard* [18] and to understand the syntax and grammar of that standard. **Interpress** does provide syntactically correct arguments and operators within the scope of the procedure calls, but the appropriate sequence of operators and the overall correctness of the master is the responsibility of the client.

Interpress: DEFINITIONS = ...

Interpress provides many “high-level” procedures which represent readily-encoded *Interpress* arguments, operators, and constructs. The calling program is free to intermix calls to these procedures since each call to **Interpress** is atomic, having no side effects besides the token output to the supplied stream. As previously stated, it is the client’s responsibility to make calls on **Interpress** that will result in a correct *Interpress* master.

Although this interface is released as part of the Print Service software, the facilities provided are independent of the *Interpress* language support implemented on a particular print server. The client should consult *Print Service 8.0 (OS 5.0) Interpress Product Description* [27] for specific limitations which should be observed when creating *Interpress* masters for Print Service 8.0 printers.

1.2 Notation and terminology

In this document, the word "Interpress" is used to define both the interface and the standard. To avoid confusing the two in plain text, **Interpress** the interface will appear in boldface while *Interpress* the standard [18] will appear as italicized text.

Frequent reference will be made to *Interpress* operators, bodies, stack, frame, Imager Variables, and other terms defined in *Interpress Electronic Printing Standard* [18]. The operator names appear in this text as SMALL CAPITAL words. The Imager Variables and other *Interpress* language components appear in this text as *italicized* words (i.e., *sequenceIdentifier*) in the sans-serif font.

An *Interpress master* is a file which starts with a valid header and an *Interpress BEGIN* token, ends with an *Interpress END* token, and in other respects obeys the syntax defined in *Interpress Electronic Printing Standard* [18].

Interface

The procedural interface, **Interpress**, provides macro-level procedures similar to those suggested in the *Interpress Electronic Printing Standard* [18] and *Introduction to Interpress* [20]. **Interpress** supplies the Mesa TYPES, constructs, and constants specific to the Interpress language as well as the client procedures. **Interpress** does not provide all the constructs and operators defined by the standard and may provide some which by themselves are not useful or which are not supported by the product print servers. It is anticipated that as product print servers implement more of the language, this interface will expand to more precisely reflect the standard and to provide more facilities to support the creation of valid *Interpress* masters. The PRIVATE procedures and TYPES defined in **Interpress** are not documented here.

The client must provide the file or other stream for the data. (It is not recommended, because of the potentially lengthy creation time, to provide a stream to the printer itself.)

All fonts defined must conform to the Xerox *Printing System Interface Standard* [30]. Character strings must contain character codes which conform to the Xerox *Character Code Standard* [4].

The following text assumes that the client program makes exclusive use of the **Interpress** interface high-level procedures and will not otherwise write onto the furnished stream. Thus statements like "EndMaster must be the last call in the creation of a master" assumes that the client program will not write the *Interpress* END token via the low-level procedures or directly onto the output stream.

2.1 Basic TYPES

ImagerVariable: TYPE = MACHINE DEPENDENT{ -- from Table 4.1 of [18]

--Persistent, restored by DOSAVEALL

DCSpx(0), DCScpy(1),

correctMX(2), correctMY(3),

--Non-Persistent, restored by DOSAVE and DOSAVEALL

T(4),

priorityImportant(5),

mediumXSize(6), mediumYSize(7),

fieldXMin(8), fieldYMin(9),

fieldXMax(10), fieldYMax(11),

showVec(12),

```

color(13),
noImage(14),
strokeWidth(15),
strokeEnd(16),
underlineStart(17),
amplifySpace(18),
correctPass(19), correctShrink(20),
correctTX(21), correctTY(22)
};

```

Defines the Imager Variables for **ISet** and **IGet**.

StrokeEnd: **TYPE** = **MACHINE DEPENDENT**{ -- from §4.8.2 of [18]
square(0), butt(1), round(2)};

Defines the argument for **SetStrokeEnd**.

CharSet: **TYPE** = [0..256];-- the character set index

Defines the character set range in [6] for the argument to **Show**.

Rational: **TYPE** = **RECORD** [num: **LONG INTEGER**, den: **LONG CARDINAL**];

Defines a rational number. **den** equal to zero is illegal.

2.2 The header and bodies

The following procedures define the boundaries of specific parts of the *Interpress skeleton*.

AppendHeader: **PROC** [**sH: Stream.Handle**];

All *Interpress* masters must begin with this call. The prescribed herald, which is used by the *Interpress* printer to determine file validity and version, is written onto the **sH** stream.

BeginMaster: **PROC** [**sH: Stream.Handle**] = **INLINE...**

BeginMaster follows **MakeHerald**, writing the *Interpress* BEGIN token onto the stream. There must be exactly one **BeginMaster** call per master.

EndMaster: **PROC** [**sH: Stream.Handle**] = **INLINE...**

EndMaster must be the last call in the creation of a master, following the completion of all bodies and closure of the current page. It writes the *Interpress* END token onto the stream.

BeginPreamble: **PROC** [**sH: Stream.Handle**];

BeginPreamble should be called once preceding all page body calls. For maximum printer efficiency, all fonts referenced in the document should be declared in the preamble. It writes the "{" token onto the stream.

EndPreamble: **PROC [sH: Stream.Handle];**

EndPreamble should be called once following the completion of the preamble and preceding all page body calls. It writes the "}" token onto the stream.

BeginPage: **PROC [sH: Stream.Handle];**

BeginPage is called at the start of each page. It writes the "{" token onto the stream.

EndPage: **PROC [sH: Stream.Handle];**

EndPage is called at the end of each page. It writes the "}" token onto the stream.

BeginBody: **PROC [sH: Stream.Handle];**

BeginBody is called to begin a new body or context. It writes the "{" token onto the stream.

EndBody: **PROC [sH: Stream.Handle];**

EndBody is called at the end of a body or context. It writes the "}" token onto the stream.

OpenBrace: **PROC [sH: Stream.Handle];**

OpenBrace may be called to begin a new body or context. It writes the "{" token onto the stream.

CloseBrace: **PROC [sH: Stream.Handle];**

CloseBrace is called at the end of a body or context. It writes the "}" token onto the stream.

Note that **BeginPreamble**, **BeginPage**, **BeginBody** and **OpenBrace** all result in the same token being inserted onto the stream—these separate procedures are provided to add semantic clarity to user programs. The same is true for **EndPreamble**, **EndPage**, **EndBody**, and **CloseBrace**.

2.3 Declaring fonts

The following procedures are used to define the fonts used within an *Interpress* master.

DefineFont: **PROC [sH: Stream.Handle,**
fIndex: CARDINAL, font: LONG STRING, scalea, scalee: Rational];

DefineFont is used to declare a specific size of the given **font**. It is a composite procedure which calls **FindFont** and **ScaleAndModifyFont**, and then outputs the **fIndex** **FSET** tokens.

ScaleAndModifyFont: **PROC [sH: Stream.Handle, scalea, scalee: Rational];**

ScaleAndModifyFont results in the output of the arguments and **SCALE** (or **SCALE2**) and **MODIFYFONT** tokens. If **scalea = scalee**, then **SCALE** is output; otherwise **SCALE2** is output.

FindFont: PROC [sH: Stream.Handle, font: LONG STRING];

FindFont outputs the parsed and encoded **font** string followed by a **FINDFONT**.

font is the string which contains the font name and must conform with the font naming convention in the Xerox *Printing System Interface Standard* [30]. Substrings are separated by spaces which are then encoded as *Interpress* vectors of *sequenceIdentifiers*. Thus the call:

FindFont[sH, "Xerox XC1-1-0 Modern-Bold-Italic"];

would result in the following sequence in the master:

<Xerox> <XC1-1-0> <Modern-Bold-Italic> 3 MAKEVEC FINDFONT.

ModifyFont: PROC [sH: Stream.Handle] = INLINE...

ModifyFont outputs the **MODIFYFONT** token.

SetFont: PROC [sH: Stream.Handle, n: CARDINAL] = INLINE...

SetFont outputs the **n SETFONT** sequence.

2.4 Imager Variable operators

The following procedures result in the output of the respective Imager Variable operators, preceded by the specified argument(s).

ISet: PROC [sH: Stream.Handle, i: ImagerVariable] = INLINE ...

ISet outputs the operator which causes the value from the top of the stack to be stored in the **i** variable.

IGet: PROC [sH: Stream.Handle, i: ImagerVariable] = INLINE ...

IGet outputs the operator which causes the value in the **i** variable to be placed on the top of the stack.

SetGray: PROC [sH: Stream.Handle, value: Rational] = INLINE ...

SetStrokeEnd: PROC [sH: Stream.Handle, n: StrokeEnd];

SetStrokeWidth: PROC [sH: Stream.Handle, n: LONG INTEGER];

SetCorrectMeasure: PROC [sH: Stream.Handle, x, y: Rational] = INLINE ...

SetCorrectTolerance: PROC [sH: Stream.Handle, x, y: Rational] = INLINE ...

Space: PROC [sH: Stream.Handle, x: Rational] = INLINE ...

SetAmplifySpace: PROC [sH: Stream.Handle, amp: Rational];

These procedures result in the output of the operators which cause the value(s) supplied to be stored in the respective Imager Variable.

2.5 Current position operators

The following procedures result in the output of the respective *current position* operator tokens, preceded by the specified argument(s), which cause the imaging coordinates to change accordingly. The arguments are in the *master coordinate* system.

SetXY: **PROC [sH: Stream.Handle, x, y: LONG INTEGER] = INLINE ...**

SetXYRel: **PROC [sH: Stream.Handle, x, y: LONG INTEGER] = INLINE ...**

SetXRel: **PROC [sH: Stream.Handle, x: LONG INTEGER] = INLINE ...**

SetYRel: **PROC [sH: Stream.Handle, y: LONG INTEGER] = INLINE ...**

These procedures result in the output of the operators which cause the current position, *DCS_{cpx}*, *DCS_{cpy}*, to be modified by converting the master coordinate(s) supplied using the current transformation *T*.

GetCP: **PROC [sH: Stream.Handle] = INLINE ...**

GetCP outputs the operator which causes the current position (x,y) from *DCS_{cpx}*, *DCS_{cpy}* to be placed on the stack.

2.6 Frame operators

These procedures result in the output of the respective frame operators, preceded by the specified frame index argument.

FSet: **PROC [sH: Stream.Handle, n: INTEGER] = INLINE ...**

FSet outputs the operator which causes the value from the top of the stack to be stored in the nth frame vector.

FGet: **PROC [sH: Stream.Handle, n: INTEGER] = INLINE ...**

FGet outputs the operator which causes the value in the nth frame vector to be placed on the top of the stack.

2.7 Vector operators

The following procedures result in the output of the respective vector operators, preceded by the specified argument(s).

MakeVec: **PROC [sH: Stream.Handle, n: INTEGER] = INLINE ...**

MakeVecLU: **PROC [sH: Stream.Handle, upper, lower: INTEGER] = INLINE ...**

2.8 Body operators

The following procedures result in the output of the respective *body operator* and *primitive body tokens* (i.e., CORRECT, { and }).

BeginCorrectBody: PROC [sH: Stream.Handle] = INLINE ...

BeginCorrectBody writes the CORRECT { tokens onto the stream. The CORRECT operator executes the literals within the {...}, correcting the masks associated with the contained SHOW operator.

EndCorrectBody: PROC [sH: Stream.Handle] = INLINE ...

EndCorrectBody writes the } token onto the stream.

BeginMakeSimpleCO: PROC [sH: Stream.Handle] = INLINE ...

BeginMakeSimpleCO writes the MAKESIMPLECO { tokens onto the stream.

EndMakeSimpleCO: PROC [sH: Stream.Handle] = INLINE ...

EndMakeSimpleCO writes the } token onto the stream.

BeginDoSaveSimpleBody: PROC [sH: Stream.Handle] = INLINE ...

BeginDoSaveSimpleBody writes the DOSAVESIMPLEBODY { tokens onto the stream.

EndDoSaveSimpleBody: PROC [sH: Stream.Handle] = INLINE ...

EndDoSaveSimpleBody writes the } token onto the stream.

2.9 Transformation operators

Interpress provides a linear transformation mechanism for mapping coordinates measured in one coordinate system into coordinates in another system, such as mapping from the *master* coordinate system to the *Interpress* coordinate system. The following procedures output the respective *Interpress* transformation operators.

Translate: PROC [sH: Stream.Handle, x, y: Rational];

Translate outputs the values and operator which creates a transformation on the stack which maps the medium origin from the *Interpress* default of the lower left-hand corner to x, y. Thus, **Translate[sH, [0, 1], [pageheight, 1]]** would create a transformation for mapping the default origin to the upper left-hand corner of the medium (where **pageheight** would be a value in the master coordinate system and would result in x oriented "up").

Rotate: PROC [sH: Stream.Handle, a: INTEGER];

Rotate outputs the value and operator which creates a transformation on the stack which causes the coordinate axes to rotate by the angle a (measured clockwise). Thus, **Rotate[sH, 90]** would create a transformation for rotating the default origin to the upper left-hand

corner of the medium with the x axis oriented along the "long" axis of the medium and the y axis oriented along the "short" axis.

Scale: `PROC[sH: Stream.Handle, s: Rational] = INLINE ...`

Scale outputs the value and operator which creates a transformation on the stack which converts (scales) the coordinates used in the master to those used in *Interpress* (meters). Thus, `Scale[sH, [1, 100000]]` would create a transformation for causing subsequent master coordinates to be interpreted as 10^{-5} meters.

Scale2: `PROC[sH: Stream.Handle, sx, sy: Rational] = INLINE ...`

Scale2 outputs the values and operator which create a transformation on the stack which can cause the axis to shift orientation, or reflect the image about an axis. Thus, `Scale2[sH, [-1, 1], [1, 1]]` would create a transformation for reflecting the image about the y axis. The scaling provided in **Scale** can also be included here; thus `Scale2[sH, [-1, 100000], [1, 100000]]` would create a transformation for causing subsequent master coordinates to be interpreted as 10^{-5} meters and for reflecting the image about the y axis.

Concat: `PROC[sH: Stream.Handle] = INLINE ...`

Concat outputs the operator which causes transformations on the stack to be concatenated, with the results left on the stack.

ConcatT: `PROC[sH: Stream.Handle] = INLINE ...`

ConcatT outputs the operator which causes the transformation on the top of the stack to be concatenated with the Imager Variable T and the results stored back into T.

Move: `PROC[sH: Stream.Handle] = INLINE ...`

Move outputs the operator which modifies the T Imager Variable so that the origin of the coordinate system maps to the current position.

Trans: `PROC[sH: Stream.Handle] = INLINE ...`

Trans outputs the operator which modifies the T Imager Variable so that the origin of the coordinate system maps to the rounded current position.

2.10 Instancing

Show: `PROC[sH: Stream.Handle, s: Environment.Block, cs: CharSet ← 0] = INLINE ...`

Show results in the output of the bytes in **s** as a *sequenceString* followed by the SHOW operator. The bytes in **s** should conform to the character codes and encoding defined in *Character Code Standard* [4] and are preceded by **cs** in accordance with that standard if a non-zero value is supplied.

ShowAndXRel: `PROC[sH: Stream.Handle, s: Environment.Block] = INLINE ...`

ShowAndXRel results in the output of the bytes in **s** as a *sequenceString* followed by the SHOWANDXREL operator. The *first* byte and alternate byte thereafter in **s** should conform to

the character codes and encoding defined in *Character Code Standard* [4]. The second byte and alternate byte thereafter is treated as an argument to SETXREL (modulo 256 and biased by 128).

2.11 Stack operators

The following procedures output operators which manipulate the *Interpress* stack.

Pop: PROC[sH: Stream.Handle] = INLINE ...

Copy: PROC[sH: Stream.Handle, depth: INTEGER] = INLINE ...

Duplicate: PROC[sH: Stream.Handle] = INLINE ...

Roll: PROC[sH: Stream.Handle, depth, moveFirst: INTEGER] = INLINE ...

Exchange: PROC[sH: Stream.Handle] = INLINE ...

Mark: PROC[sH: Stream.Handle, n: INTEGER] = INLINE ...

UnMark: PROC[sH: Stream.Handle, n: INTEGER] = INLINE ...

UnMark0: PROC[sH: Stream.Handle, n: INTEGER] = INLINE ...

Count: PROC[sH: Stream.Handle, n: INTEGER] = INLINE ...

Nop: PROC[sH: Stream.Handle, n: INTEGER] = INLINE ...

2.12 Operator operators

The following procedures output operators which apply to an *Interpress composed operator* on the stack.

Do: PROC[sH: Stream.Handle] = INLINE ...

DoSave: PROC[sH: Stream.Handle] = INLINE ...

DoSaveAll: PROC[sH: Stream.Handle] = INLINE ...

2.13 Mask operators

The following procedures output *Interpress mask* operators useful for creating graphical images. The geometrical shapes created are defined in terms of *segments*, *trajectories* and *outlines*.

MoveTo: PROC[sH: Stream.Handle, x, y: LONG INTEGER] = INLINE ...

MoveTo outputs the coordinates and operator which defines the starting point for a trajectory which is left on the stack.

LineTo: **PROC [sH: Stream.Handle, x, y: LONG INTEGER] = INLINE ...**

LineToX: **PROC [sH: Stream.Handle, x: LONG INTEGER] = INLINE ...**

LineToY: **PROC [sH: Stream.Handle, y: LONG INTEGER] = INLINE ...**

These procedures output the coordinates and operator for an extension point for a trajectory on the stack, the execution of which results in the push of the new trajectory onto the *Interpress* stack.

MakeOutline: **PROC [sH: Stream.Handle, n: LONG INTEGER] = INLINE ...**

MakeOutline outputs the operator which takes **n** trajectories off the stack and creates an outline which is placed back onto the stack.

MaskFill: **PROC [sH: Stream.Handle] = INLINE ...**

MaskFill outputs the operator which takes an outline off the stack and creates a mask, where the outer perimeter of the outline defines the boundary of the mask to be drawn on the image. Also note *Wrap-fill* conventions (Figure 4.5 in *Interpress Electronic Printing Standard*[18]).

MaskStroke: **PROC [sH: Stream.Handle] = INLINE ...**

MaskStroke outputs the operator which takes a single trajectory off the stack and uses it to define the center-line of the stroke whose width is specified by the **strokeWidth** Imager Variable and endpoints defined by **strokeEnd**. The result is laid on the page image.

MaskVector: **PROC [sH: Stream.Handle, x1, y1, x2, y2: LONG INTEGER];**

MaskVector outputs the **x1**, **y1**, **x2** and **y2** arguments followed by the **MASKVECTOR** token (convenience operator) to define a stroke whose trajectory is a single line segment.

MaskRectangle: **PROC [sH: Stream.Handle, x, y, w, h: LONG INTEGER];**

MaskRectangle outputs the **x**, **y**, **w** (width) and **h** (height) arguments followed by the **MASKRECTANGLE** token to define an arbitrary rectangle mask whose sides are parallel to the coordinate axes.

StartUnderline: **PROC [sH: Stream.Handle] = INLINE ...**

StartUnderline outputs the **STARTUNDERLINE** token which causes the current position to be stored in the **underlineStart** Imager Variable.

MaskUnderline: **PROC [sH: Stream.Handle, dy, h: INTEGER] = INLINE ...**

MaskUnderline takes the **underlineStart** Imager Variable as an origin and draws a rectangle of height **h** to the current position (parallel to the x axis) with the top a distance **dy** below the current position.

MaskTrapezoidX: PROC [sH: Stream.Handle, x1, y1, x2, x3, y3, x4: LONG INTEGER];

MaskTrapezoidX outputs the **x1, y1, x2, x3, y3** and **x4** arguments followed by the **MASKTRAPEZOIDX** token to define a trapezoid aligned with the x axis.

MaskTrapezoidY: PROC [sH: Stream.Handle, x1, y1, y2, x3, y3, y4: LONG INTEGER];

MaskTrapezoidY outputs the **x1, y1, y2, x3, y3** and **y4** arguments followed by the **MASKTRAPEZOIDY** token to define a trapezoid aligned with the y axis.

2.14 Pixel arrays

BitmapHandle: TYPE = LONG POINTER TO Bitmap;

Bitmap: TYPE = RECORD [
src: Environment.BitAddress,
srcBpl: INTEGER, -- bits per line
width, height: CARDINAL]; -- in bits

AppendPackedPixelVector: PROC [sH: Stream.Handle, bh: BitmapHandle];

AppendPackedPixelVector outputs the **bh** structure as a *packedPixelArray*.

MakePixelArray: PROC [sH: Stream.Handle];

AppendPackedPixelVector outputs the **MAKEPIXELARRAY** token.

2.15 Sampled masks

MaskPixel: PROC [sH: Stream.Handle] = INLINE ...

MaskPixel outputs the **MASKPIXEL** token.

2.16 Support procedures

The following procedures output encoded Interpress sequences. They are useful for inserting arbitrary values into the Interpress master in conjunction with other operators.

AppendInteger: PROCEDURE[sH: Stream.Handle, n: LONG INTEGER];

AppendShortInteger: PROCEDURE[sH: Stream.Handle, n: INTEGER];

AppendRational: PROCEDURE[sH: Stream.Handle, r: Rational];

These all output the specified data in the appropriate format.

XEROX



Services 8.0 Programmer's Guide

Phone Net Driver Programmer's Manual

November 1984

PRELIMINARY

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



Table of contents

1	Introduction	1-1
2	Interface	2-1
2.1	TYPES	2-1
2.2	Signals and errors	2-1
2.3	Procedures	2-2
3	Usage example	3-1

Table of contents

Introduction

This document describes **PhoneNet**, the interface to the Mesa implementation of the *Synchronous Point-to-Point Protocol* [31]. It describes the public types and procedures of the PhoneNet client interface, the implementation which allows workstations to function as Remote Workstations, connected to an internet via a synchronous point-to-point connection such as a phoneline.

PhoneNet: DEFINITIONS = ...

This interface is released as part of the Internetwork Routing Service software. The implementation for use with Remote Workstations is **RWPhoneNetConfig.bcd**.



Interface

2.1 TYPES

```
EntityClass: TYPE = MACHINE DEPENDENT{ -- from Table 3.2 of [31]
    internetworkRouter (0),
    clusterRouter (1), -- spec says "cluster system element"
    siu (2), -- spec says "interfacing system element"
    remoteHost (3) -- (Remote workstation) spec says "terminal system element"
};
```

Where `remoteHost` should be used by a remote workstation.

```
Negotiation: TYPE = {
    active,
    passive};
```

Where

active indicates that the phone net driver should actively create a connection with the far end.

passive indicates that the phone net driver should await a connection attempt from the far end.

In most cases, active mode should be used. If both system elements that wish to use the Protocol to communicate use the passive mode of operation, the communication attempt will fail. For more information, see §3.5 of [31].

2.2 Signals and errors

```
ClusternetNotInitialized: ERROR; -- ourEntityClass = clusterRouter, but
-- clusternet driver not initialized
```

```
InvalidLineNumber: ERROR; -- referring to an unknown line
```

```
IllegalEntityClass: ERROR; -- (ourEntityClass = siu) not allowed
```

2.3 Procedures

The **PhoneNet** interface includes two procedures. Procedure **Initialize** is used to start **PhoneNet** usage of a specific RS232C channel (see §6.5.3 of [26] for more information). Procedure **Destroy** is used to end **PhoneNet** usage of the RS232C channel. The channel may then be used for other purposes (such as testing).

```
Initialize: PROCEDURE [lineNumber: CARDINAL, channel: RS232C.ChannelHandle,
    commParams: RS232C.CommParamHandle,
    negotiationMode: Negotiation,
    hardwareStatsAvailable: BOOLEAN,
    -- true if the head can report stats. E.g. would be false for CIU
    -- ports since CIU can't report stats
    clientData: LONG UNSPECIFIED ← 0,
    ourEntityClass: EntityClass,
    -- we can't be a siu entity
    clientHostNumber: System.HostNumber ← System.nullHostNumber
    -- the host number that we can give out to those who lack. default means
    -- we don't have one to give out
];
-- REPORTS ClusternetNotInitialized, IllegalEntityClass
```

The RS232C channel must have been created before the **Initialize** procedure is called.

The **hardwareStatsAvailable** parameter is only used for Network Management by the phone net driver.

The **clientHostNumber** parameter is used when communicating with devices that do not possess their own 48-bit host number. If this parameter is used, the host number supplied must be unique in all space. It cannot be the same as the host number of any instance of Pilot, etc. This parameter can be defaulted when communicating with an Internetwork Routing Service or with a Shared Interface Unit (SIU).

```
Destroy: PROCEDURE [lineNumber: CARDINAL];
-- REPORTS InvalidLineNumber
```



Usage example

The following usage example can be used to start up a Remote Workstation that can be connected to an internet by using either an Internetwork Routing Service (IRS) or a Shared Interface Unit (SIU).

BEGIN

-- 1) Create a RS232C channel:

```
channelHandle: RS232C.ChannelHandle;
  -- save the channelHandle for use when destroying the channel
commParams: RS232C.CommParamObject;
lineNumber: CARDINAL = 0; -- 0 is the local port
  -- the lineNumber is also used when destroying the phonenet driver

commParams.duplex ← <half or full>; -- depends on the modem!
commParams.lineType ← bitSynchronous;
commParams.lineSpeed ← <line speed of the modem if known. Use 2400 if the line
speed isn't
  known>;
commParams.accessDetail ← directConn[];
```

```
channelHandle ← RS232C.Create[
  lineNumber, @commParams, preemptAlways, preemptNever];
```

-- 2) initialize the phonenet driver

```
PhoneNet.Initialize[lineNumber, channelHandle, @commParams,
  active, TRUE, 0, remoteHost, System.nullHostNumber];
```

-- and you're done

END;

The following usage example can be used to stop the PhoneNet driver to allow RS232C hardware testing, or for some other purpose.

BEGIN

-- 1) First stop the phonenet driver (the client of the channel)
PhoneNet.Destroy[lineNumber]; -- often takes 10 seconds (Pilot Comm feature)

-- 2) Next destroy the RS232C channel itself
RS232C.Delete[channelHandle];
-- use the channel handle that was returned in the RS232C.Create call

-- done.

END;

XEROX



Services 8.0 Programmer's Guide

External Communication Programmer's Manual

November 1984

PRELIMINARY

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



Table of contents

1	Introduction	1-1
1.1	Organization of the document	1-2
1.2	Definition of terms	1-2
2	Overview	2-1
2.1	Communicating with foreign devices and systems	2-1
2.1.1	The client interface	2-2
2.1.2	Sessions	2-3
2.1.3	Using the transport service during a session	2-5
2.1.4	Terminating the session	2-5
2.2	Relationship to other network service software.	2-6
2.2.1	Client Program	2-6
2.2.2	Client Device Filters	2-7
3	Client interface	3-1
3.1	Creating a foreign device stream	3-1
3.1.1	Session parameters	3-2
3.1.2	Defining the transport	3-2
3.1.3	Connection establishment	3-6
3.2	Data transfer	3-7
3.3	Control transfer	3-8
3.3.1	Classes of generic controls	3-8
3.3.2	List of generic controls	3-9
3.3.3	Stream operations for generic controls	3-11
3.3.4	Applicability of generic controls	3-11
3.4	Altering data transfer timeouts	3-12
3.5	Destroying a foreign device stream	3-12

Table of contents

4	Performance criteria	4-1
4.1	Delay and throughput	4-1
4.2	Security and data protection	4-1
5	Status and exception processing	5-1
5.1	Status via Stream.WaitAttention	5-1
5.2	Data errors via SubSequenceTypes	5-2
5.3	Sources of exception generation	5-2
5.4	Signals and errors	5-2
6	Reliability and maintainability	6-1
7	Multinational requirements	7-1

Appendices

A	RS-232-C communication parameters	A-1
B	Foreign device considerations	B-1
B.1	TTY terminal emulation	B-1
B.1.1	Data transfer considerations	B-1
B.1.2	Use of controls	B-1
B.1.3	Authentication	B-2
B.1.4	Device parameter setting	B-2
B.1.5	Clearinghouse entries	B-2
B.2	IBM 3270 terminal emulation	B-2
B.2.1	Data transfer considerations	B-2
B.2.2	Use of controls	B-3
B.2.3	Authentication	B-4
B.2.4	Device parameter setting	B-4
B.2.5	Clearinghouse entries	B-4



Introduction

This document describes the functionality exported by the *External Communication Service (ECS)* and the virtual terminal circuit capability of the *Gateway Access Protocol (GAP)*. The description is intended primarily for designers and implementors of client programs, i.e., communications applications such as Star TTY emulation. It provides sufficient information to allow those programmers to understand the facilities available and to write procedure calls in the Mesa language to invoke them. In particular, for each function, this document lists the calling sequences and the possible signals which can be generated.

The ECS exports a set of functions that enables a uniform method of communicating between Xerox Network Systems (NS) elements and foreign devices and systems over a variety of communication media. The facility provides a consistent method of establishing a communication channel and of managing the flow of data and communication controls on that channel. However, the content of the data is highly device-dependent and is the responsibility of the client program. See References for device-specific documentation.

The virtual terminal circuit capability of the GAP protocol enables virtual teletype-like sessions between two Xerox Network Systems elements. Virtual terminal circuits are used by the Interactive Terminal Service and the Services executive (for remote system administration) to export user interface functionality to the Internet.

The stub configuration, **GateStubConfig**, provides Mesa procedures that allow access to these functions. This configuration exports the Mesa interface, **GateStream**. **GateStream** is an interface that describes a superset of the functionality described above. This document will describe those portions of the **GateStream** interface that are provided by the stub.

1.1 Organization of the document

Section 2 presents an overview of the facilities available using the stub. Section 3 is the most important section for client programmers; it presents the procedure declarations and data types required to make use of the stub via the **GateStream** interface. Section 4 discusses performance criteria. Section 5 presents the features provided for handling exceptional situations. Not currently available are section 6 (Reliability and maintainability) and section 7 (Multinational requirements).

This manual has two appendices: Appendix A describes the RS-232-C communication parameters, and Appendix B presents some device-specific recommendations and precautions.

1.2 Definition of terms

address of a foreign device

The *address of a foreign device* is a transport-dependent data structure that defines the location and access information for the foreign device within its domain (network). Examples of parts of an address are a phone number in a telephone network.

communicating foreign device

A *communicating foreign device* is a device or system that can communicate with NS Internet system elements using conventions other than the NS Internet Transport Protocols.

connection

A *connection* is a real or virtual association between two correspondents that allows the orderly exchange of data and controls according to some protocol.

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 information on a different physical system.

information translation

Information translation is the altering of information contained in one format by expressing it in another format.

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.

service

A *service* is software that provides a function to clients on the Internet. One example is the External Communication Service that provides terminal emulation capabilities to workstations on the Internet.

session

A *session* is an association between a client and the foreign device, by which the exchange of information is managed.

stub

A *stub* is software that provides access to features exported by services. This document describes a stub that allows access to ECS and virtual terminal circuit functionality.

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 entity and a correspondent at the same level. A transport service may be made up of many levels of transport.



Overview

The purpose of this section is twofold. The first purpose is to give background and to establish a model of communication with foreign devices and systems (see §2.1). This should be of interest to both the programmer and those interested in the scope of the facilities offered. The second purpose is to give an architectural overview of the software which implements those facilities. A description is given of the way the software fits into the general structure of NS Software. Also, the structure and functions of a hypothetical client are outlined in §2.2. This should give client programmers context in which to design higher level client structures.

2.1 Communicating with foreign devices and systems

The Gateway Access Protocol (GAP) defines a set of functionality that provides a uniform method of communicating between NS Internet system elements and foreign devices and systems over a variety of communication media. A *communicating foreign device* is any device or system that does not implement the NS Internet Transport Protocols (defined in *Internet Transport Protocol* [15]). While the foreign device does not communicate via NS Internet Transport conventions, it usually communicates with other devices using a reasonably standard convention.

Before explaining the details of the model in the next sections, a little motivation for that model is in order. The goals for our model are the following:

- 1) Move information over distances

Moving information over distances is the traditional role of a communication facility. A model of transport services must be provided that allows transmission of information across many types of transmission media, both virtual and real, configured in a variety of topologies.

- 2) Support a variety of models of communication

The list of possible user and application communication models is quite long. 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 remote EDP system suggests an emulation communication model.

The communication model is independent of the content of the data. The content of data passed between an NS Internet system element and a foreign device is extremely application dependent. *Information transcription* is supported, 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. Information translation 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 devices and systems

Two complementary strategies are used to resolve the differences in foreign device communication methods. First, where possible, the most standard communication conventions are used. If many foreign devices communicate using convention (protocol) A, then convention A is supported. Our model assumes that no modification of a foreign device is necessary in order to communicate with it. Foreign devices will not be altered to conform to NS Internet communication conventions, rather NS Internet system elements must adapt to the conventions of communication defined by the foreign device.

The second strategy is to isolate those communication characteristics of a foreign device that are device-specific. Of those characteristics, if they can be altered by a local user of the foreign device, then the client is allowed to specify them. Otherwise, they are considered to be constant for that foreign device.

2.1.1 The client interface

The communication model is supported by presenting a flexible client interface. The essence of the interface is the *foreign device stream*. A foreign device stream is a Pilot stream. Therefore, it offers the following features: full duplex transmission of variable-size blocks, methods for passing control information via Pilot Stream Subsequence Types, an out-of-band signaling mechanism via the Pilot Stream Attention feature, and excellent control of block size differences. Also, client stream filters can be prefixed to the foreign device stream, giving a simple method for building higher level interfaces to foreign device communication.

Clients create foreign device streams via the MESA interface **GateStream**. **GateStream** is EXPORTed by a variety of configurations, designed to meet varying client needs. Some of these configurations assume that they reside on the same processor as the communication hardware, while others are able to communicate via the *Gateway Access Protocol (GAP)* with another machine running another configuration which EXPORTs the GAP protocol (i.e., is able to accept remote calls on the **GateStream** interface). The document limits itself to describing one of the Gateway configurations, the Gateway Access Protocol Stub:

GateStubConfig

This configuration, called the stub, exports **GateStream** to allow remote access to ECS functionality. Debugging symbols are located in **GateStubConfig.symbols**. **GateStubConfig** should be explicitly started by including it in the **CONTROL** statement.

2.1.2 Sessions

A *session* is a cooperative association between a stub client and a foreign device. A session 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 device accepts from the client an attempt to start a session. When a client is the passive participant, the session begins when a foreign device tries to start a session with the waiting (listening) client.

To start a session, the following questions must be answered: What is the type of the foreign device? Where is it? What are the unique communication needs of this foreign device? What transport services are to be used? How are the chosen transport services used?

2.1.2.1 Types of foreign devices and systems

Foreign device *types* generally correspond to product names. Each type has a set of static characteristics that describes the behavior of the foreign device. A few of the static characteristics are variations in the use of a protocol (e.g., timeouts), how the foreign device supports setting of its own communication parameters (e.g., set or exchanged remotely during session establishment or set by the foreign device operator), and the codeset (if one only is supported) used by the foreign device. Knowledge of foreign device static characteristics is kept internally, unavailable to the client.

Communication with the following foreign device types is supported: IBM 3278-2 and teletype terminals (both real and virtual). Both BSC and SNA protocols are supported for IBM 3278-2 terminals.

2.1.2.2 Session-oriented communication parameters

The client is responsible for providing session-oriented, device-specific communication information. This information corresponds to the dynamic foreign device communication parameters, i.e., those that can be set by the local operator of a foreign device and/or those that can be set remotely by a correspondent. Examples are parity, character length, and echo source.

For most foreign device types, there is very little remote setting of the operating parameters; rather, the client is responsible for knowing how the foreign device has been set up and for conforming to its settings. For instance, the client must know (and inform the stub) of the parity being used by an asynchronous dial-in host.

2.1.2.3 Transport service

A model of a layered transport service has been chosen. A *transport service* has n levels of virtual transports layered above some physical transmission medium transport. The model is used by both the client, in selecting the transport service, and the service software, in configuring it. 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. As will be discussed below, this includes providing access information and other transport-dependent information. The software 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.

2.1.2.3.1 The transports

A *transport* is a single layer of 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 connection management. A transport can be viewed as communicating with transport entities in the foreign device.

For the simple case there will be two transports, a block transport and a physical transmission medium transport. For example, when an NS Internet system element dials an asynchronous host, there is a teletype transport and an RS-232-C transmission medium transport. The teletype transport can be thought of as logically exchanging data with a teletype transport in the remote host. The RS-232-C channel can be thought of as logically exchanging bits with a similar entity in the remote host. The client must define the appropriate transport parameters, as well as the hierarchical relationship among them.

A *connection* is often required between entities that implement a transport. Connections between transports are analogous to sessions between a stub client and the application entity of the foreign device. The connection is usually made to a logical access point, which is the address or name of the transport entity as defined by the transports that communicate with one another. (Actually, if the access information helps in routing decisions, it is an address. If not, it is probably a name.)

In summary, for each level of transport, the client must give transport-specific access information and other parameters. This information comprises a transport object. The client places the transport objects into an ordered list to define the layering relationship among transports and a transmission medium.

2.1.2.3.2 The physical transmission medium

In the model of a layered transport service, the physical transmission medium is the lowest level communication facility provided. The system element is directly connected to the medium. The transmission medium interface is simply a unique transport—there is only one. It is the lowest level transport, and it corresponds to a physical resource. 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 that is supported on current NS Internet system elements are RS-232-C Controller ports.

For the RS-232-C compatible media, the access information is a telephone number. Dedicated or leased lines require no transmission medium access information.

RS-232-C channel 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 by higher priority use.

The RS-232-C medium-specific information includes line speed, duplex selection, and synchronous/asynchronous selection.

2.1.3 Using the transport service during a session

Once the transport service has been selected and a session has begun, the client can exchange data and control the interaction with the foreign device.

2.1.3.1 Sending/receiving data

The Pilot Stream facility defines the set of data transfer operations available. A timeout can be associated with every operation. Timeouts default to infinity when the foreign device stream is created and can be altered for subsequent operations by a special call.

2.1.3.2 Control during a session

Controls are directives or commands that are exchanged by communicating entities to support smooth, orderly, and reliable information exchange. A foreign device may be capable of exchanging a variety of controls. The controls supported 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, the a set of universal or *generic* controls is defined from/to which most foreign device-specific controls can be mapped. The stub client sends/receives generic controls through the Pilot Stream Subsequence Type and Attention features (see *Pilot Programmer's Manual* [26], §3.1).

2.1.4 Terminating the session

The GateStream interface allows for two kinds of session termination by the client. The client may abruptly terminate the session by deleting the foreign device stream. This method may result in lost data and possibly abnormal operation of more primitive foreign devices. The client may choose to terminate the session gracefully by waiting for some indication of termination from the foreign device side and then terminating the session.

2.2 Relationship to other network service software

It is important to understand the relationship of this software to other kinds of software found in an NS Internet system element. There are two major categories of NS software:

MESA-Pilot MESA is the programming language in which all NS software is written. Every MESA program requires a small amount of system software to support it at runtime; this is included automatically and invoked when the various MESA language features are used. Pilot is the operating system which manages the hardware resources of an NS Internet system element. This is written in MESA and its facilities are explicitly invoked by means of procedure calls in client programs.

Clients Client software performs the product-specific NS functions. These programs are written in MESA and may call upon both Pilot and functionality exported by Services for support. Services software is one class of client software. The External Communication Service, a client of Pilot, supports use of RS-232-C ports for TTY and 3270 emulation. The stub, which provides remote access to ECS and virtual terminal circuit functionality is also a client of Pilot.

The structure of a hypothetical stub client communicating with an ECS is considered below. Two modules are described, the Client Program and Client Device Filters. This is an example only and is given to provide more context in which to design higher levels of software.

2.2.1 Client Program

The Client Program is probably modularized in some way to provide a set of common functions that could be performed for all devices/processes. It utilizes its own set of Client Device Filters and a *foreign device stream* to communicate with the target device.

The ECS runs on the *preferred access system element*, the system element from which the transmission medium interface controller is accessed. Using the stub, the Client Program calls upon the ECS remotely from the preferred access system element to create the foreign device stream. The ECS registers information using the Clearinghouse Service to aid the remote client in locating the correct system element.

There is an instance of a stream for every non-NS Internet device communicating with the local system element. After obtaining the device stream handle, the client communicates with the foreign device through the standard **Stream** interface, as described in the *Pilot Programmer's Manual* [26]. The client can configure a longer stream (pipeline) by prefixing Client Device Filters to the foreign device stream.

2.2.2 Client Device Filters

Client Device Filters will perform *some* of the functions that are necessary to support high-level transactions with a foreign device.

Examples of possible Client Device Filters are:

- Translation of format information

Many transactions with non-NS Internet devices will involve transforming a document from one medium and/or format to another. The format control is usually embedded in the text itself. Since most systems choose different formatting conventions and control characters, format control translation must be done. Some format transformations may require examining the entire document; thus, a filter may not always be appropriate.

- Data and control translation

Data type conversion, such as EBCDIC to ASCII, could be provided in a client filter.

General-purpose code translation, including the ability to discard a code, could be provided. An example would be the redefinition of an attention key; another, the ignoring of DEL (the most likely noise character) on an asynchronous line.

- Encryption/de-encryption of text

Encryption of transmitted text, i.e., non-controls, could be handled at this level to provide end-to-end document encryption. However, encryption of other protocol information or device-specific controls could not be supported at this level.

- Data compression



Client interface

This section describes a subset **GateStream** functionality available from the **GateStubConfig** configuration.

3.1 Creating a foreign device stream

A foreign device stream is created using the **Create** procedure:

```
GateStream.Create: PROCEDURE [
    service: System.NetworkAddress ← System.nullNetworkAddress,
    sessionParameterHandle: SessionParameterHandle,
    transportList: LONG DESCRIPTOR FOR ARRAY OF TransportObject,
    createTimeout: WaitTime ← infiniteTime,
    conversation: Auth.Conversation ← NIL]
    RETURNS [stream: Stream.Handle];
```

service specifies the system element exporting the functionality. Local use is indicated by setting **service** to **System.nullNetworkAddress**. **sessionParameterHandle** specifies a set of device-specific session characteristics (see §3.1.1). **transportList** is an array descriptor describing the layers of the transport (see §3.1.2). **createTimeout** specifies the activation timeout. If **createTimeout** seconds elapse before the stream has been created, the **ERROR Error** with reason **mediumConnectFailed** is generated.

GateStream.WaitTime: TYPE = CARDINAL; -- *in secs*

GateStream.infiniteTime: WaitTime = LAST[CARDINAL];

conversation specifies a handle used to identify the user for network management, accounting, and access control. Specifying **NIL** passes no user identification.

The **ERROR Error** is generated if the **Create** fails. **reason** gives the failure reason. **unimplemented** is the reason if communication with the specified foreign device has not been implemented. If the stream cannot be created due to lack of some system resource, the reason is **tooManyGateStreams**. If the **Create** failed due to inability to authenticate the user (either invalid authentication parameters or Authentication/Clearinghouse Service failure), the reason is **userCannotBeAuthenticated**. If the user is not in the authorized group to use the resource, the reason is **userNotAuthorized**.

3.1.1 Session parameters

A **SessionParameterHandle** pointing to a **SessionParameterObject** describes a set of device-specific session characteristics.

```
GateStream.SessionParameterHandle: TYPE = LONG POINTER TO SessionParameterObject;
```

```
GateStream.SessionParameterObject: TYPE = MACHINE DEPENDENT RECORD [
variantPart(0): SELECT foreignDevice(0): ForeignDevice FROM
```

```
...
ttyHost, tty = > [
    charLength(1): RS232C.CharLength,
    parity(2): RS232C.Parity,
    stopBits(3): RS232C.StopBits,
    frameTimeout(4): CARDINAL], -- milliseconds
ibm3270Host = > NULL,
ENDCASE];
```

The variant tag field of the **SessionParameterObject** specifies the foreign device type. The word **Host** in a device name indicates that the Gateway Software client is communicating with a host *as though it were* the foreign device type named rather than communicating *with* the foreign device named. Thus, **ttyHost** indicates the client is communicating with a host machine *as though it were* a teletype, while **tty** indicates that the client is communicating *with* a teletype.

If the foreign device is a **tty** or **ttyHost**, **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.

If the foreign device is unimplemented, the **ERROR Error** with reason **unimplemented** is generated.

3.1.2 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.

```
GateStream.TransportObject: TYPE = MACHINE DEPENDENT RECORD [
transport(0): SELECT transportType(0): Transport FROM
```

```
rs232c = > [
    commParams(1): LONG POINTER TO RS232C.CommParamObject,
    preemptOthers(3), preemptMe(4): RS232C.ReserveType,
    phoneNumber(5): LONG STRING
    line(7): Line],
```

```
...
teletype = > NULL,
```

```
...
polledBSCTerminal, sdlcTerminal = > [
    hostControllerName(1): LONG STRING,
```

```

terminalAddress(3): TerminalAddress],
service = > [
    id(1): LONG STRING],
ENDCASE];

```

Only one- and two-level transport services are implemented. If the transport service is one-level, then that level must be a **polledBSCTerminal** or **sdlcTerminal TransportObject**. In two-level transports, the first level, the physical transmission medium transport, must be either an **rs232c TransportObject** which supports physical RS-232-C lines or a **service TransportObject** which supports virtual circuits. The second level, the block transport, must be a **teletypeTransportObject**.

If a transport specifies an illegal transport, the **ERROR Error** with reason **illegalTransport** is generated.

3.1.2.1 RS-232-C transport

The **rs232c** variant of **TransportObject** describes a transport layer implementing a transducer that supports physical RS-232-C lines. This transport is a possible bottom layer in two-layer transports.

```

GateStream.TransportObject: TYPE = MACHINE DEPENDENT RECORD [
transport(0): SELECT transportType(0): Transport FROM
.....
rs232c = > [
    commParams(1): LONG POINTER TO RS232C.CommParamObject,
    preemptOthers(3), preemptMe(4): RS232C.ReserveType,
    phoneNumber(5): LONG STRING
    line(7): SELECT reserve(6): ReserveType FROM
        reserveNeeded = > [lineNumber(7): CARDINAL],
        alreadyReserved = > [resource(7): Resource],
    ENDCASE],
.....
ENDCASE];

```

commParams is a pointer to a data structure that holds RS-232-C transmission medium parameters (see Appendix A). The **ERROR Error** with reason **inconsistentParams** is generated if the parameters pointed to by **commParamHandle** are invalid.

RS232C.CommParamObject: TYPE = ... (see Appendix A)

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.

RS232C.ReserveType: TYPE = {preemptNever, preemptAlways, preemptInactive};

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*	Preempt
	Always	Fail	Preempt	Preempt

* Preempt if inactive

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 A B C D E F * # < > =

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-F allow sending the BCD digit codes for 10-15.

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: TYPE = MACHINE DEPENDENT RECORD [
    line(0): SELECT reserve(0): ReserveType FROM
        reserveNeeded = > [lineNumber(1): CARDINAL],
        ...],
    ENDCASE],
```

The variant record **Line** specifies the RS-232-C line number. Only the **reserveNeeded** variant is supported by remotely via the stub. If no RS-232-C hardware exists or if the client selects an invalid line number, the **ERROR Error** with reason **noCommunicationHardware** is generated. If the channel is active and reservation (preemption) fails, **ERROR Error** with reason **transmissionMediumUnavailable** is generated.

3.1.2.2 Service transport

The **service** variant of **TransportObject** describes a transport which defines a virtual terminal circuit. The client is not be communicating over a physical RS-232-C line when using this transport; instead, this transport allows communicating with services that provide a virtual teletype interface to the internet. Examples are the Xerox Development Environment Remote Executive, the Services Remote Executive, and the Interactive Terminal Service. When using this transport, the second layer of the transport is always **teletype**.

```
GateStream.TransportObject: TYPE = MACHINE DEPENDENT RECORD [
transport(0): SELECT transportType(0): Transport FROM
.....
service = > [id: LONG CARDINAL],
.....
ENDCASE];
```

id identifies a particular service on the remote system element. Some standard identifiers are defined in the definitions file **TTYServiceTypes**.

The **ERROR Error** with reason **serviceTooBusy** is generated if the service specified reports it is too busy to accept additional connections. **serviceNotFound** is reported if the service cannot be located on the remote system element.

3.1.2.3 Teletype transport

The **teletype** variant of **TransportObject** describes a transport which allows communication with teletype-like terminals over asynchronous lines. This is the transport used at the second level when the bottom level is either **rs232c** or **service**.

```
GateStream.TransportObject: TYPE = MACHINE DEPENDENT RECORD [
transport(0): SELECT transportType(0): Transport FROM
.....
teletype = > NULL,
.....
ENDCASE];
```

The **ERROR Error** with reason **unimplemented** is generated if the foreign device specified in the session parameters is not a device supported by this transport.

3.1.2.4 PolledBSCTerminal and sdlcTerminal transports

The **polledBSCTerminal** and **sdlcTerminal** variants of **TransportObject** describe an IBM 3278 terminal which communicates with a host via a virtual controller provided by the ECS. When using these transports, no other levels are required since they will have been previously defined by the ECS System Administrator.

```
GateStream.TransportObject: TYPE = MACHINE DEPENDENT RECORD [
transport(0): SELECT transportType(0): Transport FROM
.....
polledBSCTerminal, sdlcTerminal = > [
hostControllerName(1): LONG STRING,
```

```
deviceAddress(3): DeviceAddress],  
....  
ENDCASE];
```

The **hostControllerName** string used to bind the terminal to a previously created virtual controller on the ECS. The field **deviceAddress** specifies the terminal's address. If **unspecifiedTerminalAddress** is specified, the terminal will be assigned any available terminal address on the controller.

```
GateStream.DeviceAddress: TYPE = CARDINAL;  
GateStream.unspecifiedDeviceAddress: TerminalAddress = ...;
```

If the controller specified by **hostControllerName** cannot be found, the **ERROR Error** is generated with a reason of **controllerDoesNotExist**. If the terminal address specified is in use or is invalid, the **ERROR Error** is generated with reasons of **terminalAddressInUse** and **terminalAddressInvalid** respectively.

3.1.3 Connection establishment

Each layer of the transport service may have its own connection establishment conventions. The client has no direct knowledge of these conventions nor 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, i.e., it may either initiate a connection or wait for initiation by the foreign device. Use varies slightly depending on which the client chooses. To give examples of the different possible situations that arise during connection establishment, five cases of RS-232-C connections are considered below:

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. Gateway Software calls the Dialup facility of Pilot to perform the dialing operation. **Create** returns after the circuit has been successfully established. The **ERROR Error**, with reason **mediumConnectFailed** is generated if dialing fails because of no answer, busy phone or activation timeout. If no dialing hardware exists or the dialing hardware is malfunctioning, the **ERROR Error**, with reason **noDialingHardware** or **dialingHardwareProblem** is generated, respectively.

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, **Stream.Delete** is called.

5) Listener using a dialed line

Same as case 4).

3.2 Data transfer

Once a session has been created and connection setup has been successfully completed, the features provided by the Pilot stream interface are available for transferring data with the device. (See *Pilot Programmer's Manual* [26] for additional semantics on stream operations.) The client is responsible for the exact sequence of operations. In general, no throughput improvement is gained by having multiple **Gets** or multiple **Puts** outstanding; rather, it is more efficient to have one operation outstanding with a large input or output block.

If the client is not able to keep up with the rate of arriving data, internal buffering and protocol flow control prevent the loss of data. Likewise, on output, the client may not keep the transport service busy sending. Transport protocol procedures prevent the remote device from complaining during periods of idleness.

Stream.GetBlock: This procedure reads a block of data from the foreign device stream sequence, per the *Pilot Programmer's Manual* [26]. The procedure **Stream.SetInputOptions** controls how **GetBlock** terminates and what SIGNALS it generates, per the *Pilot Programmer's Manual* [26]. Possible SIGNALS are **Stream.LongBlock**, **Stream.TimeOut**, and **Stream.SSTChange**. Possible ERRORS are **GateStream.DeviceOperationAborted**, **Stream.ShortBlock**, and **ABORTED**.

Stream.GetByte: This procedure gets the next byte from the input stream. It is equivalent to a call upon **Stream.GetBlock**, specifying a block containing one byte. Receiving data one byte at a time often makes inefficient use of the transmission bandwidth. Buffering of input is performed to allow for speed mismatch between the device and the consuming client. If a client's **Gets** lag excessively behind arriving data, the flow control the device will be throttled if this is possible. Possible SIGNALS are **Stream.TimeOut**, **Stream.SSTChange**, and **ABORTED**.

Stream.PutBlock: This procedure adds a block of data to the stream sequence, per the *Pilot Programmer's Manual* [26]. The **endPhysicalRecord** parameter is the means by which the size of blocks can be influenced. When **endPhysicalRecord** is **TRUE**, the software guarantees not to transmit any bytes of a subsequent block in the same block as the bytes included in and preceding this block. It has the same effect as a call to **Stream.PutBlock** with **endPhysicalRecord FALSE**, followed by a call to **Stream.SendNow**. Further decomposition of blocks will be performed as required by protocol- and device-specific limitations. Possible SIGNALS are **Stream.TimeOut** and **ABORTED**.

Stream.PutByte: A call on this procedure is equivalent to a call upon **Stream.PutBlock**, specifying a block containing one byte. Bytes will be buffered until either the maximum device frame size is exceeded or the client calls **Stream.SendNow**. Possible SIGNALS are **Stream.TimeOut** and **ABORTED**.

Stream.SendNow: This procedure sends the currently buffered output, per the *Pilot Programmer's Manual* [26]. Possible SIGNALS are **Stream.TimeOut** and **ABORTED**.

3.3 Control transfer

This section describes how the client can control the foreign device 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.

3.3.1 Classes of generic controls

The Pilot stream can be thought of as 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.

3.3.1.1 In-band

An in-band control is sent on the data channel of the stream 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 in the stream.

An in-band control is sent via the **Stream.SetSST** procedure. The control is the **sst** argument. The transition from a **sst** of **GateStream.none** to some other generic value is the event that indicates the arrival of a control in the data sequence. The client must call **Stream.SetSST** twice, once for the control desired and once to reset the **sst** to **none**. It is the client's responsibility that no data be sent between the two calls to **SetSST**.

Note: While making two calls to **SetSST** is bothersome, it preserves the feature of using **ssts** to label data. A client filter may need to use **ssts** for format conversion and/or parsing of stream data. With the method suggested, the client can send generic controls and data labelling **ssts**, with no ambiguity. One way to think of sending in-band controls is that the data to be transmitted is always labeled with a **none** control. A generic control is sent so that it never labels data; rather, the control occurs *between* data blocks and is serialized with respect to it.

Completion of data transfer operations and calls to **SetSST** from separate processes are serialized as much as possible; however, the client must ensure that the calls are initiated in the correct order. The **sst** change takes effect only after all previously initiated **Puts** have been processed. No side effect of the in-band control will cause anything to bypass previously sent data. Completion of the call indicates that all previous data operations and in-band controls have completed, and that the control will be sent as soon as the transport permits.

An in-band control is received as the **sst** result of the **Stream Get** procedures. By definition, the control takes effect at the end of the data. It does not label the data. An in-band control may be returned either with or without data. If the client does not have a **Stream Get** outstanding when a control is received, the control is saved until the next **Get** operation is performed.

3.3.1.2 Out-of-band

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 **Stream.SendAttention** procedure is used to send out-of-band controls. The control is the **byte** argument. Completion of the call indicates that the control will be sent as soon as the transport permits. All transports do not expedite generic out-of-band controls equally.

The **Stream.WaitAttention** procedure is used to receive out-of-band controls. The control is the result. A separate process is usually delegated by the client to wait for out-of-band controls.

Note: **Stream.WaitAttention** is also used to receive Gateway status **ssts** (see §5.1).

3.3.1.3 Out-of-band with mark

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, for example, that the aborting condition is synchronized with respect to the sender of the abort. In some cases, the out-of-band control and the in-band mark are generated by opposite sides of the stream.

3.3.2 List of generic controls

AbortGetTransaction

[Out-of-band-w/mark]

GateStream.abortGetTransaction: Stream.SubSequenceType = ...;

Immediately stop the transaction being received and resume at the next transaction boundary, as designated by the in-band mark **AbortMark**. The out-of-band portion of this control may be generated by either side of the stream. However, since **AbortGetTransaction** always aborts the *client's* **Get**, the in-band mark is not generated by the client, but by the stub.

AbortMark	[In-band]
GateStream.endOfTransaction: Stream.SubSequenceType = ...;	
Marks a transaction boundary in conjunction with an AbortGetTransaction or an AbortPutTransaction control.	
AbortPutTransaction	[Out-of-band-w/mark]
GateStream.abortPutTransaction: Stream.SubSequenceType = ...;	
Stop the current outgoing transaction immediately and resume at the next transaction boundary, as designated by the in-band mark AbortMark . The out-of-band portion of this control may be generated by either side of the stream. However, since AbortPutTransaction always aborts the <i>client's Put</i> , the in-band mark must be generated by the client.	
Interrupt	[Out-of-band]
GateStream.interrupt: Stream.SubSequenceType = ...;	
Temporarily halt both processing and output as quickly as possible.	
None	[In-band]
GateStream.none: Stream.SubSequenceType = ...;	
Used to return the stream to normal, indicating data is to follow.	
UnchainedCommand	[3270 emulation only] [In-band]
GateStream.unchained3270: Stream.SubSequenceType = ...;	
The following data is an unchained IBM 3270 command from the host application program. The end of the data is marked by endRecord .	
ReadModifiedData	[3270 emulation only] [In-band]
GateStream.readModified3270: Stream.SubSequenceType = ...;	
The following data is read modified data from an IBM 3270 terminal. The end of the data is marked by endRecord .	
SSCPData	[3270 emulation only] [In-band]
GateStream.sscpData: Stream.SubSequenceType = ...;	
The following data is SSCP data in character-oriented format. The end of the data is marked by endRecord . The control is only used when the transport is sdlcTerminal .	

3.3.3 Stream operations for generic controls

The following stream operations support sending and receiving of generic controls:

Stream.SetSST: Generic control functions are passed by the client as subsequence types. **SetSST** is used to generate in-band controls or the in-band mark for out-of-band with mark controls.

Stream.SetInputOptions: To be notified of the receipt of generic controls via a SIGNAL, the client must indicate that an **SSTChange** SIGNAL is to be generated whenever a control sequence corresponding to a generic control is encountered on input.

If so designated, arrival of a control that maps into a generic control will result in the generation of the following SIGNAL:

Stream.SSTChange: SIGNAL [sst: Stream.SubSequenceType, nextIndex: CARDINAL];

The client must take care to synchronize receipt of the SIGNAL and the receipt of stream blocks. For ease of synchronization it is better to receive the subsequence type as a result of a **Stream Get** procedure.

Stream.SendAttention: An out-of-band control is sent as the attention byte. **SendAttention** is also used to send the out-of-band portion of an out-of-band with mark control.

Stream.WaitAttention: Out-of-band controls are received as the result of this procedure.

Note: **Stream.WaitAttention** is also used to receive Gateway status SSTs (see §5.1).

3.3.4 Applicability of generic controls

Some devices do not support transmission and/or receipt of some of the generic controls. The table below indicates when a control is not applicable (NA), supported for sending/receiving (SR), supported for sending only (S), receiving only (R), and not implemented (NI).

DEVICES

ttyHost ibm3270

CONTROLS

Interrupt	SR ¹	NA
Abort Get Transaction	NI	R
Abort Put Transaction	NI	S
Unchained Command	NA	R
Read Modified Data	NA	S
SSCP Data	NA	SR

¹ Send only for Xerox 873. The Xerox 8000 can both send and receive an Interrupt on the local port.

3.4 Altering data transfer timeouts

Altering the timeout for subsequent stream data transfer operations is accomplished using the field `setTimeout` in the `Stream` object. The default timeout set for data transfer operations is infinite time; thus, timeouts are initially disabled.

3.5 Destroying a foreign device stream

A stream is deleted using `Stream.Delete`. This call immediately terminates a session with a foreign device. No efforts to prevent data loss will be made nor will the foreign device be notified via protocol exchange. Deletion releases all resources associated with the session, including the transmission medium connection.

No operations may be pending on the stream when Stream.Delete is called. To aid the client in aborting pending stream operations prior to a `Stream.Delete`, the client can use the Pilot Process aborting mechanism (`Process.Abort`) to force waiting processes to return in a timely manner. Aborted processes will raise the signal `ABORTED`.

Most stream procedures also raise the signal `ABORTED` if the remote side terminates the connection. `Stream.Get` calls return `endOfStream` if the remote host terminates the connection. Either of these should be used as an indication that the stream should be terminated.



Performance criteria

4.1 Delay and throughput

It is very difficult to characterize performance. This is primarily due to the fact that the configuration of a foreign device stream varies so greatly. Not only do devices communicate using different line speeds, but protocol overhead will vary depending on the packaging of data.

Increasing throughput results in lowered transmission medium cost and better utilization of the system element. Here are some general guidelines for clients that will lead to maximum throughput:

- Use the largest buffers possible for data transfer operations.
- Attempting to match client buffer sizes with foreign device medium block sizes may seem appropriate, but it is discouraged. The extra protocol overhead and client context switching incurred using small blocks offsets the advantage of eliminating block fragmentation.
- If possible, set up the foreign device to use the local storage medium with the highest throughput. For instance, reading to or writing from a floppy disk is better than a magnetic card. Never transmit to paper.

4.2 Security and data protection

Authentication of remote foreign devices is provided to the extent that protocols allow such authentication. The ECS also provides access control lists for each teletype emulation line or IBM 3270 terminal.

No other security precautions are implemented by the ECS. Data passed is encapsulated according to the conventions of standard protocols. Certain bit patterns are not allowed within the frames of some protocols. If encryption is used on the contents of a data frame while using an all text protocol, the encrypted text must not contain any characters reserved for protocol framing.

Protocol framing is never encrypted.

Status and exception processing

5.1 Status via Stream.WaitAttention

In addition to generic controls from the foreign device, the client may also receive status information from `Stream.WaitAttention`. The following `Stream.SubSequenceTypes` represent status to the client:

mediumDown

The transmission medium has gone from an *up* to a *down* condition.

mediumUp

The transmission medium has gone from a *down* to an *up* condition.

noGetData

The foreign device is sending data for which there is no corresponding client `Stream.Get`. If the foreign device is `ttyHost`, all internal buffers are full and any additional data received before the next `Stream.Get` is lost. For other foreign devices, the ECS will continue to receive data from the foreign device until all internal buffers are full and then will not accept new data from the foreign device until a `Stream.Get` is done by the client. If the client is not prepared to `Get` the data, the stream should be deleted as there is no recovery; otherwise, a `Stream.Get` should be issued.

configurationMismatch3270 [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 [3270 emulation only]

The 3270 host has not polled our controller for at least 2 minutes.

hostPolling3270 [3270 emulation only]

The 3270 host, which had not been polling our controller, is now polling our controller.

5.2 Data errors via SubSequenceTypes

For certain devices such as teletypes, data errors cannot be retried by the ECS and must be passed to the client. This is done by using the two `Stream.SubSequenceTypes` described below. In each case, the character on which the error occurred is the final character in the block returned to the client:

garbledReceiveData

The final character in the block was received with a framing error.

parityError

The final character in the block was received with a parity error.

5.3 Sources of exception generation

During the establishment and lifetime of a foreign device stream, there are many sources of exception generation. Fortunately, many of the errors that occur can be generalized and result in identical interpretation by the client. The guideline used in determining `SIGNALS` to raise is that the client must have enough information to inform a user that some corrective measure must be made to the device and/or the communication equipment.

5.4 Signals and errors

The following `SIGNALS` and `ERRORS` are generated by the stub. Client recovery actions accompany each exception condition.

`GateStream.Error: ERROR [reason: GateStream.ErrorReason];`

`reason` is one of the following `ErrorReasons`:

badAddressFormat

The phone number specified has an invalid format. No recovery; client bug.

bugInGAPCode

A non-recoverable protocol error occurred during the `Delete` call. No recovery.

dialingHardwareProblem

The dialing hardware is malfunctioning. No recovery; fix hardware or try manual dial.

gapCommunicationError

The system element specified in the parameter `service` could not be contacted. Possible client error (wrong server selected) or try again later (server is down).

gapNotExported

The Gateway Access Protocol is not exported at this time by the system element specified in the parameter `service`. Possible client error (wrong server selected) or try again later (service not currently running on the server).

illegalTransport

The transport specified is not supported. No recovery; client problem.

inconsistentParams

The parameters pointed to by **commParams** were rejected by the RS-232-C channel as unimplemented. No recovery.

mediumConnectFailed

The ECS is unable to connect to the foreign device. For example, when auto-dialing, this indicates the remote phone number was busy, did not answer, or the activation timeout occurred. Try again later.

noCommunicationHardware

No RS-232-C hardware exists or the RS-232-C line specified is invalid. No recovery; choose another server.

noDialingHardware

No auto-dialing hardware exists. No recovery; try manual dial or chooses another server.

tooManyGateStreams

One of the resources needed to make the connection is exhausted. Try again later.

transmissionMediumUnavailable

The transmission medium is currently in use by someone else. Try later or try a higher **preemptOthers** priority.

unimplemented

- 1) The foreign device specified is not implemented.
- 2) The procedure called is not implemented. No recovery; client problem.

controllerDoesNotExist [3270 emulation only]

The controller host name specified during an IBM 3270 terminal creation does not match one of the virtual controllers available on the ECS. Make sure a controller has been created or choose another name and try again.

deviceAddressInUse [3270 emulation only]

During an IBM 3270 terminal creation, the terminal address specified or all terminals are in use at this time. Try again later when terminal is available.

deviceAddressInvalid [3270 emulation only]

The terminal address specified during an IBM 3270 terminal creation is not in the range supported by the controller. Probable client error. Choose another terminal address which is in the correct range and try again.

serviceTooBusy [service transport only]

The remote service rejected the connection, probably because there were too many other users. Try again later when the service is no so busy.

userNotAuthorized

The client is not in the authorized group. Try another port or terminal which allows access by your group or have access list changed to allow access to the port or terminal you wish to use.

userNotAuthenticated

The client did not specify authentication parameters, (**conversation = NIL**), the authentication parameters were invalid, or an Authentication/Clearinghouse failure prevents the verification of the authentication parameters.

serviceNotFound [service transport only]

The service type identified by service is not available on this system element. Possible client error (the system element specified is not of the correct type) or try again later (the service is not running at this time).

networkTransmissionMediumDown
networkTransmissionMediumUnavailable
networkTransmissionMediumNotReady
networkNoAnswerOrBusy
noRouterToGAPService
gapServiceNotResponding

These are mappings of Courier errors that can occur when the connection to the remote system element is being established. Possible client error (incorrect service address specified) or try again later (if connection is temporarily down).

courierProtocolMismatch

The server does not support the compatible versions of Courier. Possible client error (install the correct version of the software) or try another server (which runs the correct version of the software).

gapVersionMismatch [service transport only]

The server does not support the versions of the protocol that you wish to use. Possible client error (install the correct version of the software) or try another server (which runs the correct version of the software).

The following Stream SIGNALS and ERRORS are generated. See *Pilot Programmer's Manual* [26] for semantics on these SIGNALS and ERRORS:

Stream.SSTChange: SIGNAL [sst: Stream.SubSequenceType, nextIndex: CARDINAL];

Stream.TimeOut: SIGNAL [nextIndex: CARDINAL];

Stream.LongBlock: SIGNAL [nextIndex: CARDINAL];

Stream.ShortBlock: ERROR;

Stream.EndOfStream: ERROR;

ABORTED: SIGNAL;

Reliability and maintainability

[TBD]

Multinational requirements

[TBD]



Appendix A

RS-232-C communication parameters

The **rs232c** variant of a **GateStream.TransportObject** contains **commParams** as a field. **commParams** is a pointer to a communication medium description, of type **RS232C.CommParamObject**, a record that defines the settings for the communication equipment. 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. See *Pilot Programmer's Manual* [26] for further information.

```
RS232C.CommParamObject: TYPE = RECORD [
    duplex: RS232C.Duplexity,
    lineType: RS232C.LineType,
    lineSpeed: RS232C.LineSpeed,
    accessDetail: SELECT netAccess: RS232C.NetAccess FROM
        directConn = > NULL,
        dialConn = > [
            dialMode: RS232C.DialMode,
            dialerNumber: CARDINAL,
            retryCount: RS232C.RetryCount],
    ENDCASE
];
RS232C.Duplexity: TYPE = {full, half}; --hardware (modem)
RS232C.NetAccess: TYPE = {directConn, dialConn};
RS232C.DialMode: TYPE = {manual, auto};
```

A

RS-232-C communication parameters

Appendix B Foreign device considerations

The ECS attempts to provide a uniform interface for communicating with a wide variety of foreign devices. While the interface may be uniform, there are aspects of it that do not apply to some foreign devices or that do not have obvious mappings into the unique operations of a particular device. In some cases, the ECS client must translate the operations that apply to a foreign device into the more generic operations provided by the Gateway Software interface. This appendix lists known device-specific peculiarities and discusses how to use Gateway Software features to handle them.

B.1 TTY terminal emulation

B.1.1 Data transfer considerations

Communication in TTY terminal emulation mode is assumed to be in an interactive mode. That is, the user is sending and receiving data without the model of transferring a large amount of data such as a document in a single direction.

When receiving data in TTY terminal emulation mode, the ECS will fill the client's buffer with as much data as is available from the remote device at the time the **Get** is done. If no data exists, the ECS will wait until some arrives. Thus, it is possible for **Gets** to return with only partially filled buffers. This should be considered normal and should not be treated as an error.

B.1.2 Use of controls

TTY terminal emulation supports only the **Interrupt** control.

Interrupt

The **Interrupt** control is used to send a **BREAK**. If a **BREAK** is received, an **Interrupt** control is generated.

B.1.3 Authentication

The ECS provides access control on a per physical port basis. If unlimited access is specified, the client need not supply authentication information (**conversation = NIL**). However, it is recommended that this information always be provided for network management and future accounting uses.

The ECS will accept either strong or weak authentication credentials. When generating strong authentication credentials, the remote name is the RS-232-C Port Clearinghouse entry that describes the RS-232-C being used.

B.1.4 Device parameter setting

The remote host may be set to send asynchronous data at speeds from 50 to 19200 baud, with no, even, or odd parity, and with data length of 5 to 8 bits.

B.1.5 Clearinghouse entries

The ECS registers all RS-232-C ports available for teletype emulation. The format of these Clearinghouse entries is defined in the file **CHEntries** and **CHPIDs**.

B.2 IBM 3270 terminal emulation**B.2.1 Data transfer considerations**

Communication in IBM 3270 terminal emulation mode is assumed to be in an interactive mode. That is, the user is sending and receiving data without the model of transferring a large amount of data such as a document in a single direction. The terminal emulated is an IBM 3278-2.

Data transfer varies depending on whether **polledBSCTerminal** or **sdlcTerminal** has been specified as the top transport layer.

If **polledBSCTerminal** is specified:

- 1) The virtual controller may be one that communicates with the foreign device (IBM host) using either the BSC or SNA protocols. Allowing a transport of **polledBSCTerminal** when using a virtual controller that communicates using SNA is provided for backward-compatibility with workstations that do not understand SNA character-oriented data.
- 2) The client receives IBM 3270 data stream commands from the ECS. If the host uses SNA protocols, the ECS converts character-oriented data on the SSCP-LU session into equivalent field-oriented data stream commands. Each command is preceded by the in-band mark **UnchainedCommand**. The end of the command is marked by **endRecord**. Each command is treated as a transaction; thus, an **AbortGetTransaction** control aborts one command. The data returned to the client begins with the **ESC** character of the command.

- 3) The client sends IBM 3270 terminal read modified data. The data is preceded by the in-band mark **ReadModifiedData**. The end of the data is marked by **endRecord**. If the host uses SNA protocols, the ECS converts field-oriented read modified data into character-oriented data when sending on the SSCP-LU session.

If **sdlcTerminal** is specified:

- 1) The virtual controller must be one that communicates with the foreign device (IBM host) using the SNA protocol.
- 2) The client receives data on the LU-LU session as IBM 3270 data stream commands. Each command is preceded by the in-band mark **UnchainedCommand**. The end of the command is marked by **endRecord**. A command is treated as a transaction; thus, an **AbortGetTransaction** control aborts one command. The data returned to the client begins with the **ESC** character of the command.
- 3) The client receives data on the SSCP-LU session as character-oriented data. The data is preceded by the in-band mark **SSCPData**. The end of the character-oriented data is marked by **endRecord**.
- 4) The client sends IBM 3270 read modified data on the LU-LU session. The data type is preceded by the in-band mark **ReadModifiedData**. The end of the read modified data is marked by **endRecord**.
- 5) The client sends character-oriented data on the SSCP-LU session. The data type is preceded by the in-band mark **SSCPData**. The end of the data is marked by **endRecord**.

B.2.2 Use of controls

UnchainedCommand [Gateway Software to Client]

The data following is an unchained IBM 3270 command. An **AbortGetTransaction** aborts the entire command transaction.

ReadModifiedData [Client to Gateway Software]

The data following is read modified data from a terminal.

SSCPData [Client to/from Gateway Software]

The data following is SSCP data in character-oriented format. This control is only used when the transport is **sdlcTerminal**.

B.2.3 Authentication

The ECS provides access control on a per device (terminal/print) basis. If unlimited access is specified, the client need not provide authentication information (**conversation = NIL**). However, it is recommended that this information always be provided for network management and possible future accounting purposes.

The ECS will accept either strong or weak authentication credentials. When generating strong authentication credentials, the remote name is the IBM 3270 Host Clearinghouse entry describing the virtual controller.

B.2.4 Device parameter setting

The remote host may be set to send synchronous data at speeds from 50 to 9600 baud, either half- or full-duplex. The number of devices **SYSGENED** into the host system should be equal to the number specified by the ECS System Administrator.

B.2.5 Clearinghouse entries

To aid the stub client in locating virtual 3270 controllers, each ECS registers its virtual controllers in the Clearinghouse. The format of the entries can be found in **CHEntries** and **CHPIDs**.

When naming a particular virtual controller, the **hostControllerName** is formed by concatenating the following substrings into a single string:

- 1) the *local name* of the port (from the Clearinghouse Service IBM 3270 Host entry),
- 2) a pound sign (#),
- 3) the controller number expressed in octal (from the Clearinghouse Service IBM3270Host entry),
- 4) a capital B (B).

For example, to specify a virtual controller with an IBM 3270 Host entry name of **PaloAltoHost:OSBU North:Xerox** and a controller number of 5, the name passed as the **hostControllerName** would be:

PaloAltoHost#5B

XEROX



Services 8.0 Programmer's Guide

Filing Programmer's Manual

November 1984

PRELIMINARY

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

(

(

(



Table of contents

1	Introduction	1-1
1.1	Filing and the services architecture	1-1
1.2	Using Filing	1-2
1.3	Organization of the Filing Programmer's Manual	1-2
2	Overview	2-1
2.1	Clients, the file system, and file service	2-1
2.2	Users, authentication, and sessions	2-1
2.3	Volumes and services	2-2
2.4	Files, content, and attributes	2-3
2.5	Directories	2-4
2.6	Handles and controls	2-4
2.7	Creating, deleting, and accessing files	2-5
2.8	Enumerating and locating files in directories	2-5
2.9	Bulk data transfer	2-5
2.10	Serializing and deserializing files	2-6
3	File/session operations	3-1
3.1	Sessions	3-1
3.1.1	Session handles	3-1
3.1.2	Establishing a session	3-2
3.1.3	Logoff	3-3
3.1.4	Probe	3-3
3.1.5	The default session	3-4
3.1.6	The default service	3-4
3.2	Naming, opening, and closing files	3-5
3.2.1	Naming	3-5
3.2.2	Opening	3-6
3.2.3	Simpler forms of Open	3-8
3.2.4	Close	3-8

Table of contents

3	File/session operations (CONTINUED)	
3.3	Handles and controls	3-9
3.3.1	Locks	3-9
3.3.2	Timeouts	3-10
3.3.3	Access	3-11
3.3.4	Retrieving and changing controls	3-11
3.4	Creating and deleting files	3-13
3.4.1	Create	3-13
3.4.2	Deleting files	3-13
3.5	Finding and listing files within directories	3-14
3.5.1	Scopes	3-14
3.5.2	Locating files	3-18
3.5.3	Listing files	3-18
3.6	Copying files	3-19
3.7	Moving files	3-20
3.8	Bulk data transfer operations	3-21
3.8.1	Single file operations	3-22
3.8.2	Subtree operations, serialized files	3-24
3.9	Macro operations	3-26
3.9.1	Child operations	3-26
3.9.2	Pathname operations	3-27
3.10	Errors	3-29
3.10.1	Access errors	3-30
3.10.2	Argument errors	3-31
3.10.3	Authentication errors	3-32
3.10.4	Clearinghouse errors	3-33
3.10.5	Connection errors	3-34
3.10.6	Handle errors	3-35
3.10.7	Insertion errors	3-36
3.10.8	Service errors	3-37
3.10.9	Range errors	3-37
3.10.10	Session errors	3-38
3.10.11	Space errors	3-38
3.10.12	Transfer errors	3-38
3.10.13	Undefined errors	3-39
4	Segment/content operations	4-1
4.1	Finding and listing segments of a file	4-1
4.2	Adding, deleting, and moving segments	4-2
4.3	Accessing and modifying segment sizes	4-3
4.4	Mapping	4-4
4.5	Errors	4-6

5	Positionable stream operations	5-1
5.1	Creating the file stream	5-1
5.2	Getting and setting the length of the stream	5-2
5.3	Miscellaneous operations	5-3
6	Attributes	6-1
6.1	Attribute model	6-1
6.2	Classes of attributes	6-1
6.2.1	Interpreted vs. uninterpreted	6-1
6.2.2	Environment vs. data	6-2
6.2.3	Primary vs. derived	6-2
6.3	Attribute descriptions	6-3
6.3.1	Identity attributes	6-3
6.3.2	File attributes	6-5
6.3.3	Activity attributes	6-8
6.3.4	Size attributes	6-10
6.3.5	Access attributes	6-10
6.3.6	Directory attributes	6-12
6.3.7	Extended attributes	6-15
6.4	Assigned types	6-15
6.4.1	Type ranges	6-16
6.4.2	Defined types	6-16
6.5	Retrieving attribute values	6-18
6.6	Modifying attribute values	6-20
6.6.1	ChangeAttributes	6-20
6.6.2	UnifyAccessLists	6-21
6.7	Manipulating attribute values	6-21
6.7.1	Copying/freeing	6-23
6.7.2	Encoding/decoding	6-24
6.8	Summary of attribute behaviors	6-25
7	Pathname parsing operations	7-1
7.1	Pathname separators and other special characters	7-1
7.1.1	Service name separators	7-1
7.1.2	Pathname component separators	7-1
7.1.3	Characters for version number constants	7-2
7.1.4	Wildcard characters	7-2
7.1.5	The escape character	7-3
7.2	The default domain and organization	7-3
7.3	Parsing qualified pathnames	7-3
7.4	Appending VPNs to Strings	7-5
7.5	Allocation and deallocation of VPNs	7-6
7.5.1	Copying VPNs	7-6
7.5.2	Freeing VPNs	7-6
7.6	Errors	7-7

Table of contents

8	System configuration and administration	8-1
8.1	Global file system variables	8-1
8.1.1	Protocol versions	8-1
8.1.2	Membership status	8-2
8.1.3	Miscellaneous operations	8-3
8.1.4	Errors	8-4
8.2	Volumes	8-4
8.2.1	Opening and closing volumes	8-5
8.2.2	The system volume	8-6
8.2.3	Initializing volumes	8-6
8.2.4	Volume attributes	8-7
8.2.5	Volume name	8-7
8.2.6	Volume scavenging	8-9
8.2.7	Errors	8-15

Tables

6.1	Add, Delete, SetSizeInBytes, SetSizeInPages	6-26
6.2	ChangeAttributes	6-27
6.3	Copy	6-28
6.4	CopyIn	6-29
6.5	CopyOut, MakeWritable, Move	6-30
6.6	Create	6-31
6.7	Create (NSFileStream)	6-32
6.8	Delete	6-33
6.9	Deserialize	6-34
6.10	Map	6-35
6.11	Move	6-36
6.12	Open	6-37
6.13	Replace	6-38
6.14	Retrieve	6-39
6.15	Serialize	6-40
6.16	SetLength	6-41
6.17	Store	6-42
6.18	UnifyAccessLists	6-43



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.)

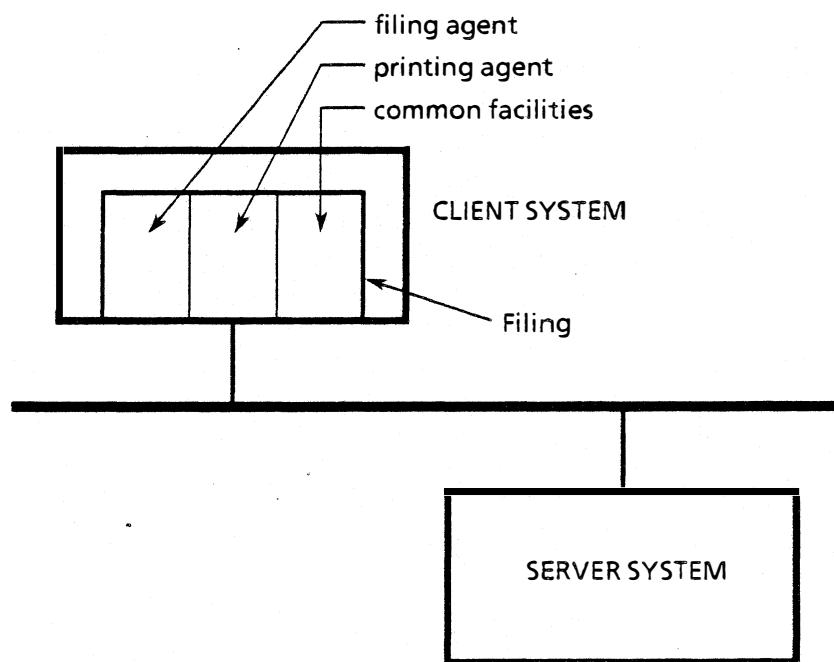


Figure 1.1 Client and server

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.



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.



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[session|invalid]]]** 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 **ANDed** 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 ← null Session];
```

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** [$filter_1, filter_2, \dots, filter_n$] is satisfied only if all of $filter_1, filter_2, \dots, filter_n$ are satisfied.

A filter whose value is **or** [$filter_1, filter_2, \dots, filter_n$] is satisfied when at least one of $filter_1, filter_2, \dots, filter_n$ 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 <code>NSString.CompareStrings</code> . 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 <code>longCardinal</code> .

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 \leftarrow nullAttributeList,
session: Session \leftarrow 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 ← null Session];

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.AttributeSet** or **NSFile.ExtendedSelections** causes a problem. An error of type **attributeValue** is raised when an attribute value specified in an **NSFile.AttributeSet** 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;
```




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 ← null Session];

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.

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.



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), numberOfWorkers(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.

<i>read</i>	Granting <i>read</i> access allows a client to: examine the contents and attributes of a file; list a directory and examine the attributes of its children (<i>List</i>); copy the file; search a directory (during <i>Find</i>).
<i>write</i>	Granting <i>write</i> 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 (<i>remove</i> permission for the parent is also required).
<i>add</i>	This permission applies only to directories. It allows the client to insert new files into the directory.
<i>remove</i>	This permission applies only to directories. It allows the client to remove children from the directory (<i>write</i> permission for the child is also required).
<i>owner</i>	Granting <i>owner</i> access allows a client to modify the access lists of a file (<i>write</i> 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.

```
NSFileascendingPositionOrdering: key Ordering = [
    key[key: position, ascending: TRUE]];
```

```
NSFiledescendingPositionOrdering: 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.

```
NSFiledefaultOrdering: key Ordering = [key[key: name, ascending: TRUE]];
```

```
NSFilenullOrdering: 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).

```
NSFileAttribute: 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.numberOfWorkers: 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.filmedBy: AttributeType = 24;
NSAssignedTypes.filmedOn: 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 \leftarrow 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
<code>accessList</code>	NC
<code>BackedUpOn</code>	NC
<code>checksum</code>	(1)
<code>childrenUniquelyNamed</code>	NC
<code>createdBy</code>	(2)
<code>createdOn</code>	(3)
<code>defaultAccessList</code>	NC
<code>extended</code>	NC
<code>filedBy</code>	NC
<code>filedOn</code>	NC
<code>fileID</code>	NC
<code>isDirectory</code>	NC
<code>isTemporary</code>	NC
<code>modifiedBy</code>	(2)
<code>modifiedOn</code>	(3)
<code>name</code>	NC
<code>numberOfChildren</code>	NC
<code>ordering</code>	NC
<code>parentID</code>	NC
<code>pathname</code>	NC
<code>position</code>	NC
<code>readBy</code>	NC
<code>readOn</code>	NC
<code>service</code>	NC
<code>sizeInBytes</code>	(4)
<code>sizeInPages</code>	(5)
<code>subtreeSize</code>	(6)
<code>subtreeSizeLimit</code>	NC
<code>type</code>	NC
<code>version</code>	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
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	In List	Resulting File Behavior	Resulting Descendants Behavior
<code>accessList</code>	NC	NC	NC	set	NC	NC
<code>backedUpOn</code>	NC	NC	NC	set	NC	NC
<code>checksum</code>	NC	NC	NC	illegal	NC	NC
<code>childrenUniquelyNamed</code>	NC	NC	NC	illegal	NC	NC
<code>createdBy</code>	NC	NC	NC	illegal	NC	NC
<code>createdOn</code>	NC	NC	NC	illegal	NC	NC
<code>defaultAccessList</code>	NC	NC	NC	set	NC	NC
<code>extended</code>	NC	NC	NC	set	NC	NC
<code>filedBy</code>	NC	NC	NC	illegal	(1)	(2)
<code>filedOn</code>	NC	NC	NC	illegal	(3)	(4)
<code>fileID</code>	NC	NC	NC	illegal	(5)	(5)
<code>isDirectory</code>	NC	NC	NC	illegal	NC	NC
<code>isTemporary</code>	NC	NC	NC	set (6)	(7)	(8)
<code>modifiedBy</code>	NC	NC	(2)	illegal	(2)	(2)
<code>modifiedOn</code>	NC	NC	(4)	illegal	(4)	(4)
<code>name</code>	NC	NC	NC	set	NC	NC
<code>numberOfChildren</code>	NC	NC	incremented	illegal	NC	NC
<code>ordering</code>	NC	NC	NC	illegal	NC	NC
<code>parentID</code>	NC	NC	NC	illegal	(9)	(10)
<code>pathname</code>	NC	NC	NC	illegal	(11)	(11)
<code>position</code>	NC	NC	NC	(12)	(13)	(14)
<code>readBy</code>	(2)	(2)	NC	illegal	nullString	nullString
<code>readOn</code>	(4)	(4)	NC	illegal	nullTime	nullTime
<code>service</code>	NC	NC	NC	(15)	(16)	(16)
<code>sizeInBytes</code>	NC	NC	NC	illegal	NC	NC
<code>sizeInPages</code>	NC	NC	NC	illegal	NC	NC
<code>subtreeSize</code>	NC	NC	(17)	illegal	NC	NC
<code>subtreeSizeLimit</code>	NC	NC	NC	set	NC	NC
<code>type</code>	NC	NC	NC	illegal	NC	NC
<code>version</code>	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
<code>accessList</code>	NC
<code>BackedUpOn</code>	NC
<code>checksum</code>	NC
<code>childrenUniquelyNamed</code>	NC
<code>createdBy</code>	NC
<code>createdOn</code>	NC
<code>defaultAccessList</code>	NC
<code>extended</code>	NC
<code>filedBy</code>	NC
<code>filedOn</code>	NC
<code>fileID</code>	NC
<code>isDirectory</code>	NC
<code>isTemporary</code>	NC
<code>modifiedBy</code>	NC
<code>modifiedOn</code>	NC
<code>name</code>	NC
<code>numberOfChildren</code>	NC
<code>ordering</code>	NC
<code>parentID</code>	NC
<code>pathname</code>	NC
<code>position</code>	NC
<code>readBy</code>	(1)
<code>readOn</code>	(2)
<code>service</code>	NC
<code>sizeInBytes</code>	NC
<code>sizeInPages</code>	NC
<code>subtreeSize</code>	NC
<code>subtreeSizeLimit</code>	NC
<code>type</code>	NC
<code>version</code>	NC

NC = No Change

CopyIn (NSSegment)

Notes:

- (1) Session's user.
- (2) Current time.

Summary of Attribute Behaviors

Table 6.5

Attribute	Behavior
<code>accessList</code>	NC
<code>BackedUpOn</code>	NC
<code>checksum</code>	(1)
<code>childrenUniquelyNamed</code>	NC
<code>createdBy</code>	(2)
<code>createdOn</code>	(3)
<code>defaultAccessList</code>	NC
<code>extended</code>	NC
<code>filedBy</code>	NC
<code>filedOn</code>	NC
<code>fileID</code>	NC
<code>isDirectory</code>	NC
<code>isTemporary</code>	NC
<code>modifiedBy</code>	(2)
<code>modifiedOn</code>	(3)
<code>name</code>	NC
<code>numberOfChildren</code>	NC
<code>ordering</code>	NC
<code>parentID</code>	NC
<code>pathname</code>	NC
<code>position</code>	NC
<code>readBy</code>	NC
<code>readOn</code>	NC
<code>service</code>	NC
<code>sizeInBytes</code>	NC
<code>sizeInPages</code>	NC
<code>subtreeSize</code>	NC
<code>subtreeSizeLimit</code>	NC
<code>type</code>	NC
<code>version</code>	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	Behavior	Resulting File	
		In List	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 [**defaultValue: 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
<code>accessList</code>	NC
<code>BackedUpOn</code>	NC
<code>checksum</code>	NC
<code>childrenUniquelyNamed</code>	NC
<code>createdBy</code>	NC
<code>createdOn</code>	NC
<code>defaultAccessList</code>	NC
<code>extended</code>	NC
<code>filedBy</code>	NC
<code>filedOn</code>	NC
<code>fileID</code>	NC
<code>isDirectory</code>	NC
<code>isTemporary</code>	NC
<code>modifiedBy</code>	NC
<code>modifiedOn</code>	NC
<code>name</code>	NC
<code>numberOfChildren</code>	NC
<code>ordering</code>	NC
<code>parentID</code>	NC
<code>pathname</code>	NC
<code>position</code>	NC
<code>readBy</code>	(1)
<code>readOn</code>	(2)
<code>service</code>	NC
<code>sizeInBytes</code>	NC
<code>sizeInPages</code>	NC
<code>subtreeSize</code>	NC
<code>subtreeSizeLimit</code>	NC
<code>type</code>	NC
<code>version</code>	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	Resulting File		Resulting Descendants Behavior
		In List	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
<code>accessList</code>	NC
<code>BackedUpOn</code>	NC
<code>checksum</code>	(1)
<code>childrenUniquelyNamed</code>	NC
<code>createdBy</code>	(2)
<code>createdOn</code>	(3)
<code>defaultAccessList</code>	NC
<code>extended</code>	NC
<code>filedBy</code>	NC
<code>filedOn</code>	NC
<code>fileID</code>	NC
<code>isDirectory</code>	NC
<code>isTemporary</code>	NC
<code>modifiedBy</code>	(2)
<code>modifiedOn</code>	(3)
<code>name</code>	NC
<code>numberOfChildren</code>	NC
<code>ordering</code>	NC
<code>parentID</code>	NC
<code>pathname</code>	NC
<code>position</code>	NC
<code>readBy</code>	(4)
<code>readOn</code>	(5)
<code>service</code>	NC
<code>sizeInBytes</code>	NC
<code>sizeInPages</code>	NC
<code>subtreeSize</code>	NC
<code>subtreeSizeLimit</code>	NC
<code>type</code>	NC
<code>version</code>	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		Descendants
			In List	Behavior	Behavior
<code>accessList</code>	NC	NC	set	NC	NC
<code>backedUpOn</code>	NC	NC	set	NC	NC
<code>checksum</code>	NC	NC	illegal	NC	NC
<code>childrenUniquelyNamed</code>	NC	NC	illegal	NC	NC
<code>createdBy</code>	NC	NC	illegal	NC	NC
<code>createdOn</code>	NC	NC	illegal	NC	NC
<code>defaultAccessList</code>	NC	NC	set	NC	NC
<code>extended</code>	NC	NC	set	NC	NC
<code>filedBy</code>	NC	NC	illegal	(1)	NC
<code>filedOn</code>	NC	NC	illegal	(2)	NC
<code>fileID</code>	NC	NC	illegal	NC	NC
<code>isDirectory</code>	NC	NC	illegal	NC	NC
<code>isTemporary</code>	NC	NC	set(3)	FALSE	NC
<code>modifiedBy</code>	(1)	(1)	illegal	(1)	NC
<code>modifiedOn</code>	(2)	(2)	illegal	(2)	NC
<code>name</code>	NC	NC	set	NC	NC
<code>numberOfChildren</code>	decremented	incremented	illegal	NC	NC
<code>ordering</code>	NC	NC	illegal	NC	NC
<code>parentID</code>	NC	NC	illegal	(4)	NC
<code>pathname</code>	NC	NC	illegal	(5)	(5)
<code>position</code>	NC	NC	(6)	(7)	NC
<code>readBy</code>	NC	NC	illegal	NC	NC
<code>readOn</code>	NC	NC	illegal	NC	NC
<code>service</code>	NC	NC	illegal	(8)	(8)
<code>sizeInBytes</code>	NC	NC	illegal	NC	NC
<code>sizeInPages</code>	NC	NC	illegal	NC	NC
<code>subtreeSize</code>	(9)	(9)	illegal	NC	NC
<code>subtreeSizeLimit</code>	NC	NC	set	set	NC
<code>type</code>	NC	NC	illegal	NC	NC
<code>version</code>	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	Behavior	File	
		In List	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
<code>accessList</code>	NC
<code>.BackedUpOn</code>	NC
<code>checksum</code>	(1)
<code>childrenUniquelyNamed</code>	NC
<code>createdBy</code>	NC
<code>createdOn</code>	NC
<code>defaultAccessList</code>	NC
<code>extended</code>	NC
<code>filedBy</code>	NC
<code>filedOn</code>	NC
<code>fileID</code>	NC
<code>isDirectory</code>	NC
<code>isTemporary</code>	NC
<code>modifiedBy</code>	NC
<code>modifiedOn</code>	NC
<code>name</code>	NC
<code>numberOfChildren</code>	NC
<code>ordering</code>	NC
<code>parentID</code>	NC
<code>pathname</code>	NC
<code>position</code>	NC
<code>readBy</code>	(2)
<code>readOn</code>	(3)
<code>service</code>	NC
<code>sizeInBytes</code>	NC
<code>sizeInPages</code>	NC
<code>subtreeSize</code>	NC
<code>subtreeSizeLimit</code>	NC
<code>type</code>	NC
<code>version</code>	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	Behavior
<code>accessList</code>	NC
<code>BackedUpOn</code>	NC
<code>checksum</code>	(1)
<code>childrenUniquelyNamed</code>	NC
<code>createdBy</code>	(2)
<code>createdOn</code>	(3)
<code>defaultAccessList</code>	NC
<code>extended</code>	NC
<code>filedBy</code>	NC
<code>filedOn</code>	NC
<code>fileID</code>	NC
<code>isDirectory</code>	NC
<code>isTemporary</code>	NC
<code>modifiedBy</code>	(2)
<code>modifiedOn</code>	(3)
<code>name</code>	NC
<code>numberOfChildren</code>	NC
<code>ordering</code>	NC
<code>parentID</code>	NC
<code>pathname</code>	NC
<code>position</code>	NC
<code>readBy</code>	NC
<code>readOn</code>	NC
<code>service</code>	NC
<code>sizeInBytes</code>	(4)
<code>sizeInPages</code>	(5)
<code>subtreeSize</code>	(6)
<code>subtreeSizeLimit</code>	NC
<code>type</code>	NC
<code>version</code>	NC

NC = No Change

SetLength (NSFileStream)

Notes:

- (1) Set to `NSFile.unknownChecksum`.
- (2) Session's user.
- (3) Currenttime.
- (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	Behavior
<code>accessList</code>	NC	set	(1)
<code>backedUpOn</code>	NC	set	<code>nullTime</code>
<code>checksum</code>	NC	set	(2)
<code>childrenUniquelyNamed</code>	NC	set	<code>FALSE</code>
<code>createdBy</code>	NC	set	(3)
<code>createdOn</code>	NC	set	(4)
<code>defaultAccessList</code>	NC	set	(1)
<code>extended</code>	NC	set	<code>empty</code>
<code>filedBy</code>	NC	illegal	(5)
<code>filedOn</code>	NC	illegal	(6)
<code>fileID</code>	NC	illegal	(7)
<code>isDirectory</code>	NC	set	<code>FALSE</code>
<code>isTemporary</code>	NC	set(8)	(9)
<code>modifiedBy</code>	(3)	illegal	(3)
<code>modifiedOn</code>	(4)	illegal	(4)
<code>name</code>	NC	set	(10)
<code>numberOfChildren</code>	incremented	illegal	0 (zero)
<code>ordering</code>	NC	set	(11)
<code>parentID</code>	NC	illegal	(12)
<code>pathname</code>	NC	illegal	(13)
<code>position</code>	NC	(14)	(15)
<code>readBy</code>	NC	illegal	<code>nullString</code>
<code>readOn</code>	NC	illegal	<code>nullTime</code>
<code>service</code>	NC	(16)	(17)
<code>sizeInBytes</code>	NC	(18)	(19)
<code>sizeInPages</code>	NC	(18)	(19)
<code>subtreeSize</code>	(20)	illegal	(2)
<code>subtreeSizeLimit</code>	NC	set	(21)
<code>type</code>	NC	set	(22)
<code>version</code>	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
<code>accessList</code>	NC	(1)
<code>backedUpOn</code>	NC	NC
<code>checksum</code>	NC	NC
<code>childrenUniquelyNamed</code>	NC	NC
<code>createdBy</code>	NC	NC
<code>createdOn</code>	NC	NC
<code>defaultAccessList</code>	NC	(1)
<code>extended</code>	NC	NC
<code>filedBy</code>	NC	NC
<code>filedOn</code>	NC	NC
<code>fileID</code>	NC	NC
<code>isDirectory</code>	NC	NC
<code>isTemporary</code>	NC	NC
<code>modifiedBy</code>	NC	(2)
<code>modifiedOn</code>	NC	(3)
<code>name</code>	NC	NC
<code>numberOfChildren</code>	NC	NC
<code>ordering</code>	NC	NC
<code>parentID</code>	NC	NC
<code>pathname</code>	NC	NC
<code>position</code>	NC	NC
<code>readBy</code>	NC	NC
<code>readOn</code>	NC	NC
<code>service</code>	NC	NC
<code>sizeInBytes</code>	NC	NC
<code>sizeInPages</code>	NC	NC
<code>subtreeSize</code>	NC	NC
<code>subtreeSizeLimit</code>	NC	NC
<code>type</code>	NC	NC
<code>version</code>	NC	NC

NC = No Change

UnifyAccessLists

Notes:

- (1) Set to **[defaulted: TRUE]**.
- (2) Session's user if modified.
- (3) Current time if modified.

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 = NSSString.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 **NSFileName.lowestVersion** and **NSFileName.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 = NSSString.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

CopyVPNTToString 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.CopyVPNTToString: 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.

AppendVPNTToString is like **CopyVPNTToString** except that it appends the fields of a **VPN** to a preallocated **String**.

NSFileName.AppendVPNTToString: 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 **VPNs** and for freeing **VPNs** 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**.



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 NSSString.String;

The default name is initially "Anonymous." It can be changed by calling **NSFileControl.SetDefaultName**.

NSFileControl.SetDefaultName: PROC [name: NSSString.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 is 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:

$$\text{size of data files (in pages)} = 2 \times (\text{number of files on volume} \div 10)$$

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.Entry**s. 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.Names .
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 masters.]
- [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.

A**References**

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

version 3 changes

```
SessionParameterObject: TYPE = CHOICE OF {
    xerox800(0) = > NULL,
    xerox850(1), xerox860(2) = > [pollProc: UNSPECIFIED],
    system6(3), cmcll(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],
```

Gateway Access Protocol

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
 bitSyncrhous(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 = 342₈

readModifiedAll3270 = 344₈

read3270 = 345₈

6 New status values

The following SPP attention bytes have been added:

puActive = 347₈

pulnactive = 350₈

OFFICE SYSTEMS DIVISION

Reader's Feedback

Xerox's Technical Publications Departments want to provide documents that meet the needs of all our product users. Your comments help us correct and improve our publications. Please take a few minutes to respond. If you have comments on the product this document describes, contact your Xerox representative.

1. Did you find any errors in this publication? What were they? On which pages?

2. Were there any areas that were hard to understand because of descriptions or wording? What were they? Where?

3. Did this publication give you all the information you needed? If not, what was missing?

4. Was this manual at the right level for your needs? If not, what other types of publications do you need?

5. What one thing could we do to improve this manual for you?

NAME _____ DATE _____

TITLE _____ COMPANY _____

ADDRESS _____

CITY _____ STATE _____ ZIP _____

XDE3.0-6001

(

(

(

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

1	Introduction	1-1
1.1	Goals	1-1
1.2	Definition of terms	1-2
2	Overview	2-1
2.1	Sessions	2-1
2.2	Types of foreign systems	2-1
2.3	Transport service	2-2
2.3.1	The transports	2-2
2.3.2	The physical transmission medium	2-3
2.4	Sending/receiving data and controls during a session	2-3
2.5	Terminating the session	2-3
3	Client interface	3-1
3.1	Reset	3-1
3.2	Creating a session	3-1
3.2.1	Session parameters	3-2
3.2.2	Defining the transport	3-3
3.2.3	Connection establishment	3-8
3.3	Reserve	3-9
3.4	Transferring sessions	3-11
3.5	Deleting a session	3-11
3.6	Freeing inactive resources	3-11
3.7	GAPCallBack protocol	3-12
3.7.1	ReserveComplete	3-12
3.7.2	Poll	3-13
3.7.3	Authenticate	3-13
4	Remote errors	4-1

Table of contents

5	Data transfer with the foreign system	5-1
5.1	Data transfer	5-1
5.2	Controls	5-1
5.2.1	Classes of generic controls	5-1
5.2.2	Generic controls	5-2
5.3	Status	5-5
5.4	Data errors.	5-6
5.5	Termination	5-6

Appendix

A	References	A-1
A.1	Mandatory references	A-1
A.2	Informational references	A-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.

<i>protocol</i>	A <i>protocol</i> is a set of conventions, particularly the allowed formats and sequences of communication, between two communicators.
<i>protocol layering</i>	<i>Protocol layering</i> 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.
<i>session</i>	A <i>session</i> is an association between a GAP client and the foreign system, by which the exchange of information is managed.
<i>transmission medium</i>	The <i>transmission medium</i> is the lowest level physical transport mechanism, e.g., leased lines, DDD circuit, and the Ethernet; also, a virtual transport mechanism.
<i>transport</i>	A <i>transport</i> is an entity that implements one layer of a transport service. The entity usually corresponds to the implementation of one layer of protocol.
<i>transport service</i>	A <i>transport service</i> 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.

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.



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, cmclI, 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,
    polledBSController, polledSDLController => [
        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 **polledBSController** transport,
- 2) a one-level **polledBSCTerminal** transport.

Note: All variants other than **rs232c**, **bsc**, **teletype**, **polledBSController**, 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 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};
```




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 many 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.



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 **306₈** [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 **315₈** [In-band]

Marks a transaction boundary in conjunction with an **abortGetTransaction** or an **abortPutTransaction**.

abortPutTransaction **307₈** [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	304₈	[Out-of-band]
Elicit a response confirming that the foreign device/process is operational.		
audibleSignal	303₈	[Out-of-band]
Send an audible signal. For some word processors, this is a signal to "go to voice".		
cleanup	320₈	[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	312₈	[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	310₈	[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	314₈	[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	333₈	[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	305₈	[Out-of-band]
Response to the areYouThere control confirming that the foreign device/process is operational.		
interrupt	301₈	[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	300₈	[In-band]
This is the normal condition. The transition from none to some other packet subtype value and back is the event that indicates the control in the data sequence.		
remoteNotResponding	331₈	[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		

Data transfer with the foreign system

mark **abortMark**. The **remoteNotResponding** control should never be generated by the GAP client.

resume **302₈** [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 **313₈** [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 **311₈** [In-band]

Tell the receiver that it may send now. This control is used only in alternating-mode transports (e.g., BSC).

chain3270 **334₈** [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 **335₈** [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 **336₈** [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 **337₈** [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 **340₈** [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 **322₈**

The transmission medium has gone from the *up* to the *down* state. This indicates that the connection to the foreign system has terminated.

mediumUp **321₈**

The transmission medium has gone from the *down* to the *up* state. This indicates that the connection to the foreign system is operational.

noGetForData **325₈**

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 **323₈**

The foreign system rejected our security id.

unsupportedProtocolFeature **326₈**

The foreign system used some protocol feature that is currently unsupported (e.g., an RVI received from a Xerox 800).

unexpectedRemoteBehavior **327₈**

The foreign system did something unusual that did not follow documented protocols.

unexpectedSoftwareFailure **330₈**

The software encountered an unrecoverable software error.

weRejectedAccessID **324₈**

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 **341₈** [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 **371₈** [3270 emulation only]

The 3270 host has not polled our controller for at least 2 minutes.

hostPolling3270 **370₈** [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 **317₈**

A framing error occurred on all characters in this packet.

parityError **316₈**

A parity error occurred 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].

A

References
