XEROX

Services Software

Services 8.0 Programmer's Guide

November 1984

Filed as: [XNS:OSBU North]<Services>8.0>Doc>Programmer's Guide>*

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

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

Filing Programmer's Manual

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

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 = > NSFile.Error[...]]
        res.file ←NSFile.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.

3

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].

4 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 **maxlength** at least as great as **bytes**, a **length** of zero, and a string body of sufficient length to hold **maxlength** 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

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

Authentication Service	or simply *AS*. The distributed service supplied by a set of cooperating authentication servers.
authentication server	a server machine running Authentication Service software; one *instance* of the Authentication Service, or, the software running on such a machine.
authentication stub	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.
authentication client	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.

2 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

```
Credentials: 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;

MakeIdentity: 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];
```

MakeIdentity 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 **MakeIdentity** (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 **FreeIdentity.**

MakeStrongIdentityUsingKey is similar to **MakeIdentity**, 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.

```
FreeIdentity: 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. **FreeIdentity** 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 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 ← 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↑**. 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 **FreeIdentity** 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.

3 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 **MakeIdentity** 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. **FreeIdentity** 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. **MakeIdentity** and **FreeIdentity** 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. FreeIdentity[@me, zj];
    Auth.Terminate[@conversation, z];
    EXIT };
z: UNCOUNTED ZONE ←← ... ;
    -- create the identity handle
userName: NSName.Name ← ... ;
userPassword: NSString.String ← ... ;
me: Auth.IdentityHandle ← Auth.MakeIdentity[userName, userPassword, 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.FreeIdentity[@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

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

Clearinghouse System	or simply *Clearinghouse* (capital C). The distributed service supplied by a set of cooperating clearinghouse servers.
clearinghouse service	or simply *clearinghouse* (small c). A server machine running Clearinghouse Service software; one *instance* of the Clearinghouse Service.
clearinghouse stub	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.
fully qualified name	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 *three part name*. Generally this term implies that the name does not contain any wildcards. Names are always presented to the clearinghouse in fully qualified form.
distinguished name	the official, full name of an object registered in the Clearinghouse (i.e., as opposed to an alias).
alias	a "nickname" for some distinguished name. When the distinguished name is deleted, the alias is automatically deleted as well.
organization name	the most significant field of a fully qualified name.
domain name	the second most significant field of a fully qualified name.
local name	the least significant field of a fully qualified name.
pattern	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.
property list	the sequence of (*propertyID*, *valueOrGroup*) pairs which is attached to a name. A property list may be empty.
propertyID	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 CHPIDs.ps(= 10001).

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

2

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:

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

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 **CHPIDs.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:

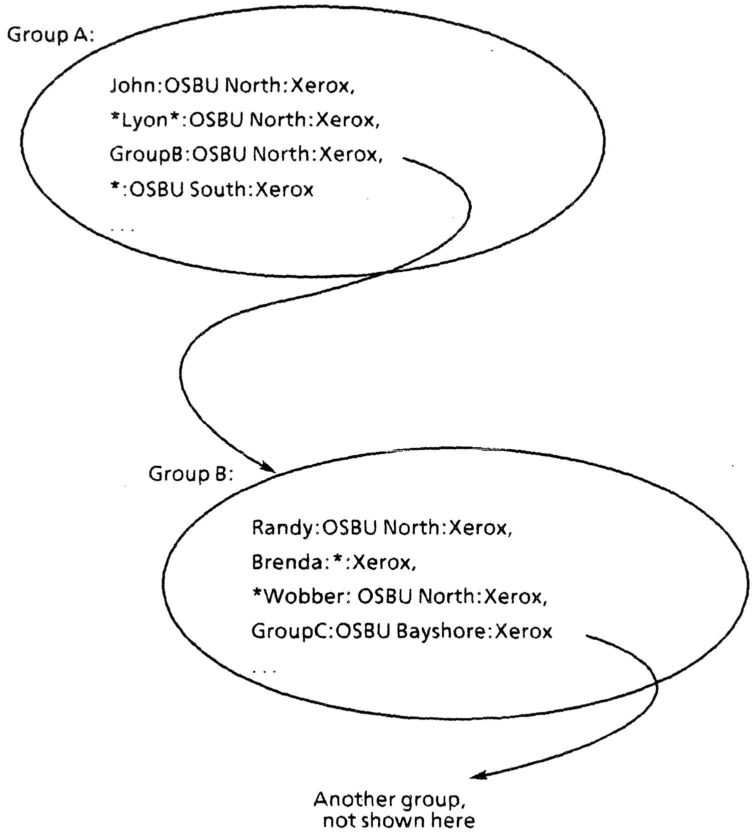

Figure 2.1 **IsMember/IsMemberClosure** example

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**. **CHPIDs.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.

3 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.Character**s. The maximum length string of **NSString.Character**s 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
    wrongPropertyType(21),-- you wanted a Group, but it was a Value, or vise 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 = LONG POINTER TO BufferArea;
BufferArea: TYPE = MACHINE DEPENDENT RECORD [
    maxlength(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* recommend 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 deserialized from **blk**.

A

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

B

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

Figures

1 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.

2

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*:

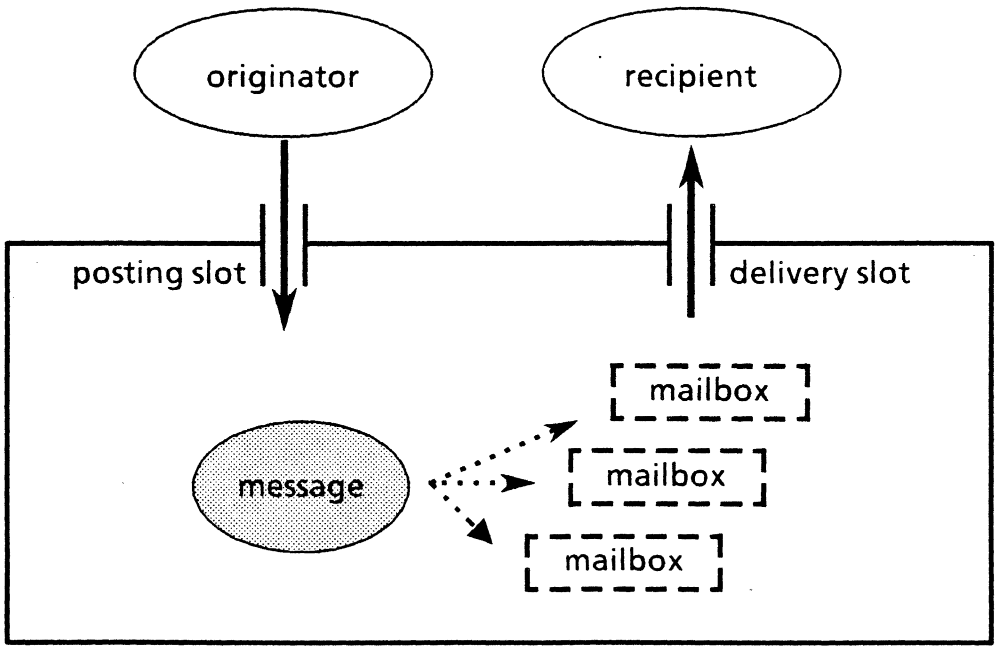

Figure 2.1 Mail transport system

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;
```

3 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.

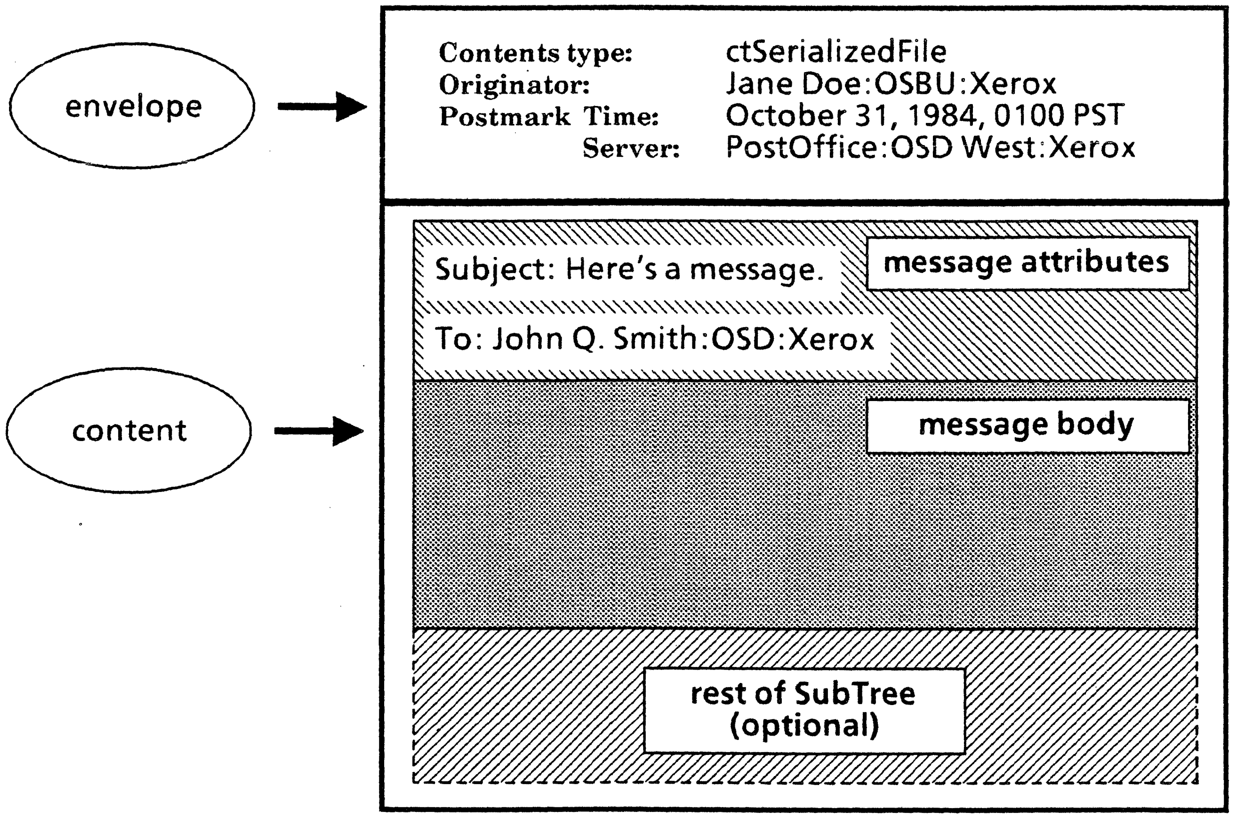
envelope
content
Contents type: ctSerializedFile
Originator: Jane Doe:OSBU:Xerox
Postmark Time: October 31, 1984, 0100 PST
Server: PostOffice:OSD West:Xerox
Subject: Here's a message.
message attributes
To: John Q. Smith:OSD:Xerox
message body
rest of SubTree (optional)

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 inbaskets frequently.

Note: Inbaskets 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 ← nullCacheVerifier,
   allowSharing: BOOLEAN ← FALSE, loc: MailTransport.Location ← 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.

4

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.UnqualifyAttributeNames: 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.AttributeList 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.

5 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

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.**

2 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 **ErrorType**s 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.**

3

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

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].

2 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
  DCScpx(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, *DCScpx, DCScpy,* 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 *DCScpx*, *DCScpy* 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

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**.

2 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
```

3

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

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.

2 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 **EXPORT**ed 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 **EXPORT**s 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

3 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 **Get**s or multiple **Put**s 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 **Get**s 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 **SST**s to label data. A client filter may need to use **SST**s for format conversion and/or parsing of stream data. With the method suggested, the client can send generic controls and data labelling **SST**s, 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 affect 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 *client's* **Put**, 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	
CONTROLS	ttyHost	ibm3270
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.

4

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.

5 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.

noGetForData

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.SubSequenceType**s 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 **SIGNAL**s and **ERROR**s are generated by the stub. Client recovery actions accompany each exception condition.

GateStream.Error: ERROR [reason: GateStream.ErrorReason];

reason is one of the following **ErrorReason**s:

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;
```

6

Reliability and maintainability

[TBD]

7 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};
```

B

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 **Get**s 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/printer) 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