Ftp Functional Specification

Version 2.0
November, 1977

XEROX

SYSTEMS DEVELOPMENT DIVISION
3408 Hillview Avenue / Palo Alto / California 94304

This document is for Xerox internal use only.

Table of Contents

Preface

This document details the procedural interface to the recently completed Mesa Ftp Package, Ftp 2.0, an outgrowth of earlier work by Smokey Wallace and Hal Murray. This document obsoletes the author's two previous design documents, [1, 2]. The code described herein is currently running and available. The reader is referred to Appendices E and F for file locations and other details concerning its use. Comments, bug reports, suggestions for change or addition, and cries for help should be addressed to the implementor, Jim White (White@Maxc).

1. Introduction

1.1. Purpose

The File Transfer Package (Ftp) is one means of several for accessing and manipulating remote files via the network. Ftp provides primitives for storing, retrieving, deleting, renaming, and enumerating remote files. Ftp trades in whole files, in contrast to a page-level access package, for example, which trades in smaller units (i.e. pages of files), or CopyDisk, which trades in larger ones (i.e. an entire disk).

Ftp provides an interface to Alto, Maxc, IFS, and Juniper file systems, and any others that implement the long-standing File Transfer Protocol (FTP) described in [3].

In addition to providing file-related services, Ftp 2.0 provides primitives for delivering mail to and retrieving mail from remote mailboxes. Ftp is thus also a means for accessing mailboxes on Maxc or any other host that implements the recently specified Mail Transfer Protocol (MTP) described in [4].

1.2. Program Structure

Every Ftp dialogue involves two parties, designated *user* and *server*, which are linked by a network connection. In point of fact, file and mail operations are implemented by separate servers and hence a dialogue in which operations of both types are carried out actually involves three parties: the local user, the remote file server, and the remote mail server. The Ftp implementation, however, disguises the distinction between the two servers and presents to the client the illusion of a single server capable of handling both types of requests. At one end, a client program initiates and controls the dialogue by calling procedures provided by a local *Ftp User*. At the other end, a passive *Ftp Server* responds and replies to requests it receives from the distant Ftp User. Several Servers can coexist within a single host and hence several independent file transfers can proceed concurrently. Ftp Servers are created by a single, resident *Ftp Listener* in response to connection requests from distant Ftp Users. Each Server is spawned as a separate process and competes for system resources with other local processes under the control of a scheduler. When the distant Ftp User terminates its dialogue with the local Ftp Server, the Server destroys itself.

The remainder of this document describes the client's interface to the Ftp User and Listener; the Ftp Server, having no external interface, is not discussed further. In the procedure descriptions presented later in this document, the terms *local* and *remote* distinguish the host containing the described procedure from the distant host to which that host is connected, respectively.

1.3. File Naming Conventions

Ftp provides the client with two separate mechanisms for designating remote files: *absolute filenames*, which must conform to the file naming conventions of the remote file system; and *virtual filenames*, having a host-independent structure, which are mapped into absolute filenames by the remote file system. The purpose of this two-fold scheme is, on the one hand, to permit the exact specification of remote filenames by human users familiar with remote file naming conventions and, on the other, to permit the mechanical generation of

filenames by clients ignorant of such conventions.

Absolute filenames are STRINGs. Any internal structure an absolute filename might possess is indicated by delimiters embedded in the STRING. Virtual filenames, on the other hand, have four components--device, directory, name, and version--each of which is a STRING:

```
VirtualFilename: TYPE = POINTER TO VirtualFilenameObject;
VirtualFilenameObject: TYPE = RECORD [device, directory, name, version: STRING];
```

As part of its mapping operation, the remote file system combines these components to form a legal absolute filename (using appropriate field delimiters where necessary). The Maxc file system maps the device, directory, and version components of a virtual filename into the corresponding Tenex filename fields and maps the name component into the name and extension fields. The Alto file system ignores the device and directory components, maps the name component into the name and extension fields, and maps the version component into the corresponding Alto filename field. IFS ignores the device component, maps the directory component into the directory and subdirectory fields, and maps the name and version components into the corresponding IFS filename fields.

The client may use either or both of the file naming schemes outlined above, or a combination of the two. Whenever the local Ftp User communicates a remote filename to the remote Ftp Server, it sends *both* an absolute filename and a virtual filename. The absolute filename is that supplied by the client as a parameter to the Ftp User procedure that initiates the exchange. The virtual filename is that supplied by the client in a previous call to the **FTPSetFilenameDefaults** procedure described in Section 3.3. If all components of the virtual filename are NIL (e.g. before **FTPSetFilenameDefaults** is called), the remote file is completely specified by the absolute filename. If the absolute filename is NIL, the remote file is completely specified by the virtual filename. If both the absolute and virtual filenames are non-NIL, the remote Ftp Server has the option of using the virtual filename to default unspecified fields in the absolute filename.

The term *file group designator* denotes a filename, either absolute or virtual or both, that names a *group* of files, rather than a single file. File group designators often contain special characters that indicate wild or unspecified portions of the filename. The Maxc file system recognizes as a legitimate value for the device, directory, name, version, and/or extension field, the special character, asterisk ('*), denoting an arbitrary field value. The Alto file system recognizes the two special characters, asterisk ('*), denoting zero or more arbitrary characters, and pound sign ('#), denoting exactly one arbitrary character. IFS recognizes the special character, asterisk ('*), denoting zero or more arbitrary characters.

1.4. Exception Handling

Exceptional conditions encountered by Ftp are reported to the client by means of a single ERROR signal, **FTPError**; its one parameter, an enumerated type, pinpoints the error. Exceptional conditions reported in this way include not only those explicitly detected by Ftp but also those that originate as signals within the Pup package, as well as some that originate as signals within the Alto file package. Ftp restores itself to a consistent state after every error (in response to the UNWIND signal). Therefore, for example, if the client's connection to a remote Ftp Server breaks (e.g. because the remote host breaks), the client can close the connection and open a new one (in this case, to another host) without first having to destroy and recreate the local Ftp User.

The errors that Ftp may report to the client are summarized below. The most prominent of the errors generated by particular procedures are listed (using a notation something like that used for Mesa declarations) with the descriptions of those procedures later in this document. The errors classed below as *implementation* or *unidentified* errors can be generated (at least in principle) by nearly every procedure. Because they are so pervasive in principle and rare in practice, such errors are excluded from the descriptions of the individual procedures to which they nevertheless apply:

```
FTPError: ERROR [ftpError: FtpError];

FtpError: TYPE = {

  -- credential errors
  missingCredentials, noSuchPrimaryUser, noSuchSecondaryUser,
   incorrectPrimaryPassword, incorrectSecondaryPassword, requestedAccessDenied,

  -- communication errors
  noSuchHost, connectionTimedOut, connectionRejected, connectionClosed,
   noRouteToNetwork,

  -- file errors
  illegalFilename, noSuchFile, noRoomForFile, fileDataError, unexpectedEOF,

  -- dump errors
  errorBlockInDumpFile, unrecognizedBlockInDumpFile, blockInDumpFileTooLong,
   dumpFileCheckSumInError,

  -- mail errors
  noValidRecipients, noSuchMailbox, noSuchForwardingHost, noSuchDmsName,
   mailboxIsBusy,

  -- client state errors
  duplicateListener, filesModuleNotLoaded, mailModuleNotLoaded, missingConnection,
   duplicateConnection, busyConnection, unopenedForFiles, unopenedForMail,
   enumeratedFileProcessedOutOfSequence,

  -- implementation errors
  protocolVersionMismatch, notImplementedLocally, notImplementedRemotely,
   unrecognizedMarkByte, unexpectedMark, inappropriateMarkByte, stringTooLong,
   missingPropertyList, malformedPropertyList, unrecognizedProperty,
   duplicateProperty, illegalBooleanProperty, illegalExceptionIndex,
   duplicateMailboxException, illegalExceptionErrorCode, missingMessageLength,
   inappropriateCommandReported, malformedPropertyListReported,
   illegalFileCharacteristicReported, pupGlitch,
```

```
    -- unidentified errors
    unidentifiedTransientError, unidentifiedPermanentError, unidentifiedError};
```

2. Ftp

2.1. Program Management Primitives

Ftp provides two procedures for controlling its overall operation. The first, **FTPInitialize**, initializes Ftp for operation by preparing the necessary internal data structures and initializing the Pup Package (via a call to **PupDefs.PupPackageMake**). *The client must call this procedure before calling any other Ftp procedures.* Redundant calls simply increment a use count:

```
FTPInitialize: PROCEDURE;
```

The second procedure, **FTPFinalize**, finalizes Ftp's operation by finalizing the Pup Package (via a call to **PupDefs.PupPackageDestroy**) and disposing of Ftp's internal data structures. Before calling this procedure, the client should destroy any Users and Listener that may exist. *The client must call no other Ftp procedures (except FTPInitialize) once this procedure has been invoked.* Calls corresponding to redundant calls to **FTPInitialize** simply decrement the use count:

```
FTPFinalize: PROCEDURE;
```

3. Ftp User

3.1. Program Management Primitives

Ftp provides two procedures for controlling Ftp Users, several of which can coexist within a single host. The first procedure, **FTPCreateUser**, creates a new Ftp User:

```
FTPCreateUser: PROCEDURE [clientFilePrimitives: ClientFilePrimitives, clientData:
    UNSPECIFIED] RETURNS [ftpuser: FTPUser];

ClientFilePrimitives: TYPE = POINTER TO ClientFilePrimitivesObject;
ClientFilePrimitivesObject: TYPE = RECORD [...];
FTPUser: TYPE = POINTER TO FTPUserObject;
FTPUserObject: PRIVATE TYPE = RECORD[...];
```

The client must retain the result **ftpuser**, since each of the other procedures described in this section requires it as a parameter. The **ftpuser** is a pointer to a private record containing all of the state information the newly created Ftp User requires to function properly.

By supplying the parameter **clientFilePrimitives** (rather than setting it to NIL), the client can provide its own local file system interface, rather than accept the interface to the standard Alto file system which Ftp otherwise supplies. If it elects to exercise this option, the client

may also provide a parameter, clientData, to be passed by Ftp as an argument to each of the file primitives the client supplies. The reader is referred to Appendix D for detailed motivation for and instruction in the use of this option.

The second procedure, FTPDestroyUser, destroys a previously created Ftp User, reclaiming any local resources that may have been allocated to it and, if necessary, closing its connection to the remote host (which may involve a delay as control messages are exchanged via the network):

```
FTPDestroyUser: PROCEDURE [ftpuser: FTPUser];
```

3.2. Connection Management Primitives

Ftp provides three procedures for controlling communication with remote Ftp Servers. The first, FTPOpenConnection, establishes a connection to an Ftp Server at the designated host (which is specified by any string acceptable to PupDefs.GetAddress). A single Ftp User can support just one open connection at a time. The client must specify, by means of the purpose parameter, the class(es) of remote objects--files, mail, or filesAndMail--it intends to manipulate via the connection. The Ftp User employs this information to make contact with the appropriate server(s):

```
FTPOpenConnection: PROCEDURE [ftpuser: FTPUser, host: STRING, purpose: Purpose];

Purpose: TYPE = {files, mail, filesAndMail};

    FTPError: ERROR [{noSuchHost, connectionTimedOut, connectionRejected,
      connectionClosed, noRouteToNetwork, filesModuleNotLoaded,
      mailModuleNotLoaded, duplicateConnection}];
```

The second procedure, FTPRenewConnection, prevents a previously established but long inactive connection from being timed out and broken by the remote Ftp Server. A Server created by Ftp 2.0, for example, will break its connection to a remote Ftp User after ten minutes of inactivity. Calling the procedure below, while performing no real operation, is sufficient to convince a remote Ftp Server that the local client is alive, well, and interested in maintaining the connection. The client may call this procedure at any time (except during the course of a remote file enumeration or dump file inventory) without ill effect upon the connection. Because file and mail operations are implemented by different servers, the client's connection to one can be timed out because of inactivity while its connection to the other remains intact. To avoid such anomalies, the client must take care to exercise each connection with the required frequency:

```
FTPRenewConnection: PROCEDURE [ftpuser: FTPUser];

    FTPError: ERROR [{connectionTimedOut, connectionClosed,
      noRouteToNetwork, missingConnection, busyConnection}];
```

The third procedure, FTPCloseConnection, breaks a previously established connection to a remote Ftp Server. Redundant calls upon this procedure are treated as no operations:

```
FTPCloseConnection: PROCEDURE [ftpuser: FTPUser];
```

The three connection management procedures described above block the client until the connection to the remote Ftp Server has been established, renewed, or broken, respectively (which may involve a delay as control messages are exchanged via the network).

3.3. File Access and Specification Primitives

Ftp provides two procedures for obtaining access to remote files and for assisting in the formulation of remote filenames. The first, **FTPSetCredentials**, specifies the credentials to be implicitly used to access the remote files or mailboxes specified in subsequent procedure calls. This procedure declares either the client's **primary** or **secondary** identity, which the Ftp User associates with the first or second remote filename, respectively, in subsequent parameter lists. Mail primitives use the client's primary credentials only. To rename a file, for example, the client must present *two* sets of credentials, one (**primary**) to access the file and the other (**secondary**) to access its new location. The credentials are inspected only when access to the remote file is actually attempted (as the result of a subsequent procedure call). The client can retract previously specified **primary** or **secondary** credentials by calling **FTPSetCredentials** with both **user** and **password** set to NIL (or, of course, by assigning **user** and **password** new values):

```
FTPSetCredentials: PROCEDURE [ftpuser: FTPUser, status: Status, user, password:
    STRING];

Status: TYPE = {primary, secondary};
```

The second procedure, **FTPSetFilenameDefaults**, specifies the virtual filename to be implicitly associated with the remote (absolute) filenames specified in subsequent procedure calls. The reader is referred to Section 1.3 for a discussion of virtual filenames and their use. This procedure declares either the client's **primary** or **secondary** virtual filename, which the Ftp User associates with the first or second remote filename, respectively, in subsequent parameter lists. To rename a file, for example, the client may have to specify *two* virtual filenames, one (**primary**) to identify the file to be renamed, the other (**secondary**) to specify its new name. The virtual filename is interpreted only when access to the remote file is actually attempted (as the result of a subsequent procedure call). The client can retract (or decline to specify) components of a previously specified **primary** or **secondary** virtual filename by calling **FTPSetFilenameDefaults** with those components set to NIL (or, of course, by assigning them new values).:

```
FTPSetFilenameDefaults: PROCEDURE [ftpuser: FTPUser, status: Status,
    virtualFilename: VirtualFilename];

Status: TYPE = {primary, secondary};
VirtualFilename: TYPE = POINTER TO VirtualFilenameObject;
```

```
VirtualFilenameObject: TYPE = RECORD [device, directory, name, version: STRING];
```

3.4. File Enumeration Primitives

Ftp provides one procedure for enumerating the members of a remote file group. This procedure, FTPEnumerateFiles, supplies in turn to a client-provided procedure, processFile, the absolute and virtual filename of each file in the remote file group whose file group designator, remoteFiles, is specified, along with various pieces of information about the file and an additional parameter, processFileData, supplied by the client. Unknown or unspecified file information is specified as unknown, zero, or NIL, as appropriate. The reader is referred to Section 1.3 for a discussion of virtual filenames and their use. The order in which the filenames are presented to the client is host-dependent:

```
FTPEnumerateFiles: PROCEDURE [ftpuser: FTPUser, remoteFiles: STRING, intent: Intent,
    processFile: PROCEDURE [UNSPECIFIED, STRING, VirtualFilename, FileInfo],
    processFileData: UNSPECIFIED];

Intent: TYPE = {enumeration, retrieval, deletion, renaming, unspecified};
VirtualFilename: TYPE = POINTER TO VirtualFilenameObject;
VirtualFilenameObject: TYPE = RECORD [device, directory, name, version: STRING];
FileInfo: TYPE = POINTER TO FileInfoObject;
FileInfoObject: TYPE = RECORD [
    fileType: FileType, byteSize: CARDINAL, byteCount: InlineDefs.LongCARDINAL,
    creationDate, writeDate, readDate, author: STRING];
FileType: TYPE = {text, binary, unknown};

        FTPError: ERROR [{missingCredentials, noSuchPrimaryUser,
            noSuchSecondaryUser, incorrectPrimaryPassword,
            incorrectSecondaryPassword, requestedAccessDenied, connectionTimedOut,
            connectionClosed, noRouteToNetwork, illegalFilename, noSuchFile,
            noRoomForFile, fileDataError, missingConnection, busyConnection,
            unopenedForFiles}];
```

As a part of requesting the enumeration, the client indicates to the Ftp User, via the parameter intent, how it intends to process the files in the group. This information enables the Ftp User to intelligently select from among several possible strategies for effecting the enumeration. Since most of the enumeration strategies occupy the remote Ftp Server until the enumeration is complete, Ftp prohibits processFile from calling local Ftp User procedures, other than those implied by intent, that communicate with the remote Server. The client may specify any of the following intents:

1. An intent of enumeration declares that the client seeks the names (and file information) of the designated files (e.g. for presentation to a human user) and intends to manipulate the files in no other way during the course of the enumeration. More specifically, the client guarantees (and Ftp enforces) that processFile will make no calls to local Ftp User procedures that communicate with the remote Ftp Server.

2. An intent of **retrieval** declares that the client seeks to retrieve some or all (but possibly none) of the designated files and to manipulate them in no other way during the course of the enumeration. The client's **processFile** procedure may retrieve the file whose name is presented to it by supplying that name to the **FTPRetrieveFile** procedure described in Section 3.5. More specifically, then, the client guarantees (and Ftp enforces) that **processFile** will make no calls to local Ftp User procedures (other than **FTPRetrieveFile**) that communicate with the remote Ftp Server.

3. An intent of **deletion** declares that the client seeks to delete some or all (but possibly none) of the designated files and to manipulate them in no other way during the course of the enumeration. The client's **processFile** procedure may delete the file whose name is presented to it by supplying that name to the **FTPDeleteFile** procedure described in Section 3.6. More specifically, then, the client guarantees (and Ftp enforces) that **processFile** will make no calls to local Ftp User procedures (other than **FTPDeleteFile**) that communicate with the remote Ftp Server.

4. An intent of **renaming** declares that the client seeks to rename some or all (but possibly none) of the designated files and to manipulate them in no other way during the course of the enumeration. The client's **processFile** procedure may rename the file whose name is presented to it by supplying that name to the **FTPRenameFile** procedure described in Section 3.6. More specifically, then, the client guarantees (and Ftp enforces) that **processFile** will make no calls to local Ftp User procedures (other than **FTPRenameFile**) that communicate with the remote Ftp Server. In point of fact, **renaming** is currently little more than a synonym for **unspecified**, described below; all filenames are spooled onto a local scratch file before any are presented to the client.

5. An intent of **unspecified** declares that the client seeks unconstrained access to the designated files. The client's **processFile** procedure may retrieve, delete, or rename the file whose name is presented to it (or any other file, for that matter) by calling the appropriate Ftp User procedure. More specifically, **processFile** may make calls on any local Ftp User procedures it chooses since, in this case, Ftp spools the filenames onto a local scratch file (which Ftp promptly deletes once it has served its purpose) before presenting them to the client.

3.5. File Transfer Primitives

Ftp provides two procedures for transferring files between the local and remote file systems. The first, **FTPStoreFile**, stores a copy of the designated **localFile** in the remote file system, creating a new **remoteFile** with the indicated name and returning the size, **byteCount**, of the transmitted file in bytes:

```
FTPStoreFile: PROCEDURE [ftpuser: FTPUser, localFile, remoteFile: STRING, fileType:
  FileType] RETURNS [byteCount: InlineDefs.LongCARDINAL];
```

```
FileType: TYPE = {text, binary, unknown};

    FTPError: ERROR [{missingCredentials, noSuchPrimaryUser,
      noSuchSecondaryUser, incorrectPrimaryPassword,
      incorrectSecondaryPassword, requestedAccessDenied, connectionTimedOut,
      connectionClosed, noRouteToNetwork, illegalFilename, noSuchFile,
      noRoomForFile, fileDataError, missingConnection, busyConnection,
      unopenedForFiles, enumeratedFileProcessedOutOfSequence}];
```

As an additional parameter to FTPStoreFile, the client specifies the **fileType**--**text** or **binary**--of the file to be stored. The remote Ftp Server uses this information in deciding how it should store the file in its file system (e.g. on Maxc, text files are stored as 7-bit bytes, binary files as 8-bit bytes). If the client declines to provide this information (e.g. because it doesn't have it), FTPStoreFile makes an educated guess about the file's type. The reader is referred to the discussion of the OpenFile procedure in Appendix D for a description of the algorithm used in making this determination.

The client can effect the remote storage of a whole *group* of local files by using FTPStoreFile (to store a single file) in conjunction with the EnumerateFiles procedure described in Appendix D (to enumerate the files to be stored).

The FTPStoreFile procedure has yet another use in connection with the construction of remote dump files, as described in Appendix A.

The second procedure, FTPRetrieveFile, stores a copy of the designated **remoteFile** in the local file system, creating a new **localFile** with the indicated name and returning the size, **byteCount**, of the transmitted file in bytes:

```
FTPRetrieveFile: PROCEDURE [ftpuser: FTPUser, localFile, remoteFile: STRING]
  RETURNS [byteCount: InlineDefs.LongCARDINAL];

    FTPError: ERROR [{missingCredentials, noSuchPrimaryUser,
      noSuchSecondaryUser, incorrectPrimaryPassword,
      incorrectSecondaryPassword, requestedAccessDenied, connectionTimedOut,
      connectionClosed, noRouteToNetwork, illegalFilename, noSuchFile,
      noRoomForFile, fileDataError, errorBlockInDumpFile,
      unrecognizedBlockInDumpFile, blockInDumpFileTooLong,
      dumpFileCheckSumInError, missingConnection, busyConnection,
      unopenedForFiles, enumeratedFileProcessedOutOfSequence}];
```

The client can effect the local storage of a whole *group* of remote files by using FTPRetrieveFile (to retrieve a single file) in conjunction with the FTPEnumerateFiles procedure described in Section 3.4 (to enumerate the files to be retrieved).

The FTPRetrieveFile procedure has yet another use in connection with the retrieval of remote dump files, as described in Appendix A.

3.6. File Manipulation Primitives

Ftp provides two procedures for manipulating existing remote files. The first, **FTPDeleteFile**, deletes the specified **remoteFile**, reclaiming the space it occupied on secondary storage:

```
FTPDeleteFile: PROCEDURE [ftpuser: FTPUser, remoteFile: STRING];

    FTPError: ERROR [{missingCredentials, noSuchPrimaryUser,
      noSuchSecondaryUser, incorrectPrimaryPassword,
      incorrectSecondaryPassword, requestedAccessDenied, connectionTimedOut,
      connectionClosed, noRouteToNetwork, illegalFilename, noSuchFile,
      fileDataError, missingConnection, busyConnection, unopenedForFiles,
      enumeratedFileProcessedOutOfSequence}];
```

The client can effect the deletion of a whole *group* of remote files by using **FTPDeleteFile** (to delete a single file) in conjunction with the **FTPEnumerateFiles** procedure described in Section 3.4 (to enumerate the files to be deleted).

The second procedure, **FTPRenameFile**, renames the remote file whose current name, **currentFile**, is specified, assigning it the new name, **newFile**:

```
FTPRenameFile: PROCEDURE [ftpuser: FTPUser, currentFile, newFile: STRING];

    FTPError: ERROR [{missingCredentials, noSuchPrimaryUser,
      noSuchSecondaryUser, incorrectPrimaryPassword,
      incorrectSecondaryPassword, requestedAccessDenied, connectionTimedOut,
      connectionClosed, noRouteToNetwork, illegalFilename, noSuchFile,
      noRoomForFile, fileDataError, missingConnection, busyConnection,
      unopenedForFiles, enumeratedFileProcessedOutOfSequence}];
```

The client can effect the renaming of a whole *group* of remote files by using **FTPRenameFile** (to rename a single file) in conjunction with the **FTPEnumerateFiles** procedure described in Section 3.4 (to enumerate the files to be renamed).

4. Ftp Listener

4.1. Program Management Primitives

Ftp provides two procedures for controlling the local Ftp Listener. The first procedure, **FTPCreateListener**, creates the Listener. The client must specify, by means of the **purpose** parameter, the class(es) of local objects--**files**, **mail**, or **filesAndMail**--it wishes be made accessible to remote Ftp Users. The Listener employs this information to monitor the appropriate contact socket(s). The newly created Ftp Listener will create local Ftp Servers in response to requests from remote Ftp Users. A Server is destroyed when the remote User explicitly terminates its dialog with the Server, or after ten minutes of inactivity:

```
FTPCreateListener: PROCEDURE [purpose: Purpose, clientListenerAppendages:
    ClientListenerAppendages, clientFilePrimitives: ClientFilePrimitives, clientData:
    UNSPECIFIED];

Purpose: TYPE = {files, mail, filesAndMail};
ClientListenerAppendages: TYPE = POINTER TO ClientListenerAppendagesObject;
ClientListenerAppendagesObject: TYPE = RECORD [...];
ClientFilePrimitives: TYPE = POINTER TO ClientFilePrimitivesObject;
ClientFilePrimitivesObject: TYPE = RECORD [...];

    FTPError: ERROR [{duplicateListener, filesModuleNotLoaded,
      MailModuleNotLoaded}];
```

By supplying the parameter **clientListenerAppendages** (rather than setting it to NIL), the client can provide procedures to be called at strategic points in the operation of the Ftp Listener and any Servers it creates. The reader is referred to Appendix C for detailed motivation for and instruction in the use of this option.

By supplying the parameter **clientFilePrimitives** (rather than setting it to NIL), the client can provide its own local file system interface, rather than accept the interface to the standard Alto file system which Ftp otherwise supplies. If it elects to exercise this option, the client may also provide a parameter, **clientData**, to be passed by Ftp as an argument to each of the file primitives the client supplies. (Alternately, the client may supply this parameter on a per-Server basis by means of the **NoteServerCreation** listener appendage.) The reader is referred to Appendix D for detailed motivation for and instruction in the use of this option.

The second procedure, **FTPDestroyListener**, destroys the previously created Ftp Listener, reclaiming any local resources that may have been allocated to it and, if so instructed by the **abortServers** parameter, destroying any of its Servers that may exist at the time (rather than waiting for them to terminate normally). Destroying an Ftp Server may require closing its connection to the remote host, which may in turn involve a delay as control messages are exchanged via the network. Redundant calls upon this procedure are treated as no operations:

```
FTPDestroyListener: PROCEDURE [abortServers: BOOLEAN];
```

References

1. Jim White, "Mesa FTP Specification", 15 June 1977.
2. Jim White, "Mail Update to Mesa FTP Specification", 9 August 1977.
3. John Shoch, "A File Transfer Protocol Using the BSP -- 2nd edition", 15 June 1976.
4. Ed Taft, "Pup Mail Transfer Protocol (Edition 4)", 4 September 1977.
5. Ted Myer and Austin Henderson, "Message Transmission Protocol" (RFC 680), 15 May 1975.
6. Hal Murray, "How to get at the Pup Package", 19 October 1977.
7. Hal Murray, "Specifications for the current Mesa Pup Package", 19 October 1977.

Appendix A: Additional Primitives

A.1. Introduction

In addition to the primitives described in the body of this document, Ftp supplies a number of less frequently used procedures, described below.

A.2. Infrequently Used Connection Management Primitives

Ftp provides three procedures for controlling communication with remote Ftp Servers in a debugging context. The first, **FTPSetContactSocket**, specifies the remote socket at which the local Ftp User should expect to find the remote Ftp Listener, in subsequent calls to **FTPOpenConnection**. Experimental Ftp Listeners are often attached to non-standard sockets during their checkout phase; the **FTPSetContactSocket** procedure permits communication with such Listeners. The client must specify, by means of the **purpose** parameter, the class(es) of remote objects--**files**, **mail**, or **filesAndMail**--it expects to find accessible at this socket (file- and mail-related transactions being supported by distinct servers created by distinct listeners on distinct sockets). A socket number of zero resets the affected socket(s) to their standard, default values (i.e. 3 for **files** and 7 for **mail**):

```
FTPSetContactSocket: PROCEDURE [ftpuser: FTPUser, socket: PupDefs.Pair, purpose:
    Purpose];

PupDefs.Pair: TYPE = MACHINE DEPENDENT RECORD [first, second: CARDINAL];
Purpose: TYPE = {files, mail, filesAndMail};
```

The second procedure, **FTPEnableTrace**, causes a textual representation of all subsequent interactions between the local Ftp User and the remote Ftp Server to be presented to the client in zero or more calls to a **writeString** procedure it supplies. Successive STRINGs represent successive segments of the character stream describing the dialogue; STRING boundaries are insignificant. Redundant calls to **FTPEnableTrace** are treated as no operations. Be advised that passwords may appear in the trace:

```
FTPEnableTrace: PROCEDURE [ftpuser: FTPUser, writeString: PROCEDURE [STRING]];
```

The third procedure, **FTPDisableTrace**, prevents the textual representation of User/Server interaction from being reported to the client, and disassociates from the Ftp User the **writeString** procedure supplied by the client in a previous call to **FTPEnableTrace**. Redundant calls to **FTPDisableTrace** are treated as no operations:

```
FTPDisableTrace: PROCEDURE [ftpuser: FTPUser];
```

A.3. Dump File Primitives

Ftp provides three procedures for manipulating remote dump files. The first, **FTPInventoryDumpFile**, supplies in turn to a client-provided procedure, **processFile**, the (absolute) filename of each file in a specified **remoteDumpFile**, along with an additional parameter, **processFileData**, supplied by the client:

```
FTPInventoryDumpFile: PROCEDURE [ftpuser: FTPUser, remoteDumpFile: STRING,
    intent: DumpFileIntent, processFile: PROCEDURE [UNSPECIFIED, STRING],
    processFileData: UNSPECIFIED];

Intent: TYPE = {enumeration, retrieval, deletion, renaming, unspecified};
DumpFileIntent: TYPE = Intent[enumeration..retrieval];

        FTPError: ERROR [{missingCredentials, noSuchPrimaryUser,
            noSuchSecondaryUser, incorrectPrimaryPassword,
            incorrectSecondaryPassword, requestedAccessDenied, connectionTimedOut,
            connectionClosed, noRouteToNetwork, illegalFilename, noSuchFile,
            fileDataError, errorBlockInDumpFile, unrecognizedBlockInDumpFile,
            blockInDumpFileTooLong, dumpFileCheckSumInError, missingConnection,
            busyConnection, unopenedForFiles}];
```

As a part of requesting the inventory, the client indicates to the Ftp User, via the parameter **intent**, how it intends to process the files contained in the dump file. This information enables the Ftp User to intelligently select from among several possible strategies for effecting the inventory. Since each of the inventory strategies occupies the remote Ftp Server until the inventory is complete, Ftp prohibits **processFile** from calling local Ftp User procedures, other than those implied by **intent**, that communicate with the remote Server. The client may specify either of the following intents:

1. An intent of **enumeration** declares that the client seeks the names of the dumped files (e.g. for presentation to a human user) but has no interest in retrieving their contents. More specifically, the client guarantees (and Ftp enforces) that **processFile** will make no calls to local Ftp User procedures that communicate with the remote Ftp Server.

2. An intent of **retrieval** declares that the client seeks to retrieve some or all (but possibly none) of the dumped files. The client's **processFile** procedure may retrieve the file whose name is presented to it by supplying that name to the **FTPRetrieveFile** procedure described elsewhere. More specifically, then, the client guarantees (and Ftp enforces) that **processFile** will make no calls to local Ftp User procedures (other than **FTPRetrieveFile**) that communicate with the remote Ftp Server.

The second procedure, **FTPBeginDumpFile**, initializes a new remote **dumpFile** and prepares it to receive files via the **FTPStoreFile** procedure described elsewhere. In the presence of an open dump file, **FTPStoreFile**'s invocation is interpreted as a request to dump the specified **localFile**. In this context, **FTPStoreFile**'s **remoteFile** parameter is interpreted as the name by which the file is to be known *within the remote dump file*. Since the construction of a remote dump file occupies the remote Ftp Server until the dump file is complete, Ftp prohibits the client from calling local Ftp User procedures, other than **FTPStoreFile**, that communicate with the remote Ftp Server while the dump file is under construction (i.e. until the **FTPEndDumpFile** procedure described below is invoked):

```
FTPBeginDumpFile: PROCEDURE [ftpuser: FTPUser, remoteDumpFile: STRING];

    FTPError: ERROR [{missingCredentials, noSuchPrimaryUser,
      noSuchSecondaryUser, incorrectPrimaryPassword,
      incorrectSecondaryPassword, requestedAccessDenied, connectionTimedOut,
      connectionClosed, noRouteToNetwork, illegalFilename, noRoomForFile,
      fileDataError, missingConnection, busyConnection, unopenedForFiles}];
```

The third procedure, FTPEndDumpFile, finalizes a newly created remote dump file after all the desired files have been added to it:

```
FTPEndDumpFile: PROCEDURE [ftpuser: FTPUser];

    FTPError: ERROR [{connectionTimedOut, connectionClosed,
      noRouteToNetwork, noRoomForFile, fileDataError, missingConnection,
      busyConnection}];
```

Appendix B: Mail Primitives

B.1. Introduction

In addition to file primitives, Ftp supplies a family of procedures, described below, for manipulating remote mailboxes.

B.2. Delivery Primitives

Ftp provides three procedures for delivering (and/or forwarding) mail to remote mailboxes. The first, **FTPBeginDeliveryOfMessage**, initiates the delivery and/or forwarding of a message by enumerating its intended recipients via a linked list, **mailboxList**. In the simpler case, called *delivery*, in which a recipient's mailbox resides on the remote host (a case which Ftp distinguishes by finding **mailboxHostName** set to NIL), the corresponding list element need contain only the host-specific name, **mailboxName**, of the remote mailbox to which a copy of the message is to be appended, and a pointer, **next**, to the next element in the list (NIL signalling the end of the list). In the more complex case, called *forwarding* (which not all Ftp Servers will support), in which a recipient's mailbox resides on a third host, the corresponding list element must also contain the name, **mailboxHostName**, of that third host and (optionally) the full **dmsName** of the target mailbox (which the forwarder may be able to use to locate the recipient if he has moved):

```
FTPBeginDeliveryOfMessage: PROCEDURE [ftpuser: FTPUser, mailboxList: Mailbox,
   allocateString: PROCEDURE [CARDINAL] RETURNS [STRING]];

Mailbox: TYPE = POINTER TO MailboxObject;
MailboxObject: TYPE = RECORD [next: Mailbox,
   mailboxName, mailboxHostName, dmsName: STRING,
   errorCode: ErrorCode, errorMessage: STRING];
ErrorCode: TYPE = {ok, noSuchMailbox, noForwardingProvided,
   unspecifiedTransientError, unspecifiedPermanentError, unspecifiedError};

      FTPError: ERROR [{noSuchPrimaryUser, noSuchSecondaryUser,
         incorrectPrimaryPassword, incorrectSecondaryPassword,
         connectionTimedOut, connectionClosed, noRouteToNetwork,
         noValidRecipients, noSuchForwardingHost, noSuchDmsName, mailboxIsBusy,
         missingConnection, busyConnection, unopenedForMail}];
```

Delivery of the message succeeds or fails for each of its intended recipients independently. Either **FTPBeginDeliveryOfMessage** or the **FTPEndDeliveryOfMessage** procedure described below may report the failure of an individual delivery attempt by despositing in the appropriate list element an **errorCode** intended for examination by the client (**ok** signalling successful delivery, but only tentatively by **FTPBeginDeliveryOfMessage**) and, if **errorCode** is one of the three having the form **unspecified...Error**, an **errorMessage** intended for examination by a human user. Storage for any error messages that may be returned is allocated via the **allocateString** procedure provided by the client, which assumes responsibility for releasing the storage.

The second procedure, FTPSendBlockOfMessage, specifies a portion of the text of a message and is called repetitively after the message's recipients have been identified to FTPBeginDeliveryOfMessage. Successive calls specify the location in the client's address space, source, and the length in bytes, byteCount, of successive blocks of text. The text of the message must include a message header conforming to ARPANET standards as defined in [5]. Throughout the message, end of line is indicated via a carriage return (CR):

```
FTPSendBlockOfMessage: PROCEDURE [ftpuser: FTPUser, source: POINTER,
  byteCount: CARDINAL];

    FTPError: ERROR [{connectionTimedOut, connectionClosed,
      noRouteToNetwork, missingConnection, busyConnection,
      unopenedForMail}];
```

The third procedure, FTPEndDeliveryOfMessage, signals the end of the sequence of calls to FTPSendBlockOfMessage and, therefore, of the message's text, and effects the message's delivery and/or enqueues it for forwarding:

```
FTPEndDeliveryOfMessage: PROCEDURE [ftpuser: FTPUser];

    FTPError: ERROR [{connectionTimedOut, connectionClosed,
      noRouteToNetwork, missingConnection, busyConnection,
      unopenedForMail}];
```

Like the FTPBeginDeliveryOfMessage procedure already described, FTPEndDeliveryOfMessage reports its failure to deliver the message to one of its intended recipients by depositing in the corresponding element of the recipient list supplied to FTPBeginDeliveryOfMessage, an errorCode intended for examination by the client (ok here signalling successful delivery with finality) and, if errorCode is one of the three having the form unspecified...Error, an errorMessage intended for examination by a human user. Storage for any error messages that may be returned is again allocated via the allocateString procedure provided by the client, which assumes responsibility for releasing the storage.

B.3. Retrieval Primitives

Ftp provides four procedures for retrieving the contents of (and then resetting to empty) a remote mailbox. The first, FTPBeginRetrievalOfMessages, initiates retrieval of the contents of the remote mailbox whose host-specific name, mailboxName, is specified. To obtain access to the mailbox, the client must first have supplied the necessary credentials (if any) by calling the FTPSetCredentials procedure described elsewhere:

```
FTPBeginRetrievalOfMessages: PROCEDURE [ftpuser: FTPUser, mailboxName: STRING];

    FTPError: ERROR [{missingCredentials, noSuchPrimaryUser,
      noSuchSecondaryUser, incorrectPrimaryPassword,
      incorrectSecondaryPassword, requestedAccessDenied, connectionTimedOut,
      connectionClosed, noRouteToNetwork, noSuchMailbox, mailboxIsBusy,
      missingConnection, busyConnection, unopenedForMail}];
```

The second procedure, **FTPIdentifyNextMessage**, retrieves information about one of the messages in the mailbox identified in a previous call to **FTPBeginRetrievalOfMessages**. **FTPIdentifyNextMessage** is called repetitively until a **byteCount** of zero (signalling no more messages) is returned. Successive calls return information about successive messages stored in the mailbox. (The client may elect to leave some or all of the mailbox's contents unretrieved, in which case whatever remains will be sent by the remote Ftp Server but discarded by the local Ftp User in the final call to the **FTPEndRetrievalOfMessages** procedure described later.) The information returned by the procedure is deposited in a record, **messageInfo**, supplied by the client, and consists of the messsage's size in bytes, **byteCount**; the date and time, **deliveryDate**, at which the message was deposited in the mailbox (the required STRING being supplied by the client); and whether or not the message has been **opened** (i.e. examined) or **deleted** while in the mailbox (Maxc mailboxes, for example, can be manipulated directly via the MSG subsystem):

```
FTPIdentifyNextMessage: PROCEDURE [ftpuser: FTPUser, messageInfo: MessageInfo];

MessageInfo: TYPE = POINTER TO MessageInfoObject;
MessageInfoObject: TYPE = RECORD [byteCount: CARDINAL, deliveryDate: STRING,
   opened, deleted: BOOLEAN];

      FTPError: ERROR [{connectionTimedOut, connectionClosed,
         noRouteToNetwork, missingConnection, busyConnection,
         unopenedForMail}];
```

The third procedure, **FTPRetrieveBlockOfMessage**, retrieves a portion of the text of the message identified in a previous call to **FTPIdentifyNextMessage**. **FTPRetrieveBlockOfMessage** is called repetitively until an **actualByteCount** of zero (signalling no more blocks) is returned. Successive calls return successive blocks of the message. (The client may elect to leave some or all of the message's text unretrieved, in which case whatever remains will be sent by the remote Ftp Server but discarded by the local Ftp User in the next call to **FTPIdentifyNextMessage**.) Note that the client can anticipate the end of a message on the basis of the byte count returned by **FTPIdentifyNextMessage**. The text returned by the procedure is deposited in the buffer whose location in the client's address space, **destination**, and whose length *in words*, **maxWordCount**, are specified by the client. The procedure returns the length *in bytes*, **actualByteCount**, of the block of text actually retrieved (which may be shorter than the block requested). The text of the message includes a message header conforming to ARPANET standards as defined in [5]. Throughout the message, end of line is indicated via a carriage return (CR):

```
FTPRetrieveBlockOfMessage: PROCEDURE [ftpuser: FTPUser, destination: POINTER,
   maxWordCount: CARDINAL] RETURNS [actualByteCount: CARDINAL];

      FTPError: ERROR [{connectionTimedOut, connectionClosed,
         noRouteToNetwork, pupGlitch, missingConnection, busyConnection,
         unopenedForMail}];
```

The fourth procedure, FTPEndRetrievalOfMessages, terminates the retrieval operation and resets the mailbox to empty. FTPBeginRetrievalOfMessages and FTPEndRetrievalOfMessages are implemented in such a way that no new messages are lost during the retrieval process and the contents of the mailbox are discarded only when (if) FTPEndRetrievalOfMessages is invoked:

```
FTPEndRetrievalOfMessages: PROCEDURE [ftpuser: FTPUser];

    FTPError: ERROR [{connectionTimedOut, connectionClosed,
      noRouteToNetwork, missingConnection, busyConnection,
      unopenedForMail}];
```

Appendix C: Client Listener Appendages

C.1. Description of the Option

An Ftp Listener is (at least in principle) a process, distinct from its client, that creates local Ftp Servers in response to connection requests from remote Ftp Users. Ftp Servers are also processes and several can coexist within the local host. When a remote Ftp User terminates its dialogue with a local Ftp Server, the Server destroys itself. In the absence of more specific instructions from the client, this background activity continues, unattended and unobserved, until the client orders its termination via a call to **FTPDestroyListener**.

If it wishes, however, the client can monitor or even influence the activity of the local Ftp Listener and any Servers it creates by providing procedures, called *listener appendages*, that are given control by Ftp at key points in the Listener's operation. By means of such appendages, a client can, for example:

1. Handle ERROR signals raised within an Ftp Server.
2. Place an upper bound on the number of Ftp Servers that may coexist within the local host.
3. Control access to the local file system on a per-host or per-network basis.
4. Maintain a log of Listener/Server activity.

The Ftp implementation includes one set of listener appendages which are used by default if the client fails to supply its own. The default appendages supplied by Ftp are declared as PUBLIC procedures and are exported as part of the Ftp interface. The client can therefore implement certain appendages while relying on Ftp for others, or use the Ftp implementations as building blocks for its own implementations. For example, a client could log Server errors by supplying an implementation of the **NoteServerError** procedure (described in Section C.4) that records the error and then calls Ftp's **NoteServerError** procedure to decide whether or not to attempt a recovery.

C.2. Exercising the Option

The client exercises the option described above by means of the **clientListenerAppendages** parameter accepted by the **FTPCreateListener** procedure. This parameter is a POINTER to a RECORD containing the PROCEDUREs that constitute the appendages. Setting this parameter to NIL causes the Ftp Listener to employ the default appendages supplied by Ftp; setting it non-NIL causes the Listener to use the appendages contained in the record. Ftp does not copy the client's RECORD of procedures, which must therefore be preserved intact by the client until **FTPDestroyListener** is called:

```
ClientListenerAppendages: TYPE = POINTER TO ClientListenerAppendagesObject;
ClientListenerAppendagesObject: TYPE = RECORD [
```

```
    NoteServerCreation: PROCEDURE [purpose: Purpose, originOfRequest:
      PupDefs.PAddress] RETURNS [allowServerCreation: BOOLEAN, clientData:
      UNSPECIFIED],
    NoteServerError: PROCEDURE [clientData: UNSPECIFIED, ftpError: FtpError] RETURNS
      [allowServerContinuance: BOOLEAN],
    NoteServerDestruction: PROCEDURE [clientData: UNSPECIFIED]];

  Purpose: TYPE = {files, mail, filesAndMail};
  PupDefs.PAddress: TYPE = POINTER TO PupDefs.Address;
  PupDefs.Address: TYPE = RECORD [
    network: PupDefs.Net, host: PupDefs.Host, socket: PupDefs.Pair];
  PupDefs.Net: TYPE = [0..377B];
  PupDefs.Host: TYPE = [0..377B];
  PupDefs.Pair: TYPE = MACHINE DEPENDENT RECORD [first, second: CARDINAL];
```

C.3. General Characteristics

Each of the listener appendages employed by Ftp and suppliable by the client is described below. Statements that apply to all valid implementations of an appendage (i.e. both Ftp's implementation and any a client might supply) are rendered in the standard font. Statements that apply only to the default implementation supplied by Ftp are rendered in a smaller font (like this). The procedure declarations that accompany the descriptions below apply to both Ftp and client implementations.

Since they execute in a multi-processing environment controlled by a non-preemptive scheduler, client listener appendages should periodically surrender control by calling the procedure, **SchedDefs.ScheduleeYields**.

While client listener appendages may respond to exceptional conditions by signalling, such errors will not be communicated to the remote Ftp User in any meaningful way. The local Ftp Server will simply be destroyed and its connection to the remote Ftp User broken.

C.4. Server Encapsulation Appendages

Ftp or its client provides three procedures that serve to encapsulate an Ftp Server. The first, **NoteServerCreation**, is called upon the creation of each new Ftp Server. The procedure receives as parameters the **purpose** for which the Server is being created and the origin, **originOfRequest**, of the connection request. By returning with **allowServerCreation** set to FALSE, the procedure may abort the Server (before it can communicate with the remote Ftp User) and cause its connection to the remote Ftp User to be broken. If it confirms the Server's creation, the procedure may return an arbitrary parameter, **clientData**, which Ftp will present to other appendages it asks to manipulate the Server, as well as to any client file primitives associated with the Listener. The default implementation of this procedure always sets **allowServerCreation** to TRUE:

```
NoteServerCreation: PROCEDURE [purpose: Purpose, originOfRequest:
  PupDefs.PAddress] RETURNS [allowServerCreation: BOOLEAN, clientData:
  UNSPECIFIED];

Purpose: TYPE = {files, mail, filesAndMail};
PupDefs.PAddress: TYPE = POINTER TO PupDefs.Address;
PupDefs.Address: TYPE = RECORD [
  network: PupDefs.Net, host: PupDefs.Host, socket: PupDefs.Pair];
PupDefs.Net: TYPE = [0..377B];
PupDefs.Host: TYPE = [0..377B];
PupDefs.Pair: TYPE = MACHINE DEPENDENT RECORD [first, second: CARDINAL];
```

The second procedure, **NoteServerError**, is called whenever a signal reaches the top of an Ftp Server's control thread without having been handled. The procedure receives as parameters the **clientData** associated with the Server at creation and an indication, **ftpError**, of the nature of the error encountered. The procedure may abort the Server, causing its connection to the remote Ftp User to be broken, by returning with **allowServerContinuance** set to FALSE. The default implementation of this procedure aborts the Server only if the error represents an Ftp protocol violation or a broken network connection:

```
NoteServerError: PROCEDURE [clientData: UNSPECIFIED, ftpError: FtpError] RETURNS
  [allowServerContinuance: BOOLEAN];
```

The third procedure, **NoteServerDestruction**, is called whenever an Ftp Server is destroyed. The procedure receives as a parameter the **clientData** associated with the Server at creation. The default implementation of this procedure is a no operation:

```
NoteServerDestruction: PROCEDURE [clientData: UNSPECIFIED];
```

Appendix D: Client File Primitives

D.1. Description of the Option

An Ftp User or Server manipulates its local file system by means of a family of procedures called *file primitives.* This family includes, for example, procedures for enumerating the members of a local file group, for reading and writing the contents of local files, and for deleting and renaming local files. The Ftp implementation includes one set of primitives for manipulating the standard Alto file system. In the absence of more specific instructions from the client, this set becomes the basis upon which each new Ftp User or Server interacts with the local file system.

Rather than accept Ftp's interface to the standard Alto file system, the client may, if it wishes, provide its own file primitives to a particular Ftp User or Listener. By so doing, a client may use Ftp to, for example:

1. Interface to another local file system (e.g. Juniper).
2. Interface to a pseudo file system (e.g. a printer).
3. Produce or consume files on the fly (i.e. files that never exist on secondary storage).
4. Transform filenames (e.g. convert abstract filenames to concrete ones).
5. Control access to particular functions on a per-user or per-host basis.
6. Maintain a log of file system activity.

The Alto file primitives supplied by Ftp are declared as PUBLIC procedures and are exported as part of the Ftp interface. The client can therefore implement certain file primitives while relying on Ftp for others, or use the Ftp implementations as building blocks for its own implementations. For example, a client could log file access attempts by supplying an implementation of the **OpenFile** procedure (described in Section D.7) that records the event and then calls Ftp's **OpenFile** procedure to actually open the file.

D.2. Exercising the Option

The client exercises the option described above by means of the **clientFilePrimitives** parameter accepted by both the **FTPCreateUser** and **FTPCreateListener** procedures. This parameter is a POINTER to a RECORD containing the PROCEDUREs by which the newly created Ftp User or the Servers that result from the newly created Ftp Listener are to access the local file system. Setting this parameter to NIL causes the Ftp User or Servers to employ the standard Alto file primitives supplied as part of Ftp; setting it non-NIL causes them to use the file primitives contained in the record. Ftp does not copy the client's RECORD of procedures, which must therefore be preserved intact by the client until **FTPDestroyUser** or **FTPDestroyListener** is called:

```
ClientFilePrimitives: TYPE = POINTER TO ClientFilePrimitivesObject;
ClientFilePrimitivesObject: TYPE = RECORD [

    -- filename manipulation primitives
```

```
    DecomposeFilename: PROCEDURE [clientData: UNSPECIFIED, absoluteFilename:
      STRING, virtualFilename: VirtualFilename],
    ComposeFilename: PROCEDURE [clientData: UNSPECIFIED, absoluteFilename: STRING,
      virtualFilename: VirtualFilename],

      -- access control primitives
    InspectCredentials: PROCEDURE [clientData: UNSPECIFIED, status: Status, user,
      password: STRING],

      -- file enumeration primitives
    EnumerateFiles: PROCEDURE [clientData: UNSPECIFIED, files: STRING, processFile:
      PROCEDURE [UNSPECIFIED, STRING, FileInfo], processFileData: UNSPECIFIED],

      -- file transfer primitives
    OpenFile: PROCEDURE [clientData: UNSPECIFIED, file: STRING, mode: Mode,
      fileTypePlease: BOOLEAN] RETURNS [handle: UNSPECIFIED, fileType: FileType],
    ReadFile: PROCEDURE [clientData: UNSPECIFIED, handle: UNSPECIFIED, sendBlock:
      PROCEDURE [UNSPECIFIED, POINTER, CARDINAL], sendBlockData: UNSPECIFIED],
    WriteFile: PROCEDURE [clientData: UNSPECIFIED, handle: UNSPECIFIED, receiveBlock:
      PROCEDURE [UNSPECIFIED, POINTER, CARDINAL] RETURNS [CARDINAL],
      receiveBlockData: UNSPECIFIED],
    CloseFile: PROCEDURE [clientData: UNSPECIFIED, handle: UNSPECIFIED, aborted:
      BOOLEAN],

      -- file manipulation primitives
    DeleteFile: PROCEDURE [clientData: UNSPECIFIED, file: STRING],
    RenameFile: PROCEDURE [clientData: UNSPECIFIED, currentFile, newFile: STRING]];

  VirtualFilename: TYPE = POINTER TO VirtualFilenameObject;
  VirtualFilenameObject: TYPE = RECORD [device, directory, name, version: STRING];
  Status: TYPE = {primary, secondary};
  FileInfo: TYPE = POINTER TO FileInfoObject;
  FileInfoObject: TYPE = RECORD [
    fileType: FileType, byteSize: CARDINAL, byteCount: InlineDefs.LongCARDINAL,
    creationDate, writeDate, readDate, author: STRING];
  Mode: TYPE = {read, write};
  FileType: TYPE = {text, binary, unknown};
```

D.3. General Characteristics

Each of the file primitives employed by Ftp and suppliable by the client is described below. Statements that apply to all valid implementations of a primitive (i.e. both Ftp's implementation and any a client might supply) are rendered in the standard font. Statements that apply only to the default implementation supplied by Ftp are rendered in a smaller font (like this). The procedure declarations that accompany the descriptions below are those of the Ftp implementations. The reader will note, therefore, that file handles are declared as StreamDefs.DiskHandles, rather than as UNSPECIFIEDs, as would be appropriate in the more general case.

Each client file primitive receives as its first parameter the clientData supplied to Ftp via the FTPCreateUser or FTPCreateListener procedure (or the NoteServerCreation listener appendage). Since they execute in a multi-processing environment controlled by a non-preemptive scheduler, client file primitives should periodically surrender control by calling the procedure, SchedDefs.ScheduleeYields.

In accordance with standard Mesa exception handling conventions, file primitives report errors by signalling. The standard Ftp ERROR signal, **FTPError** (described elsewhere in this document), should be used for this purpose. Doing so enables the Ftp User or Server to communicate the error to the remote Ftp Server or User in a meaningful way. **FTPError**'s single parameter, an enumerated type, identifies the error. Each procedure's description below includes a list of the parameter values that seem, to the author, most appropriate for that primitive. Should the implementor of a client file primitive need a richer vocabulary, the author will gladly entertain suggestions for additional parameter values.

D.4. Filename Manipulation Primitives

Ftp or its client provides two procedures that serve to encapsulate Ftp's knowledge of local file naming conventions. The first, **DecomposeFilename**, used by both Ftp User and Server, constructs a **virtualFilename** from an **absoluteFilename**, verifying the syntax of the absolute filename as a side effect. The caller provides the STRINGs into which the components of the virtual filename are to be placed. Components that are without meaning to the local file system should be represented as zero-length (rather than NIL) STRINGs. The Alto implementation of this procedure sets the length of the device and directory components to zero, since these components are without meaning to the Alto file system; returns the name component always; and returns a version component if an exclamation point in the absolute filename signals its presence:

```
DecomposeFilename: PROCEDURE [clientData: UNSPECIFIED, absoluteFilename: STRING,
    virtualFilename: VirtualFilename];

VirtualFilename: TYPE = POINTER TO VirtualFilenameObject;
VirtualFilenameObject: TYPE = RECORD [device, directory, name, version: STRING];

    FTPError: ERROR [{illegalFilename}];
```

The second procedure, **ComposeFilename**, used by both Ftp User and Server, inverts the operation performed by the first procedure by constructing an **absoluteFilename** from a **virtualFilename**. The caller provides the STRING into which the absolute filename is to be placed. Unspecified components of the virtual filename are represented by zero-length (rather than NIL) STRINGs. Components that are present but without meaning to the local file system should be ignored. The Alto implementation of this procedure ignores the device and directory components, since they are without meaning to the Alto file system; insists upon a name component; and accepts a version component if it is present:

```
ComposeFilename: PROCEDURE [clientData: UNSPECIFIED, absoluteFilename: STRING,
    virtualFilename: VirtualFilename];

VirtualFilename: TYPE = POINTER TO VirtualFilenameObject;
VirtualFilenameObject: TYPE = RECORD [device, directory, name, version: STRING];

    FTPError: ERROR [{illegalFilename}];
```

D.5. Access Control Primitives

Ftp or its client provides one procedure for inspecting credentials presented by the remote client. This procedure, **InspectCredentials**, used only by Ftp Server, verifies the credentials to be implicitly used to access the remote files specified in subsequent file primitive calls. This procedure declares either the client's **primary** or **secondary** identity, which the Ftp Server is to associate with the first or second remote filename, respectively, in subsequent parameter lists. To rename a file, for example, the client must present *two* sets of credentials, one (**primary**) to access the file and the other (**secondary**) to access its new location. **InspectCredentials** verifies the existence of the indicated **user** and the correctness of the supplied **password**, and records for later use the fact that they were correctly supplied. **InspectCredentials** simply establishes the remote client's identity; other primitives determine whether that client has access of the appropriate kind to specific file system objects (e.g. the **DeleteFile** primitive described in Section D.8 verifies that the client has delete access to the target file before honoring the delete request). The Alto implementation of this procedure is a no operation:

```
InspectCredentials: PROCEDURE [clientData: UNSPECIFIED, status: Status, user,
    password: STRING];

Status: TYPE = {primary, secondary};

    FTPError: ERROR [{noSuchPrimaryUser, noSuchSecondaryUser,
      incorrectPrimaryPassword, incorrectSecondaryPassword}];
```

D.6. File Enumeration Primitives

Ftp or its client provides one procedure for enumerating the members of a local file group. This procedure, **EnumerateFiles**, used by both Ftp User and Server, supplies in turn to an Ftp-provided procedure, **processFile**, the (absolute) filename of each file in the local file group whose file group designator, **files**, is specified, along with various pieces of information about the file and an additional parameter, **processFileData**, supplied by Ftp. Unknown or unspecified file information should be specified as **unknown**, zero, or NIL, as appropriate. Ftp imposes no constraints on the order in which **EnumerateFiles** presents files to it; alphabetical order is one reasonable choice. The Alto implementation of this procedure recognizes in the file group designator, the two special characters, asterisk ('*), denoting zero or more arbitrary characters, and pound sign ('#), denoting exactly one arbitrary character. The procedure returns to its caller all those files in the local file system that satisfy this mask. No file information is returned. The files are presented in the order determined by **DirectoryDefs.EnumerateDirectory**:

```
EnumerateFiles: PROCEDURE [clientData: UNSPECIFIED, files: STRING, processFile:
    PROCEDURE [UNSPECIFIED, STRING, FileInfo], processFileData: UNSPECIFIED];

FileInfo: TYPE = POINTER TO FileInfoObject;
FileInfoObject: TYPE = RECORD [
    fileType: FileType, byteSize: CARDINAL, byteCount: InlineDefs.LongCARDINAL,
    creationDate, writeDate, readDate, author: STRING];
FileType: TYPE = {text, binary, unknown};
```

```
FTPError: ERROR [{missingCredentials, requestedAccessDenied, illegalFilename,
    noSuchFile, fileDataError}];
```

D.7. File Transfer Primitives

Ftp or its client provides four procedures for transferring files to and from the local file system. The first, **OpenFile**, used by both Ftp User and Server, verifies either the existence of an old file (if **mode** is **read**) or the availability of space for a new one (if **mode** is **write**), establishes the remote client's access to it (in conjunction with the **InspectCredentials** procedure described in Section D.5), prepares the file to be read or written, and returns a handle to it. If **fileTypePlease** is TRUE (in which case **mode** will be **read**), the procedure also returns the **fileType**--**text** or **binary**--of the file being opened. The Alto implementation of this procedure attaches a byte stream to the file and returns its handle. If the file's type is requested, the procedure scans the file until it encounters a byte with the high-order bit set to one (in which case the file is classified as **binary**) or reaches the end of the file (in which case the file is classified as **text**):

```
OpenFile: PROCEDURE [clientData: UNSPECIFIED, file: STRING, mode: Mode,
    fileTypePlease: BOOLEAN] RETURNS [handle: StreamDefs.DiskHandle, fileType:
    FileType];

Mode: TYPE = {read, write};
FileType: TYPE = {text, binary, unknown};

    FTPError: ERROR [{missingCredentials, requestedAccessDenied, illegalFilename,
        noSuchFile, noRoomForFile, fileDataError}];
```

The second procedure, **ReadFile**, used by both Ftp User and Server, transmits to the remote Ftp Server or User the contents of the file (previously opened for read) whose **handle** is specified. **ReadFile** supplies in turn to an Ftp-provided procedure, **sendBlock**, the location and length in bytes of successive segments of the file, along with an additional parameter, **sendBlockData**, supplied by Ftp. After the entire file has been output in this manner, **ReadFile** signals end of file by calling **sendBlock** a final time with a byte count of zero. The Alto implementation of this procedure simply allocates a buffer, reads successive blocks of the file from the disk stream into the buffer and presents them to **sendBlock**, and then releases the buffer:

```
ReadFile: PROCEDURE [clientData: UNSPECIFIED, handle: StreamDefs.DiskHandle,
    sendBlock: PROCEDURE [UNSPECIFIED, POINTER, CARDINAL], sendBlockData:
    UNSPECIFIED];

    FTPError: ERROR [{fileDataError}];
```

The third procedure, **WriteFile**, used by both Ftp User and Server, accepts from the remote Ftp Server or User the contents of the file (previously opened for write) whose **handle** is specified. This procedure receives successive segments of the file in turn from an Ftp-provided procedure, **receiveBlock**. **WriteFile** supplies **receiveBlock** with the location and length *in words* of a buffer into which the next segment may be placed, along with an additional parameter, **receiveBlockData**, supplied by Ftp; **receiveBlock** returns the segment left-adjusted in the buffer, along with its length *in bytes*. After the entire file has been

input in this manner, receiveBlock signals end of file by returning a byte count of zero. The Alto implementation of this procedure simply allocates a buffer, reads successive blocks of the file into the buffer and appends them to the disk stream, and then releases the buffer:

```
WriteFile: PROCEDURE [clientData: UNSPECIFIED, handle: StreamDefs.DiskHandle,
    receiveBlock: PROCEDURE [UNSPECIFIED, POINTER, CARDINAL] RETURNS
    [CARDINAL], receiveBlockData: UNSPECIFIED];

        FTPError: ERROR [{noRoomForFile, fileDataError}];
```

The fourth procedure, CloseFile, used by both Ftp User and Server, closes the previously opened file whose handle is specified after the contents of the file have been transmitted to or from the remote Ftp Server or User. If the transfer had to be aborted for some reason, that fact is indicated to the procedure. If the transfer was aborted and the local file was being written, the procedure should discard the partial file. The Alto implementation of this procedure simply destroys the disk stream and then deletes the file if the transfer was aborted and the stream was being written:

```
CloseFile: PROCEDURE [clientData: UNSPECIFIED, handle: StreamDefs.DiskHandle,
    aborted: BOOLEAN];

        FTPError: ERROR [{noRoomForFile, fileDataError}];
```

D.8. File Manipulation Primitives

Ftp or its client provides two procedures for manipulating existing local files. The first, DeleteFile, used only by Ftp Server, deletes the specified local file, reclaiming the space it occupied on secondary storage. The Alto implementation of this procedure simply deletes the file:

```
DeleteFile: PROCEDURE [clientData: UNSPECIFIED, file: STRING];

        FTPError: ERROR [{missingCredentials, requestedAccessDenied, illegalFilename,
            noSuchFile, fileDataError}];
```

The second procedure, RenameFile, used only by Ftp Server, renames the local file whose current name, currentFile, is specified, assigning it the new name, newFile. The Alto implementation of this procedure creates a new file (with the appropriate name), copies the contents of the file to it, and then deletes the original file:

```
RenameFile: PROCEDURE [clientData: UNSPECIFIED, currentFile, newFile: STRING];

        FTPError: ERROR [{missingCredentials, requestedAccessDenied, illegalFilename,
            noSuchFile, noRoomForFile, fileDataError}];
```

Appendix E: Sample Configuration and Program

E.1. Introduction

The sample configuration and program presented below illustrate the Ftp's use. The reader is referred to Appendix F for the location of the necessary files. This stand-alone program retrieves a single file from a remote file system.

E.2. Sample Configuration

The client must include in its configuration the Ftp Package, the Pup Package, and a Scheduler. In the sample configuration below, just the Ftp User code is included, since the sample program creates no Ftp Listener:

```
FTPSampleConfiguration: CONFIGURATION
  IMPORTS
    DirectoryDefs, DoubleDefs, FrameDefs, ImageDefs, IODefs,
    ProcessDefs, SegmentDefs, StatsDefs, StreamDefs, StringDefs,
    SystemDefs, TrapDefs
  CONTROL FTPSample =
  BEGIN

-- sample program
  FTPSample;

-- required packages
  FTPUser;
  Pup;
  Sched;

  END. -- of FTPSampleConfiguration
```

E.3. Sample Program

The sample program first initializes Ftp; creates an Ftp User, using the default client file primitives supplied by Ftp; extracts the login user name and password from the Alto operating system and uses them as its credentials; opens a connection to IFS; retrieves the file, FtpServer.Bcd, from the remote file system; closes the connection to IFS; destroys the Ftp User; and finalizes Ftp:

```
DIRECTORY
  FTPDefs:      FROM "FTPDefs",
  OsStaticDefs: FROM "OsStaticDefs",
  PupDefs:      FROM "PupDefs",
  StringDefs:   FROM "StringDefs";

FTPSample: PROGRAM
  IMPORTS FTPDefs, PupDefs, StringDefs =
  BEGIN

-- variables
```

```
ftpuser: FTPDefs.FTPUser;
user: STRING ← [40];
password: STRING ← [40];

-- initialize pup and ftp packages
FTPDefs.FTPInitialize[];

-- create ftp user
ftpuser ← FTPDefs.FTPCreateUser[NIL, NIL];

-- set credentials to login user and password
StringDefs.BcplToMesaString[OsStaticDefs.OsStatics↑.UserName, user];
StringDefs.BcplToMesaString[OsStaticDefs.OsStatics↑.UserPassword, password];
IF user.length # 0 AND password.length # 0 THEN
  FTPDefs.FTPSetCredentials[ftpuser, primary, user, password];

-- open connection, retrieve file, close connection
FTPDefs.FTPOpenConnection[ftpuser, "IFS", files];
[] ← FTPDefs.FTPRetrieveFile[ftpuser,
  "FtpServer.Bcd", "<MesaPup>Ftp2.0>FtpServer.Bcd"];
FTPDefs.FTPCloseConnection[ftpuser];

-- destroy ftp user
FTPDefs.FTPDestroyUser[ftpuser];

-- finalize pup and ftp packages
FTPDefs.FTPFinalize[];

END. -- of FTPSample
```

Appendix F: Production Configurations and File Locations

The following production Ftp configurations presently exist; others will be created as the need arises:

FTPUser: Ftp User, file primitives only (i.e. the primitives of Sections 2 and 3 and Appendix A). In **FTPOpenConnection**, **purpose** must be **files**.

FTPServer: Ftp Listener, file primitives only (i.e. the primitives of Sections 2 and 4). In **FTPCreateListener**, **purpose** must be **files**.

FTPSystem: Ftp User and Listener, file primitives only (i.e. the primitives of Sections 2, 3, and 4, and Appendix A). In **FTPOpenConnection** and **FTPCreateListener**, **purpose** must be **files**.

MTPUser: Ftp User, mail primitives only (i.e. the primitives of Sections 2 and 3.1-3.3 and Appendix B). In **FTPOpenConnection**, **purpose** must be **mail**.

Ftp 2.0 resides at IFS as the following files:

Documentation:

[IFS]<MesaPup>Ftp2.0>FtpSpecification1.Memo & .Ears
[IFS]<MesaPup>Ftp2.0>FtpSpecification2.Memo & .Ears
[IFS]<MesaPup>Ftp2.0>FtpSpecification3.Memo & .Ears

Object files:

[IFS]<MesaPup>Ftp2.0>FtpDefs.Bcd
[IFS]<MesaPup>Ftp2.0>FtpUser.Bcd & .Symbols
[IFS]<MesaPup>Ftp2.0>FtpServer.Bcd & .Symbols
[IFS]<MesaPup>Ftp2.0>FtpSystem.Bcd & .Symbols
[IFS]<MesaPup>Ftp2.0>MtpUser.Bcd & .Symbols

Source files:

[IFS]<MesaPup>Ftp2.0>FtpSources.Dm
[IFS]<MesaPup>Ftp2.0>FtpSubsystemSources.Dm
[IFS]<MesaPup>Ftp2.0>FtpUtilities.Dm

Ftp 2.0 requires (that is, imports) the 17 Oct 1977 version of **PupDefs** and **SchedDefs** (described in [6, 7]) and version 3.0 of **DirectoryDefs**, **DoubleDefs**, **SegmentDefs**, **StreamDefs**, **StringDefs**, **SystemDefs**, and **TrapDefs**. (**FTPUser** does not need **SchedDefs**. **MTPUser** does not need **DirectoryDefs**, **SchedDefs**, **SegmentDefs**, or **StreamDefs**.)