Mesa FTP Functional Specification

Version 6.0
October, 1980

XEROX

SYSTEMS DEVELOPMENT DEPARTMENT
3333 Coyote Hill Road / Palo Alto / California 94304

Table of Contents

Preface

This document details the procedural interface to Version 6.0 of the Mesa File Transfer Package (FTP). FTP 6.0 provides a wide range of capabilities, only a subset of which are typically required by any particular application. To use FTP for simple file transfer, for example, the programmer need only consult the body of this document and Appendices H and I. Comments, bug reports, suggestions for change or addition, and cries for help should be addressed to your support group.

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 (that is, pages of files), or CopyDisk, which trades in larger ones (that is, 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 [1].

In addition to providing file-related services, FTP provides primitives for delivering mail to and retrieving mail from remote mailboxes. FTP is thus also a means for accessing mailboxes on Maxc and any other host that implements the Mail Transfer Protocol (MTP) described in [2, 3].

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 one or more resident *FTP Listeners* in response to connection requests from distant FTP Users. Each Server is spawned as a separate Mesa process and competes for system resources with other local processes under the control of the 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 has no real external interface. In the procedure descriptions presented throughout this document, the terms *local* and *remote* distinguish the host containing the described procedure from the distant host to which the first host is connected.

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 which 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 (for example, if **FTPSetFilenameDefaults** is never 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, which 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, extension, and/or version 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 signal, **FTPError**. In rare circumstances, the Mesa Runtime System may generate errors or signals that are neither handled by FTP nor reissued as **FTPError**s. This fact complicates the client implementation in theory but typically not in practice. Although declared as a SIGNAL, **FTPError** is generally raised as an ERROR and so cannot be resumed by the client. However, exceptional conditions reported to a server backstop are issued as SIGNALs and *can* be resumed by the client, as described in Section 4.2. FTP restores itself to a consistent state after every error (in response to the UNWIND signal). Therefore if the client's connection is timed out by the remote FTP Server, for example, the client can close and then reopen the connection without first having to destroy and recreate the local FTP User. Warning: don't call FTP again from within a catch phrase.

FTPError has two parameters that pinpoint the exceptional condition encountered by FTP: an enumerated type, **ftpError**, to be interpreted by the client; and a STRING, **message**, to be interpreted by the human user. All error messages issued by FTP are centralized in a single FTPAccessories module (see Appendix I). To avoid incurring the space overhead associated with such strings, the programmer can omit this module from his configuration, causing FTP to supply a NIL whenever it would otherwise obtain a **message** from FTPAccessories.

Exceptional conditions reported via **FTPError** include not only those explicitly detected by FTP but also those that originate as signals within the local file, mail, or communication system. Furthermore, many of the errors reported by a local FTP User are actually detected by the remote FTP Server. In such cases, the User relays to the client the message supplied by the Server. If the Server provides no message, the User supplies the appropriate message from FTPAccessories in its place. Regardless of source, **message** STRINGs presented to the client may be assumed to remain intact only until the signal is unwound.

The errors that FTP may report to the client are summarized below and are explained in detail in Appendix A. The most prominent exceptional conditions which may be encounted by particular procedures are also listed with the descriptions of those procedures throughout this document. The errors classed below as *protocol, internal,* or *unidentified* errors theoretically can be generated 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: SIGNAL [ftpError: FtpError, message: STRING];

FtpError: TYPE = {

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

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

 -- file errors
 illegalFilename, noSuchFile, fileAlreadyExists, fileBusy, noRoomForFile,
   fileDataError,

 -- dump errors
 errorBlockInDumpFile, unrecognizedDumpFileBlock, dumpFileBlockTooLong,
   dumpFileCheckSumInError,

 -- mail errors
 noValidRecipients, noSuchMailbox, noSuchForwardingHost, noSuchDmsName,
 -- client errors
 filePrimitivesNotSpecified, mailPrimitivesNotSpecified,
   communicationPrimitivesNotSpecified, filesModuleNotLoaded,
   mailModuleNotLoaded, noConnectionEstablished,
   connectionAlreadyEstablished, connectionNotOpenedForFiles,
   connectionNotOpenedForMail, illegalProcedureCallSequence,
   fileGroupDesignatorUnexpected, filenameUnexpected,

 -- protocol errors
 protocolVersionMismatch, functionNotImplemented,
   inputDiscontinuityUnexpected, outputDiscontinuityUnexpected,
   illegalProtocolSequence, protocolParameterListMissing,
   illegalProtocolParameterList, unrecognizedProtocolParameter,
   missingProtocolParameter, duplicateProtocolParameter,
   illegalBooleanParameter, illegalFileAttribute, illegalFileType,
   unrecognizedProtocolErrorCode, noSuchRecipientNumber,
   duplicateMailboxException, unrecognizedMailboxExceptionErrorCode,
   missingMessageLength, messageLongerThanAdvertised,
   messageShorterThanAdvertised,

 -- internal errors
 stringTooLong, queueInconsistent, unexpectedEndOfFile,

 -- unidentified errors
 unidentifiedTransientError, unidentifiedPermanentError, unidentifiedError};


CommunicationError: TYPE = FtpError[noSuchHost..noNameLookupResponse];
CredentialError: TYPE = FtpError[credentialsMissing..requestedAccessDenied];
FileError:   TYPE = FtpError[illegalFilename..fileDataError];
DumpError: TYPE = FtpError[errorBlockInDumpFile..dumpFileCheckSumInError];
MailError: TYPE = FtpError[noValidRecipients..noSuchDmsName];
ClientError: TYPE = FtpError[filePrimitivesNotSpecified..filenameUnexpected];
ProtocolError: TYPE =
   FtpError[protocolVersionMismatch..messageShorterThanAdvertised];
InternalError: TYPE = FtpError[stringTooLong..unexpectedEndOfFile];
UnidentifiedError: TYPE =
   FtpError[unidentifiedTransientError..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. *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 disposing of FTP's internal data structures after the client has destroyed any Users and Listener it created. *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 local FTP Users, several of which can coexist within a single host. The first procedure, **FTPCreateUser**, creates a new FTP User founded upon the specified file and communication systems; the FTP User will access those systems *solely* by means of the specified **filePrimitives** and **communicationPrimitives**, respectively. The procedure returns a handle, **ftpuser**, to the newly created FTP User, which the client must retain and later present to any of the other procedures described in this section it invokes. The **ftpuser** is a pointer to a private record containing all of the state information the FTP User requires to function properly:

```
FTPCreateUser: PROCEDURE [filePrimitives: FilePrimitives,
    communicationPrimitives: CommunicationPrimitives] RETURNS [ftpuser:
    FTPUser];

FilePrimitives: TYPE = POINTER TO FilePrimitivesObject;
FilePrimitivesObject: TYPE = RECORD [...];
CommunicationPrimitives: TYPE = POINTER TO CommunicationPrimitivesObject;
CommunicationPrimitivesObject: TYPE = RECORD [...];
FTPUser: TYPE = POINTER TO FTPUserObject;
FTPUserObject: PRIVATE TYPE = RECORD[...];
```

Exceptions: communicationPrimitivesNotSpecified.

The **filePrimitives** parameter supplied by the client is a pointer to a public record containing descriptors for all of the procedures an FTP User requires to manipulate the local file system. An implementation for the Alto file system is provided as part of FTP, as described in Section 5.1. The client is also free to supply its own file primitives. The reader is referred to Appendix E for detailed motivation for and instruction in the use of this option.

The **communicationPrimitives** parameter supplied by the client is a pointer to a public record containing descriptors for all of the procedures an FTP User requires to manipulate the local communication system. An implementation for the Pup communication system is provided as part of FTP, as described in Section 5.3. The client is also free to supply its own communication primitives. The reader is referred to Appendix G 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 allocated to it and, if necessary, closing its connection to the remote FTP Server (which may involve a delay as control messages are exchanged via the network):

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

FTPUser: TYPE **=** POINTER TO **FTPUserObject;**
FTPUserObject: PRIVATE TYPE **=** RECORD**[...];**

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** for the **purpose** of manipulating either remote **files**, **mail**, or **filesAndMail**. A single FTP User can support only one open connection at a time. The FTP User makes contact with the appropriate server(s) based upon the client's stated **purpose**. Unless **remoteInsignia** is NIL, the procedure returns the *insignia* of the remote FTP Server. An insignia is textual information (for example, the Server's host name, version number, and date of installation) which the client may wish to present to its human user. File and mail servers supply separate insignias. If **purpose** is **filesAndMail**, FTP concatenates the two (separated by a carriage return), unless the two insignias are identical, in which case it returns just one of them:

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

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

Exceptions: noSuchHost, connectionTimedOut, connectionRejected, connectionClosed, noRouteToNetwork, noNameLookupResponse, filePrimitivesNotSpecified, filesModuleNotLoaded, mailModuleNotLoaded, connectionAlreadyEstablished.

The second procedure, **FTPRenewConnection**, prevents a previously established but long inactive connection from being timed out and broken by the remote FTP Server. A Mesa FTP Server, for example, will break its connection to a remote FTP User after three minutes of inactivity. To preserve its connection, the client must punctuate such idle periods with calls to **FTPRenewConnection** (which may be invoked at any time, except during a remote file enumeration or dump file inventory, or during mail delivery or retrieval). 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];**

Exceptions: connectionTimedOut, connectionClosed, noRouteToNetwork, noConnectionEstablished, illegalProcedureCallSequence.

The third procedure, **FTPCloseConnection**, breaks a previously established connection to a remote FTP Server. Redundant calls 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 **primary** or **secondary** credentials--**user** and **password**--that are implicitly to be employed in all subsequent procedure calls that attempt to access a remote file or mailbox. The credentials (that is, the contents of the STRINGs) are saved by **FTPSetCredentials** and transmitted to the remote FTP Server for inspection only when access to the remote file system is actually attempted. Primary credentials typically identify the user upon whose behalf the access is attempted (*a la* the Tenex Login command) while secondary credentials, when necessary, usually identify another area of the file system--in addition to the user's own workspace--to which the user claims access (*a la* the Tenex Connect command). By setting both **user** and **password** to NIL, the client effectively retracts any previously specified credentials:

> **FTPSetCredentials: PROCEDURE [ftpuser: FTPUser, status: Status, user, password: STRING];**
>
> **Status: TYPE = {primary, secondary};**

The second procedure, **FTPSetFilenameDefaults**, specifies the **primary** or **secondary virtualFilename** that is implicitly to be employed (in combination with an explicitly specified primary or secondary *absolute* filename) in all subsequent procedure calls that attempt to manipulate a remote file. The reader is referred to Section 1.3 for a discussion of virtual filenames and their use. The virtual filename (that is, the contents of the RECORD and STRINGs) is saved by **FTPSetFilenameDefaults** and transmitted to the remote FTP Server for interpretation only when access to the remote file system is actually attempted. In this context, the adjectives *primary* and *secondary* refer to the first and second remote filenames in a procedure's argument list. To rename a file, for example, the client must, in general, specify *two* virtual filenames, one (**primary**) specifying the file to be renamed, the other (**secondary**) its new name. By setting one or more components of the **virtualFilename** to NIL, the client effectively declines to specify, or retracts previously specified component value(s):

> **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, **FTPEnumerateFiles**, for enumerating the members of a remote file group. For each file in the group whose file group designator, **remoteFiles**, is specified, the procedure supplies to a client-provided procedure, **processFile**, the client's **processFileData**, the file's absolute and virtual filenames, and a variety of other file information (**FileInfo**). The information provided in **FileInfo** is only as reliable as the remote file server. Bravo and/or the Alto file system are notoriously unreliable about file lengths (**byteCount**). The order in which filenames are presented to the client is host-dependent; alphabetical order is typical. The reader is referred to Section 1.3 for a discussion of virtual filenames and their use. Unknown or unspecified file information is rendered as **unknown**, zero, or NIL, as appropriate:

```
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: LONG CARDINAL,
   creationDate, writeDate, readDate, author: STRING];
FileType: TYPE = {text, binary, unknown};
```

Exceptions: connectionTimedOut, connectionClosed, noRouteToNetwork, credentialsMissing, noSuchPrimaryUser, noSuchSecondaryUser, incorrectPrimaryPassword, incorrectSecondaryPassword, requestedAccessDenied, illegalFilename, noSuchFile, noRoomForFile, fileDataError, noConnectionEstablished, connectionNotOpenedForFiles, illegalProcedureCallSequence.

The **intent** parameter supplied by the client declares the manner in which the client expects to manipulate the files whose names are presented to it. This information enables the FTP User to select intelligently from among several possible protocol strategies for effecting the enumeration. Since most such 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 simply seeks the names of (and file information for) the members of the designated file group (for presentation to a human user, for example) and intends to manipulate the files in no other way during the course of the enumeration. More specifically, the client declares (and FTP insures) 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 declares (and FTP insures) 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 declares (and FTP insures) 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 declares (and FTP insures) 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 a synonym for **unspecified** (described below), and 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 to any local FTP User procedures it chooses, since FTP will have spooled all of the filenames onto a local scratch file (which FTP promptly deletes once it has served its purpose) before any are presented 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 in the remote file system a copy of the **localFile** the name of which and **fileType**--**text** or **binary**--are specified, creating a new **remoteFile** with the indicated name and returning its size in bytes, **byteCount**. The **fileType** parameter supplied by the client is used by the remote FTP Server in determining how to store the file in its file system (for example, on Maxc, text files are stored as 7-bit bytes, binary files as 8-bit bytes). The client may report the file's type as **unknown**, in which case the local FTP User will attempt to determine it. The reader is referred to the discussion of the **OpenFile** procedure in Appendix E for a description of the algorithm used in making this determination:

FTPStoreFile: PROCEDURE [ftpuser: FTPUser, localFile, remoteFile: STRING, fileType: FileType] RETURNS [byteCount: LONG CARDINAL];

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

Exceptions: connectionTimedOut, connectionClosed, noRouteToNetwork, credentialsMissing, noSuchPrimaryUser, noSuchSecondaryUser, incorrectPrimaryPassword, incorrectSecondaryPassword, requestedAccessDenied, illegalFilename, noSuchFile, fileAlreadyExists, fileBusy, noRoomForFile, fileDataError, noConnectionEstablished, connectionNotOpenedForFiles, illegalProcedureCallSequence, filenameUnexpected.

Once the file has been stored, the client may invoke the **FTPNoteFilenameUsed** primitive described in Appendix D to determine the fully qualified absolute and/or virtual filename used by the remote FTP Server. 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 E (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 B.

The second procedure, **FTPRetrieveFile**, stores in the local file system a copy of the **remoteFile** whose name and **fileType**--**text** or **binary**--are specified, creating a new **localFile** with the indicated name and returning its size in bytes, **byteCount**. In rare cases, the **fileType** parameter supplied by the client is used by the remote FTP Server to disambiguate between two like-named files of different types. The client may (and often does) report the file's type as **unknown**, in which case the remote FTP Server must select a file without it:

FTPRetrieveFile: PROCEDURE [ftpuser: FTPUser, localFile, remoteFile: STRING, fileType: FileType] RETURNS [byteCount: LONG CARDINAL];

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

Exceptions: connectionTimedOut, connectionClosed, noRouteToNetwork, credentialsMissing, noSuchPrimaryUser, noSuchSecondaryUser, incorrectPrimaryPassword, incorrectSecondaryPassword, requestedAccessDenied, illegalFilename, noSuchFile, fileBusy, noRoomForFile, fileDataError, errorBlockInDumpFile, unrecognizedDumpFileBlock, dumpFileBlockTooLong, dumpFileCheckSumInError, noConnectionEstablished, connectionNotOpenedForFiles, illegalProcedureCallSequence, fileGroupDesignatorUnexpected, filenameUnexpected.

Once the file has been retrieved, the client may invoke the **FTPNoteFilenameUsed** primitive described in Appendix D.3 to determine the fully qualified absolute and/or virtual filename used by the remote FTP Server. 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 loading of remote dump files, as described in Appendix B.

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];**

> Exceptions: connectionTimedOut, connectionClosed, noRouteToNetwork, credentialsMissing, noSuchPrimaryUser, noSuchSecondaryUser, incorrectPrimaryPassword, incorrectSecondaryPassword, requestedAccessDenied, illegalFilename, noSuchFile, fileBusy, fileDataError, noConnectionEstablished, connectionNotOpenedForFiles, illegalProcedureCallSequence, fileGroupDesignatorUnexpected, filenameUnexpected.

Once the file has been deleted, the client may invoke the **FTPNoteFilenameUsed** primitive described in Appendix D to determine the fully qualified absolute and/or virtual filename used by the remote FTP Server. 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 the current name for which is specified by **currentFile**, assigning it the new remote name specified by **newFile**:

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

> Exceptions: connectionTimedOut, connectionClosed, noRouteToNetwork, credentialsMissing, noSuchPrimaryUser, noSuchSecondaryUser, incorrectPrimaryPassword, incorrectSecondaryPassword, requestedAccessDenied, illegalFilename, noSuchFile, fileAlreadyExists, fileBusy, noRoomForFile, fileDataError, noConnectionEstablished, connectionNotOpenedForFiles, illegalProcedureCallSequence, filenameUnexpected.

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 local FTP Listeners, several of which can coexist within a single host. The first procedure, **FTPCreateListener**, creates a new FTP Listener for the **purpose** of allowing remote manipulation of either local **files**, **mail**, or **filesAndMail**. The FTP Listener monitors the appropriate well-known socket(s) based upon the client's stated **purpose**. The newly created FTP Listener and any FTP Servers it creates are founded upon the specified file, mail, and communication systems and will access those systems *solely* by means of the specified **filePrimitives**, **mailPrimitives,** and **communicationPrimitives**, respectively. A Server is destroyed when the User breaks its connection to it or after four minutes of inactivity. The procedure returns a handle, **ftplistener**, to the newly created FTP Listener, which the client must retain and later present to the **FTPDestroyListener** procedure described below. The **ftplistener** is a pointer to a private record containing all of the state information the FTP Listener requires to function properly:

```
FTPCreateListener: PROCEDURE [purpose: Purpose, filePrimitives: FilePrimitives,
    mailPrimitives: MailPrimitives, communicationPrimitives:
    CommunicationPrimitives,
    backstopServer: POINTER TO BackstopServer,
    backstopServerData: UNSPECIFIED,
    filter: PROCEDURE [STRING, Purpose] _ RejectNothing]
    RETURNS [ftplistener: FTPListener];
RejectNothing: PROCEDURE [STRING, Purpose];
RejectThisConnection: ERROR [error: STRING];

Purpose: TYPE = {files, mail, filesAndMail};
FilePrimitives: TYPE = POINTER TO FilePrimitivesObject;
FilePrimitivesObject: TYPE = RECORD [...];
MailPrimitives: TYPE = POINTER TO MailPrimitivesObject;
MailPrimitivesObject: TYPE = RECORD [...];
CommunicationPrimitives: TYPE = POINTER TO CommunicationPrimitivesObject;
CommunicationPrimitivesObject: TYPE = RECORD [...];
BackstopServer: TYPE = PROCEDURE [backstopServerData: UNSPECIFIED, purpose:
    SingularPurpose, originOfRequest, localInsignia: STRING, server: PROCEDURE];
SingularPurpose: TYPE = Purpose[files..mail];
FTPListener: TYPE = POINTER TO FTPListenerObject;
FTPListenerObject: PRIVATE TYPE = RECORD[...];
```

Exceptions: filePrimitivesNotSpecified, mailPrimitivesNotSpecified, communicationPrimitivesNotSpecified, filesModuleNotLoaded, mailModuleNotLoaded.

The **filePrimitives** parameter supplied by the client is a pointer to a public record containing descriptors for all of the procedures an FTP User requires to manipulate the local file system. An implementation for the Alto file system is provided as part of FTP, as described in Section 5.1. The client is also free to supply its own file primitives. The reader is referred to Appendix E for detailed motivation for and instruction in the use of this option.

The **mailPrimitives** parameter supplied by the client is a pointer to a public record containing descriptors for all of the procedures an FTP Server requires to manipulate the local mail system. A primitive set for a simple-minded mail system based upon the file primitives specified by the client are provided as part of the FTP implementation, as described in Section 5.2. The client is also free to supply its own mail primitives. The reader is referred to Appendix F for detailed motivation for and instruction in the use of this option.

The **communicationPrimitives** parameter supplied by the client is a pointer to a public record containing descriptors for all of the procedures an FTP User requires to manipulate the local communication system. An implementation for the Pup communication system is provided as part of FTP, as described in Section 5.3. The client is also free to supply its own communication primitives. The reader is referred to Appendix G for detailed motivation for and instruction in the use of this option.

The **backstopServer** and **backstopServerData** parameters optionally supplied (rather then set to NIL) enable the client to monitor the creation, execution, and destruction of FTP Servers. The reader is referred to Section 4.2 for detailed motivation for and instruction in the use of this option.

An optional **filter** procedure may be used to reject undesired connections. This is useful to keep a server from crashing because it runs out of resources. (The default doesn't reject anything.) To reject a connection, **filter** should raise the **ERROR RejectThisConnection**. The text will be be returned to the machine attempting to establish the connection. If a connection is rejected, **backstopServer** is never called. If **filter** returns the connection will be accepted.

The second procedure, **FTPDestroyListener**, destroys a previously created FTP Listener, either destroying any of its Servers that remain in existence or waiting for them to terminate normally, as directed by **abortServers**, and reclaiming any local resources allocated to it. Destroying an FTP Server requires closing its connection to the remote FTP User (which may involve a delay as control messages are exchanged via the network):

```
FTPDestroyListener: PROCEDURE [ftplistener: FTPListener, abortServers:
   BOOLEAN];

FTPListener: TYPE = POINTER TO FTPListenerObject;
FTPListenerObject: PRIVATE TYPE = RECORD[...];
```

4.2. Server Monitoring

At least in principle, an FTP Listener is a process that creates local FTP Servers in response to connection requests from remote FTP Users. Each FTP Server is also a process. When a remote FTP User terminates its dialogue with a local FTP Server, the latter 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, to a limited extent, influence the activity of local FTP Servers by supplying to **FTPCreateListener** a *server backstop* procedure to sit directly above each FTP Server in its thread of control. By means of such a procedure, a client can, for example:

1. dictate the handling of exceptional conditions encountered by an FTP Server.
2. control access to the local file or mail system on a per-host or per-network basis.
3. maintain a log of Listener/Server activity.

The client's server backstop is called upon to oversee the execution of each new FTP Server. As parameters, it receives the **backstopServerData** supplied by the client in its call to **FTPCreateListener**; the **purpose**--either **files** or **mail** (never **filesAndMail**)--for which the new FTP Server is being created; the remote host, **originOfRequest**, on which the requesting FTP User resides; the **localInsignia** of the FTP Server, which the remote FTP User will return to its client via **FTPOpenConnection**; and a procedure, **server**, that represents the top-most level of the FTP Server itself:

BackstopServer: TYPE **=** PROCEDURE **[backstopServerData:** UNSPECIFIED**, purpose: SingularPurpose, originOfRequest, localInsignia:** STRING**, server:** PROCEDURE**];**

The server backstop is called immediately prior to the birth of each new FTP Server. It should initiate the **server** by calling it. Before doing so, it may modify or replace the **localInsignia** to be returned to the remote client. The insignia is textual information (for example, the Server's host name, version number, and date of installation) which the remote client may wish to present to its human user.

The server backstop sits immediately atop the **server** throughout its lifetime and, therefore, by means of an appropriate catchphrase, can note and/or influence the processing of each **FTPError** encountered by the **server**. If the backstop RESUMES such a SIGNAL, the **server** will abort the transaction that provoked the error, report the error to the remote FTP User, and await the initiation of another transaction. Before resuming the signal, the backstop may override (by modification) the message that will otherwise be reported to the remote FTP User. If the backstop CONTINUES the signal and returns to its caller, the **server** will terminate the remote FTP User's session, the connection will be closed, and the local FTP Server destroyed.

When the session is voluntarily terminated by the remote FTP User, the **server** will return to the server backstop. At that point, the backstop should return to its caller, and the local FTP Server will be destroyed.

The default server backstop provided by FTP simply catches and dispatches **FTPError**s. It CONTINUES the first **CommunicationError** or **ProtocolError**. Unless the catching of unidentified errors has been disabled via **FTPCatchUnidentifiedErrors**, it also CONTINUES the first **UnidentifiedError**. All other errors it RESUMES.

Notice that the interfaces do not provide any convient way for the client provided file system to interact with a particular instance of a server. For example, it is difficult for discover the address of the remote client so that it may be included in an error message. One possible solution to this problem is to provide a backstop procedure, and call it via the **SIGNAL**ing mechanisim. This requires catching of unidentified errors to be disabled.

5. FTP Support Systems

5.1. File Primitives

FTP provides a procedure for obtaining a file primitive set of the sort required by **FTPCreateUser** and **FTPCreateListener**. **AltoFilePrimitives**, returns a pointer to a public record containing descriptors for all of the procedures that an FTP User or Server requires to manipulate the standard Alto file system. *It, along with the file primitive set to which it provides access, is implemented as a separate module(s), which must be bound together as described in Appendix I*:

AltoFilePrimitives: PROCEDURE RETURNS **[filePrimitives: FilePrimitives];**

FilePrimitives: TYPE **=** POINTER TO **FilePrimitivesObject;**
FilePrimitivesObject: TYPE **=** RECORD **[...];**

Alternatively, the client can provide its own local file system interface by constructing the record of procedure descriptors it supplies to **FTPCreateUser** and/or **FTPCreateListener**. The reader is referred to Appendix E for detailed motivation for and instruction in the use of this option.

5.2. Mail Primitives

FTP provides one procedure, **SysMailPrimitives** (alias **SomeMailPrimitives**), for obtaining mail primitive sets of the sort required by **FTPCreateListener**. **SysMailPrimitives** returns a pointer to a public record containing descriptors for all of the procedures that an FTP Server requires to manipulate a simple-minded mail system implemented *by means of the file primitives supplied by the client*:

SysMailPrimitives, SomeMailPrimitives: PROCEDURE RETURNS **[mailPrimitives: MailPrimitives];**

MailPrimitives: TYPE **=** POINTER TO **MailPrimitivesObject;**
MailPrimitivesObject: TYPE **=** RECORD **[...];**

Alternatively, the client can provide its own local mail system interface by constructing the record of procedure descriptors it supplies to **FTPCreateListener**. The reader is referred to Appendix F for detailed motivation for and instruction in the use of this option.

5.3. Communication Primitives

FTP provides a procedure for obtaining a communication primitive set of the sort required by **FTPCreateUser** and **FTPCreateListener**. **PupCommunicationPrimitives**, returns a pointer to a public record containing descriptors for all of the procedures that an FTP User, Listener, or Server requires to manipulate the standard Pup communication system:

PupCommunicationPrimitives: PROCEDURE RETURNS **[communicationPrimitives: CommunicationPrimitives];**

```
CommunicationPrimitives: TYPE = POINTER TO CommunicationPrimitivesObject;
CommunicationPrimitivesObject: TYPE = RECORD [...];
```

Alternatively, the client can provide its own local communication system interface by constructing the record of procedure descriptors it supplies to **FTPCreateUser** and/or **FTPCreateListener**. The reader is referred to Appendix G for detailed motivation for and instruction in the use of this option.

References

1. John Shoch, "A File Transfer Protocol Using the BSP -- 4th edition," 15 July 1978, [Maxc1]<Pup>FtpSpec.press and FtpFigs.press
2. Ed Taft, "Pup Mail Transfer Protocol (Edition 6)," 11 February 1979, [Maxc1]<Pup>MailTransfer.press
3. Dave Crocker, John Vittal, Ken Pogran, and Austin Henderson, "Standard for the Format of ARPA Network Text Messages," 21 November 1977.

Appendix A: Error Summary

A.1. Introduction

Exceptional conditions encountered by FTP are reported to the client by means of a single signal, **FTPError**. **FTPError** has two parameters that pinpoint the error: an enumerated type, **ftpError**, to be interpreted by the client; and a STRING, **message**, to be interpreted by the human user. Listed alphabetically below are the values that **ftpError** can assume, along with descriptions of the causes of these errors and, where appropriate, their possible remedies.

A.2. Errors

communicationPrimitivesNotSpecified

A client error signalled by **FTPCreateUser** or **FTPCreateListener**. The **communicationPrimitives** parameter supplied to one of these procedures is NIL. Provided the corresponding program module has been bound into the client's configuration (see Appendix I), an acceptable value for this parameter is the one returned by the FTP procedure **PupCommunicationPrimitives**.

connectionAlreadyEstablished

A client error signalled by **FTPOpenConnection**. The specified **ftpuser** already supports an open connection. A local FTP User can simultaneously support at most one connection to a remote FTP Server. The client should either close the existing connection via **FTPCloseConnection** or create another FTP User via **FTPCreateUser**.

connectionClosed

A communication error signalled by many primitives. The connection to the remote FTP Server (or User) has been closed. It was either deliberately closed by the Server (User), or it was closed by the Server's (User's) communication system when the Server (User) crashed. The client should clear the local FTP User of the connection-closed condition via **FTPCloseConnection**, and then try to reestablish communication with (a new FTP Server at) the remote host by recalling **FTPOpenConnection**. A typical FTP Server will break off communication with a remote FTP User that allows its connection to the Server to remain idle for more than a few minutes. The client should, therefore, either punctuate such idle periods with calls to **FTPRenewConnection** or be prepared to reestablish the connection whenever it discovers it closed.

connectionNotOpenedForFiles

A client error signalled by most file primitives. A file operation has been attempted using a connection that was opened for mail operations only. The client should close the existing connection via **FTPCloseConnection** and reopen it via a call to **FTPOpenConnection** with **purpose** specified as either **files** or **filesAndMail**.

connectionNotOpenedForMail

A client error signalled by most mail primitives. A mail operation has been attempted using a connection that was opened for file operations only. The client should close the existing connection via **FTPCloseConnection** and reopen it via a call to **FTPOpenConnection** with **purpose** specified as either **mail** or **filesAndMail**.

connectionRejected

A communication error signalled by **FTPOpenConnection**. Either no FTP Listener is resident in the remote host, or that Listener is rejecting requests for new connections for lack of resources. At a later time, the client should reattempt connection via **FTPOpenConnection**.

connectionTimedOut

A communication error signalled by many primitives. Either the remote FTP Server (or User) has crashed, or its communication hardware has broken and effectively cut it off from the local FTP User (Server). The client should clear the local FTP User of the connection-timed-out condition via **FTPCloseConnection**, and at a later time try to reestablish communication with (a new FTP Server at) the remote host by recalling **FTPOpenConnection**.

credentialsMissing

A credentials-related client or user error signalled by many primitives. Before it can carry out the requested operation, the remote FTP Server requires the local user’s credentials. The client should supply them via **FTPSetCredentials** and then reattempt the operation.

dumpFileBlockTooLong

A dump file format error signalled by **FTPInventoryDumpFile** or **FTPRetrieveFile**. One or more of the blocks within the dump file is declared by its header to contain more than 256 bytes of data. The program that created the dump file is in error, and the dump file cannot be read by FTP.

dumpFileCheckSumInError

A dump file format error signalled by **FTPInventoryDumpFile** or **FTPRetrieveFile**. One or more of the blocks within the dump file contains in its header a checksum that is inconsistent with the data in the block. The contents of the file have been altered as a result of faulty storage or transmission, and the dump file cannot be read by FTP.

duplicateMailboxException

A File Transfer Protocol (FTP) violation signalled by **FTPIdentifyNextRejectedRecipient**. The remote FTP Server transmitted to the local FTP User, two or more mailbox exceptions for a single recipient. This occurrence should be reported to your support group; it represents a bug in the remote FTP implementation.

duplicateProtocolParameter

A File Transfer Protocol (FTP) violation signalled by many primitives. The remote FTP Server transmitted to the local FTP User, two or more instances of a single parameter type in a single parameter list. This occurrence should be reported to your support group; it represents a bug in the remote FTP implementation.

errorBlockInDumpFile

A dump file error signalled by **FTPInventoryDumpFile** or **FTPRetrieveFile**. The program that (partially) created the dump file failed to complete it successfully. The fact that an error occurred while writing the file is explicitly and permanently recorded within it, and the dump file cannot be read by FTP.

fileAlreadyExists

A file error reported by **FTPStoreFile**. A remote file with the specified name already exists within the remote file system and cannot be overwritten. The client should either first delete the existing file via **FTPDeleteFile** and then reattempt the store operation via **FTPStoreFile**, or select another name or version for the **remoteFile**.

fileBusy

A file error reported by several primitives. Because the specified remote file is currently being manipulated by another (either local or remote) client, the file operation requested by the local client cannot now be carried out. The client should reattempt the operation at another time.

fileDataError

A file error reported by several primitives. A permanent error was encountered by the remote (or local) file system while reading or writing the remote (local) file. The client may wish to reattempt the operation, although the file may well be permanently damaged.

fileGroupDesignatorUnexpected

A client error signalled by **FTPRetrievelFile** or **FTPDeleteFile**. The specified **remoteFile** designates a group of remote files, rather than a single remote file. To retrieve or delete all files in the group, the client should enumerate the files via **FTPEnumerateFiles** and retrieve or delete each file in turn via **FTPRetrieveFile** or **FTPDeleteFile**.

filenameUnexpected

A client error signalled by **FTPRetrieveFile**, **FTPDeleteFile**, or **FTPRenameFile**. A file enumeration operation is in progress, and the specified **remoteFile** is not that most recently identified to the client by **FTPEnumerateFiles**. During the course of an enumeration, the client may manipulate the remote file system in no way other than that implied by **FTPEnumerateFiles' intent** parameter. Specifying an **intent** of **unspecified** grants the client unconstrained access to the remote file system.

filePrimitivesNotSpecified

A client error signalled by **FTPCreateUser** or **FTPCreateListener**. The **filePrimitives** parameter supplied to one of these procedures is NIL.

filesModuleNotLoaded

A client error signalled by **FTPOpenConnection** or **FTPCreateListener**. The client specified a **purpose** of **files** or **filesAndMail**, yet the **FTPUserFiles** or **FTPServerFiles** module is absent from the client's configuration. The client should either obtain a more appropriate standard configuration or, if none is currently provided, request one from his support group.

functionNotImplemented

A File Transfer Protocol (FTP) violation signalled by many primitives. The remote FTP Server failed to recognize a valid operation code sent to it by the local FTP User. This occurrence should be reported to your support group; it represents a bug or omission in the remote FTP implementation.

illegalBooleanParameter

A File Transfer Protocol (FTP) violation signalled by several primitives. The remote FTP Server transmitted to the local FTP User, a boolean parameter whose value was neither TRUE nor FALSE. This occurrence should be reported to your support group; it represents a bug in the remote FTP implementation.

illegalFileAttribute

A File Transfer Protocol (FTP) violation signalled by several primitives. The remote FTP Server reported that it received an illegal file attribute parameter from the local FTP User. This occurrence should be reported to your support group; it represents a bug in the local FTP implementation.

illegalFilename

A file error signalled by many primitives. The specified local or remote filename violates the syntax conventions of the local or remote file system. The client should correct the filename and reattempt the operation.

illegalFileType

A File Transfer Protocol (FTP) violation signalled by several primitives. The remote FTP Server (or User) transmitted to the local FTP User (or Server) a file type parameter the value of which was neither TEXT nor BINARY. This occurrence should be reported to your support group; it represents a bug in the remote FTP implementation.

illegalProcedureCallSequence

A client error signalled by several primitives. A composite operation (for example, the creation of a dump file) involving the successive invocation of several FTP primitives is in progress. The requested operation cannot be carried out until that composite operation is completed.

illegalProtocolParameterList

A File Transfer Protocol (FTP) violation signalled by many primitives. The remote FTP Server (or User) transmitted an ill-formed parameter list to the local FTP User (or Server). This occurrence should be reported to your support group; it represents a bug in the remote FTP implementation.

illegalProtocolSequence

A File Transfer Protocol (FTP) violation signalled by many primitives. The remote FTP Server (or User) transmitted to the local FTP User (or Server) a command or reply that was out of place in the protocol sequence. This occurrence should be reported to your support group; it represents a bug in the remote FTP implementation.

incorrectPrimaryPassword

A credentials-related client or user error signalled by many primitives. The primary **password** supplied to the remote FTP Server via **FTPSetCredentials** is incorrect. The client should supply a valid user name and its associated password via **FTPSetCredentials** and then reattempt the operation.

incorrectSecondaryPassword

A credentials-related client or user error signalled by many primitives. The secondary **password** supplied to the remote FTP Server via **FTPSetCredentials** is incorrect. The client should supply a valid user name and its associated password via **FTPSetCredentials** and then reattempt the operation.

inputDiscontinuityUnexpected

A File Transfer Protocol (FTP) violation signalled by many primitives. The remote FTP Server (or User) transmitted a truncated command to the local FTP User (or Server). This occurrence should be reported to your support group; it represents a bug in the remote FTP implementation.

mailModuleNotLoaded

A client error signalled by **FTPCreateListener**. The client specified a **purpose** of **mail** or **filesAndMail**, yet the **FTPUserMail** or **FTPServerMail** module is absent from the client's configuration. The client should either obtain a more appropriate standard configuration or, if none is currently provided, request one from his support group.

mailPrimitivesNotSpecified

A client error signalled by **FTPCreateListener**. The **mailPrimitives** parameter supplied to this procedure is NIL. Provided the corresponding program module has been bound into the client's configuration (see Appendix I), acceptable values for this parameter include those returned by the FTP procedures **SysMailPrimitives** and **SomeMailPrimitives**.

messageLongerThanAdvertised

A File Transfer Protocol (FTP) violation signalled by **FTPRetrieveBlockOfMessage**, **FTPIdentifyNextMessage**, or **FTPEndRetrievalOfMessages**. The remote FTP Server transmitted to the local FTP User a message that was longer than indicated by the byte count that preceded the message. This occurrence should be reported to your support group; it represents a bug in the remote FTP implementation.

messageShorterThanAdvertised

A File Transfer Protocol (FTP) violation signalled by **FTPRetrieveBlockOfMessage**, **FTPIdentifyNextMessage**, or **FTPEndRetrievalOfMessages**. The remote FTP Server transmitted to the local FTP User a message that was shorter than indicated by the byte count that preceded the message. This occurrence should be reported to your support group; it represents a bug in the remote FTP implementation.

missingMessageLength

A File Transfer Protocol (FTP) violation signalled by **FTPIdentifyNextMessage** or **FTPEndRetrievalOfMessages**. The remote FTP Server failed to transmit to the local FTP User the length of the message that was to follow. This occurrence should be reported to your support group; it represents a bug in the remote FTP implementation.

missingProtocolParameter

A File Transfer Protocol (FTP) violation signalled by many primitives. The remote FTP User (or Server) failed to transmit a required parameter to the local FTP Server (or User). This occurrence should be reported to your support group; it represents a bug in the remote FTP implementation.

noConnectionEstablished

A client error signalled by many primitives. The client has attempted to manipulate remote files or mail without first having opened a connection to a remote FTP Server. The client should open a connection via **FTPOpenConnection** and then reattempt the operation.

noNameLookupResponse

A communication error signalled by **FTPOpenConnection**. The specified host name cannot be decoded because all of the name lookup servers on all of the directly connected networks (normally just one Ethernet) are broken or are inaccessible. The client should reattempt connection via **FTPOpenConnection** at a later time.

noRoomForFile

A file error reported by **FTPStoreFile**. The specified remote file cannot be written because the remote disk allocation has been (or would be) exceeded. The client should delete one or more unwanted files via **FTPDeleteFile** and then reattempt the store operation via **FTPStoreFile**.

noRouteToNetwork

A communication error signalled by many primitives. The internetwork has been partitioned in such a way that the remote host is (no longer) accessible. The client should clear the no-route-to-network condition via **FTPCloseConnection** and reattempt connection via **FTPOpenConnection** at a later time.

noSuchFile

A file error signalled by many primitives. The specified local or remote file does not exist. The client should respecify the local or remote filename and reattempt the operation.

noSuchHost

A communication error signalled by **FTPOpenConnection**. The specified **host** name is unrecognized by the local name lookup server. The client should respecify the host name and reattempt the operation via **FTPOpenConnection**.

noSuchMailbox

A mail error signalled by **FTPBeginRetrievalOfMessages**. The specified **mailboxName** is unrecognized by the remote FTP Server. The client should respecify the mailbox name and reattempt the operation via **FTPBeginRetrievalOfMessages**.

noSuchPrimaryUser

A credentials-related client or user error signalled by many primitives. The primary **user** name supplied to the remote FTP Server via **FTPSetCredentials** is unrecognized by the Server (that is, the name is invalid). The client should supply a valid user name and its associated password via **FTPSetCredentials** and then reattempt the operation.

noSuchRecipientNumber

A File Transfer Protocol (FTP) violation signalled by **FTPIdentifyNextRejectedRecipient**. The remote FTP Server transmitted to the local FTP User, a mailbox exception, the identifying number of which was out of range. This occurrence should be reported to your support group; it represents a bug in the remote FTP implementation.

noSuchSecondaryUser

A credentials-related client or user error signalled by many primitives. The secondary **user** name supplied to the remote FTP Server via **FTPSetCredentials** is unrecognized by the Server (that is, the name is invalid). The client should supply a valid user name and its associated password via **FTPSetCredentials** and then reattempt the operation.

noValidRecipients

A mail error reported by **FTPIdentifyNextRejectedRecipient**. None of the recipients was accepted by the remote FTP Server; all were rejected. The client should clear the error condition via **FTPEndDeliveryOfMessage**, correct the distribution list, and reattempt the mail delivery operation via **FTPBeginDeliveryOfMessage**.

outputDiscontinuityUnexpected

A File Transfer Protocol (FTP) violation signalled by many primitives. The local FTP User (or Server) transmitted a truncated command to the remote FTP Server (or User). This occurrence should be reported to your support group; it represents a bug in the local FTP implementation.

protocolParameterListMissing

A File Transfer Protocol (FTP) violation signalled by many primitives. The remote FTP User (or Server) failed to transmit the required parameter list to the local FTP Server (or User). This occurrence should be reported to your support group; it represents a bug in the remote FTP implementation.

protocolVersionMismatch

A File Transfer Protocol (FTP) problem signalled by **FTPOpenConnection** or **FTPRenewConnection**. The local FTP User and remote FTP Server disagree on the version (that is, vintage) of the Protocol to be used. This occurrence should be reported to your support group; it represents an incompatibility between the local and remote FTP implementations.

queueInconsistent

An error internal to the local FTP implementation signalled by several primitives. One of the queues maintained by FTP (for example, the queue of FTP Servers created by a particular FTP Listener) is not valid. This occurrence should be reported to your support group; it represents a bug in the local FTP implementation.

requestedAccessDenied

A credentials error signalled by many primitives. The access privileges associated with the previously specified primary or secondary credentials are insufficient to allow the remote FTP Server to carry out the requested operation. The client may wish to supply the name and password of a more privileged user via **FTPSetCredentials** and then reattempt the operation.

stringTooLong

An error internal to the local FTP implementation signalled by many primitives. An attempt has been made to write more characters into a Mesa STRING than are permitted by its maximum length. This occurrence should be reported to your support group; it may represent a bug in the local FTP implementation.

unexpectedEndOfFile

An error internal to the local FTP implementation signalled by several primitives. An attempt has been made to read more characters from a system or scratch file (for example, that used to buffer the results of a file enumeration) than were previously written into that file. This occurrence should be reported to your support group; it represents a bug in the local FTP implementation.

unidentifiedError

An unidentified error signalled by many primitives. The requested operation was aborted due to an unidentified error. It is unknown whether the error is likely to recur. The client may wish, therefore, to treat the error as a transient one and reattempt the operation.

unidentifiedPermanentError

An unidentified error signalled by many primitives. The requested operation was aborted due to an unidentified error that is believed likely to recur if the operation is reattempted.

unidentifiedTransientError

An unidentified error signalled by many primitives. The requested operation was aborted due to an unidentified error that is believed unlikely to recur. The client may wish, therefore, to reattempt the operation.

unrecognizedDumpFileBlock

A dump file format error signalled by **FTPInventoryDumpFile** or **FTPRetrieveFile**. One or more of the blocks within the dump file is of unknown type. The program that created the dump file is in error, and the dump file cannot be read by FTP.

unrecognizedMailboxExceptionErrorCode

A File Transfer Protocol (FTP) violation signalled by **FTPIdentifyNextRejectedRecipient**. The remote FTP Server transmitted an unrecognized mailbox exception error code to the local FTP User. This occurrence should be reported to your support group; it represents a bug in the remote FTP implementation or an incompatibility between the local and remote implementations.

unrecognizedProtocolErrorCode

A File Transfer Protocol (FTP) violation signalled by many primitives. The remote FTP Server (or User) transmitted an unrecognized error code to the local FTP User (or Server). This occurrence should be reported to your support group; it represents a bug in the remote FTP implementation or an incompatibility between the local and remote implementations.

unrecognizedProtocolParameter

A File Transfer Protocol (FTP) violation signalled by many primitives. The remote FTP User (or Server) transmitted an unrecognized parameter to the local FTP Server (or User). This occurrence should be reported to your support group; it represents a bug in the remote FTP implementation or an incompatibility between the local and remote implementations.

Appendix B: Dump Primitives

B.1. Introduction

Besides its more general file-manipulation primitives, FTP supplies a family of procedures for composing and decomposing remote *dump files*, as described below. A dump file is a single *physical* file that contains one or more *logical* files.

B.2. Inventory Primitives

FTP provides one procedure, **FTPInventoryDumpFile**, for enumerating the members of a remote dump file. For each logical file in the physical file whose name, **remoteDumpFile**, is specified, the procedure supplies to a client-provided procedure, **processFile**, the client's **processFileData** and the logical file's (absolute) filename. The order in which filenames are presented to the client is that in which the corresponding files were written into the dump file:

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

DumpFileIntent: TYPE = Intent[enumeration..retrieval];
Intent: TYPE = {enumeration, retrieval, deletion, renaming, unspecified};
FileInfo: TYPE = POINTER TO FileInfoObject;
FileInfoObject: TYPE = RECORD [
    fileType: FileType, byteSize: CARDINAL, byteCount: LONG CARDINAL,
    creationDate, writeDate, readDate, author: STRING];
FileType: TYPE = {text, binary, unknown};
```

Exceptions: connectionTimedOut, connectionClosed, noRouteToNetwork, credentialsMissing, noSuchPrimaryUser, noSuchSecondaryUser, incorrectPrimaryPassword, incorrectSecondaryPassword, requestedAccessDenied, illegalFilename, noSuchFile, fileBusy, fileDataError, errorBlockInDumpFile, unrecognizedDumpFileBlock, dumpFileBlockTooLong, dumpFileCheckSumInError, noConnectionEstablished, connectionNotOpenedForFiles, illegalProcedureCallSequence, fileGroupDesignatorUnexpected.

Notice that the **TYPE** of **processFile** is the same as that required by **FTPEnumerateFiles**. However, when **processFile** is called, the only interesting portion of the **FileInfo** is the **creationDate**.

The **intent** parameter supplied by the client declares the manner in which the client expects to manipulate the files the names of which are presented to it. This information enables the FTP User to intelligently select from among several possible protocol strategies for effecting the inventory. Since most such strategies occupy 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 any of the following intents:

1. An intent of **enumeration** declares that the client seeks the names of the members of the designated dump file (for presentation to a human user, for example) and intends to manipulate the files in no other way during the course of the enumeration. More specifically, the client declares (and FTP insures) 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 member files and to manipulate them in no other way during the course of the enumeration. The client's **processFile** procedure may retrieve the member file the name of which is presented to it by supplying that name to the **FTPRetrieveFile** procedure described elsewhere. The **byteCount** returned by **FTPRetrieveFile** will reflect only the (original) contents of the member file; it will not reflect the additional bytes of formatting information contained in the dump file, which are stripped away by FTP. More specifically, then, the client declares (and FTP insures) that **processFile** will make no calls to local FTP User procedures (other than **FTPRetrieveFile**) that communicate with the remote FTP Server.

Once the dump file has been inventoried, the client may invoke the **FTPNoteFilenameUsed** primitive described in Appendix D to determine the fully qualified absolute and/or virtual filename used by the remote FTP Server.

B.3. Construction Primitives

FTP provides two procedures for constructing remote dump files. The first, **FTPBeginDumpFile**, initializes (to empty) a new **remoteDumpFile** and prepares it to receive member files via the **FTPStoreFile** procedure described in Section 3.5. In the presence of an open dump file, **FTPStoreFile**'s invocation is interpreted as a request to add the specified **localFile** to the open dump file as a new member. **FTPStoreFile**'s **remoteFile** parameter is interpreted as the name by which the file is to be known *within the remote dump file*. The **byteCount** returned by **FTPStoreFile** will reflect only the (original) contents of the member file; it will not reflect the additional bytes of formatting information contained in the dump file, which are supplied by FTP. Since the construction of a remote dump file occupies the remote FTP Server until the process is complete, FTP prohibits the client from calling local FTP User procedures (other than **FTPStoreFile**) that communicate with the remote FTP Server until the dump file is complete (that is, until the **FTPEndDumpFile** procedure described below has been invoked):

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

Exceptions: connectionTimedOut, connectionClosed, noRouteToNetwork, credentialsMissing, noSuchPrimaryUser, noSuchSecondaryUser, incorrectPrimaryPassword, incorrectSecondaryPassword, requestedAccessDenied, illegalFilename, fileAlreadyExists, fileBusy, noRoomForFile, fileDataError, noConnectionEstablished, connectionNotOpenedForFiles, illegalProcedureCallSequence.

The second procedure, **FTPEndDumpFile**, finalizes a newly created remote dump file after all member files have been added to it. Only by calling this procedure can the client leave the dump file construction mode entered via successful invocation of **FTPBeginDumpFile**:

FTPEndDumpFile: PROCEDURE [ftpuser: FTPUser];

Exceptions: connectionTimedOut, connectionClosed, noRouteToNetwork, noRoomForFile, fileDataError, noConnectionEstablished, connectionNotOpenedForFiles, illegalProcedureCallSequence.

Appendix C: Mail Primitives

C.1. Introduction

Besides its file-related primitives, FTP supplies a family of procedures for delivering mail to and retrieving mail from remote mailboxes, as described below.

C.2. Delivery Primitives

FTP provides five procedures for delivering mail directly to remote mailboxes and/or for forwarding it to its ultimate destination via a third party. The first procedure, **FTPBeginDeliveryOfMessage**, initiates the delivery and/or forwarding of a single message to one or more remote recipients. Successful invocation of this first mail delivery procedure conditions the local FTP User to accept calls to the other four. Since the mail delivery process occupies the remote FTP Server until delivery is complete, FTP prohibits the client from calling other local FTP User procedures that communicate with the remote Server until **FTPEndDeliveryOfMessage** procedure described below has been invoked:

FTPBeginDeliveryOfMessage: PROCEDURE [ftpuser: FTPUser];

Exceptions: connectionTimedOut, connectionClosed, noRouteToNetwork, noConnectionEstablished, connectionNotOpenedForMail, illegalProcedureCallSequence.

The second procedure, **FTPSendRecipientOfMessage**, identifies one of the message's recipients. After **FTPBeginDeliveryOfMessage** has been invoked, this procedure is called repetitively until all of the recipients of the message have been specified. **mailboxHostName** and **dmsName** are leftover from Mesa 4 and some old ideas about how to forward mail. They will be deleted from the interface when FTP is converted to Mesa 6.

FTPSendRecipientOfMessage: PROCEDURE [ftpuser: FTPUser, mailboxName: STRING,
mailboxHostName: STRING _ NIL, dmsName: STRING _ NIL];

Exceptions: connectionTimedOut, connectionClosed, noRouteToNetwork, noConnectionEstablished, connectionNotOpenedForMail, illegalProcedureCallSequence.

The third procedure, **FTPIdentifyNextRejectedRecipient**, reports to the client the rejection by the remote FTP Server of one of the intended recipients of the message. Delivery of the message succeeds or fails for each of its intended recipients independently. The procedure returns the number, **recipientNumber**, of the rejected recipient (the numbering starting at one) and two additional results that pinpoint the reason for the rejection: an enumerated type, **recipientError**, to be interpreted by the client; and a STRING, **errorMessage** (supplied by the client and filled in by the procedure), to be interpreted by the human user. The recipient errors reported by a local FTP User are actually detected by the remote FTP Server, and hence the User normally relays to the client the message supplied by the Server. If the Server provides no message, the User supplies an appropriate message in its place. All error messages issued by FTP are centralized in a single FTPAccessories module (see Appendix I). To avoid incurring the space overhead associated with these strings, some configurations omit this module, causing FTP to supply a zero-length string whenever it would otherwise obtain an **errorMessage** from FTPAccessories. *Once*, after **FTPSendRecipientOfMessage** has been invoked repetitively to specify the recipients of the message, and, *once again*, after the text of the message has been specified (as described later), **FTPIdentifyNextRejectedRecipient** must be called repetitively until a **recipientNumber** of zero is returned, indicating the end of the list. Most recipient errors (for example, **noSuchMailbox**) are detected and reported in the first round of calls to this procedure, but others (for example, **unspecifiedTransientError**) may not occur until the remote FTP Server actually attempts to append the message to the recipient's mailbox:

FTPIdentifyNextRejectedRecipient: PROCEDURE [ftpuser: FTPUser, errorMessage: STRING] RETURNS [recipientNumber: CARDINAL, recipientError: RecipientError];

RecipientError: TYPE = {noSuchMailbox, noForwardingProvided, unspecifiedTransientError, unspecifiedPermanentError, unspecifiedError};

Exceptions: connectionTimedOut, connectionClosed, noRouteToNetwork, noValidRecipients, noConnectionEstablished, connectionNotOpenedForMail, illegalProcedureCallSequence.

Old Exceptions: noSuchForwardingHost, noSuchDmsName.

The fourth procedure, **FTPSendBlockOfMessage**, specifies a portion of the text of the message and is called repetitively after the recipients of the message have been specified and the first round of recipient rejections accepted, as described above. 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 the Arpanet standard detailed in [3]. Throughout the message, end of line is indicated via a carriage return (CR):

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

Exceptions: connectionTimedOut, connectionClosed, noRouteToNetwork, noConnectionEstablished, connectionNotOpenedForMail, illegalProcedureCallSequence.

The fifth procedure, **FTPEndDeliveryOfMessage**, finalizes the delivery process. Only by calling this procedure can the client leave the mail delivery mode entered via successful invocation of **FTPBeginDeliveryOfMessage**:

FTPEndDeliveryOfMessage: PROCEDURE [ftpuser: FTPUser];

Exceptions: connectionTimedOut, connectionClosed, noRouteToNetwork, noConnectionEstablished, connectionNotOpenedForMail, illegalProcedureCallSequence.

C.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 the host-specific name of whish is specified by **mailboxName**. To obtain access to the mailbox, the client must first have supplied the necessary credentials (if any) by calling the **FTPSetCredentials** procedure described elsewhere. Successful invocation of this first mail retrieval procedure conditions the local FTP User to accept calls to the other three. Since the mail retrieval process occupies the remote FTP Server until it is complete, FTP prohibits the client from calling other local FTP User procedures that communicate with the remote Server until the retrieval is complete (that is, until the **FTPEndRetrievalOfMessages** procedure described below has been invoked):

> **FTPBeginRetrievalOfMessages: PROCEDURE [ftpuser: FTPUser, mailboxName: STRING];**

> Exceptions: connectionTimedOut, connectionClosed, noRouteToNetwork, credentialsMissing, noSuchPrimaryUser, noSuchSecondaryUser, incorrectPrimaryPassword, incorrectSecondaryPassword, requestedAccessDenied, noSuchMailbox, noConnectionEstablished, connectionNotOpenedForMail, illegalProcedureCallSequence.

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 contents of the mailbox unretrieved, in which case whatever remains will be sent by the remote FTP Server but will be 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 includes the size of the messsage 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** (that is, examined) or **deleted** while in the mailbox (Maxc mailboxes, for example, can be manipulated directly via the Tenex MSG subsystem):

> **FTPIdentifyNextMessage: PROCEDURE [ftpuser: FTPUser, messageInfo: MessageInfo];**
>
> **MessageInfo: TYPE = POINTER TO MessageInfoObject;**
> **MessageInfoObject: TYPE = RECORD [byteCount: LONG CARDINAL, deliveryDate: STRING, opened, deleted: BOOLEAN];**

> Exceptions: connectionTimedOut, connectionClosed, noRouteToNetwork, noConnectionEstablished, connectionNotOpenedForMail, illegalProcedureCallSequence.

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 text of the message unretrieved, in which case whatever remains will be sent by the remote FTP Server but will be 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 the Arpanet standard detailed in [3]. Throughout the message, end of line is indicated via a carriage return (CR):

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

Exceptions: connectionTimedOut, connectionClosed, noRouteToNetwork, noConnectionEstablished, connectionNotOpenedForMail, illegalProcedureCallSequence.

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 **FTPEndRetrievalOfMessages** is invoked. Only by calling this procedure can the client leave the mail retrieval mode entered via successful invocation of **FTPBeginRetrievalOfMessages**:

FTPEndRetrievalOfMessages: PROCEDURE [ftpuser: FTPUser];

Exceptions: connectionTimedOut, connectionClosed, noRouteToNetwork, noConnectionEstablished, connectionNotOpenedForMail, illegalProcedureCallSequence.

Appendix D: Miscellaneous Primitives

D.1. Introduction

In addition to the primitives described in the body of this document and the preceding appendices, FTP supplies a number of less frequently used procedures, as described below.

D.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, in subsequent calls to **FTPOpenConnection**, the local FTP User should expect to contact remote FTP Listeners for the **purpose** of manipulating either remote **files**, **mail**, or **filesAndMail**. Recall that file- and mail-related transactions are supported by distinct FTP Servers created by distinct FTP Listeners monitoring distinct well-known sockets. This procedure permits use of experimental FTP Listeners, which are often assigned to non standard (that is, not-so-well-known) sockets during their checkout phase. A socket number of zero resets the affected socket(s) to its(their) standard, default values (that is, 3 for **files** and 7 for **mail**):

> **FTPSetContactSocket:** PROCEDURE **[ftpuser: FTPUser, socket:** LONG CARDINAL**,**
> **purpose: Purpose];**

> **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 (for example, IODefs.WriteString) 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];**

D.3. Infrequently Used File Transfer Primitives

FTP provides three procedures that supplement the primary file transfer procedures described elsewhere. The first, **FTPTransferFile**, retrieves from the remote file system addressed by **srcFtpuser** and stores (under the name **dstFile**) in the remote file system addressed by **dstFtpuser** a copy of the remote file the name of which is specified by **srcFile** and the type of which is specified by **fileType**--**text** or **binary**. It also returns the size in bytes, **byteCount**, of the transferred file. In rare cases, the **fileType** parameter supplied by the client is used by the source FTP Server to disambiguate between two like-named files of different types. The client may (and often does) report the file's type as **unknown**, in which case the remote FTP Server must select a file without it. In any case, the destination FTP Server uses the file type reported by the source FTP Server in determining how to store the file in its file system (for example, on Maxc, text files are stored as 7-bit bytes, binary files as 8-bit bytes) Even though data are double buffered as they move from source to destination host, because of input and output interference on the Ether, throughput is currently very low compared to that achieved by using **FTPRetrieveFile** and **FTPStoreFile**:

```
FTPTransferFile: PROCEDURE [srcFtpuser: FTPUser, srcFile: STRING, dstFtpuser:
    FTPUser, dstFile: STRING, fileType: FileType,
    transferFile: POINTER TO TransferFile,
    transferFileData: UNSPECIFIED] RETURNS [byteCount: LONG CARDINAL];

FileType: TYPE = {text, binary, unknown};
TransferFile: TYPE = PROCEDURE [transferFileData: UNSPECIFIED,
    receiveBlock: PROCEDURE [UNSPECIFIED, POINTER, CARDINAL] RETURNS [CARDINAL],
    receiveBlockData: UNSPECIFIED,
    sendBlock: PROCEDURE [UNSPECIFIED, POINTER, CARDINAL],
    sendBlockData: UNSPECIFIED];
```

Exceptions: connectionTimedOut, connectionClosed, noRouteToNetwork, credentialsMissing, noSuchPrimaryUser, noSuchSecondaryUser, incorrectPrimaryPassword, incorrectSecondaryPassword, requestedAccessDenied, illegalFilename, noSuchFile, fileAlreadyExists, fileBusy, noRoomForFile, fileDataError, errorBlockInDumpFile, unrecognizedDumpFileBlock, dumpFileBlockTooLong, dumpFileCheckSumInError, noConnectionEstablished, connectionNotOpenedForFiles, illegalProcedureCallSequence, fileGroupDesignatorUnexpected, filenameUnexpected.

Rather than use the local file system as a way station between the remote source and destination file systems, **FTPTransferFile** immediately outputs to the destination FTP Server each successive segment of the file it inputs from the source FTP Server. The task of transferring the contents of the file from source to destination is normally performed by an FTP-provided procedure of type **TransferFile**. The client can examine and/or modify the contents of the file as it passes from source to destination by supplying its own implementation of this procedure. The client exercises this option by specifying a **transferFile** procedure and arbitrary **transferFileData**, to be passed by FTP as an argument to that procedure. If **transferFile** is NIL, FTP supplies its own implementation, which simply double-buffers the contents of the file as it passes it unexamined and unchanged to the destination.

The client's **transferFile** procedure consumes successive segments of the file in turn using an FTP-provided procedure, **receiveBlock**. The client supplies **receiveBlock** with the **receiveBlockData** supplied by FTP and the location and length *in words* of a buffer into which the next segment is to be placed. In response, **receiveBlock** returns the segment left-adjusted in the buffer, along with its length *in bytes.* **ReceiveBlock** eventually signals end of file by returning a byte count of zero.

The client's **transferFile** procedure produces successive segments of the file in turn, using a second FTP-provided procedure, **sendBlock**. The client supplies **sendBlock** with the **sendBlockData** supplied by FTP and the location and length *in bytes* of the next segment.

TransferFile eventually signals end of file by calling **sendBlock** a final time with a byte count of zero.

The input and output functions performed by **receiveBlock** and **sendBlock**, respectively, are completely independent. The client's **transferFile** procedure may consume and produce the incoming and outgoing streams in any manner it sees fits. Its only responsibilities are to provoke (eventually) an end-of-file indication from **receiveBlock** and signal an end-of-file indication to **sendBlock**.

Once the file has been transferred, the client may invoke the **FTPNoteFilenameUsed** primitive described below to determine the fully qualified absolute and/or virtual filenames used by the remote source and destination FTP Servers. The client can effect the transfer of a whole *group* of remote files by using **FTPTransferFile** (to transfer a single file) in conjunction with the **FTPEnumerateFiles** described in Section 3.4 or **FTPInventoryDumpFile** procedure described in Appendix B.2 (to enumerate or inventory the source files to be transferred). **FTPTransferFile** can also be used to construct remote dump files, as described in Appendix B.

The second procedure, **FTPNoteFilenameUsed**, returns the fully qualified absolute and/or virtual filename used by the remote FTP Server in the immediately preceding **FTPStoreFile**, **FTPRetrieveFile**, **FTPDeleteFile**, **FTPInventoryDumpFile**, or **FTPTransferFile** operation. For example, if the client had defaulted the version number in the remote filename it supplied to **FTPStoreFile**, and if the remote FTP Server, therefore, had created a new version of an already existing file, **FTPNoteFilenameUsed** could be employed by the client to determine the version number assigned by the Server. Any manipulation of the FTP User between execution of the target store, retrieve, delete, inventory, or transfer operation and invocation of **FTPNoteFilenameUsed** may invalidate the information the primitive returns. If **absoluteFilename** is specified (that is, non-NIL), the primitive returns a fully-qualified absolute filename in the STRING supplied by the client. If **virtualFilename** is specified (that is, non-NIL), the primitive returns a fully qualified virtual filename in the STRINGs supplied by the client. The reader is referred to Section 1.3 in this document for a discussion of virtual filenames and their use:

```
FTPNoteFilenameUsed: PROCEDURE [ftpuser: FTPUser, absoluteFilename: STRING,
   virtualFilename: VirtualFilename];

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

The third procedure, **FTPSetBufferSize**, alters the size of the buffers that local FTP Users and Servers use to transfer data between the file and communication systems. The default value is four. Since it is the **ReadFile** and **WriteFile** file primitives described in Appendix E that actually make use of this parameter, if the client supplies its own implementation of either or both primitives, **FTPSetBufferSize** will have no effect upon the corresponding file operation(s). The procedure accepts as its only parameter the size, **pages**, of the buffers in 256-word pages. The new buffer size takes effect with the next file transfer:

```
FTPSetBufferSize: PROCEDURE [pages: CARDINAL];
```

D.4. Other Primitives

FTP provides one procedure, **FTPCatchUnidentifiedErrors**, for specifying whether FTP should catch, and report as an **unidentifiedError** via **FTPError**, signals raised unexpectedly by the Mesa System (for example, **StringBoundsFault**). The procedure accepts as its only parameter the new switch **setting**, which takes effect at once. The default value is TRUE (that is, catch such errors):

```
FTPCatchUnidentifiedErrors: PROCEDURE [setting: BOOLEAN];
```

Appendix E: Client File Primitives

E.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 an implementation manipulating the standard Alto file system.

Rather than the file system interface offered by FTP, the client may, if it wishes, provide its own file primitives to a particular FTP User or Listener. By so doing, a client can use FTP, for example to:

1. interface to another local file system (for example, Juniper).
2. interface to a pseudo file system (for example, a printer).
3. produce or consume files on the fly (that is, files that never exist on secondary storage).
4. transform filenames (for example, 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 client can also 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 E.8) that records the event and then calls FTP's **OpenFile** procedure to actually open the file.

E.2. Exercising the Option

The client exercises the option described above by means of the **filePrimitives** parameter required 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 any FTP Servers created by the newly created FTP Listener, are to access the local file system. The client may supply its own version of this record, rather than rely upon the standard version offered by FTP. In constructing the record and/or implementing the procedures it contains, the client may draw upon any of the file primitives it finds in the FTP-provided record. Since FTP will not copy the record presented to it, the client must preserve it intact until **FTPDestroyUser** or **FTPDestroyListener** is called:

```
FilePrimitives: TYPE = POINTER TO FilePrimitivesObject;
FilePrimitivesObject: TYPE = RECORD [

  -- program management primitives
 CreateFileSystem: PROCEDURE [bufferSize: CARDINAL] RETURNS [fileSystem:
   FileSystem],
 DestroyFileSystem: PROCEDURE [fileSystem: FileSystem],
  -- filename manipulation primitives
 DecomposeFilename,
 ComposeFilename: PROCEDURE [fileSystem: FileSystem, absoluteFilename:
   STRING, virtualFilename: VirtualFilename],

  -- access control primitives
 InspectCredentials: PROCEDURE [fileSystem: FileSystem, status: Status, user,
   password: STRING],

  -- file enumeration primitives
 EnumerateFiles: PROCEDURE [fileSystem: FileSystem, files: STRING, intent:
   EnumerateFilesIntent,
   processFile: PROCEDURE [UNSPECIFIED, STRING, FileInfo],
   processFileData: UNSPECIFIED],

  -- file transfer primitives
 OpenFile: PROCEDURE [fileSystem: FileSystem, file: STRING, mode: Mode,
   fileTypePlease: BOOLEAN, info: FileInfo] RETURNS [fileHandle: FileHandle,
   fileType: FileType],
 ReadFile: PROCEDURE [fileSystem: FileSystem, fileHandle: FileHandle,
   sendBlock: PROCEDURE [UNSPECIFIED, POINTER, CARDINAL],
   sendBlockData: UNSPECIFIED],
 WriteFile: PROCEDURE [fileSystem: FileSystem, fileHandle: FileHandle,
   receiveBlock: PROCEDURE [UNSPECIFIED, POINTER, CARDINAL] RETURNS [CARDINAL],
   receiveBlockData: UNSPECIFIED],
 CloseFile: PROCEDURE [fileSystem: FileSystem, fileHandle: FileHandle, aborted:
   BOOLEAN],

  -- file manipulation primitives
 DeleteFile: PROCEDURE [fileSystem: FileSystem, file: STRING],
 RenameFile: PROCEDURE [fileSystem: FileSystem, currentFile, newFile: STRING]];

FileSystem: TYPE = POINTER TO FileSystemObject;
FileSystemObject: TYPE = RECORD [dummy: UNSPECIFIED];
VirtualFilename: TYPE = POINTER TO VirtualFilenameObject;
VirtualFilenameObject: TYPE = RECORD [device, directory, name, version: STRING];
Status: TYPE = {primary, secondary};
EnumerateFilesIntent: TYPE = Intent[enumeration..deletion];
Intent: TYPE = {enumeration, retrieval, deletion, renaming, unspecified};
FileInfo: TYPE = POINTER TO FileInfoObject;
FileInfoObject: TYPE = RECORD [
   fileType: FileType, byteSize: CARDINAL, byteCount: LONG CARDINAL,
   creationDate, writeDate, readDate, author: STRING];
FileType: TYPE = {text, binary, unknown};
Mode: TYPE = {read, write, append, writeThenRead, readThenWrite};
FileHandle: TYPE = POINTER TO FileHandleObject;
FileHandleObject: TYPE = RECORD [dummy: UNSPECIFIED];
```

E.3. General Characteristics

The file primitives employed by FTP and suppliable by the client are described below. Statements that apply to all valid implementations of a primitive (that is, the FTP implementation for the Alto and any a client might supply) are rendered in the standard font. Statements that apply only to the implementation supplied by FTP are rendered in a smaller font. Except where stated to the contrary, filenames supplied as arguments to file primitives are *absolute* filenames and have thus already been processed by the **ComposeFilename** primitive described in Section E.5.

In accordance with standard Mesa exception handling conventions, file primitives report errors by signalling. FTP catches any signal that reaches it and aborts the current transaction (with the help of the remote FTP User or Server, as necessary). Wherever possible, client file primitives should use the standard FTP signal, **FTPError** (described in Section 1.4), to report errors. Doing so enables the FTP User or Server to communicate the error to the remote FTP Server or User in a meaningful way. The description of each procedure below includes a list of the **FtpError** values that seem, to the author, most appropriate for that primitive. Requests by file primitive implementors for new **FtpError** values will be gladly entertained.

E.4. Program Management Primitives

FTP or its client provides two procedures for creating and destroying instances of the local file system. The first, **CreateFileSystem**, used by both FTP User and Server, creates a new instance of the local file system and specifies the size in pages, **bufferSize**, of the buffers to be used by its **ReadFile** and **WriteFile** primitives (see Section E.8). The procedure returns a handle, **fileSystem**, to the newly created file system instance, which the caller must retain and later present to any of the other file primitives it invokes. The **fileSystem** is a pointer to a private record containing all of the state information the file system instance requires to function properly. The Alto implementation of this procedure simply records the buffer size:

```
CreateFileSystem: PROCEDURE [bufferSize: CARDINAL] RETURNS [fileSystem:
   FileSystem];

FileSystem: TYPE = POINTER TO FileSystemObject;
FileSystemObject: TYPE = RECORD[dummy: UNSPECIFIED];
```

The second procedure, **DestroyFileSystem**, used by both FTP User and Server, destroys a previously created instance of the local file system, reclaiming any local resources allocated to it. Before invoking this procedure, the caller must close all open files. The Alto implementation of this procedure is essentially a no operations:

```
DestroyFileSystem: PROCEDURE [fileSystem: FileSystem];

FileSystem: TYPE = POINTER TO FileSystemObject;
FileSystemObject: TYPE = RECORD[dummy: UNSPECIFIED];
```

E.5. 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 are rendered 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 **[fileSystem: FileSystem, absoluteFilename:**
> STRING**, virtualFilename: VirtualFilename];**
>
> **VirtualFilename:** TYPE **=** POINTER TO **VirtualFilenameObject;**
> **VirtualFilenameObject:** TYPE **=** RECORD **[device, directory, name, version:** STRING**];**
>
> Exceptions: illegalFilename.

The second procedure, **ComposeFilename**, used by both FTP User and Server, constructs a fully qualified **absoluteFilename** from a **virtualFilename** and an unqualified or partially specified **absoluteFilename** (which is overwritten in the process). If the virtual filename is unspecified (that is, if all of its components are of zero length), the absolute filename to be returned by the procedure is completely specified by the absolute filename supplied to it. If the absolute filename supplied to the procedure is unspecified (that is, if it is of zero length), the absolute filename to be returned is completely specified by the virtual filename. If both absolute and virtual filenames are specified, the procedure optionally uses the virtual filename to default unspecified fields in the absolute filename. An implementation of this procedure could, for example, construct the fully qualified absolute filename "<Mesa>FTP>FTPDefs.Mesa" from a virtual filename the directory component of which is "Mesa" and an absolute filename the value of which is "FTP>FTPDefs.Mesa". Components of the virtual filename 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, but uses the name and version components to default components that may be missing from the absolute filename. Having applied the available defaults, it then insists upon a name component and accepts a version component if it is present:

> **ComposeFilename:** PROCEDURE **[fileSystem: FileSystem, absoluteFilename:** STRING**,**
> **virtualFilename: VirtualFilename];**
>
> **VirtualFilename:** TYPE **=** POINTER TO **VirtualFilenameObject;**
> **VirtualFilenameObject:** TYPE **=** RECORD **[device, directory, name, version:** STRING**];**
>
> Exceptions: illegalFilename.

E.6. Access Control Primitives

FTP or its client provides one procedure, **InspectCredentials**, for inspecting credentials presented by the remote client. Used only by FTP Server, this procedure verifies and records the **primary** or **secondary** credentials--**user** and **password**--with which the next file system access will be implicitly attempted. The procedure verifies the user's existence and the password's correctness and records the fact that they were (correctly) supplied; the file primitive through which access to a particular file is subsequently attempted then determines whether those credentials entitle the remote FTP User to manipulate the file in the manner requested (for example, the **DeleteFile** primitive described in Section E.9 verifies that the client has delete access to the target file before honoring the delete request). Primary credentials typically identify the user upon whose behalf the access is attempted (*a la* the Tenex Login command) while secondary credentials, when necessary, usually identify another area of the file system--in addition to the user's own workspace--to which the user claims access (*a la* the Tenex Connect command). Primary and secondary credentials (if any) are presented for inspection *immediately before* the call to the file primitive--**EnumerateFiles**, **OpenFile**, **DeleteFile**, or **RenameFile**--which attempts the access, and implicitly are discarded by the file system instance *immediately after* that operation. The Alto implementation of this procedure is a no operation:

```
InspectCredentials: PROCEDURE [fileSystem: FileSystem, status: Status, user,
    password: STRING];

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

Exceptions: noSuchPrimaryUser, noSuchSecondaryUser, incorrectPrimaryPassword, incorrectSecondaryPassword.

E.7. File Enumeration Primitives

FTP or its client provides one procedure, **EnumerateFiles**, for enumerating the members of a local file group. For each file in the group whose file group designator, **files**, is specified, the procedure, used by both FTP User and Server, supplies to a caller-provided procedure, **processFile**, the caller's **processFileData**, the absolute filename of the file, and a variety of other file information (**FileInfo**). The order in which filenames are presented to the caller is file-system-dependent, but alphabetical order is typical. Unknown or unspecified file information is rendered as **unknown**, zero, or NIL, as appropriate. 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. The files are presented in the order determined by **DirectoryDefs.EnumerateDirectory** and no use is made of the **intent** parameter:

```
EnumerateFiles: PROCEDURE [fileSystem: FileSystem, files: STRING, intent:
    EnumerateFilesIntent,
    processFile: PROCEDURE [UNSPECIFIED, STRING, FileInfo],
    processFileData: UNSPECIFIED];

EnumerateFilesIntent: TYPE = Intent[enumeration..deletion];
Intent: TYPE = {enumeration, retrieval, deletion, renaming, unspecified};
FileInfo: TYPE = POINTER TO FileInfoObject;
FileInfoObject: TYPE = RECORD [
    fileType: FileType, byteSize: CARDINAL, byteCount: LONG CARDINAL,
    creationDate, writeDate, readDate, author: STRING];
FileType: TYPE = {text, binary, unknown};
```

Exceptions: credentialsMissing, requestedAccessDenied, illegalFilename, noSuchFile, fileDataError.

The **intent** parameter supplied by the caller declares the manner in which the caller expects to manipulate the files whose names are presented to it. This information enables the file system, for example, to determine the version(s) of a file it should present to the caller (for instance, the procedure might return the most recent versions of files being retrieved and the oldest versions of files being deleted). The caller may specify any of the following intents:

1. An intent of **enumeration** declares that the caller simply seeks the names of (and file information for) the members of the designated file group (for presentation to a human user, for example) and intends to manipulate the files in no other way during the course of the enumeration.

2. An intent of **retrieval** declares that the caller 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 caller's **processFile** procedure may retrieve the file whose name is presented to it by opening that file for read via **OpenFile**, obtaining the contents of the file via **ReadFile**, and then closing the file via **CloseFile** (see Section E.8).

3. An intent of **deletion** declares that the caller 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 **DeleteFile** procedure described in Section E.9.

E.8. File Transfer Primitives

FTP or its client provides four procedures for transferring files to and from the local file system. *The client must supply its own version of all or none of these procedures, since they interact with one another by means of the file handle returned by the first.* The first procedure, **OpenFile**, used by both FTP User and Server, verifies either the existence of an old file (if **mode** is **read**, **append**, or **readThenWrite**) or the availability of space for a new one (if **mode** is **write** or **writeThenRead**), establishes the remote client's access to that file (in conjunction with the **InspectCredentials** procedure described in Section E.6), prepares the file to be read or written, and returns a handle, **fileHandle**, to the newly opened file. 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. If **mode** is **write**, and **info** is not **NIL**, and **info.creationDate** is not **NIL**, then **info.creationDate** contains the creation date and time of the file which should be saved as the creation date and time of the local file so that various automatic updating heuristics will operate correctly. If **mode** is **read** or **readThenWrite**, and **info** is not **NIL**, and **info.creationDate** is not **NIL**, then the creation date and time of the local file should be appended to **info.creationDate** so that it can be passed to the remote file system where it will be saved as the creation date and time of the new file. The Alto implementation of this procedure attaches a byte stream to the file and returns its handle. If the file's type is requested, it 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 **[fileSystem: FileSystem, file:** STRING**, mode: Mode,**
    fileTypePlease: BOOLEAN**, info: FileInfo]** RETURNS **[fileHandle: FileHandle,**
    fileType: FileType];

Mode: TYPE **= {read, write, append, writeThenRead, readThenWrite};**
FileHandle: TYPE **=** POINTER TO **FileHandleObject;**
FileHandleObject: TYPE **=** RECORD**[dummy:** UNSPECIFIED**];**
FileType: TYPE **= {text, binary, unknown};**

Exceptions: credentialsMissing, requestedAccessDenied, illegalFilename, noSuchFile, fileAlreadyExists, fileBusy, noRoomForFile, fileDataError.

The mode **writeThenRead** enables calls to both **WriteFile** and **ReadFile**, in that order; the mode **readThenWrite** enables calls to the same procedures in the other order. If mode is **write** or **writeThenRead** and the length of the **file** STRING is zero, the procedure opens a scratch file and return its name in the STRING. The reader is referred to Section E.10 for information about the manner in which this and other file primitives are used by FTP.

The second procedure, **ReadFile**, used by both FTP User and Server, presents to its caller the contents of the file (previously opened for read) the handle of which is specified by **fileHandle**. The procedure supplies to a caller-provided procedure, **sendBlock**, the **sendBlockData** supplied by **ReadFile**'s caller and the location and length in bytes of successive segments of the file. 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 **[fileSystem: FileSystem, fileHandle: FileHandle,**
    sendBlock: PROCEDURE **[**UNSPECIFIED**,** POINTER**,** CARDINAL**],**
    sendBlockData: UNSPECIFIED**];**

FileHandle: TYPE **=** POINTER TO **FileHandleObject;**
FileHandleObject: TYPE **=** RECORD**[dummy:** UNSPECIFIED**];**

Exceptions: connectionTimedOut, connectionClosed, noRouteToNetwork, fileDataError.

The third procedure, **WriteFile**, used by both FTP User and Server, accepts from its caller the contents of the file (previously opened for write) the handle of which is specified by **fileHandle**. The procedure supplies in zero or more calls to a caller-provided procedure, **receiveBlock**, the **receiveBlockData** supplied by **WriteFile**'s caller and the location and length *in words* of a buffer into which the next segment of the file is to be placed. In response, **receiveBlock** returns the segment left-adjusted in the buffer, along with its length *in bytes.* **ReceiveBlock** eventually 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 **[fileSystem: FileSystem, fileHandle: FileHandle,**
    receiveBlock: PROCEDURE **[**UNSPECIFIED**,** POINTER**,** CARDINAL**]** RETURNS **[**CARDINAL**],**
    receiveBlockData: UNSPECIFIED**];**

FileHandle: TYPE **=** POINTER TO **FileHandleObject;**
FileHandleObject: TYPE **=** RECORD**[dummy:** UNSPECIFIED**];**

Exceptions: connectionTimedOut, connectionClosed, noRouteToNetwork, noRoomForFile, fileDataError.

The fourth procedure, **CloseFile**, used by both FTP User and Server, closes the previously opened file the handle of which is specified by **fileHandle**. If the caller's attempt to read or write the file (via **ReadFile** or **WriteFile**) had to be **aborted** for some reason, the caller so indicates. If the file whose transfer was aborted was opened for **write**, **writeThenRead**, or **readThenWrite**, the procedure discards (that is, deletes) 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 **[fileSystem: FileSystem, fileHandle: FileHandle, aborted:**
  BOOLEAN**];**

FileHandle: TYPE **=** POINTER TO **FileHandleObject;**
FileHandleObject: TYPE **=** RECORD**[dummy:** UNSPECIFIED**];**

Exceptions: noRoomForFile, fileDataError.

E.9. 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 **[fileSystem: FileSystem, file:** STRING**];**

Exceptions: credentialsMissing, requestedAccessDenied, illegalFilename, noSuchFile, fileBusy, fileDataError.

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

RenameFile: PROCEDURE **[fileSystem: FileSystem, currentFile, newFile:** STRING**];**

Exceptions: credentialsMissing, requestedAccessDenied, illegalFilename, noSuchFile, fileAlreadyExists, fileBusy, noRoomForFile, fileDataError.

E.10. Usage of File Primitives by FTP

In principle, the programmer who elects to implement his own file primitives should provide FTP with a complete and coherent set which meets the above specifications. In practice, however, only a subset of the primitives may actually be required in any particular application (for example, an FTP File Listener). In such cases, the required implementation effort can be reduced by knowledge of which primitives FTP uses to implement its various features.

What follows, therefore, is an exhaustive specification of the use *currently* made by FTP of the file primitives supplied by the client. The programmer may wish to exploit this information and implement only those primitives required by his particular configuration and application. Doing so, however, *obliges* him to keep careful tabs on the corresponding usage data for subsequent FTP releases, the file primitive usage for which may change or expand.

The twelve file primitives are currently used as follows:

CreateFileSystem

1. invoked by **FTPOpenConnection** unless **purpose** is **mail**.
2. invoked as part of the initialization of each new FTP File (but not Mail) Server.
3. invoked by the FTP-provided **CreateMailSystem** mail primitive (see Appendix F).

DestroyFileSystem

1. invoked by **FTPCloseConnection** unless **FTPOpenConnection**'s **purpose** was **mail**.
2. invoked as part of the finalization of each retiring FTP File (but not Mail) Server.
3. invoked by the FTP-provided **DestroyMailSystem** mail primitive (see Appendix F).

DecomposeFilename

1. invoked by FTP File Servers asked to enumerate, store, retrieve, or delete files.
2. invoked by the Alto FTP-provided **ComposeFilename** primitive.

ComposeFilename

1. invoked by FTP File Servers asked to enumerate, store, retrieve, delete, or rename files.

InspectCredentials

1. invoked by FTP File Servers asked to enumerate, store, retrieve, delete, or rename files.

EnumerateFiles

1. invoked by FTP File Servers asked to enumerate, retrieve, or delete files.

OpenFile

1. invoked by **FTPEnumerateFiles** if **intent** is **renaming** or **unspecified** to open a scratch file for filenames **writeThenRead**.
2. invoked by **FTPStoreFile** to open the source file **read**.
3. invoked by **FTPRetrieveFile** to open the destination file **write**.
4. invoked by FTP File Servers to open the destination file for a store **write**.
5. invoked by FTP File Servers to open the source file for a retrieve **read**.
6. invoked by the Alto FTP-provided **RenameFile** primitive to open the current file **read** and to open the new file **write**.
7. invoked by the FTP-provided **CreateMailSystem** mail primitive (see Appendix F) to open its mailbox directory file (that is, FTPSysMail-Directory.Bravo) **read** and to open a scratch file for incoming messages **writeThenRead**.
8. invoked by the FTP-provided **RetrieveMessages** mail primitive (see Appendix F) to open the mailbox file (named in FTPSysMail-Directory.Bravo) **readThenWrite**.
9. invoked by the FTP-provided **DeliverMessage** mail primitive (see Appendix F) to open the mailbox file (named in FTPSysMail-Directory.Bravo) **append**.

ReadFile

1. invoked by **FTPEnumerateFiles** if **intent** is **renaming** or **unspecified** to read a scratch file for filenames.
2. invoked by **FTPStoreFile** to read the source file.
3. invoked by FTP File Servers to read the source file for a retrieve.
4. invoked by the Alto FTP-provided **RenameFile** primitive to read the current file.
5. invoked by the FTP-provided **LocateMailboxes** mail primitive (see Appendix F) to read its mailbox directory file (that is, FTPSysMail-Directory.Bravo).

6. invoked by the FTP-provided **RetrieveMessages** mail primitive (see Appendix F) to read the mailbox file (named in FTPSysMail-Directory.Bravo).
7. invoked by the FTP-provided **DeliverMessage** mail primitive (see Appendix F) to read the scratch file for incoming messages.

WriteFile

1. invoked by **FTPEnumerateFiles** if **intent** is **renaming** or **unspecified** to write a scratch file for filenames.
2. invoked by **FTPRetrieveFile** to write the destination file.
3. invoked by FTP File Servers to write the destination file for a store.
4. invoked by the Alto FTP-provided **RenameFile** primitive to write the new file.
5. invoked by the FTP-provided **StageMessage** mail primitive (see Appendix F) to write a scratch file for incoming messages.
6. invoked by the FTP-provided **DeliverMessage** mail primitive (see Appendix F) to write (that is, append to) the mailbox file (named in FTPSysMail-Directory.Bravo).

CloseFile

1. invoked by **FTPEnumerateFiles** if **intent** is **renaming** or **unspecified** to close a scratch file for filenames.
2. invoked by **FTPStoreFile** to close the source file.
3. invoked by **FTPRetrieveFile** to close the destination file.
4. invoked by FTP File Servers to close the destination file for a store.
5. invoked by FTP File Servers to close the source file for a retrieve.
6. invoked by the Alto FTP-provided **RenameFile** primitive to close the current file and to close the new file.
7. invoked by the FTP-provided **DestroyMailSystem** mail primitive (see Appendix F) to close its mailbox directory file (that is, FTPSysMail-Directory.Bravo) and to close a scratch file for incoming messages.
8. invoked by the FTP-provided **RetrieveMessages** mail primitive (see Appendix F) to close the mailbox file (named in FTPSysMail-Directory.Bravo).
9. invoked by the FTP-provided **DeliverMessage** mail primitive (see Appendix F) to close the mailbox file (named in FTPSysMail-Directory.Bravo).

DeleteFile

1. invoked by FTP File Servers when asked to delete a file.
2. invoked by the Alto FTP-provided **RenameFile** primitive to delete the current file.

RenameFile

1. invoked by FTP File Servers when asked to rename a file.

Appendix F: Client Mail Primitives

F.1. Description of the Option

An FTP Server manipulates its local mail system by means of a family of procedures called *mail primitives.* This family includes, for example, procedures for appending a message to one or more local mailboxes, for emptying a local mailbox of its contents, and for forwarding a message to one or more remote mailboxes. The FTP implementation includes one set of primitives, which implements a simple-minded mail system in terms of the file primitives described in Appendix E.

Rather than use the mail system interfaces offered by FTP, the client may, if it wishes, provide its own mail primitives to a particular FTP Listener. By so doing, a client can use FTP, for example to:

1. interface to another local mail system.
2. interface to a pseudo mail system (for example, a printer).
3. transform mailbox names (for example, convert abstract mailbox names to concrete ones).
4. control access to particular functions on a per-user or per-host basis.
5. maintain a log of mail system activity.

The client can also implement certain mail 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 mail deliveries by supplying an implementation of the **DeliverMessage** procedure (described in Section F.7) that records the event and then calls FTP's **DeliverMessage** procedure to actually deliver the message.

F.2. Exercising the Option

The client exercises the option described above by means of the **mailPrimitives** parameter required by the **FTPCreateListener** procedure. This parameter is a POINTER to a RECORD containing the PROCEDUREs by which any FTP Servers created by the newly created FTP Listener are to access the local mail system. The client may supply its own version of this record rather than rely upon the standard version offered by FTP. In constructing the record and/or implementing the procedures it contains, the client may draw upon any of the mail primitives it finds in the FTP-provided record. Since FTP will not copy the record presented to it, the client must preserve it intact until **FTPDestroyListener** is called:

```
MailPrimitives: TYPE = POINTER TO MailPrimitivesObject;
MailPrimitivesObject: TYPE = RECORD [

  -- program management primitives
 CreateMailSystem: PROCEDURE [filePrimitives: FilePrimitives, bufferSize:
   CARDINAL] RETURNS [mailSystem: MailSystem, forwardingProvided: BOOLEAN],
 DestroyMailSystem: PROCEDURE [mailSystem: MailSystem],

  -- access control primitives
 InspectCredentials: PROCEDURE [mailSystem: MailSystem, status: Status, user,
   password: STRING],
  -- identification primitives
 LocateMailboxes: PROCEDURE [mailSystem: MailSystem, localMailboxList:
   Mailbox],

  -- delivery primitives
 StageMessage: PROCEDURE [mailSystem: MailSystem,
   receiveBlock: PROCEDURE [UNSPECIFIED, POINTER, CARDINAL] RETURNS [CARDINAL],
   receiveBlockData: UNSPECIFIED],
 DeliverMessage: PROCEDURE [mailSystem: MailSystem, localMailboxList:
   Mailbox],
 ForwardMessage: PROCEDURE [mailSystem: MailSystem, remoteMailboxList:
   Mailbox],

  -- retrieval primitives
 RetrieveMessages: PROCEDURE [mailSystem: MailSystem, localMailbox: Mailbox,
   processMessage: PROCEDURE [MessageInfo],
   sendBlock: PROCEDURE [UNSPECIFIED, POINTER, CARDINAL],
   sendBlockData: UNSPECIFIED]];

FilePrimitives: TYPE = POINTER TO FilePrimitivesObject;
FilePrimitivesObject: TYPE = RECORD [...];
MailSystem: TYPE = POINTER TO MailSystemObject;
MailSystemObject: TYPE = RECORD [dummy: UNSPECIFIED];
Status: TYPE = {primary, secondary};
Mailbox: TYPE = POINTER TO MailboxObject;
MailboxObject: TYPE = RECORD [
   number: CARDINAL, mailbox, location: STRING, located, delivered: BOOLEAN,
   nextMailbox: Mailbox];
MessageInfo: TYPE = POINTER TO MessageInfoObject;
MessageInfoObject: TYPE = RECORD [
   byteCount: LONG CARDINAL, deliveryDate: STRING, opened, deleted: BOOLEAN];
```

F.3. General Characteristics

The mail primitives employed by FTP and suppliable by the client are described below. Statements that apply to all valid implementations of a primitive (that is, FTP's implementation and any a client might supply) are rendered in the standard font. Statements that apply only to the standard implementation supplied by FTP are rendered in a smaller font.

In accordance with standard Mesa exception handling conventions, mail primitives report errors by signalling. FTP catches any signal that reaches it and aborts the current transaction (with the help of the remote FTP User, as necessary). Wherever possible, client mail primitives should use the standard FTP signal, **FTPError** (described in Section 1.4), to report errors. Doing so enables the FTP Server to communicate the error to the remote FTP User in a meaningful way. The description of each procedure below includes a list of the **FtpError** values that seem, to the author, most appropriate for that primitive. Requests by mail primitive implementors for new **FtpError** values will be gladly entertained.

F.4. Program Management Primitives

FTP or its client provides two procedures for creating and destroying instances of the local mail system. The first, **CreateMailSystem**, creates a new instance of the local mail system founded upon the specified file system; the mail system will use *only* the file system whose **filePrimitives** and **bufferSize** (passed to **CreateFileSystem**) are specified. The procedure returns a handle, **mailSystem**, to the newly created mail system instance, which the caller must retain and later present to any of the other mail primitives it invokes, along with an indication, **forwardingProvided**, of whether the mail system implements the **ForwardMessage** primitive described in Section F.7. The **mailSystem** is a pointer to a private record containing all of the state information the mail system instance requires to function properly. The FTP implementation of this procedure records the specified file primitives, creates an instance of the corresponding file system, opens for read the mailbox directory file named FTPSysMail-Directory.Bravo (which is assumed to contain zero or more lines of the form: mailboxname@filename), opens for writeThenRead a scratch file in which incoming messages will be staged for delivery, and then returns with **forwardingProvided** set to FALSE:

```
CreateMailSystem: PROCEDURE [filePrimitives: FilePrimitives, bufferSize:
    CARDINAL] RETURNS [mailSystem: MailSystem, forwardingProvided: BOOLEAN];

FilePrimitives: TYPE = POINTER TO FilePrimitivesObject;
FilePrimitivesObject: TYPE = RECORD [...];
MailSystem: TYPE = POINTER TO MailSystemObject;
MailSystemObject: TYPE = RECORD [dummy: UNSPECIFIED];
```

Exceptions: credentialsMissing, requestedAccessDenied, illegalFilename, noSuchFile, fileAlreadyExists, fileBusy, noRoomForFile, fileDataError, filePrimitivesNotSpecified.

The second procedure, **DestroyMailSystem**, destroys a previously created instance of the local mail system, reclaiming any local resources allocated to it. The FTP implementation of this procedure closes the staging and directory files and then destroys the previously created file system instance:

```
DestroyMailSystem: PROCEDURE [mailSystem: MailSystem];

MailSystem: TYPE = POINTER TO MailSystemObject;
MailSystemObject: TYPE = RECORD [dummy: UNSPECIFIED];
```

Exceptions: noRoomForFile, fileDataError.

F.5. Access Control Primitives

FTP or its client provides one procedure, **InspectCredentials**, for inspecting credentials presented by the remote client. This procedure verifies and records the **primary** or **secondary** credentials--**user** and **password**--with which the next mail system access will be implicitly attempted. It is customary to require credentials when an attempt is made to empty (that is, read) a mailbox, but not when an attempt is made to deliver a message (that is, append) to a mailbox. The procedure verifies the existence of the user and the correctness of the password's and records the fact that they were (correctly) supplied; the mail primitive through which access to a particular mailbox is subsequently attempted then determines whether those credentials entitle the remote FTP User to manipulate the mailbox in the manner requested (for example, the **RetrieveMessages** primitive described in Section F.8 verifies that the client has read access to the target mailbox before honoring the retrieve request). Primary credentials typically identify the user upon whose behalf the access is attempted (*a la* the Tenex Login command) while secondary credentials, when necessary, usually identify another area of the mail system--in addition to the mailbox of the user--to which the user claims access (*a la* the Tenex Connect command). (The distinction between primary and secondary credentials is really more germane to file systems than to mail systems.)

Primary and secondary credentials (if any) are presented for inspection *immediately before* the call to the mail primitive--**DeliverMessage**, **ForwardMessage**, or **RetrieveMessages**--which attempts the access, and implicitly are discarded by the file system instance *immediately after* that operation. The FTP implementation of this procedure is a no operation:

InspectCredentials: PROCEDURE [mailSystem: MailSystem, status: Status, user, password: STRING];

Status: TYPE = {primary, secondary};

Exceptions: noSuchPrimaryUser, noSuchSecondaryUser, incorrectPrimaryPassword, incorrectSecondaryPassword.

F.6. Mailbox Identification Primitives

FTP or its client provides one procedure, **LocateMailboxes**, for verifying the existence of (and optionally locating) one or more local mailboxes. The procedure receives among its parameters a linked list, **localMailboxList**, of one or more mailbox objects. The value **NIL** in the **nextMailbox** field signals the end of the list. For every mailbox object whose **located** field is FALSE, the procedure sets **located** to TRUE if the **mailbox** named by the object exists. In addition, it optionally records in the object's **location** field, information (for example, a filename) that might assist a subsequently called primitive deliver mail to or empty the mailbox. If it makes use of the **location** field at all (FTP initializes it to **NIL**), the procedure must store in it a STRING allocated from the heap, which FTP will return to the heap when the subsequent delivery or retrieval operation is complete. The FTP implementation of this procedure searches the directory file for each mailbox name and records the corresponding filename as the location of the mailbox's:

LocateMailboxes: PROCEDURE [mailSystem: MailSystem, localMailboxList: Mailbox];

Mailbox: TYPE = POINTER TO MailboxObject;
MailboxObject: TYPE = RECORD [
number: CARDINAL, mailbox, location: STRING, located, delivered: BOOLEAN, nextMailbox: Mailbox];

F.7. Mail Delivery Primitives

FTP or its client provides three procedures for delivering messages to local and remote mailboxes. The first procedure, **StageMessage**, accepts from the remote FTP User the text of a message that subsequently is to be delivered to one or more local or remote mailboxes. The procedure supplies in zero or more calls to a caller-provided procedure, **receiveBlock**, the **receiveBlockData** supplied by **StageMessage**'s caller, and the location and length *in words* of a buffer into which the next segment of the message is to be placed. In response, **receiveBlock** returns the segment left-adjusted in the buffer, along with its length *in bytes.* **ReceiveBlock** eventually signals end of message by returning a byte count of zero. The FTP implementation of this procedure writes the text of the message onto the previously opened scratch file by means of the **WriteFile** file primitive:

```
StageMessage: PROCEDURE [mailSystem: MailSystem,
    receiveBlock: PROCEDURE [UNSPECIFIED, POINTER, CARDINAL] RETURNS [CARDINAL],
    receiveBlockData: UNSPECIFIED];
```

Exceptions: connectionTimedOut, connectionClosed, noRouteToNetwork, noRoomForFile, fileDataError.

The second procedure, **DeliverMessage**, appends a previously staged message to one or more previously verified local mailboxes. The procedure receives among its parameters a linked list, **localMailboxList**, of one or more mailbox objects. The value NIL in the **nextMailbox** field signals the end of the list. For every mailbox object whose **located** field is TRUE and **delivered** field is FALSE, the procedure attempts to deliver the message to the named **mailbox** and sets **delivered** to TRUE if successful. The procedure uses the contents of the **location** field, as set by **LocateMailboxes**, to help it deliver the mail. The FTP implementation of this procedure appends the contents of the scratch file in which the message was staged to the file named by **location**, preceding it with a header containing the current date and time and the length of the message:

```
DeliverMessage: PROCEDURE [mailSystem: MailSystem, localMailboxList: Mailbox];

Mailbox: TYPE = POINTER TO MailboxObject;
MailboxObject: TYPE = RECORD [
    number: CARDINAL, mailbox, location,: STRING, located, delivered: BOOLEAN,
    nextMailbox: Mailbox];
```

Exceptions: credentialsMissing, requestedAccessDenied, illegalFilename, noSuchFile, fileAlreadyExists, fileBusy, noRoomForFile, fileDataError.

The third procedure, **ForwardMessage**, enqueues the previously staged message for delivery to one or more remote mailboxes. The procedure receives among its parameters a linked list, **remoteMailboxList**, of one or more mailbox objects. The value NIL in the **nextMailbox** field signals the end of the list. For each mailbox object, the procedure attempts to enqueue the message for later delivery to the named **mailbox** at the remote host the name of which is stored in the **location** field, and sets **located** to TRUE if successful. The FTP implementation of this procedure, which will never be called because **CreateMailSystem** returns with **forwardingProvided** set to FALSE, is a no operation:

```
ForwardMessage: PROCEDURE [mailSystem: MailSystem, remoteMailboxList:
    Mailbox];

Mailbox: TYPE = POINTER TO MailboxObject;
MailboxObject: TYPE = RECORD [
    number: CARDINAL, mailbox, location: STRING, located, delivered: BOOLEAN,
    nextMailbox: Mailbox];
```

F.8. Mail Retrieval Primitives

FTP or its client provides one procedure, **RetrieveMessages**, for retrieving the contents of (and then resetting to empty) a local mailbox. For each of the zero or more messages in the specified **localMailbox**, the procedure first invokes the caller-supplied **processMessage** procedure with information about the message: its size in bytes, **byteCount**; the date and time, **deliveryDate**, at which the message was deposited in the local mailbox; and whether or not the message has been **opened** (that is, examined) or **deleted** while in the mailbox (Maxc mailboxes, for example, can be manipulated directly via the Tenex MSG subsystem). Following each invocation of **processMessage**, **RetrieveMessages** supplies in zero or more calls to a second FTP-provided procedure, **sendBlock**, the **sendBlockData** supplied by **RetrieveMessages**' caller and the location and length in bytes of successive segments of the message. After the entire message has been output in this manner, **RetrieveMessages** signals end of message by calling **sendBlock** a final time with a byte count of zero. After all messages have been returned, the procedure resets the contents of the mailbox to empty. **RetrieveMessages** is implemented in such a way that no new messages are lost during the retrieval process. The FTP implementation of this procedure retrieves each message's **byteCount** and **deliveryDate** from the message header and forces **opened** and **deleted** to FALSE:

```
RetrieveMessages: PROCEDURE [mailSystem: MailSystem, localMailbox: Mailbox,
    processMessage: PROCEDURE [MessageInfo],
    sendBlock: PROCEDURE [UNSPECIFIED, POINTER, CARDINAL],
    sendBlockData: UNSPECIFIED];

Mailbox: TYPE = POINTER TO MailboxObject;
MailboxObject: TYPE = RECORD [
    number: CARDINAL, mailbox, location: STRING, located, delivered: BOOLEAN,
    nextMailbox: Mailbox];
MessageInfo: TYPE = POINTER TO MessageInfoObject;
MessageInfoObject: TYPE = RECORD [
    byteCount: LONG CARDINAL, deliveryDate: STRING, opened, deleted: BOOLEAN];
```

Exceptions: connectionTimedOut, connectionClosed, noRouteToNetwork, credentialsMissing, requestedAccessDenied, illegalFilename, noSuchFile, fileAlreadyExists, fileBusy, noRoomForFile, fileDataError.

Appendix G: Client Communication Primitives

G.1. Description of the Option

An FTP User, Server, or Listener manipulates its local communication system by means of a family of procedures called *communication primitives.* This family includes, for example, procedures for opening and closing network connections, for activating and deactivating network ports (well-known sockets), and for sending and receiving data. The FTP implementation includes a set of primitives for manipulating the standard Pup communication system.

Rather than use the communication system offered by FTP, the client may, if it wishes, provide its own communication primitives to a particular FTP User or Listener. By so doing, a client can use FTP to, for example:

1. interface to another local communication system.
2. transform host names (for example, convert abstract host names to concrete ones).
3. control access to particular functions on a per-host basis.
4. maintain a log of communication system activity.

The client can also implement certain communication 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 incoming requests for connection by supplying an implementation of the **ActivatePort** procedure (described in Section G.5) that calls FTP's **ActivatePort** procedure to actually activate the port.

G.2. Exercising the Option

The client exercises the option described above by means of the **communicationPrimitives** parameter required 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 Listener (and any Servers it may create) are to access the local communication system. The client may supply its own version of this record, rather than rely upon either of the standard versions offered by FTP. In constructing the record and/or implementing the procedures it contains, the client may draw upon any of the communication primitives it finds in the FTP-provided records. Since FTP will not copy the record presented to it, the client must preserve it intact until **FTPDestroyUser** or **FTPDestroyListener** is called:

```
CommunicationPrimitives: TYPE = POINTER TO CommunicationPrimitivesObject;
CommunicationPrimitivesObject: TYPE = RECORD [

   -- program management primitives
  CreateCommunicationSystem: PROCEDURE RETURNS [communicationSystem:
     CommunicationSystem],
  DestroyCommunicationSystem: PROCEDURE [communicationSystem:
     CommunicationSystem],
    -- connection management primitives
   OpenConnection: PROCEDURE [communicationSystem: CommunicationSystem,
     remoteHost: STRING, remoteSocket: LONG CARDINAL, receiveSeconds: CARDINAL]
     RETURNS [connection: Connection],
   CloseConnection: PROCEDURE [communicationSystem: CommunicationSystem,
     connection: Connection],
   ActivatePort: PROCEDURE [communicationSystem: CommunicationSystem,
     localSocket: LONG CARDINAL,
     serviceConnection: PROCEDURE [UNSPECIFIED, Connection, STRING],
     serviceConnectionData: UNSPECIFIED, receiveSeconds: CARDINAL] RETURNS [port:
     Port],
   DeactivatePort: PROCEDURE [communicationSystem: CommunicationSystem,
     port: Port],

    -- data transmission and receipt primitives
   SendBytes: PROCEDURE [communicationSystem: CommunicationSystem,
     connection: Connection, bytePointer: BytePointer],
   ReceiveBytes: PROCEDURE [communicationSystem: CommunicationSystem,
     connection: Connection, bytePointer: BytePointer, settleForLess: BOOLEAN],
   SendByte: PROCEDURE [communicationSystem: CommunicationSystem,
     connection: Connection, byte: Byte],
   ReceiveByte: PROCEDURE [communicationSystem: CommunicationSystem,
     connection: Connection, settleForNone: BOOLEAN] RETURNS [byte: Byte,
     settledForNone: BOOLEAN],
   ProduceDiscontinuity,
   ConsumeDiscontinuity,
   ForceOutput: PROCEDURE [communicationSystem: CommunicationSystem,
     connection: Connection]];

  CommunicationSystem: TYPE = POINTER TO CommunicationSystemObject;
  CommunicationSystemObject: TYPE = RECORD [dummy: UNSPECIFIED];
  Connection: TYPE = POINTER TO ConnectionObject;
  ConnectionObject: TYPE = RECORD [dummy: UNSPECIFIED];
  Port: TYPE = POINTER TO PortObject;
  PortObject: TYPE = RECORD [dummy: UNSPECIFIED];
  BytePointer: TYPE = POINTER TO BytePointerObject;
  BytePointerObject: TYPE = RECORD [address: POINTER, offset: BOOLEAN, count:
     CARDINAL];
  Byte: TYPE = [0..377B];
```

G.3. General Characteristics

The communication primitives employed by FTP and suppliable by the client are described below. Statements that apply to all valid implementations of a primitive (that is, one or both of FTP's implementations and any a client might supply) are rendered in the standard font. Statements which apply only to one or both of the standard implementations supplied by FTP are rendered in a smaller font.

In accordance with standard Mesa exception handling conventions, communication primitives report errors by signalling. FTP catches any signal that reaches it and aborts the current transaction (with the help of the remote FTP User, Listener, or Server, as necessary). Wherever possible, client communication primitives should use the standard FTP signal, **FTPError** (described in Section 1.4), to report errors. Doing so enables the FTP User or Server to communicate the error to the remote FTP Server or User in a meaningful way. The description of each procedure below includes a list of the **FtpError** values that seem, to the author, most appropriate for that primitive. Requests by communication primitive implementors for new **FtpError** values will be gladly entertained.

The communication system provides stream-like *connections* that are used by FTP to interconnect FTP Users with FTP Servers. Over such connections flow streams of data partitioned into logical records by *discontinuities* in the streams. The communication system provides primitives for producing and consuming discontinuities, as well as for sending and receiving data. In general, data transmitted by the client (that is, FTP) are buffered by the communication system until an amount consistent with efficient use of the transmission medium has been accumulated. Before waiting for a response from the remote FTP User or Server to previously transmitted data, therefore, FTP must insure that it has actually been sent, by invoking the **ForceOutput** primitive, described in Section G.6.

Some of the data transmission and receipt primitives require among their parameters a *byte pointer* that defines the block of storage to be emptied or filled. A byte pointer is a POINTER to a RECORD containing the **address** of a storage block, the **offset** from that address to the first byte of data, and a **count** of the number of bytes to be read or written:

```
BytePointer: TYPE = POINTER TO BytePointerObject;
BytePointerObject: TYPE = RECORD [address: POINTER, offset: BOOLEAN, count:
    CARDINAL];
```

If **offset** is FALSE, the first byte of data is the left most (that is, high order) byte of the addressed word; if **offset** is TRUE, the first byte of data is the right most (that is, low order) byte of that word. In practice, the vast majority of calls on the primitives requiring byte pointers will specify an offset of FALSE. However, the generality of the communication protocols employed by FTP requires this same generality at the interface to the communication system.

Each primitive that accepts a byte pointer among its arguments has the side effect of updating that byte pointer to address the first byte of data beyond the last byte actually transmitted or received. Since the **SendBytes** procedure described in Section G.6 always transmits the entire contents of the storage block, it returns with the byte pointer updated to address the first byte beyond that block. Since the **ReceiveBytes** primitive described in Section G.7 may receive less than the requested number of bytes, it may return with the byte pointer updated to a place somewhere within the specified storage block.

G.4. Program Management Primitives

FTP or its client provides two procedures for creating and destroying instances of the local communication system. The first, **CreateCommunicationSystem**, used by both FTP User and Listener, creates a new instance of the local communication system. The procedure returns a handle, **communicationSystem**, to the newly created communication system instance, which the caller must retain and later present to any of the other communication primitives it invokes. The **communicationSystem** is a pointer to a private record containing all of the state information the communication system instance requires to function properly. The Alto implementation of this procedure turns on the Pup Package:

```
CreateCommunicationSystem: PROCEDURE RETURNS [communicationSystem:
    CommunicationSystem];
```

```
CommunicationSystem: TYPE = POINTER TO CommunicationSystemObject;
CommunicationSystemObject: TYPE = RECORD [dummy: UNSPECIFIED];
```

The second procedure, **DestroyCommunicationSystem**, used by both FTP User and Listener, destroys a previously created instance of the local communication system, reclaiming any local resources allocated to it. Before invoking this procedure, the caller must close all open connections and deactivate all active ports. The Alto implementation of this procedure destroys the Pup Package:

```
DestroyCommunicationSystem: PROCEDURE [communicationSystem:
    CommunicationSystem];

CommunicationSystem: TYPE = POINTER TO CommunicationSystemObject;
CommunicationSystemObject: TYPE = RECORD [dummy: UNSPECIFIED];
```

G.5. Connection Management Primitives

FTP or its client provides four procedures for creating and destroying network connections. The first, **OpenConnection**, used only by FTP User, establishes a **connection** to the specified **remoteHost** and **remoteSocket** and returns a handle to it. The caller specifies the interval in seconds, **receiveSeconds**, after which subsequently unfulfilled calls to the **ReceiveBytes** or **ReceiveByte** procedure described in Section G.7 are to be timed out and aborted. The distinguished value LAST[CARDINAL] requests the maximum allowed timeout interval, which may be infinite. The Alto implementation of this procedure interprets **remoteHost** as either a host name or Ethernet address, and creates a network stream to the specified socket at that host:

```
OpenConnection: PROCEDURE [communicationSystem: CommunicationSystem,
    remoteHost: STRING, remoteSocket: LONG CARDINAL, receiveSeconds: CARDINAL]
    RETURNS [connection: Connection];

Connection: TYPE = POINTER TO ConnectionObject;
ConnectionObject: TYPE = RECORD [dummy: UNSPECIFIED];
```

Exceptions: noSuchHost, connectionRejected, noRouteToNetwork, noNameLookupResponse.

The second procedure, **CloseConnection**, used only by FTP User, closes the previously opened **connection** with the specified handle. The Alto implementation of this procedure simply deletes the network stream:

```
CloseConnection: PROCEDURE [communicationSystem: CommunicationSystem,
    connection: Connection];

Connection: TYPE = POINTER TO ConnectionObject;
ConnectionObject: TYPE = RECORD [dummy: UNSPECIFIED];
```

The third procedure, **ActivatePort**, used only by FTP Listener, causes the local host to become responsive to incoming requests for connection to the specified **localSocket**, and returns a handle to the newly established local communication **port**. The caller specifies the interval in seconds, **receiveSeconds**, after which subsequently unfulfilled calls to the **ReceiveBytes** or **ReceiveByte** procedure described in Section G.7 are to be timed out and aborted. The distinguished value LAST[CARDINAL] requests the maximum allowed timeout interval, which may be infinite. As long as the port is active, the communication system will automatically create a connection for each incoming request, and invoke the caller-supplied **serviceConnection** procedure to service that connection. This procedure receives as arguments the **serviceConnectionData** supplied to **ActivatePort**, a handle for the newly established connection, and the host name (possibly expressed as a network address, depending upon the implementation) of the host that initiated the connection request. When **serviceConnection** returns, its caller (within the communication system) will close the connection. The Alto implementation of this procedure simply creates a Pup listener for the indicated socket:

```
ActivatePort: PROCEDURE [communicationSystem: CommunicationSystem,
    localSocket: LONG CARDINAL,
    serviceConnection: PROCEDURE [UNSPECIFIED, Connection, STRING],
    serviceConnectionData: UNSPECIFIED, receiveSeconds: CARDINAL] RETURNS [port:
    Port];

Connection: TYPE = POINTER TO ConnectionObject;
ConnectionObject: TYPE = RECORD [dummy: UNSPECIFIED];
Port: TYPE = POINTER TO PortObject;
PortObject: TYPE = RECORD [dummy: UNSPECIFIED];
```

The fourth procedure, **DeactivatePort**, used only by FTP Listener, deactivates the previously activated **port** with the specified handle. Existing connections remain unaffected and will continue to be serviced by the client, but no new incoming requests for connection to the affected socket will be honored The Alto implementation of this procedure sets a flag, pokes the listening process, and waits for it to destroy itself:

```
DeactivatePort: PROCEDURE [communicationSystem: CommunicationSystem, port:
    Port];

Port: TYPE = POINTER TO PortObject;
PortObject: TYPE = RECORD [dummy: UNSPECIFIED];
```

G.6. Data Transmission Primitives

FTP or its client provides four procedures for transmitting data via a previously opened connection. The first, **SendBytes**, used by both FTP User and Server, enqueues for transmission via the specified **connection**, the contents of the block of storage denoted by the specified **bytePointer**. As a side effect, the procedure updates the byte pointer to address the first byte beyond the storage block. The procedure returns after the caller's storage block has been emptied but, in general, before the data has actually been transmitted to the remote host. Data are transmitted only when an amount consistent with efficient use of the transmission medium has been accumulated by the communication system, or when the **ForceOutput** procedure described below is invoked, whichever occurs first. The Alto implementation of this procedure simply outputs the block of data on the network stream and advances the byte pointer:

```
SendBytes: PROCEDURE [communicationSystem: CommunicationSystem,
    connection: Connection, bytePointer: BytePointer];

Connection: TYPE = POINTER TO ConnectionObject;
ConnectionObject: TYPE = RECORD [dummy: UNSPECIFIED];
BytePointer: TYPE = POINTER TO BytePointerObject;
BytePointerObject: TYPE = RECORD [address: POINTER, offset: BOOLEAN, count:
    CARDINAL];
```

Exceptions: connectionTimedOut, connectionClosed, noRouteToNetwork.

The second procedure, **SendByte**, used by both FTP User and Server, enqueues for transmission via the specified **connection**, the indicated **byte** of data. In general, the procedure returns before the data has actually been transmitted to the remote host. Data are transmitted only when an amount consistent with efficient use of the transmission medium has been accumulated by the communication system, or when the **ForceOutput** procedure described below is invoked, whichever occurs first. The Alto implementations of this procedure simply outputs the byte of data on the network stream, unless it is the first byte following a discontinuity, in which case it transmits the byte in the form of a change in subsequence type:

```
SendByte: PROCEDURE [communicationSystem: CommunicationSystem,
    connection: Connection, byte: Byte];

Connection: TYPE = POINTER TO ConnectionObject;
ConnectionObject: TYPE = RECORD [dummy: UNSPECIFIED];
Byte: TYPE = [0..377B];
```

Exceptions: connectionTimedOut, connectionClosed, noRouteToNetwork.

The third procedure, **ProduceDiscontinuity**, used by both FTP User and Server, creates a discontinuity in the stream being transmitted via the specified **connection**, at a point immediately following the last byte of data sent. Multiple calls upon this procedure without intervening data are treated as no operations. The Alto implementation of this procedure simply makes note of the requested discontinuity so that the next byte of data sent via **SendByte** will be transmitted in the form of a change in subsequence type:

> **ProduceDiscontinuity:** PROCEDURE **[communicationSystem:
> CommunicationSystem, connection: Connection];**
>
> **Connection:** TYPE **=** POINTER TO **ConnectionObject;**
> **ConnectionObject:** TYPE **=** RECORD **[dummy:** UNSPECIFIED**];**
>
> Exceptions: connectionTimedOut, connectionClosed, noRouteToNetwork.

The fourth procedure, **ForceOutput**, used by both FTP User and Server, causes all data previously sent via the specified **connection** actually to be transmitted to the remote host. In the absence of calls upon this procedure, data are transmitted only when an amount consistent with efficient use of the transmission medium has been accumulated by the communication system. Multiple calls upon this procedure without intervening data are treated as no operations. The Alto implementation of this procedure simply invokes the network stream's **SendNow** primitive:

> **ForceOutput:** PROCEDURE **[communicationSystem: CommunicationSystem,
> connection: Connection];**
>
> **Connection:** TYPE **=** POINTER TO **ConnectionObject;**
> **ConnectionObject:** TYPE **=** RECORD **[dummy:** UNSPECIFIED**];**
>
> Exceptions: connectionTimedOut, connectionClosed, noRouteToNetwork.

G.7. Data Receipt Primitives

FTP or its client provides three procedures for receiving data transmitted via a previously opened connection. The first, **ReceiveBytes**, used by both FTP User and Server, accepts from the specified **connection**, some or all of the data required to fill the block of storage denoted by the specified **bytePointer**. If **settleForLess** is FALSE, the procedure completely fills the block of storage, blocking the client as long as is required to obtain the requested amount of data. If **settleForLess** is TRUE, the procedure returns whatever data the communication system may have already received, up to the capacity of the storage block, blocking the client if necessary to obtain at least one byte of data. Regardless of the value of **settleForLess**, a discontinuity in the input stream will terminate the receive operation at that point. As a side effect, the procedure updates the byte pointer to address the byte just beyond the last byte read, thereby informing the client of the quantity of data actually returned by the procedure. The Alto implementation of this procedure simply invoke the network stream's **GetBlock** procedure, terminating on end of physical record (if **settleForLess** is TRUE) or a change in subsequence type:

> **ReceiveBytes:** PROCEDURE **[communicationSystem: CommunicationSystem,
> connection: Connection, bytePointer: BytePointer, settleForLess:** BOOLEAN**];**
>
> **Connection:** TYPE **=** POINTER TO **ConnectionObject;**
> **ConnectionObject:** TYPE **=** RECORD **[dummy:** UNSPECIFIED**];**
> **BytePointer:** TYPE **=** POINTER TO **BytePointerObject;**
> **BytePointerObject:** TYPE **=** RECORD **[address:** POINTER**, offset:** BOOLEAN**, count:**
> CARDINAL**];**

Exceptions: connectionTimedOut, connectionClosed, noRouteToNetwork.

The second procedure, **ReceiveByte**, used by both FTP User and Server, returns to the caller the next **byte** of data transmited via the specified **connection**. A discontinuity in the input stream either aborts the procedure, if **settleForNone** if FALSE, or causes the procedure to return empty-handed, if **settleForNone** is TRUE. In either case, the procedure returns an indication of whether it **settledForNone**. The Alto implementation of this procedure simply invoke the network stream's **GetByte** procedure, noting and acting upon a change in subsequence type:

```
ReceiveByte: PROCEDURE [communicationSystem: CommunicationSystem,
   connection: Connection, settleForNone: BOOLEAN] RETURNS [byte: Byte,
   settledForNone: BOOLEAN];

Connection: TYPE = POINTER TO ConnectionObject;
ConnectionObject: TYPE = RECORD [dummy: UNSPECIFIED];
Byte: TYPE = [0..377B];
```

Exceptions: connectionTimedOut, connectionClosed, noRouteToNetwork.

The third procedure, **ConsumeDiscontinuity**, used by both FTP User and Server, consumes any discontinuity that might exist at the current point in the input stream being received via the specified **connection**. Unnecessary calls upon this procedure are treated as no operations. The Alto implementation of this procedure simply clears the discontinuity-pending condition, if it exists:

```
ConsumeDiscontinuity: PROCEDURE [communicationSystem:
   CommunicationSystem,  connection: Connection];

Connection: TYPE = POINTER TO ConnectionObject;
ConnectionObject: TYPE = RECORD [dummy: UNSPECIFIED];
```

Exceptions: connectionTimedOut, connectionClosed, noRouteToNetwork.

Appendix H: Sample Configuration and Program

H.1. Introduction

The sample configuration and stand-alone program presented below, which store a single file on a remote file system, illustrate FTP's use on the Alto. The reader is referred to Appendix I for the location of the necessary object files.

H.2. Sample Configuration

The programmer must include in his configuration: FTP; file, mail, and communication primitives, as appropriate; and (on the Alto) the Pup Package. In the sample configuration below, just the FTP User code is included, since the sample program creates no FTP Listener:

```
FTPCSample: CONFIGURATION
  -- import list
    IMPORTS DiskKDDefs, FrameDefs, ImageDefs, IODefs, ProcessDefs,
      SegmentDefs, StreamDefs, StringDefs, SystemDefs, TimeDefs
  -- control module
    CONTROL FTPSample
  = BEGIN

-- sample program
  FTPSample;

-- ftp
  FTPUser;

-- communication
  TinyPup;

  END. -- of FTPCSample
```

H.3. Sample Program

The sample program first initializes FTP; creates an FTP User using the file and communication 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 IRIS; stores a copy of the local file, **FTPCSample.Bcd**, in the remote file system; closes the connection to IRIS; destroys the FTP User; and finalizes FTP:

```
DIRECTORY
 FTPDefs:        FROM "FTPDefs"         USING [
   FTPCloseConnection, FTPCreateUser, FTPDestroyUser, FTPError, FTPFinalize,
   FTPInitialize, FTPOpenConnection, FTPSetCredentials, FTPStoreFile, FTPUser,
   PupCommunicationPrimitives, AltoFilePrimitives],
 ImageDefs:      FROM "ImageDefs"       USING [StopMesa],
 IODefs:         FROM "IODefs"          USING [WriteLine],
 OsStaticDefs:   FROM "OsStaticDefs"    USING [OsStatics],
 StringDefs:     FROM "StringDefs"      USING [BcplToMesaString];

FTPSample: PROGRAM
 -- import list
   IMPORTS FTPDefs, ImageDefs, IODefs, StringDefs
 = BEGIN OPEN FTPDefs, ImageDefs, IODefs, OsStaticDefs, StringDefs;

-- variables
ftpInitialized: BOOLEAN _ FALSE;
ftpuser, copy: FTPUser _ NIL;
user: STRING _ [40];
password: STRING _ [40];

-- intercept errors
BEGIN ENABLE
 BEGIN
 FTPError =>
   BEGIN
   IF message # NIL THEN WriteLine[message];
   CONTINUE;
   END;
 UNWIND =>
   BEGIN
   IF ftpuser # NIL THEN FTPDestroyUser[ftpuser];
   IF ftpInitialized THEN FTPFinalize[];
   END;
 END;

-- initialize ftp
FTPInitialize[];  ftpInitialized _ TRUE;
```

```
-- create ftp user
ftpuser _ FTPCreateUser[AltoFilePrimitives[], PupCommunicationPrimitives[]];

-- set credentials to login user and password
BcplToMesaString[OsStatics^.UserName, user];
BcplToMesaString[OsStatics^.UserPassword, password];
FTPSetCredentials[ftpuser, primary, user, password];

-- open connection, store self, and close connection
FTPOpenConnection[ftpuser, "Iris"L, files, NIL];
[] _ FTPStoreFile[ftpuser, "FTPCSample.BCD"L, "FTPCSample.BCD"L, binary];
FTPCloseConnection[ftpuser];

-- destroy ftp user
copy _ ftpuser;  ftpuser _ NIL;  FTPDestroyUser[ftpuser];

-- finalize ftp
ftpInitialized _ FALSE;  FTPFinalize[];
END; -- enable

-- return to exec
StopMesa[];

END. -- of FTPSample
```

Appendix I: Production Configurations and File Locations

I.1. Introduction

FTP is offered in a number of configurations, described below. In theory at least, it is possible to build a configuration that includes only the desired facilities. Here is a summary of what the various optional pieces do:

FTPAltoFile implements an interface to the Alto file system as described in Appendix E.

FTPPupComCool and **FTPPupComHot** implement the Pup version of the communication interface described in Appendix G. They require either TinyPup or FatPup to be included in your configuration.

FTPUserStore implements **FTPStoreFile**, as described in Section 3.5.

FTPUserRetrieve implements **FTPRetrieveFile**, as described in Section 3.5.

FTPUserDump implements the Dump Primitives described in Appendix B. If it is ommitted, **DumpBlock** (from **FTPUserStore**) and/or **LoadBlock** (from **FTPUserRetrieve**) will need dummy implementations to satisfy the Binder.

FTPUserFiles implements most of the other user file primitives (**FTPEnumerateFiles**, **FTPDeleteFile**, **FTPRenameFile**, and **FTPSetFilenameDefaults**) described in Section 3.

FTPUserXfer implements **FTPTransferFile**, as described in Appendix D.3.

FTPUserMailIn implements the mail retrieval primitives described in Appendix C.3.

FTPUserMailOut implements the mail delivery primitives described in Appendix C.2.

FTPAccessories translates error codes into text. If the text is not needed, it may be omitted if dummy routines are provided.

FTPTrace implements the trace facilities described in Appendix D, Section D.2. It may be omitted if dummy routines are provided to satisfy the Binder.

FTPSysMail plus a local file system interface of the client’s choosing is used to implement a mail server as described in Appendix F.

Dummy* implement the various routines needed to glue FTP together and/or to satisfy the Binder.

I.2. Production Configurations

The client can determine FTP's version number by means of the following constants defined in **FTPDefs**:

```
ftpMajorVersion: Byte = 6;
ftpMinorVersion: Byte = 1;

Byte: TYPE = [0..377B];
```

FTP 6.0 is written in Mesa 6.0 and FTP proper imports **Process**, **String**, and **Storage**. The FTP-provided file, mail and, communication interfaces import additional components of Mesa, and/or Pup.

The following production FTP configurations presently exist; others will be created as need for them is expressed by the user community:

FTPUser: FTP User file primitives (that is, the primitives of Sections 2 and 3 and of Appendices B and D). In calls to **FTPOpenConnection**, **purpose** must be **files**.

FTPServers: FTP Listener file and mail primitives only (that is, the primitives of Sections 2 and 4, and of Appendix D, Sections D.2 and D.4).

MTPUser: FTP User mail primitives only (that is, the primitives of Appendices C.2, C.3, and D.4). In calls to **FTPOpenConnection**, **purpose** must be **mail**. This is a minimal system primarily for use by Laurel. It does not contain **FTPAccessories** or the trace facilities of Appendix D.2.

FTPMTPUser: This configuration contains all of the user facilities for files and/or mail.

FTPAll: This configuration contains all of the user and server facilities for files and/or mail.

PupAndFTP: This is simply **TinyPup**, **EFTPSend**, and **FTPMTPUser** all packaged into one big lump to eliminate the complexities of including the correct packages.

Appendix J: Utilities

FTP needs a few routines that may be useful to other programs. They are packaged separately so that they can be included in you configuration even if you do not need the appropiate parts of FTP.

J.1. TimeExtraDefs

TimeExtraDefs is exported by **FTPUser**, **FTPMTPUser**, and **PupAndFTP**. If one of them is not needed, **TimeExtras** can be included directly.

If the argument is **NIL** or its contents cannot be parsed as a reasonable time, it returns 0. There are probably many strings easily recognizable by a person that **PackedTimeFromString** won't be able to handle, but it doesn't have any trouble with the output of the existing FTP programs (Maxc, IFS, Alto BCPL, Juniper) and **TimeDefs.AppendDayTime**.

```
PackedTimeFromString: PROCEDURE [STRING] RETURNS [TimeDefs.PackedTime];
```

J.2. DirExtraDefs

The server half of **FTPAltoFile** uses **DirExtras** to do * and # expansion. It is not exported by any of the standard **FTP** configurations.

There is only one procedure in **DirExtras**: **EnumerateDirectoryMasked**. It scans the directory, and presents to **proc** the matching files in directory order. The trailing "." on Alto file names has already been removed when **proc** is called.

```
EnumerateDirectoryMasked: PROCEDURE [
  files: STRING,
  proc: PROCDURE [fp: POINTER TO FP, file: STRING] RETURNS [BOOLEAN] ];
```