ChorusOS man pages section
3FTPD: FTP Daemon Library

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900
U.S.A.

Part No: 806-3331
December 10, 1999

Contents

PREFACE

Overview

A man page is provided for both the naive user, and sophisticated user who is familiar with the ChorusOS™ operating system and is in need of on-line information. A man page is intended to answer concisely the question “What does it do?” The man pages in general comprise a reference manual. They are not intended to be a tutorial.

The following is a list of sections in the ChorusOS man pages and the information it references:

- *Section 1CC: User Utilities; Host and Target Utilities*
- *Section 1M: System Management Utilities*
- *Section 2DL: System Calls; Data Link Services*
- *Section 2K: System Calls; Kernel Services*
- *Section 2MON: System Calls; Monitoring Services*
- *Section 2POSIX: System Calls; POSIX System Calls*
- *Section 2RESTART: System Calls; Hot Restart and Persistent Memory*
- *Section 2SEG: System Calls; Virtual Memory Segment Services*
- *Section 3FTPD: Libraries; FTP Daemon*
- *Section 3M: Libraries; Mathematical Libraries*
- *Section 3POSIX: Libraries; POSIX Library Functions*
- *Section 3RPC: Libraries; RPC Services*
- *Section 3STDC: Libraries; Standard C Library Functions*
- *Section 3TELD: Libraries; Telnet Services*
- *Section 4CC: Files*

- *Section 5FEA: ChorusOS Features and APIs*
- *Section 7P: Protocols*
- *Section 7S: Services*
- *Section 9DDI: Device Driver Interfaces*
- *Section 9DKI: Driver to Kernel Interface*
- *Section 9DRV: Driver Implementations*

ChorusOS man pages are grouped in Reference Manuals, with one reference manual per section.

Below is a generic format for man pages. The man pages of each manual section generally follow this order, but include only needed headings. For example, if there are no bugs to report, there is no BUGS section. See the `intro` pages for more information and detail about each section, and `man`(1) for more information about man pages in general.

NAME — This section gives the names of the commands or functions documented, followed by a brief description of what they do.

SYNOPSIS — This section shows the syntax of commands or functions. When a command or file does not exist in the standard path, its full pathname is shown. Options and arguments are alphabetized, with single letter arguments first, and options with arguments next, unless a different argument order is required.

The following special characters are used in this section:

[ ] — The option or argument enclosed in these brackets is optional. If the brackets are omitted, the argument must be specified.

. . . — Ellipses. Several values may be provided for the previous argument, or the previous argument can be specified multiple times, for example, ‘ "`filename` . . ." .

| — Separator. Only one of the arguments separated by this character can be specified at time.

{ } — Braces. The options and / or arguments enclosed within braces are

	interdependent, such that everything enclosed must be treated as a unit.
FEATURES	This section provides the list of features which offer an interface. An API may be associated with one or more system features. The interface will be available if one of the associated features has been configured.
DESCRIPTION	This section defines the functionality and behavior of the service. Thus it describes concisely what the command does. It does not discuss OPTIONS or cite EXAMPLES.. Interactive commands, subcommands, requests, macros, functions and such, are described under USAGE.
OPTIONS	This lists the command options with a concise summary of what each option does. The options are listed literally and in the order they appear in the SYNOPSIS section. Possible arguments to options are discussed under the option, and where appropriate, default values are supplied.
OPERANDS	This section lists the command operands and describes how they affect the actions of the command.
OUTPUT	This section describes the output - standard output, standard error, or output files - generated by the command.
RETURN VALUES	If the man page documents functions that return values, this section lists these values and describes the conditions under which they are returned. If a function can return only constant values, such as 0 or –1, these values are listed in tagged paragraphs. Otherwise, a single paragraph describes the return values of each function. Functions declared void do not return values, so they are not discussed in RETURN VALUES.
ERRORS	On failure, most functions place an error code in the global variable `errno` indicating why they failed. This section lists alphabetically all error codes a function can generate and describes the conditions that cause each error. When more than one condition can cause the same error, each condition is described in a separate paragraph under the error code.

USAGE	This section is provided as a guidance on use. This section lists special rules, features and commands that require in-depth explanations. The subsections listed below are used to explain built-in functionality: Commands Modifiers Variables Expressions Input Grammar
EXAMPLES	This section provides examples of usage or of how to use a command or function. Wherever possible a complete example including command line entry and machine response is shown. Whenever an example is given, the prompt is shown as `example%` or if the user must be superuser, `example#`. Examples are followed by explanations, variable substitution rules, or returned values. Most examples illustrate concepts from the SYNOPSIS, DESCRIPTION, OPTIONS and USAGE sections.
ENVIRONMENT VARIABLES	This section lists any environment variables that the command or function affects, followed by a brief description of the effect.
EXIT STATUS	This section lists the values the command returns to the calling program or shell and the conditions that cause these values to be returned. Usually, zero is returned for successful completion and values other than zero for various error conditions.
FILES	This section lists all filenames referred to by the man page, files of interest, and files created or required by commands. Each is followed by a descriptive summary or explanation.
SEE ALSO	This section lists references to other man pages, in-house documentation and outside publications.
DIAGNOSTICS	This section lists diagnostic messages with a brief explanation of the condition causing the error.
WARNINGS	This section lists warnings about special conditions which could seriously affect your working conditions. This is not a list of diagnostics.
NOTES	This section lists additional information that does not belong anywhere else on the page. It takes the form of an aside to the user, covering points of special interest. Critical information is never covered here.

BUGS	This section describes known bugs and wherever possible, suggests workarounds.

CHAPTER

FTPD Library Functions

NAME

Intro – introduction to the FTPD library

SYNOPSIS

```
#include <arpa/ftpd/ftpd.h>
#include <arpa/ftpd/systemLog.h>
#include <arpa/ftpd/systemSleep.h>
#include <arpa/ftpd/systemAuth.h>
#include <arpa/ftpd/systemFilesys.h>
```

DESCRIPTION

The FTPD library is a module to be linked with application code in order to enable the server to implement the FTP protocol, but with authentication, logging, and file system operations in a manner suitable to the particular application. The FTPD library provides all the code that is deemed invariant by the FTP protocol, that is, everything needed to meet the requirements of a standard FTP client. The application should provide all the code that may change according to the underlying operating system semantics, or according to the objectives of a particular application.

An example of applying this flexible approach to the FTPD implementation is the implementation of a file-systemless FTP server, which stores and retrieves data to and from a medium managed entirely by the application code, without involvement of the operating system's file management features.

FTPD has been split into two main parts, reflecting how the FTPD library and the application code interact:

- One part, called "The FTPD Library", or "The Library Code", is implemented by the FTPD library. It takes care of as much as was deemed possible of what is invariant in an FTP server, for example: interpreting the commands, opening and closing data connections with the client.
- The other part is called "The Application Code" or "The Application Side" and is provided by the application code linked with the FTPD library (see routines below). This part is expected to supply the thread or threads that run the whole code, to perform authentication and logging, and to store / retrieve data to / from files.

```
int          in;                    /* control connection (in file desc.) */
int          out;                   /* control connection (out file desc.) */
char         host_name[];           /* client host name */
struct       sockaddr_in   host_addr; /* client host address */
char         name[];                /* user's logging name */
char*        passwd;                /* password entered by user */
int          guest;                 /* user is a guest */
char*        fileName;              /* actual file name */
char*        dir;                   /* current directory */
char*        shell;                 /* user's shell */
off_t        byte_count;            /* bytes transferred so far */
CleanupFunc  cleanup;               /* called on ABORT */
```

The application code could extend this structure with application-dependent state information. For example, the application definition of a connection could look like this:

```
typedef struct _ClxFtpConn {
  FtpConn ftpconn;

  /* cleanup state */
  FILE*   toFclose;
  void*   toFree;
  int     toClose;
  glob_t* toGlobfree;
  DIR*    toClosedir;
} ClxFtpConn;
```

The main routine is part of the application and controls how the service is set up. The generic code supplies three routines to help to set up the service:

```
int ftpdStartSrv(int portNumber, int argc, char* argv[], char** envp);
```

Interprets the command-line arguments, creates the main port and listens on it. It then returns the newly created socket. All that needs to be provided is the port number that should be listened on.

```
int ftpdGetCnx(FtpConn* conn, int socket);
```

Initializes the state data for one connection, accepts that one connection, and returns. At this point the application side can create a new thread to continue handling the connection. All that needs be provided is the space to store the connection data and the socket from which to accept data (as returned by the previous routine).

```
int ftpdHandleCnx(FtpConn* conn);
```

Manages the new connection. All that needs be provided is the connection state as initialized by the previous routine. This routine only returns when the session is finished. Until then, the treatment is driven by the generic code, which calls back routines provided by the application.

In some implementations some of these steps are not needed; typically, in a traditional UNIX implementation, the first two steps are taken care of by `inetd`. In that case, the control connection is already open and accessible via `stdin` and `stdout`. All that `main` has to do is to set up the connection so that the in and out channels point at `stdin` and `stdout`, and call *ftpdHandleCnx*.

The following is an example of a main routine:

```
    int
main(int argc, char* argv[], char** envp)
{
    FtpConn* conn;
    int mainSock;
    int error;

    mainSock = ftpdStartSrv(2600, argc, argv, envp);
    if (mainSock < 0) exit(1);

    conn = (FtpConn*) malloc(sizeof(ClxFtpConn));
    if (conn == NULL) exit(1);

    while (1) {
        bzero(conn, sizeof(ClxFtpConn));
        if (ftpdGetCnx(conn, mainSock) < 0) exit(1);

        if (ftpdHandleCnx(conn) != 0) {
            printf("session aborted\\n");
        } else {
            printf("session terminated normaly\\n");
        }
    }
}
```

The reason for allocating the connection structure here is that the system-dependent implementation will probably need to extend the structure with extra state information. If this extra information needs to be initialized, this is the right place to do it (hence the `bzero` in this example).

You may also prefer to spawn a new thread which would call *ftpdHandleCnx()* and have the main thread go back to *ftpdGetCnx()* to accept a new session. This example handles one session at a time.

The system routines provided by the application and invoked by the FTPD library to perform FTP commands may have to output error messages, the cause of an error being specific to the application. The FTPD library also provides an interface for that:

```
void reply(FtpConn* conn, int number, const char* message, ...);

void perror_reply(FtpConn* conn, int number, const char* message, ...);


void lreply(FtpConn* conn, int number, const char* message, ...);
```

In all three cases, the message and the following arguments follow the same rules as *printf*. The number is defined by the FTP protocol to reflect the reason for the message being issued. For each routine only certain numbers are valid, as

defined by RFC 959. Use the numbers listed in the manual page of the routine. Note, however, that the lists are not exhaustive. The ones listed in the man pages are those actually used by the BSD implementation. If you require other reply types, check in RFC 959.

Both *reply()* and *perror_reply()* are final, and only one of either type should be issued per invocation of any system module routine. The difference between *reply* and *perror_reply* is that *perror_reply* automatically adds the standard string implied by the current value of `errno`.

If a multiple line reply is needed, use *lreply*. Multiple *lreply()* calls can be used, followed by one final *reply()*. Not all routines are expected to supply an error reply, no routine may use *reply* when returning an OK result. The OK reply is always performed by the library. Only *systemUser()* and *systemPasswd()* are allowed to use *lreply()* when returning an OK status. In general, supply an error or OK reply or *lreply* only if the manual page mentions one. The routines that the application code must provide fall into the following four categories:

Authentication:

```
int systemUser(FtpConn* conn);
int systemPass(FtpConn* conn);
void systemBesuper(FtpConn* conn);
void systemBeuser(FtpConn* conn);
void systemBeany(FtpConn* conn);
```

Logging:

```
void systemLog(int level, const char* format);
void systemVlog(int level, const char* format, va_list ap);
void systemSetThreadTitle(const char *fmt, ...);
void systemLogwtmp(FtpConn* conn);
```

File System:

```
int systemChdir(FtpConn* conn, char* name);
int systemFileSize(FtpConn* conn, char* name, off_t* size);
int systemMkdir(FtpConn* conn, char* name);
int systemRmdir(FtpConn* conn, char* name);
int systemDelete(FtpConn* conn, char* name);
int systemRename(FtpConn* conn, char* old, char* new);
char * systemGunique(FtpConn* conn, char* local);
off_t systemAsciiOff(FtpConn* conn, char* name, int lines);
int systemReceiveBin(FtpConn* conn, FILE* instr, char* name, off_t offset);
int systemReceiveAscii(FtpConn* conn, FILE* instr, char* name, off_t offset);
int systemSendBin(FtpConn* conn, char* name, FILE* outstr, off_t offset);
int systemSendAscii(FtpConn* conn, char* name, FILE* outstr, off_t offset);
void systemCommand(FtpConn* conn, char* cmd, FILE* outstr);
int systemListFiles(FtpConn* conn, char* name, FILE* outstr, int isAscii);
```

Sleep:

```
void systemSleep(int t);
```

The File System category needs to be fairly completely implemented, while Logging may well do nothing at all and Authentication may mostly be inaccurate. Providing Sleep is optional. The manual pages describe what they are expected to do.

The FTP protocol allows the user to abort a file transfer at any time (usually by pressing Ctrl-C). To avoid relying on an asynchronous signal delivery model, the FTPD library supports synchronous checking of this event and handles its occurrence automatically. It works in the following way:

The FTPD library provides a function called *tpdOob* that checks for the occurrence of exceptional events on the control connection (such as the one generated by an ABORT command from the client). It is the responsibility of the application side to call that routine from time to time when performing a lengthy file transfer. Typically, calling this routine between every block read or write should be sufficient. The full prototype of this routine is:

```
void ftpdOob(FtpConn* conn);
```

If an abort command is issued to cancel the transfer, the flow of control will *longjmp* out of the file transfer operation and call a cleanup routine to release the resources associated with the transfer. This cleanup routine must be provided by the application side, and the *cleanup* member of the connection structure must point to this routine. If no cleanup is needed, *conn–>cleanup* may be left to NULL.

A typical file transfer routine is as follows (error checks have been removed for clarity) :

```
    int systemSendBin(FtpConn* conn, char* name, FILE* outstr, off_t offset)
{
    int cnt;
    ClxFtpConn* myConn = (ClxFtpConn*) conn;
    int fdin = open(name, O_RDONLY);
    int fdout = fileno(outstr);
    char * buf = malloc(BLOCKSIZE);

    myConn–>toFree = buf;
    myConn–>toClose = fdout;

    conn–>cleanup = (CleanupFunc) cleanup;

    while ((cnt = read(fdin, buf, BLOCKSIZE)) > 0) {
      write(fdout, buf, cnt);
          /* chek urg */
      ftpdOob(conn);
    }

    close(fdin);
    free(buf);
    return 0;
}
```

In this case, the cleanup function would look as follows:

```
static void cleanup(ApplFtpConn* conn)
{
    if (conn->toFree != NULL) free(conn->toFree);
    if (conn->toClose >= 0) close(conn->toClose);
    ((FtpConn*) conn)->cleanup = NULL;
}
```

The above example assumes that the application has extended the `FtpConn` structure with two members; *toClose* and *toFree*. In the case of a single connection application, you can use global variables instead of extending the connection structure.

LIBRARY ENTRY-POINTS

These are: *reply*, *lreply*, *perror_reply*, *ftpdStartSrv*, *ftpdGetCnx*, *ftpdHandleCnx*.

APPLICATION-SIDE ROUTINES

These are: *systemUser*, *systemPass*, *systemInituser*, *systemBeuser*, *systemBesuper*, *systemBeany*, *systemLog*, *systemVlog*, *systemSetThreadTitle*, *systemLogwtmp*, *systemChdir*, *systemFileSize*, *systemMkdir*, *systemRmdir*, *systemDelete*, *systemRename*, *systemGunique*, *systemAsciiOff*, *systemReceiveBin*, *systemReceiveAscii*, *systemSendBin*, *systemSendAscii*, *systemCommand*, *systemListFiles*, *systemSleep*.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

Name	Description
`ftpdGetCnx`(3FTPD)	Accepts a new FTP connection
`ftpdHandleCnx`(3FTPD)	Manages an FTP connection
`ftpdOob`(3FTPD)	Check for out of band data on the control connection
`ftpdStartSrv`(3FTPD)	Initializes FTP service
`lreply`(3FTPD)	See `reply`(3FTPD)
`perror_reply`(3FTPD)	See `reply`(3FTPD)
`reply`(3FTPD)	Reply to an FTP client
`systemAsciiOff`(3FTPD)	Reports offset of text offset in file

systemBeany(3FTPD)	See systemBeuser(3FTPD)
systemBesuper(3FTPD)	See systemBeuser(3FTPD)
systemBeuser(3FTPD)	Switch and lock user id
systemChdir(3FTPD)	Change the current directory for the given connection
systemCommand(3FTPD)	Performs the given command
systemDelete(3FTPD)	Removes file specified
systemFileSize(3FTPD)	Reports the presence and size of the file specified
systemGunique(3FTPD)	Creates a name for a new file
systemLinesToOff(3FTPD)	Reports offset of line in text file
systemListFiles(3FTPD)	Lists the files matching the pattern specified
systemLog(3FTPD)	Adds the text given to the log
systemLogwtmp(3FTPD)	Record the given connection to wtmp
systemMkdir(3FTPD)	Create a directory of the name specified
systemPass(3FTPD)	Checks the user password
systemReceiveAscii(3FTPD)	Stores text or binary data in the file specified
systemReceiveBin(3FTPD)	See systemReceiveAscii(3FTPD)
systemRename(3FTPD)	Moves a file
systemRmdir(3FTPD)	Removes the directory specified
systemSendAscii(3FTPD)	Retrieves text or binary data from the file specified
systemSendBin(3FTPD)	See systemSendAscii(3FTPD)
systemSetThreadTitle(3FTPD)	Names the current thread with the text given
systemSleep(3FTPD)	Sleep for the given number of seconds
systemUser(3FTPD)	Checks the user login
systemVlog(3FTPD)	See systemLog(3FTPD)

NAME

ftpdGetCnx – Accepts a new FTP connection

SYNOPSIS

#include <arpa/ftpd/ftpd.h>
int **ftpdGetCnx**(FtpConn **conn*, int *socket*);

DESCRIPTION

The *ftpdGetCnx* function initializes the connection structure pointed to by *conn*, accepts one incoming connection from the socket *socket*, connects the new socket to the connection's in and out descriptors, and returns. After calling this routine, *conn–>in* and *conn–>out* are the file descriptors to use for reading and writing the control connection.

RETURN VALUE

Returns 0 if one connection was successfully accepted, -1 otherwise.

NOTE

When this routine returns, the application can create a new thread to continue the next treatment, if required. In most cases, *socket* should be the value returned by a previous call to *ftpdStartSrv*(3FTPD).

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

ftpdHandleCnx – Manages an FTP connection

SYNOPSIS

```
#include <arpa/ftpd/ftpd.h>
int ftpdHandleCnx(FtpConn *conn);
```

DESCRIPTION

The *ftpdHandleCnx* function manages a connection until the client disconnects. The main flow of execution takes place in the FTPD library which interprets the client's commands, and translates them into actions to be performed. Some of these actions will call back a number of routines to be provided by the application. These routines are described in the introduction manual page. This routine interprets *conn–>in* and *conn–>out* as the file descriptors corresponding to the input and output of the control connection (they may be, and usually are, identical). If *conn* was the first argument to a prior call to *ftpdGetCnx*(3FTPD), *conn–>in* and *conn–>out* are initialized correctly.

RETURN VALUES

The *ftpdHandleCnx* function returns the traditional ftpd status code: `0` if the session ended normally; `1` otherwise.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

ftpdOob – Check for out of band data on the control connection

SYNOPSIS

```
#include <arpa/ftpd/ftpd.h>
void ftpdOob(FtpConn *conn);
```

DESCRIPTION

The *ftpdOob* function checks for the occurrence of exceptional conditions on the control connection. An exceptional condition is either an ABORT request sent by the client, or the closing of the control connection by the client. File transfer or directory listing operations can only be aborted due to an exceptional condition by calling *ftpdOob*. Application routines that perform lengthy operations should call *ftpdOob* from time to time. Typically, a file transfer routine should call it between each block read from or written to a file. Calling *ftpdOob* is not mandatory. The consequence of not calling it is that ABORT commands are processed only when the ongoing transfer is finished. However, only certain routines may call *ftpdOob*. These are the following: *systemSendAscii*, *systemSendBin*, *systemReceiveBin*, *systemReceiveAscii* and *systemListFiles*.

NOTE

As a result of calling this function, the flow of control may *longjmp*(3STDC) out of the routine being executed. To perform a cleanup of the global state, which may have been affected by the interrupted routine, the following function call will be performed automatically *PRIOR* to effecting the longjmp:

```
*(conn->cleanup)(conn)
```

The routine that calls *ftpdOob* must ensure that *conn–>cleanup* points to the correct cleanup routine *PRIOR* to calling *ftpdOob*. If no cleanup routine is needed, *conn–>cleanup* should be set to NULL. If the supplied cleanup routine needs any information to perform its task, this information should be stored in the connection structure. The *ftpConn* structure can be extended by the application into a bigger, compatible one in which to store the information. As *ftpdOob* is called explicitly by the application, the application need not keep the connection ready for cleanup all the time. It should be ready for cleanup only when *ftpdOob* is called.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

ftpdStartSrv – Initializes FTP service

SYNOPSIS

```
#include <arpa/ftpd/ftpd.h>
int ftpdStartSrv(int portNb, int argc, char *argv[], char **envp);
```

DESCRIPTION

The *ftpdStartSrv* function interprets the command-line arguments passed, creates a socket, binds it to *portNb*, and listens to it using *listen*(2POSIX). It is possible to have the FTP server built using the FTPD library recognize the command-line options that the BSD implementation of FTPD recognizes. The behaviors that these command-line options affect are controlled entirely by the FTPD library. However, as it is the application's main routine that receives these options as its arguments, the application must hand over these arguments to the FTPD library for processing. This is achieved by setting *argc*, *argv* and *envp* to the values of the application's main routine parameters. This the simplest method, but it does imply that the application accepts the exact set of arguments that a BSD FTP server accepts. If this is not the case, the application must build an argument vector and count that reflect the arguments that must be passed to the FTPD library. During normal operation, no arguments are required. If *argc* is set to 0, *argv* and *envp* are ignored and *ftpdStartSrv* will behave as though *argv[0]* pointed to the string "ftpd" and *argc* was 1. Otherwise, the meaning of the arguments is the following:

OPTIONS

–d	Debugging information is logged to the application–defined log mechanism.
–l	Each FTP session is logged to the application–defined log mechanism.
–t *timeout*	Set the inactivity timeout period to *timeout* seconds. The maximum is two hours, the default is 15 minutes.

RETURN VALUES

The file descriptor of the socket created, or -1 if not successful.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

reply, lreply, perror_reply – Reply to an FTP client

SYNOPSIS

#include <arpa/ftpd/ftpd.h>
void **reply**(FtpConn * *conn*, int *number*, char * *format*, ...);

void **lreply**(FtpConn * *conn*, int *number*, char * *format*, ...);

void **perror_reply**(FtpConn * *conn*, int *number*, char * *format*, ...);

DESCRIPTION

These three routines all issue a message to the client with an associated condition code. These routines are used by the FTPD library as well as by the application-side routines to report the result of an operation. The *message* and the following arguments follow the same rules as *printf(3STDC)* . The *number* argument is defined by the FTP protocol to reflect the reason for the message being issued. For each routine, only certain numbers are valid, as defined by RFC 959. Each application-side routine manual page lists a number of valid replies along with the type of reason to issue each of them. The lists are not exhaustive, the ones mentioned in the manual pages are those actually used by the BSD implementation. If you require other reply types, check in RFC 959. The *reply* and *perror_reply* functions are final, only one of either type should be called per invocation of any application-side routine. The difference between *reply* and *perror_reply* is that *perror_reply* automatically adds the standard string implied by the current value of errno. The *lreply* function should be used if a multiple–line reply is required. Multiple *lreply()* calls can be used, followed by one final *reply()* .

NOTES

Not all routines are expected to issue an error reply. Only *systemUser* (3FTPD), and *systemPass* (3FTPD) can supply an OK reply, by using *lreply* . The final *lreply* is supplied by the FTPD library. Supply an error or OK reply only if the manual page mentions one.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

reply, lreply, perror_reply – Reply to an FTP client

SYNOPSIS

```
#include <arpa/ftpd/ftpd.h>
void reply(FtpConn * conn, int number, char * format, ... );

void lreply(FtpConn * conn, int number, char * format, ... );

void perror_reply(FtpConn * conn, int number, char * format, ... );
```

DESCRIPTION

These three routines all issue a message to the client with an associated condition code. These routines are used by the FTPD library as well as by the application-side routines to report the result of an operation. The *message* and the following arguments follow the same rules as *printf(3STDC)* . The *number* argument is defined by the FTP protocol to reflect the reason for the message being issued. For each routine, only certain numbers are valid, as defined by RFC 959. Each application-side routine manual page lists a number of valid replies along with the type of reason to issue each of them. The lists are not exhaustive, the ones mentioned in the manual pages are those actually used by the BSD implementation. If you require other reply types, check in RFC 959. The *reply* and *perror_reply* functions are final, only one of either type should be called per invocation of any application-side routine. The difference between *reply* and *perror_reply* is that *perror_reply* automatically adds the standard string implied by the current value of errno. The *lreply* function should be used if a multiple–line reply is required. Multiple *lreply()* calls can be used, followed by one final *reply()* .

NOTES

Not all routines are expected to issue an error reply. Only *systemUser* (3FTPD), and *systemPass* (3FTPD) can supply an OK reply, by using *lreply* . The final *lreply* is supplied by the FTPD library. Supply an error or OK reply only if the manual page mentions one.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

reply, lreply, perror_reply – Reply to an FTP client

SYNOPSIS

```
#include <arpa/ftpd/ftpd.h>
void reply(FtpConn * conn, int number, char * format, ... );

void lreply(FtpConn * conn, int number, char * format, ... );

void perror_reply(FtpConn * conn, int number, char * format, ... );
```

DESCRIPTION

These three routines all issue a message to the client with an associated condition code. These routines are used by the FTPD library as well as by the application-side routines to report the result of an operation. The *message* and the following arguments follow the same rules as *printf(3STDC)* . The *number* argument is defined by the FTP protocol to reflect the reason for the message being issued. For each routine, only certain numbers are valid, as defined by RFC 959. Each application-side routine manual page lists a number of valid replies along with the type of reason to issue each of them. The lists are not exhaustive, the ones mentioned in the manual pages are those actually used by the BSD implementation. If you require other reply types, check in RFC 959. The *reply* and *perror_reply* functions are final, only one of either type should be called per invocation of any application-side routine. The difference between *reply* and *perror_reply* is that *perror_reply* automatically adds the standard string implied by the current value of errno. The *lreply* function should be used if a multiple–line reply is required. Multiple *lreply()* calls can be used, followed by one final *reply()* .

NOTES

Not all routines are expected to issue an error reply. Only *systemUser* (3FTPD), and *systemPass* (3FTPD) can supply an OK reply, by using *lreply* . The final *lreply* is supplied by the FTPD library. Supply an error or OK reply only if the manual page mentions one.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

systemAsciiOff – Reports offset of text offset in file

SYNOPSIS

```
#include <arpa/ftpd/systemFilesys.h>
off_t systemAsciiOff(FtpConn *conn, char *name, int offset);
```

DESCRIPTION

The *systemAsciiOff* function converts an offset expressed relatively to the FTP-encoded version (that is, EOL is \r\n) of the text contained by the file given, to the corresponding offset in the file itself. The routine behaves as though the file had been read, converting it on the fly to the FTP text format. When the resulting text is *offset* bytes long, the actual number of characters read is returned.

RETURN VALUES

If succesful, the offset thus computed is returned, or -1 in case of failure.

ERROR MESSAGES

perror_reply(conn, 451, "...")	For a resource allocation problem
perror_reply(conn, 553, "(file name)")	For a file name problem
perror_reply(conn, 550, "(file name)")	For a file seek problem (including an incorrect number of characters in the file)

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

systemBeuser, systemBesuper, systemBeany – Switch and lock user id

SYNOPSIS

#include <arpa/ftpd/systemAuth.h>
void **systemBeuser**(FtpConn * *conn*);

void **systemBesuper**(FtpConn * *conn*);

void **systemBeany**(FtpConn * *conn*);

DESCRIPTION

These routines deal with the credentials attached to a particular connection. As identity is expected to be a global property of the server, there must be some synchronization if several connections with differing credentials are handled simultaneously. The model is the following: Normally, the server's effective uid does not matter. When one thread actually needs a particular uid for a particular operation, a designated lock is taken and the euid is changed. When the operation is completed, the lock is released. If the application is mono-threaded, no lock is necessary. If credentials are irrelevant, these routines do nothing. The *systemBeuser* function acquires the identity lock and sets the server's identity to that associated with the connection. The *systemBesuser* function acquires the identity lock and sets the server's identity to that of the superuser. The *systemBeany* function releases the identity lock. The following is an example of the routines that deal with multiple connections and real credentials. They assume that *conn–>user_uid* and *conn–>super_user* are application-dependent extensions of the *FtpConn* structure, initialized by *systemPass* (3FTPD). The user's name can also be used.

```
static KnMutex credMutex = K_KNMUTEX_INITIALIZER;

void
systemBeuser(ApplFtpConn* conn) {
  mutexGet(&credMutex);
  seteuid(conn–>user_uid);
}
void
systemBesuper(ApplFtpConn* conn) {
  mutexGet(&credMutex);
  seteuid(conn–>super_user);
}
void
systemBeany(ApplFtpConn* conn) {
  mutexRel(&credMutex);
}
```

NOTES

The FTP library only uses *systemBesuper* and *systemBeany* , but it is expected that application side routines that open files will use *systemBeuser* before opening a file, and use *systemBeany* once the file is open.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

systemBeuser, systemBesuper, systemBeany – Switch and lock user id

SYNOPSIS

#include <arpa/ftpd/systemAuth.h>
void **systemBeuser**(FtpConn * *conn*);

void **systemBesuper**(FtpConn * *conn*);

void **systemBeany**(FtpConn * *conn*);

DESCRIPTION

These routines deal with the credentials attached to a particular connection. As identity is expected to be a global property of the server, there must be some synchronization if several connections with differing credentials are handled simultaneously. The model is the following: Normally, the server's effective uid does not matter. When one thread actually needs a particular uid for a particular operation, a designated lock is taken and the euid is changed. When the operation is completed, the lock is released. If the application is mono-threaded, no lock is necessary. If credentials are irrelevant, these routines do nothing. The *systemBeuser* function acquires the identity lock and sets the server's identity to that associated with the connection. The *systemBesuser* function acquires the identity lock and sets the server's identity to that of the superuser. The *systemBeany* function releases the identity lock. The following is an example of the routines that deal with multiple connections and real credentials. They assume that *conn–>user_uid* and *conn–>super_user* are application-dependent extensions of the *FtpConn* structure, initialized by *systemPass* (3FTPD). The user's name can also be used.

```
static KnMutex credMutex = K_KNMUTEX_INITIALIZER;

void
systemBeuser(ApplFtpConn* conn) {
  mutexGet(&credMutex);
  seteuid(conn–>user_uid);
}
void
systemBesuper(ApplFtpConn* conn) {
  mutexGet(&credMutex);
  seteuid(conn–>super_user);
}
void
systemBeany(ApplFtpConn* conn) {
  mutexRel(&credMutex);
}
```

NOTES

The FTP library only uses *systemBesuper* and *systemBeany* , but it is expected that application side routines that open files will use *systemBeuser* before opening a file, and use *systemBeany* once the file is open.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

systemBeuser, systemBesuper, systemBeany – Switch and lock user id

SYNOPSIS

#include <arpa/ftpd/systemAuth.h>
void **systemBeuser**(FtpConn * *conn*);

void **systemBesuper**(FtpConn * *conn*);

void **systemBeany**(FtpConn * *conn*);

DESCRIPTION

These routines deal with the credentials attached to a particular connection. As identity is expected to be a global property of the server, there must be some synchronization if several connections with differing credentials are handled simultaneously. The model is the following: Normally, the server's effective uid does not matter. When one thread actually needs a particular uid for a particular operation, a designated lock is taken and the euid is changed. When the operation is completed, the lock is released. If the application is mono-threaded, no lock is necessary. If credentials are irrelevant, these routines do nothing. The *systemBeuser* function acquires the identity lock and sets the server's identity to that associated with the connection. The *systemBesuser* function acquires the identity lock and sets the server's identity to that of the superuser. The *systemBeany* function releases the identity lock. The following is an example of the routines that deal with multiple connections and real credentials. They assume that *conn–>user_uid* and *conn–>super_user* are application-dependent extensions of the *FtpConn* structure, initialized by *systemPass* (3FTPD). The user's name can also be used.

```
static KnMutex credMutex = K_KNMUTEX_INITIALIZER;

void
systemBeuser(ApplFtpConn* conn) {
  mutexGet(&credMutex);
  seteuid(conn–>user_uid);
}
void
systemBesuper(ApplFtpConn* conn) {
  mutexGet(&credMutex);
  seteuid(conn–>super_user);
}
void
systemBeany(ApplFtpConn* conn) {
  mutexRel(&credMutex);
}
```

NOTES

The FTP library only uses *systemBesuper* and *systemBeany* , but it is expected that application side routines that open files will use *systemBeuser* before opening a file, and use *systemBeany* once the file is open.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

systemChdir – Change the current directory for the given connection

SYNOPSIS

```
#include <arpa/ftpd/systemFilesys.h>
int systemChdir(FtpConn *conn, char *name);
```

DESCRIPTION

Changes the current directory for that particular connection to *name*. The change applies to that connection only. The effect depends on the implementation. If the server is meant to service precisely one connection, this may be implemented by actually changing the global current directory of the server. If the server is supposed to serve simultaneous connections, the new current directory may be associated with the connection either as a string (the *conn–>dir* member is readily available for that purpose); or as more application-dependent data in the application-dependent part of the connection structure (fd of the open directory, for example). If the server does not support directories, nothing will be done, and an error will be reported. Whatever is put into the string pointed to by *conn–>dir* will be subsequently reported to the user as being the current directory. Other than this, the FTPD library does not use the concept of current directory.

RETURN VALUES

Returns 0 if the current directory was changed successfully, -1 otherwise.

NOTES

All functions of the application side that accept a pathname parameter are assumed to interpret it in the context of the connection's current directory, according to the application's semantics. This is not emphasized in the other manual pages; all pathname arguments are simply referred to as "the file name", or "the directory name."

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

systemCommand – Performs the given command

SYNOPSIS

#include <arpa/ftpd/systemFilesys.h>
void **systemCommand**(FtpConn **conn*, char **cmd*, FILE **outstr*);

DESCRIPTION

The *systemCommand* function executes the command specified by *cmd*. The validity and interpretation is entirely up to the implementer, and writes the output of that command to *outstr*. Possible error messages related to the command are directed to the *outstr* with a simple *printf*(3STDC).

ERROR MESSAGES

`perror_reply(conn, 421, "control connection")` for problems with *outstr*

OK MESSAGES

`reply(226, "Transfer complete.")` to mark the end of the command output

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

systemDelete – Removes file specified

SYNOPSIS

#include <arpa/ftpd/systemFilesys.h>
int **systemDelete**(FtpConn **conn*, char **name*);

DESCRIPTION

Removes the file refernced by *name*. After successful completion, the file is no longer shown by any list command, any attempt to retrieve that file fails, and it is possible to create a new file of that name.

RETURN VALUES

Returns 0 if the file was deleted, -1 otherwise.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

systemFileSize – Reports the presence and size of the file specified

SYNOPSIS

```
#include <arpa/ftpd/systemFilesys.h>
int systemFileSize(FtpConn *conn, char *name, off_t *size, int isAscii);
```

DESCRIPTION

Report the size of the file *name* to **size*. If *isAscii* is non 0, the size reported is that of the file if it contains FTP-encoded text (that is, with lines terminated by \r\n).

RETURN VALUES

Returns 0 if a size was reported. Otherwise, returns -1 and does not change **size*.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

systemGunique – Creates a name for a new file

SYNOPSIS

```
#include <arpa/ftpd/systemFilesys.h>
char *systemGunique(FtpConn *conn, char *local);
```

DESCRIPTION

Creates a valid file name not already in use for an existing file, starting with *local*. The pathname returned is such that it is possible to create a file of that name. If applicable, the pathname returned is relative to the current directory, rather than *local*.

RETURN VALUES

Returns the new pathname if one could be elaborated. Otherwise, NULL is returned.

ERROR MESSAGES

perror_reply(conn, 451, "...")	For a resource allocation problem
perror_reply(conn, 553, "(directory)")	If *local* is not valid
reply(conn, 452, "...")	If no unique name could be found

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

systemLinesToOff – Reports offset of line in text file

SYNOPSIS

```
#include <arpa/ftpd/systemFilesys.h>
off_t systemLinesToOff(FtpConn *conn, char *name, int lines);
```

DESCRIPTION

Read the file referenced by *name*, until just past the number of lines specified by *lines* and return the current offset.

RETURN VALUES

The offset of the specified line, or -1 in case of failure.

ERROR MESSAGES

perror_reply(conn, 451, "...")	For a resource allocation problem
perror_reply(conn, 553, "(file name)")	For a file name problem
perror_reply(conn, 550, "(file name)")	For a file seek problem (including insufficient lines in the file)

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

systemListFiles – Lists the files matching the pattern specified

SYNOPSIS

```
#include <arpa/ftpd/systemFilesys.h>
void systemListFiles(FtpConn *conn, char *name, FILE *outstr, int isAscii);
```

DESCRIPTION

Output a simple listing of the file or files referred to by *name*. The interpretation of the file name and the format of the list is implementation dependent. If isAscii != 0 the lines of output are formatted as text lines (that is, terminated by \r\n).

RETURN VALUES

Returns 0 if the list could be created; -1 otherwise.

ERROR MESSAGES

perror_reply(conn, 451, "...")	For a resource allocation problem
perror_reply(conn, 426, "Data connection")	For a problem with *outstr*
reply(conn, 550, "not found")	If *name* does not refer to anything

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

systemLog, systemVlog – Adds the text given to the log

SYNOPSIS

```
#include <arpa/ftpd/systemLog.h>
void systemLog(int level, const char * format, ... );

void systemVlog(int level, const char * format, va_list ap);
```

DESCRIPTION

Records, in any way the implementer requires, the message represented by *format* and the following arguments. The *level* argument indicates the type of information as a small number. The possible types of information are defined in `systemLog.h` as one of:

```
LOG_INFO LOG_WARNING LOG_DEBUG LOG_NOTICE LOG_PID LOG_ERR LOG_NDELAY LOG_FTP
```

The format obeys the same syntax as the format argument of *printf* . The *systemLog* function accepts a variable number of arguments while *systemVlog* accepts a vector.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

systemLogwtmp – Record the given connection to wtmp

SYNOPSIS

```
#include <arpa/ftpd/systemLog.h>
void systemLogwtmp(FtpConn *conn);
```

DESCRIPTION

Logs the connection in *wtmp* or the equivalent, if applicable.

NOTES

The *wtmp* function may become inaccessible after performing a *chroot*(2POSIX). It is the responsibility of the application to make sure that *wtmp* is opened before using *chroot*(2POSIX).

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

systemMkdir – Create a directory of the name specified

SYNOPSIS

```
#include <arpa/ftpd/systemFilesys.h>
int systemMkdir(FtpConn *conn, char *name);
```

DESCRIPTION

Creates a new directory with the name *name*. After successful completion, a call to *systemChdir*(3FTPD) with the same arguments should succeed.

RETURN VALUES

Returns 0 if the directory was created, -1 otherwise.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

systemPass – Checks the user password

SYNOPSIS

#include <arpa/ftpd/systemAuth.h>
int **systemPass**(FtpConn **conn*);

DESCRIPTION

Checks that the password given by the user is the correct password according to implementation–defined criteria. The name of the user is *conn–>name*, and the password entered by the user is *conn–>passwd*. If the password is accepted, the connection is initialized to reflect the initial defaults associated with the user according to implementation–defined data bases. This may include changing the current directory to the home directory of the user (updating *conn–>dir* in multi-threaded conditions), and changing the root directory (if supported by the implementation). This may also include changing the shell (*conn–>shell*), if spawning of shell commands is supported. The *conn–>dir* parameter is already initialized to "/" and *conn–>shell* is already initialized to "". If the application changes the value of *conn–>dir* or *conn–>shell*, it should first free the memory pointed to by *conn–>dir* or *conn–>shell* using *free*(3STDC). When the session is finished, the FTPD library calls *free*(3STDC) on *conn–>dir* or *conn–>shell* if their value is not set to NULL. In a traditional UNIX implementation, if *conn–>guest* is 1, the password is not checked.

RETURN VALUES

Returns 0 if the password is correct, -1 otherwise.

OK REPLIES

lreply(conn, 230, "(Comments)")	Optional and only if access granted

ERROR MESSAGES

reply(conn, 530, "...")	For a bad password
reply(conn, 550, "...")	For a problem with the data base file

NOTES

The *conn–>dir* and *conn–>shell* parameters are for use by the application side; the FTPD library does not interpret them. If credentials are important, the necessary information should be initialized in the application-dependent part of the connection, to allow the *systemBeuser*(3FTPD) and *systemBesuper*(3FTPD) routines to perform correctly. (See these routines.)

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

systemReceiveAscii, systemReceiveBin – Stores text or binary data in the file specified

SYNOPSIS

#include <arpa/ftpd/systemFilesys.h>
int **systemReceiveAscii**(FtpConn * *conn*, FILE * *instr*, char * *name*, off_t *offset*);

int **systemReceiveBin**(FtpConn * *conn*, FILE * *instr*, char * *name*, off_t *offset*);

DESCRIPTION

Both routines read data from *FILE instr* until `EOF` is reached, and store it in the file called *name* , starting from offset *offset* . If the file does not exist yet, it is created. The effect of storing data in a file is implementation–dependent. The *systemReceiveAscii* routine gets lines of text (in the format defined by FTP) from the input stream and converts them to the file representation of text lines. The *systemReceiveBin* routine stores the data exactly as it receives it from the input stream. Both routines increment *conn–>byte_count* by the number of bytes received. These routines periodically call *ftpdOob* (3FTPD) to check for any urgent conditions on the control line. When calling *ftpdOob* (3FTPD), *conn–>byte_count* is up-to-date. In a traditional UNIX implementation, after successful completion of *systemReceiveAscii* , a call to *systemSendAscii* (3FTPD) using the same arguments should send a sequence of lines that begins with the same set of lines that were just received. In addition, if the file already contained lines prior to *systemReceiveAscii* , the lines not contained between *offset* and *offset* + `<number-of-bytes-received>` should be unchanged. In a traditional UNIX implementation, after successful completion of *systemReceiveBin* a call to *systemSendBin* (3FTPD) using the same arguments should send a flow of data that begins with the same stream of bytes that were just received. In addition, if the file already contained data prior to *systemReceiveBin* , the data not contained between *offset* and *offset* + `<number-of-bytes-received>` should be unchanged.

RETURN VALUES

Both routines return 0 if the transfer completed successfully, -1 otherwise.

ERROR MESSAGES

perror_reply(conn, 426, "...")	For a problem with *instr*
perror_reply(conn, 451, "...")	For a resource allocation problem
perror_reply(conn, 452, "...")	For a file write problem
perror_reply(conn, 550, "(file name)")	For a file seek problem
perror_reply(conn, 553, "(file name)")	For a file name problem

NOTES

The *ftpdOob* (3FTPD) routine may *longjmp* (3STDC) to an older stack frame instead of returning. See this routine for the necessary precautions.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

systemReceiveAscii, systemReceiveBin – Stores text or binary data in the file specified

SYNOPSIS

#include <arpa/ftpd/systemFilesys.h>
int **systemReceiveAscii**(FtpConn * *conn*, FILE * *instr*, char * *name*, off_t *offset*);

int **systemReceiveBin**(FtpConn * *conn*, FILE * *instr*, char * *name*, off_t *offset*);

DESCRIPTION

Both routines read data from *FILE instr* until `EOF` is reached, and store it in the file called *name* , starting from offset *offset* . If the file does not exist yet, it is created. The effect of storing data in a file is implementation–dependent. The *systemReceiveAscii* routine gets lines of text (in the format defined by FTP) from the input stream and converts them to the file representation of text lines. The *systemReceiveBin* routine stores the data exactly as it receives it from the input stream. Both routines increment *conn–>byte_count* by the number of bytes received. These routines periodically call *ftpdOob* (3FTPD) to check for any urgent conditions on the control line. When calling *ftpdOob* (3FTPD), *conn–>byte_count* is up-to-date. In a traditional UNIX implementation, after successful completion of *systemReceiveAscii* , a call to *systemSendAscii* (3FTPD) using the same arguments should send a sequence of lines that begins with the same set of lines that were just received. In addition, if the file already contained lines prior to *systemReceiveAscii* , the lines not contained between *offset* and *offset* + `<number-of-bytes-received>` should be unchanged. In a traditional UNIX implementation, after successful completion of *systemReceiveBin* a call to *systemSendBin* (3FTPD) using the same arguments should send a flow of data that begins with the same stream of bytes that were just received. In addition, if the file already contained data prior to *systemReceiveBin* , the data not contained between *offset* and *offset* + `<number-of-bytes-received>` should be unchanged.

RETURN VALUES

Both routines return 0 if the transfer completed successfully, -1 otherwise.

ERROR MESSAGES

perror_reply(conn, 426, "...")	For a problem with *instr*
perror_reply(conn, 451, "...")	For a resource allocation problem
perror_reply(conn, 452, "...")	For a file write problem
perror_reply(conn, 550, "(file name)")	For a file seek problem
perror_reply(conn, 553, "(file name)")	For a file name problem

NOTES

The *ftpdOob* (3FTPD) routine may *longjmp* (3STDC) to an older stack frame instead of returning. See this routine for the necessary precautions.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

systemRename – Moves a file

SYNOPSIS

```
#include <arpa/ftpd/systemFilesys.h>
int systemRename(FtpConn *conn, char *old, char *new);
```

DESCRIPTION

After successful completion, *old* is an invalid file name. This behaves in the same way as a successful *systemDelete*(3FTPD). The *new* parameter is a valid file name. The data formerly contained in *old* is now contained in *new*.

RETURN VALUES

Returns 0 if successful, -1 otherwise.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

systemRmdir – Removes the directory specified

SYNOPSIS

```
#include <arpa/ftpd/systemFilesys.h>
int systemRmdir(FtpConn *conn, char *name);
```

DESCRIPTION

Removes the directory specified by *name*. After successful completion, a call to *systemChdir* using the same arguments should fail, and a call to *systemMkdir* with the same arguments should succeed.

RETURN VALUES

Returns 0 if the directory was removed; -1 otherwise.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

systemSendAscii, systemSendBin – Retrieves text or binary data from the file specified

SYNOPSIS

```
#include <arpa/ftpd/systemFilesys.h>
int systemSendAscii(FtpConn * conn, char * name, FILE * outstr, off_t offset);

int systemSendBin(FtpConn * conn, char * name, FILE * outstr, off_t offset);
```

DESCRIPTION

Both routines read data from the file specified by *name* until end of file, starting from the offset *offset* , and write it to the output stream *outstr* . The *systemSendAscii* function outputs the data in the text format defined by FTP. The data in the file are assumed to be organized as text lines which must be output as strings of characters terminated by \\r\ , regardless of the text line representation in the file. The *systemSendBin* function is expected to output the data exactly as it was in the file. Both routines increment *conn–>byte_count* by the number of bytes sent. These routines periodically call *ftpdOob* (3FTPD) to check for any urgent condition on the control line. When calling *ftpdOob* (3FTPD), *conn–>byte_count* is updated.

RETURN VALUES

Returns 0 if the file was transferred successfully.Otherwise, returns -1.

ERROR MESSAGES

perror_reply(conn, 426, "...")	For a problem with *outsrt*
perror_reply(conn, 451, "...")	For a resource allocation problem
perror_reply(conn, 452, "...")	For a file read problem
perror_reply(conn, 550, "(file name)")	For a file seek problem
perror_reply(conn, 553, "(file name)")	For a file name problem

NOTES

The *ftpdOob* (3FTPD) routine may *longjmp* (3STDC) to an older stack frame instead of returning. See *ftpdOob* (3FTPD) for the necessary precautions.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

systemSendAscii, systemSendBin – Retrieves text or binary data from the file specified

SYNOPSIS

#include <arpa/ftpd/systemFilesys.h>
int **systemSendAscii**(FtpConn * *conn*, char * *name*, FILE * *outstr*, off_t *offset*);

int **systemSendBin**(FtpConn * *conn*, char * *name*, FILE * *outstr*, off_t *offset*);

DESCRIPTION

Both routines read data from the file specified by *name* until end of file, starting from the offset *offset* , and write it to the output stream *outstr* . The *systemSendAscii* function outputs the data in the text format defined by FTP. The data in the file are assumed to be organized as text lines which must be output as strings of characters terminated by `\\r\` , regardless of the text line representation in the file. The *systemSendBin* function is expected to output the data exactly as it was in the file. Both routines increment *conn–>byte_count* by the number of bytes sent. These routines periodically call *ftpdOob* (3FTPD) to check for any urgent condition on the control line. When calling *ftpdOob* (3FTPD), *conn–>byte_count* is updated.

RETURN VALUES

Returns 0 if the file was transferred successfully.Otherwise, returns -1.

ERROR MESSAGES

perror_reply(conn, 426, "...")	For a problem with *outsrt*
perror_reply(conn, 451, "...")	For a resource allocation problem
perror_reply(conn, 452, "...")	For a file read problem
perror_reply(conn, 550, "(file name)")	For a file seek problem
perror_reply(conn, 553, "(file name)")	For a file name problem

NOTES

The *ftpdOob* (3FTPD) routine may *longjmp* (3STDC) to an older stack frame instead of returning. See *ftpdOob* (3FTPD) for the necessary precautions.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

systemSetThreadTitle – Names the current thread with the text given

SYNOPSIS

#include <arpa/ftpd/systemLog.h>
void **systemSetThreadTitle**(const char **fmt*, ...);

DESCRIPTION

Each time the generic code of the FTPD library receives a new FTP command, it calls *systemSetThreadTitle* with a title that contains the client FTP host name, the client FTP user name and the name of the command which is being processed.

systemSetThreadTitle may ignore this title or use it to set the current thread name (this is useful when debugging multi-threaded FTPD servers).

Traditional UNIX implementations use this title to set the name of the FTPD process so that it can be viewed using *ps*.

The arguments of *systemSetThreadTitle* obey the same rules as the *printf* arguments.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

systemSleep – Sleep for the given number of seconds

SYNOPSIS

```
#include <arpa/ftpd/systemSleep.h>
void systemSleep(int t);
```

DESCRIPTION

Suspends the execution of the current thread for *t* seconds. Providing this function is optional; the FTPD library includes a default one.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

systemUser – Checks the user login

SYNOPSIS

```
#include <arpa/ftpd/systemAuth.h>
int systemUser(FtpConn *conn);
```

DESCRIPTION

Checks that the user can log in. The user's name is in *conn–>name*. If the user is a guest, *conn–>guest* is set to 1, otherwise it is set to 0.

RETURN VALUES

Returns 0 if the user is granted access, -1 otherwise.

OK REPLIES

lreply(conn, 331, "(Comments)") Optional and only if access is granted

ERROR MESSAGES

reply(conn, 530, "(Grumble)") If access is denied

NOTES

The *conn–>guest* parameter is interpreted by the generic code only insofar as the password of guest users is logged, while the password of non-guest users is not logged.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

systemLog, systemVlog – Adds the text given to the log

SYNOPSIS

#include <arpa/ftpd/systemLog.h>
void **systemLog**(int *level*, const char * *format*, ...);

void **systemVlog**(int *level*, const char * *format*, va_list *ap*);

DESCRIPTION

Records, in any way the implementer requires, the message represented by *format* and the following arguments. The *level* argument indicates the type of information as a small number. The possible types of information are defined in `systemLog.h` as one of:

```
LOG_INFO LOG_WARNING LOG_DEBUG LOG_NOTICE LOG_PID LOG_ERR LOG_NDELAY LOG_FTP
```

The format obeys the same syntax as the format argument of *printf* . The *systemLog* function accepts a variable number of arguments while *systemVlog* accepts a vector.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

Index